Table of Contents

WebRTC 文档翻译项目

中文版 WebRTC 1.0官方文档，由 RTC 开发者社区及 WebRTC 中文网发起的翻译、校对项目。旨在为 WebRTC 开发者们提供一份标准的 WebRTC API 文档，易于大家学习与开发。

目前，我们已经通过编写好的脚本程序（稍后开源，供有需要的人使用）完成了文档的初步翻译工作，翻译后的文档正在逐步添加中。欢迎参与校对。

贡献力量

如果希望为 RTC(Real-Time Communication) 社区贡献力量，欢迎：

- 参与校对
- 提出修改建议
- 协助改进术语翻译

如果你有兴趣参与进来，请仔细阅读：

- 为了有利阅读，翻译时请遵循 翻译排版指南.md
- 翻译流程，请参照 翻译校对流程.md

感谢本项目的贡献者们

请见《WebRTC 官方文档-中文版》贡献榜

参考

WebRTC 代码可参考官网或这个地址

术语表

术语	注解
STUN	Session Traversal Utilities for NAT，NAT 会话传输应用程序，一种网络协议，它允许位于 NAT（或多重 NAT）后的客户端找出自己的公网地址，查出自己位于哪种类型的 NAT 之后以及NAT为某一个本地端口所绑定的 Internet 端端口。这些信息被用来在两个同时处于 NAT 路由器之后的主机之间建立 UDP 通信。详见 RFC5389。

1.介绍

这部分内容与技术规范无关。

这份技术规范中引述了基于 HTML 的视频会议与点对点通信的一系列功能：

- 利用 NAT 穿透技术连接远程节点，如 ICE、STUN 和 TURN。
- 发送本地流至远端，或是从远端接收流。
- 发送任意类型数据至远端。

本文档定义了实现以上功能的接口。这份技术规范由 IETF RTCWEB 小组和 Media Capture Task Force 协同开发，前者开发贡献了一份协议技术规范，后者开发贡献了一套用于接入本地媒体设备的接口规范。你可以在 [RTCWEB-OVERVIEW] 和 [RTCWEB-SECURITY] 中找到系统概述。

2.一致性

如果有章节被标记为与技术规范无关（non-normative）时，这意味着其中的作者指引、图表、示例和注释内容均不属于技术规范。除此以外的其它章节均为技术规范。

在 [RTC2119] 中，定义描述了关键词**可能（MAY）**、**必须（MUST）**、**一定不（MUST NOT）**、**将（SHALL）**、**应该（SHOULD）**。

以任何形式出现的用于实践的算法、特定步骤，只要结果是正确的，即为符合一致性要求的表述。（注意，本规范说明中定义的算法旨在易用性，而不考虑性能。）

本说明规范中使用 ECMAScript 来实现的接口**必须**与 [WEBIDL-1] 中定义的 ECMAScript 约束一致。

1. Terminology zh:3.术语

The EventHandler interface, representing a callback used for event handlers, and the ErrorEvent interface are defined in [HTML51].

zh:事件处理器接口，代表用于响应事件的回调函数，错误事件的接口定义可以在[HTML51]中找到。

The concepts queue a task and networking task source are defined in [HTML51].

zh:队列任务与网络任务源的概念可以在[HTML51]中找到定义。

The concept fire an event is defined in [DOM].

zh:事件触发的概念可以在[DOM]中找到定义。

The terms event, event handlers and event handler event types are defined in [HTML51].

zh:术语"事件"、"事件处理器"和"事件类型"可以在[HTML51]中找到定义。

performance.timeOrigin and performance.now() are defined in [HIGHRES-TIME].

zh:performance.timeOrigin 和 performance.now() 分别可以在[HIGHRES-TIME]中找到定义。

The terms serializable objects, serialization steps, and deserialization steps are defined in [HTML].

zh:术语"可序列化对象"，"序列化步骤"和"反序列化步骤"可以在[HTML]中找到定义。

The terms MediaStream, MediaStreamTrack, and MediaStreamConstraints are defined in [GETUSERMEDIA]. Note that MediaStream is extended in the MediaStream section in this document while MediaStreamTrack is extended in the MediaStreamTrack section in this document.

zh:术语"MediaStream"，"MediaStreamTrack" 和 "MediaStreamConstraints" 可以在[GETUSERMEDIA]中找到定义。请注意，MediaStream 在本文档的 MediaStream 部分中进行了扩展，而 MediaStreamTrack 在本文档的 MediaStreamTrack 部分中进行了扩展。

The term Blob is defined in [FILEAPI].

zh:术语"Blob"可以在[FILEAPI]中找到定义。

The term media description is defined in [RFC4566].

zh:术语"媒体描述"可以在[RFC4566]中找到定义。

The term media transport is defined in [RFC7656].

zh:术语"媒体传输"可以在[RFC7656]中找到定义。

The term generation is defined in [TRICKLE-ICE] Section 2.

zh:术语"生成"可以在[TRICKLE-ICE]第2节中找到定义。

The terms RTCStatsType, stats object and monitored object are defined in [WEBRTC-STATS].

zh:术语"RTCStatsType"，"stats对象"和"被监控对象"可以在[WEBRTC-STATS]中找到定义。

When referring to exceptions, the terms throw and create are defined in [WEBIDL-1].

zh:在讨论异常时，术语 "throw" 和 "create" 可以在[WEBIDL-1]中找到定义。

The term "throw" is used as specified in [INFRA]: it terminates the current processing steps.

zh:术语 "throw" 按[INFRA]中的规定使用：它会即时终止当前的处理步骤。

The terms fulfilled, rejected, resolved, pending and settled used in the context of Promises are defined in [ECMASCRIPT-6.0].

zh:在Promises的上下文中使用的满足，拒绝，解决，待决和结算的术语可以在[ECMASCRIPT-6.0]中找到定义。

The terms bundle, bundle-only and bundle-policy are defined in [JSEP].

zh:术语"bundle"，"bundle-only"和"bundle-policy"可以在[JSEP]中找到定义。

The OAuth Client and Authorization Server roles are defined in [RFC6749] Section 1.1.

zh:OAuth客户端和授权服务器角色可以在[RFC6749]第1.1节中找到定义。

The terms isolated stream, peeridentity, request an identity assertion and validate the identity are defined in [WEBRTC-IDENTITY].

zh: 术语"隔离流"，"同等性"，"请求身份断言"和"验证身份"可以在[WEBRTC-IDENTITY]中找到定义。

The general principles for Javascript APIs apply, including the principle of run-to-completion and no-data-races as defined in [API-DESIGN-PRINCIPLES]. That is, while a task is running, external events do not influence what's visible to the Javascript application. For example, the amount of data buffered on a data channel will increase due to "send" calls while Javascript is executing, and the decrease due to packets being sent will be visible after a task checkpoint. It is the responsibility of the user agent to make sure the set of values presented to the application is consistent - for instance that getContributingSources() (which is synchronous) returns values for all sources measured at the same time.

zh: 本文档遵循基本的 Javascript 接口原则，包括[API-DESIGN-PRINCIPLES]中定义的 run-to-completion 和 no-data-race 的原则。也就是说，在任务运行时，外部事件不会影响 Javascript 应用程序可见的内容。例如，在执行 JavaScript 应用时，数据通道上缓冲的数据量将因为"发送"的调用增加，而由于数据发送成功导致的数据通道上缓冲数据量的减少，只有在这个任务的检查点之后才能观察到。用户端需要确保在这一刻展现给应用的数值是恒定不变的 - 例如同一时间内 getContributingSources（）（同步调用）返回的值。

4.点对点连接

4.1简介

RTCPeerConnection实例允许应用程序与另一个浏览器中的另一个RTCPeerConnection实例或另一个实现了所需协议的端点建立对等通信。通过信令信道交换控制消息（称为信令协议）来协调通信，信令信道没有指定的实现方法，通常由页面中的脚本连接服务器提供。例如，使用XMLHttpRequest [XMLHttpRequest]或WebSockets[WEBSOCKETS-API]。

4.2配置

4.2.1 RTCConfiguration字典

RTCConfiguration定义了一组参数，用于配置 RTCPeerConnection 如何建立或重新建立点对点通信。

```
dictionary RTCConfiguration {
  sequence<RTCIceServer> iceServers;
  RTCIceTransportPolicy iceTransportPolicy = "all";
  RTCBundlePolicy bundlePolicy = "balanced";
  RTCRtcpMuxPolicy rtcpMuxPolicy = "require";
  DOMString peerIdentity;
  sequence<RTCCertificate> certificates;
  [EnforceRange]
  octet iceCandidatePoolSize = 0;
};
```

字典RTCConfiguration成员

iceServers of type sequence: zh:类的 `iceServers`

```
一组可供ICE使用的服务器的描述，例如 STUN 和 TURN 服务器。
```

一组可供ICE使用的服务器的描述，例如 STUN 和 TURN 服务器。

iceTransportPolicy of type RTCIceTransportPolicy, defaulting to "all": zh:RTCIceTransportPolicy 类型的 iceTransportPolicy，默认为“all”

```
指示允许ICE代理使用哪些候选者。
```

指示允许ICE代理使用哪些候选者。

bundlePolicy of type RTCBundlePolicy, defaulting to "balanced": zh:类型为 RTCBundlePolicy 的 bundlePolicy ，默认为“balanced”

```
指示在收集ICE候选项时要使用的媒体捆绑策略。
```

指示在收集ICE候选项时要使用的媒体捆绑策略。

rtcpMuxPolicy of type RTCRtcpMuxPolicy, defaulting to "require": zh:RTCPtcpMuxPolicy类型的rtcpMuxPolicy，默认为“require”

```
指示收集ICE候选项时要使用的 rtcp-mux 策略。
```

指示收集ICE候选项时要使用的 rtcp-mux 策略。

peerIdentity of type DOMString: zh:DOMString类型的peerIdentity

```
设置 RTCPeerConnection 目标对端的标识。 RTCPeerConnection 不会与对端建立连接，除非对端提供
```

设置 RTCPeerConnection 目标对端的标识。 RTCPeerConnection 不会与对端建立连接，除非对端提供名称并用该名称成功验证身份。

certificates of type sequence: 序列号的证书。

RTCPeerConnection 用于进行身份验证的一组证书。

通过调用 generateCertificate 函数创建此参数的有效值

虽然任何给定的 DTLS 连接仅使用一个证书，但此属性允许调用者提供支持不同算法的多个证书。将根据 DTLS 握手允许哪些证书，来选择最终证书。RTCPeerConnection 实现选择将哪个证书用于给定连接;如何选择证书超出了本规范的范围。

如果此值不存在，则为每个 RTCPeerConnection 实例生成一组默认证书。

此选项允许应用程序建立密钥连续性。 RTCCertificate 可以保存在 [INDEXEDDB] 中并重复使用。持久性和重用也避免了密钥生成的成本。

最初选择此值后，此配置选项的值不能更改

octet类型的iceCandidatePoolSize，默认为0

```
 Size of the prefetched ICE pool as defined in [JSEP] (section 3.5.4. and section 4.1.
zh: [JSEP]（第3.5.4节和第4.1.1节）中定义的预取ICE池的大小。
```

Size of the prefetched ICE pool as defined in [JSEP] (section 3.5.4. and section 4.1.1.).

[JSEP]（第3.5.4节和第4.1.1节）中定义的预取ICE池的大小。

4.2.2 RTCIceCredentialType Enum

```
enum RTCIceCredentialType {
  "password",
  "oauth"
};
```

Enumeration description	
password	The credential is a long-term authentication username and password, as described in [RFC5389], Section 10.2.
oauth	基于OAuth 2.0的身份验证方法，如[RFC7635]中所述。 对于OAuth身份验证，ICE代理需要三个凭据信息。凭证由用于 kid，macKey 和 accessToken 组成。其中kid用于RTCIceServer 成员变量，macKey 和 accessToken放置在RTCOAuthCredential字典中的。 此规范未定义应用程序（充当OAuth客户端）如何从授权服务器获取accessToken，kid和macKey，因为WebRTC仅处理ICE代理和TURN服务器之间的交互。例如，应用程序可以使用OAuth 2.0 Implicit Grant类型和PoP（Proof-of-Possession）令牌类型，如[RFC6749]和[OAUTH-POP-KEY-DISTRIBUTION]中所述; [RFC7635]附录B中提供了这方面的一个例子。 NOTE：作为OAuth客户端的应用程序负责在accessToken到期之前刷新凭据信息并使用新的凭据更新ICE代理。 OAuth客户端可以使用 RTCPeerConnection 的 setConfiguration 方法定期刷新 TURN 凭据。 HMAC密钥的长度（RTCOAuthCredential.macKey）可以是大于 20（160位）的任何整数字节数。 NOTE：根据[RFC7635]第 4.1 节，HMAC密钥必须是对称密钥，因为非对称密钥会导致大的访问令牌，这可能不适合单个 STUN 消息。 NOTE：目前，STUN / TURN 协议仅使用SHA-1和SHA-2系列哈希算法进行消息完整性保护，如[RFC5389]第15.4节和[STUN-BIS]第14.6节中所定义。

4.2.3 RTCOAuthCredential Dictionary

RTCOAuthCredential字典用于描述OAuth auth凭证信息，STUN / TURN 客户端（在 ICE 代理内部）使用该信息对 STUN / TURN 务器进行身份验证，如[RFC7635]中所述。请注意，kid参数不在此字典中，而是在RTCIceServer的成员变量 username 中。

```
dictionary RTCOAuthCredential {
  required DOMString macKey;
  required DOMString accessToken;
};
```

字典RTCOAuthCredential成员

DOMString类型的`macKey`，需要：

```
`mac_key`，如[RFC7635]第 6.2 节中所述，采用 base64-url 编码格式。它用于 STUN 消息完整性哈希
```

DOMString类型的`accessToken`，需要：

```
`access_token`，如[RFC7635]第6.2节中所述，采用 base64 编码格式。这是一个加密的自包含令牌，
```

RTCOAuthCredential字典的一个示例是：

```
{
  macKey: 'WmtzanB3ZW9peFhtdm42NzUzNG0=',
  accessToken: 'AAwg3kPHWPfvk9bDFL936wYvkoctMADzQ5VhNDgeMR3+ZlZ35byg972fW8QjpEl7bx91YL
}
```

4.2.4 RTCIceServer字典

RTCIceServer 字典用于描述可以被 ICE 代理用于与对等方建立连接的STUN和TURN服务器。

```
dictionary RTCIceServer {
  required (DOMString or sequence<DOMString>) urls;
  DOMString username;
  (DOMString or RTCOAuthCredential) credential;
  RTCIceCredentialType credentialType = "password";
};
```

字典RTCIceServer成员

`URL` 的类型是 DOMString或序列，必填

```
zh: [RFC7064]和[RFC7065]或其他URI类型中定义的STUN或TURN URI。
```

`username` 的类型是 DOMString

如果此 RTCIceServer 对象表示 TURN 服务器，并且 credentialType 为“password”，则此属性指定用于该TURN服务器的用户名。

如果此RTCIceServer对象表示TURN服务器，并且credentialType为“oauth”，则此属性指定共享对称密钥的密钥ID（kid），该密钥在TURN服务器和授权服务器之间共享，如[RFC7635]中所述。它是一个短暂而唯一的密钥标识符。密钥ID（kid）允许TURN服务器选择适当的密钥材料来解密Access-Token，因此这个密钥ID（kid）识别的密钥被用于“access_token”的加密。 kid值与OAuth响应“kid”参数相同，如[RFC7515]第4.1.4节中所定义。

`credential` 的类型是 DOMString 或 RTCOAuthCredential

如果此 RTCIceServer 对象表示 TURN 服务器，则此属性是要与该 TURN 服务器一起使用的证书。

如果credentialType是“password”，则证书是DOMString类型，并表示长期验证密码，如[RFC5389]第10.2节中所述。

如果credentialType是“oauth”，则证书是RTCOAuthCredential类型，其中包含OAuth访问令牌和MAC密钥。

`credentialType` 的类型是 RTCIceCredentialType，默认为“password”

如果此RTCIceServer对象表示TURN服务器，则此属性指定如何在该 TURN 服务器请求授权时使用证书。

An example array of RTCIceServer objects is:

一个 RTCIceServer 数组的例子：

```
[
  {urls: 'stun:stun1.example.net'},
  {urls: ['turns:turn.example.org', 'turn:turn.example.net'],
    username: 'user',
    credential: 'myPassword',
    credentialType: 'password'},
  {urls: 'turns:turn2.example.net',
    username: '22BIjxU93h/IgwEb',
    credential: {
      macKey: 'WmtzanB3ZW9peFhtdm42NzUzNG0=',
      accessToken: 'AAwg3kPHWPfvk9bDFL936wYvkoctMADzQ5VhNDgeMR3+ZlZ35byg972fW8QjpEl7bx
    },
    credentialType: 'oauth'}
];
```

4.2.5 RTCIceTransportPolicy 枚举

如[JSEP]（第4.1.1节）中所述，如果指定了RTCConfiguration的iceTransportPolicy成员，则它定义了浏览器获取ICE候选地址的策略[JSEP]（第3.5.3节）;只有这些候选地址才会用于连接性检查。

```
enum RTCIceTransportPolicy {
  "relay",
  "all"
};
```

Enumeration description (non-normative)	
relay	ICE代理仅使用媒体中继候选地址，例如通过TURN服务器的候选地址。 NOTE：这可以用于防止远端了解到用户的IP地址，这可能用于一些特定的情况下。例如，在基于“呼叫”的应用程序中，应用程序可能不希望未知呼叫者知道被呼叫者的IP地址，直到被呼叫者以某种方式同意为止。
all	指定此值时，ICE代理可以使用任何类型的候选地址。 该实现仍然可以使用其自己的候选过滤策略，以限制暴露IP地址给应用程序，如RTCIceCandidate.address的描述中所述。

4.2.6 RTCBundlePolicy 枚举

如[JSEP]（第4.1.1节）中所述，如果远程端点不支持bundle策略，会影响协商哪些媒体轨道的协商，以及会收集那些 ICE 候选地址。如果远端支持 bundle 策略，则所有媒体轨道和数据通道都捆绑在同一传输路径上。

```
enum RTCBundlePolicy {
  "balanced",
  "max-compat",
  "max-bundle"
};
```

Enumeration description (non-normative)	
balanced	为正在使用的每种媒体类型（音频，视频和数据）收集 ICE 候选地址。如果远程端点不支持 bundle 策略，则在一个传输路径上仅协商一个音频和视频轨道。
max-compat	收集每个媒体轨道的 ICE 候选地址。如果远程端点不支持 bundle，则在一个传输路径上协商所有媒体轨道。
max-bundle	只收集一个媒体轨道的 ICE 候选地址。如果远程端点不支持 bundle 策略，则仅协商一个媒体轨道。

4.2.7 RTCRtcpMuxPolicy 枚举

如[JSEP]（第4.1.1节）中所述，RtcpMuxPolicy会影响为支持非多路复用而收集哪些ICE候选项。

```
enum RTCRtcpMuxPolicy {
  // At risk due to lack of implementers' interest.
  "negotiate",
  "require"
};
```

Enumeration description (non-normative)	
negotiate	同时收集RTP和RTCP的ICE候选项。如果远程端点能够复用RTCP，则在RTP候选地址上复用RTCP。如果不是，请分开使用RTP和RTCP候选项。请注意，如[JSEP]（第4.1.1节）中所述，用户代理可能无法实现非多路复用的RTCP，在这种情况下，它将拒绝使用协商策略创建 RTCPeerConnection 实例的尝试。
require	只收集RTC候选地址和在RTP上复用了RTCP的候选地址。如果远程端点不支持rtcp-mux，则会话协商将失败。

> FEATURE AT RISK 1
>
> 支持非多路复用RTP / RTCP的本规范的各个方面被标记为存在风险的特征，因为实施者没有明确的承诺。这包括：
>
> 1. 对于negotiate值，实施者没有明确承诺与此相关的行为。
> 2. 支持RTCRtpSender和RTCRtpReceiver中的rtcpTransport属性。

4.2.8 Offer/answer option 提供、应答选项

这些词典描述了可用于控制 offer/answer 创建过程的选项。

```
dictionary RTCOfferAnswerOptions {
  boolean voiceActivityDetection = true;
};
```

字典 `RTCOfferAnswerOptions` 的成员

`voiceActivityDetection` 类型为 boolean，默认为 `true`

```
  许多编解码器和系统能够通过诸如不传输任何媒体的这类方式来检测“静音”并改变它们的行为。在许多情况
```

```
dictionary RTCOfferOptions : RTCOfferAnswerOptions {
  boolean iceRestart = false;
};
```

字典 `RTCOfferOptions` 的成员

iceRestart 的类型为 boolean ， 默认为 false

```
  当此词典成员的值为 true 时，生成的 description 将具有与当前凭据不同的 ICE 凭证（在localDes

当此词典成员的值为 true 时，生成的 description 将具有与当前凭据不同的 ICE 凭证（在 localDes
```

> NOTE
>
> 当 `iceConnectionState` 转换为“failed”时，建议执行 ICE 重启。应用程序可另外选择侦听 `iceConnectionState` 转换为“disconnected”，然后使用其他信息源（例如使用 `getStats` 来测量在接下来的几秒内发送或接收的字节数是否增加）以确定是否应该重新启动 ICE。

`RTCAnswerOptions` 字典描述特定于类型为answer的会话描述的选项（在此版本的规范中没有）。

```
dictionary RTCAnswerOptions : RTCOfferAnswerOptions {
};
```

4.3 状态定义

4.3.1 RTCSignalingState 枚举

```
enum RTCSignalingState {
  "stable",
  "have-local-offer",
  "have-remote-offer",
  "have-local-pranswer",
  "have-remote-pranswer",
  "closed"
};
```

Enumeration description	
stable	没有 offer/answer 交换正在进行。这也是初始状态，在这种情况下，本地和远程描述为空。
have-local-offer	成功使用“offer”类型的本地描述。
have-remote-offer	成功使用“offer”类型的远程描述。
have-local-pranswer	zh：已成功应用“offer”类型的远程描述，并已成功应用“pranswer”类型的本地描述。
have-remote-pranswer	已成功应用“offer”类型的本地描述，并且已成功应用“pranswer”类型的远程描述。
closed	`RTCPeerConnection`已关闭;它的[[IsClosed]]值是 true 的。

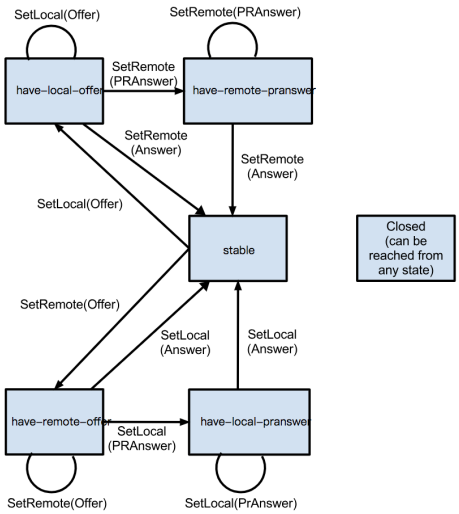


Figure 1 Non-normative signalling state transitions diagram. Method calls abbreviated.

An example set of transitions might be:

一组状态转换示例：

呼叫者状态转换：

- new RTCPeerConnection(): stable
- setLocalDescription(offer): have-local-offer
- setRemoteDescription(pranswer): have-remote-pranswer
- setRemoteDescription(answer): stable

被呼叫者状态转换：

- new RTCPeerConnection(): stable
- setRemoteDescription(offer): have-remote-offer
- setLocalDescription(pranswer): have-local-pranswer
- setLocalDescription(answer): stable

4.3.2 RTCIceGatheringState 枚举

```
enum RTCIceGatheringState {
  "new",
  "gathering",
  "complete"
};
```

Enumeration description	
new	任何 RTCIceTransport 都处于“new”收集状态，并且没有任何传输处于“gathering”状态，或者没有传输。
gathering	任何 RTCIceTransport 都处于“gathering”收集状态。
complete	至少存在一个 RTCIceTransport，并且所有RTCIceTransport 都处于“已完成”的收集状态。

4.3.3 RTCPeerConnectionState 枚举

```
enum RTCPeerConnectionState {
  "closed",
  "failed",
  "disconnected",
  "new",
  "connecting",
  "connected"
};
```

Enumeration description	
closed	RTCPeerConnection对象的[[IsClosed]]值为true。
failed	之前的状态不适用，任何RTCIceTransports或RTCDtlsTransports都处于“failed”状态。
disconnected	之前的状态不适用，任何RTCIceTransports或RTCDtlsTransports都处于“disconnected”状态。
new	以前的状态都不适用，并且所有RTCIceTransports或RTCDtlsTransport都处于“new”或“closed”状态，或者当前没有传输。
connecting	以前的状态都不适用，并且所有RTCIceTransports或RTCDtlsTransport都处于“new”，“connecting”或”checking”状态。
connected	以前的状态均不适用，并且所有RTCIceTransports和RTCDtlsTransports都处于“connected”，“completed”或“closed”状态。

4.3.4 RTCIceConnectionState 枚举

```
enum RTCIceConnectionState {
  "closed",
  "failed",
  "disconnected",
  "new",
  "checking",
  "completed",
  "connected"
};
```

Enumeration description	
closed	RTCPeerConnection对象的[[IsClosed]]值为true。
failed	之前的状态不适用，并且任何RTCIceTransport都处于“failed”状态。
disconnected	以前的状态都不适用，并且任何RTCIceTransport都处于“disconnected”状态。
new	以前的状态都不适用，并且所有RTCIceTransport都处于“new”或“closed”状态，或者没有传输。
checking	以前的状态都不适用，并且任何RTCIceTransport都处于new”或“checking”状态。
completed	以前的状态均不适用，并且所有RTCIceTransport都处于“completed”或“closed”状态。
connected	以前的状态均不适用，并且所有RTCIceTransport都处于“connected”，“completed”或“closed”状态。

注意，如果由于信令（例如，RTCP多路复用或捆绑）而丢弃RTCIceTransport，或者由于信令（例如，添加新媒体描述）而创建RTCIceTransport，则状态可以直接从一个状态前进到另一个状态。

4.4 RTCPeerConnection 接口

[JSEP]规范作为一个整体，描述了 `RTCPeerConnection` 如何运行的细节。适当时提供对[JSEP]特定小节的参考。

4.4.1 操作

调用new RTCPeerConnection（configuration）会创建RTCPeerConnection对象。

configuration.servers 包含 ICE 用于查找和访问服务器的信息。应用程序可以提供同种类的服务器的多个实例，并且任何TURN服务器也可以用作STUN服务器，以便收集服务器自反候选者。

RTCPeerConnection对象具有信令状态，连接状态，ICE收集状态和ICE连接状态。这些状态在创建对象时被初始化。

RTCPeerConnection的ICE协议实现用 ICE 代理[ICE]表示。某些RTCPeerConnection方法涉及与 ICE 代理 的交互被命名为：addIceCandidate，setConfiguration，setLocalDescription，setRemoteDescription和close。这些交互在本文档和[JSEP]的相关章节中进行了描述。当 RTCIceTransport 的内部表示状态发生变化时，ICE 代理还向用户代理提供指示，如5.6 RTCIceTransport接口中所述。

本节中列出的任务的任务源是网络任务源。

4.4.1.1 构造函数

当调用 `RTCPeerConnection()` 构造函数时，用户代理必须运行以下步骤：

1. 如果由于此处未指定的原因而导致下面列举的任何步骤失败，则抛出UnknownError并将“message”字段设置为适当的描述。
2. 让connection成为新创建的RTCPeerConnection对象。
3. 如果配置中的证书集为非空，请对证书集中的每个证书运行以下步骤：
    i. 如果证书expires属性的值不在将来，则抛出InvalidAccessError。
    ii. 如果证书[[Origin]]值与当前证书的[[Origin]]不同，则抛出InvalidAccessError。
    iii. 存储证书。
4. 然后，使用此 RTCPeerConnection 实例生成一个或多个新的 RTCCertificate 实例并存储它们。这可能是异步发生的，所以后续步骤的证书值可能仍未定义。如[RTCWEB-SECURITY]第4.3.2.3节所述，WebRTC使用自签名而不是公钥基础结构（PKI）证书，因此到期检查是为了确保密钥不会无限期使用，并且不需要额外的证书检查。
5. 初始化连接的ICE代理。
6. 让连接有[[Configuration]]内部值。设置 configuration 参数指定的配置。
7. 让连接有一个[[IsClosed]]内部值，初始化为 false。
8. 让连接有一个[[NegotiationNeeded]]内部值，初始化为 false。
9. 让连接有一个[[SctpTransport]]内部值，初始化为 null。
10. 让连接具有[[Operations]]内部值，表示操作队列，初始化为空列表。
11. 让连接有一个[[LastOffer]]内部值，初始化为“”。
12. 让连接有一个[[LastAnswer]]内部值，初始化为“”。
13. 将连接的信令状态设置为“stable”。
14. 将连接的ICE连接状态设置为“new”。
15. 将连接的ICE收集状态设置为“new”。
16. 将连接的连接状态设置为“new”。
17. 让连接有一个[[PendingLocalDescription]]内部值，初始化为null。
18. 让连接有一个[[CurrentLocalDescription]]内部值，初始化为null。
19. 让连接有一个[[PendingRemoteDescription]]内部值，初始化为null。
20. 让连接有一个[[CurrentRemoteDescription]]内部值，初始化为null。
21. 返回连接。

4.4.1.2 将操作排入队列

RTCPeerConnection对象有一个操作队列[[Operations]]，它确保队列中只有一个异步操作同时执行。如果下一个 promise 请求被调用时之前的 promise 还没返回，则将它们添加到队列中并等到队列前面所有的 promise 执行完毕并返回后再执行。

要将操作加入 RTCPeerConnection 对象的操作队列，请运行以下步骤：

1. 让连接成为 RTCPeerConnection 对象。
2. 如果 connection 的[[IsClosed]] 值为true，用 promise 包装一个新创建的 InvalidStateError 并以reject返回。
3. 让操作成为即将入队的那一项。
4. 创建新 promise ， 名为 p
5. 将操作附加到[[操作]]列表中。
6. 如果[[操作]]的长度恰好为1，则执行操作。
7. promise返回后，如果状态是 fulfillment 或 rejection，请执行以下步骤：
    i. 如果connection的[[IsClosed]]值为true，则中止这些步骤。
    ii. 如果操作返回的 promise 已完成并有值，则将 p 以 fulfill 的状态返回该值。
    iii. 如果操作返回的 promise 被拒绝并有值，则将 p 以 rejected 的状态返回该值。
    iv. 完成或拒绝p后，执行以下步骤：
        i. 如果connection的[[IsClosed]]值为true，则中止这些步骤。
        ii. 删除[[操作]]的第一个元素。
        iii. 如果[[操作]]非空，则执行[[操作]]队列中的第一个操作。
1. 返回 p。

4.4.1.3 更新连接状态

RTCPeerConnection 对象具有聚合连接状态。每当RTCDtlsTransport或RTCIceTransport的状态发生变化或[[IsClosed]]值变为true时，用户代理必须通过加入任务到队列来更新连接状态，任务执行以下步骤：

1. 让连接成为此RTCPeerConnection对象。
2. 把 newState 的值设为新状态的值，这些值是 RTCPeerConnectionState 中的枚举值。
3. 如果连接的状态值等于newState的值，则中止这些步骤。
4. 把newState的值赋给连接的状态值。
5. 触发名为 connectionstatechange 的事件。

4.4.1.4 更新 ICE 收集状态

要更新RTCPeerConnection实例连接的ICE收集状态，用户代理必须把运行以下步骤的任务加入队列：

1. 如果connection的[[IsClosed]]值为true，则中止这些步骤。
2. 把 newState 的值设为新状态的值，这些值是 RTCIceGatheringState 中的枚举值。
3. 如果连接的ICE收集状态的值等于newState的值，则中止这些步骤。
4. 将连接的 ICE 采集状态值设置为newState的值。
5. 触发名为icegatheringstatechange的事件。
6. 如果 newState 的值为“completed”，则使用 RTCPeerConnectionIceEvent 接口触发名为 icecandidate 的事件，并在候选连接时将候选属性设置为null。

> NOTE
>
> 会触发null候选事件以确保旧版兼容性。新代码应监视 RTCIceTransport 和/或 RTCPeerConnection 的收集状态。

4.4.1.5 更新 ICE 连接状态

要更新RTCPeerConnection实例连接的ICE连接状态，用户代理必须把运行以下步骤的任务加入队列：

1. 如果connection的[[IsClosed]]值为true，则中止这些步骤。
2. 把 newState 的值设为新状态的值，这些值是 RTCIceConnectionState 中的枚举值。
3. 如果connection的ICE连接状态值等于newState的值，则中止这些步骤。
4. 将connection的连接状态值设置为newState的值。
5. 触发名为iceconnectionstatechange的事件。

4.4.1.6 设置 RTCSessionDescription

要在 RTCPeerConnection 对象连接上设置 RTCSessionDescription 描述，请将运行以下步骤加入 connection 的操作队列：

1. 让 p 成为新的 promise。
2. 同时，按照[JSEP]（第5.5节和第5.6节）中的描述，启动应用 description 的流程。
    i. 如果应用 description 的过程因任何原因失败，则用户代理必须将运行以下步骤的任务加入队列：
        i. 如果 connection 的[[IsClosed]] 值为true，则中止这些步骤。
        ii. 如果 description 的类型对于当前信令连接状态无效，如[JSEP]（第5.5节和第5.6节）中所述，则使用新创建的 InvalidStateError 错误，p 以拒绝的状态返回这个错误，并中止这些步骤。
        iii. 如果 description 被设置为本地描述，如果 escription.type 是 offer 并且description.sdp 不等于 connection 的[[LastOffer]]，则创建新的 InvalidModificationError 错误，p 以拒绝的状态返回这个错误，并中止这些步骤。
        iv. 如果将 description 设置为本地描述，如果 description.type 为“rollback”且信令状态为“stable”，则创建新的 InvalidStateError 错误，p以拒绝的状态返回这个错误，并中止这些步骤。
        v. 如果description被设置为本地描述，如果description.type是“answer”或“pranswer”并且description.sdp不等于connection的[[LastAnswer]]，则创建新的 InvalidModificationError 错误，p 以拒绝的状态返回这个错误，并中止这些步骤。
        vi. zh: 如果 description 内容不是有效的SDP语法，则使用 RTCError 拒绝 p（将errorDetail设置为“sdp-syntax-error”并将 sdpLineNumber 属性设置为检测到语法错误的SDP中的行号）并中止这些步骤。
        vii. 如果将 description 设置为远程描述，则连接的 RTCRtcpMuxPolicy 是必需的，远程描述不使用 RTCP mux，然后创建新的 InvalidAccessError 错误，p 以拒绝的状态返回这个错误，并中止这些步骤。
        viii. 如果 description 内容无效，则创建新的 InvalidAccessError 错误，p 以拒绝的状态返回这个错误，并中止这些步骤。
        ix. 对于所有其他错误，创建新的 OperationError 错误，p 以拒绝的状态返回这个错误。
    ii. zh:如果成功应用 description，则用户代理必须对运行以下步骤的任务进行排队：
        i. 如果connection的[[IsClosed]] 值为true，则中止这些步骤。
        ii. 如果将 description 设置为本地描述，则运行以下步骤之一：

- 如果 description 是“offer”类型，则将connection的[[PendingLocalDescription]]设置为从 description 构造出的新 RTCSessionDescription 对象，并将信令状态设置为“have-local-offer”。
- 如果 description 的类型为“answer”，那么这就完成了 offer/answer 协商。将[[CurrentLocalDescription]]连接到从 description 构造的新 RTCSessionDescription 对象，并设置 connection的[[PendingRemoteDescription]]为connection的[[CurrentRemoteDescription]]。将connection的[[PendingRemoteDescription]]和connection的[[PendingLocalDescription]]设置为null。最后将连接的信令状态设置为“stable”。
- 如果 description 是“rollback”类型，那么这是一个回滚。将connection的[[PendingLocalDescription]]设置为null，并将信令状态设置为“stable”。
- 如果 description 是“pranswer”类型，则将connection的[[PendingLocalDescription]]设置为从 description 构造出的新 RTCSessionDescription 对象，并将信令状态设置为“have-local-pranswer”。

iii. 否则，如果将 description 设置为远程描述，则运行以下步骤之一：

- 如果将 description 设置为远程描述，如果description.type为“rollback”且信令状态为“stable”，则使用新创建的InvalidStateError拒绝p并中止这些步骤。
- 如果 description 是“offer”类型，则将connection的[[PendingRemoteDescription]]属性设置为从 description 构造出的新 RTCSessionDescription 对象，并将信令状态设置为“have-remote-offer”。
- 如果 description 的类型为“answer”，那么这就完成了 offer/answer 协商。将connection的[[CurrentRemoteDescription]]属性设置为从 description 构造出的新 RTCSessionDescription 对象，并设置connection的[[CurrentLocalDescription]]到connection的[[PendingLocalDescription]]。将connection的[[PendingRemoteDescription]]和connection的[[PendingLocalDescription]]设置为null。最后将连接的信令状态设置为“stable”。
- 如果 description 是“rollback”类型，那么这是一个回滚。将connection的[[PendingRemoteDescription]]设置为 null，并将信令状态设置为“stable”。
- 如果 description 是“pranswer”类型，则将connection的[[PendingRemoteDescription]]设置为从 description 构造出的新 RTCSessionDescription 对象。最后将连接的信令状态设置为“have-remote-pranswer”。

iv. 如果 description 是“answer”类型，并且它启动现有 SCTP 关联的关闭，如[SCTP-SDP]第10.3和10.4节中所定义，则将 connection 的[[SctpTransport]]的值设置为null。

v. 如果 description 的类型为“answer”或“pranswer”，则执行以下步骤：

    i. 如果 description 启动了新 SCTP 关联的建立，如[SCTP-SDP]第10.3和10.4节中所定义，则创建一个初始状态为“connecting”的RTCSctpTransport ，并将结果分配给[[SctpTransport]]。

    ii. 否则，如果建立了 SCTP 关联，但更新了“max-message-size” SDP 属性，则更新connection的[[SctpTransport]]的最大消息大小。

    iii. 如果 description 协商 SCTP 传输的 DTLS 角色，并且存在具有空 id 的 RTCDataChannel，则根据[RTCWEB-DATA-PROTOCOL]生成ID。如果无法生成可用的ID，请运行以下步骤：

        i. 设channel为无法生成ID的 RTCDataChannel 对象。

        ii. 将channel的[[ReadyState]]设置为“closed”。

        iii. 使用 RTCErrorEvent 接口触发名为 error 的事件，并在 channel 中将 errorDetail 属性设置为“data-channel-failure”。

        iv. 在 channel 上触发一个名为 close 的事件。

vi. 让trackEventInits，muteTracks，addList和removeList为空列表。

vii. 如果将 description 设置为本地描述，则运行以下步骤：

    i. 对 description 中的每个媒体描述运行以下步骤：

        i. 如果媒体描述尚未与 RTCRtpTransceiver 对象关联，请运行以下步骤：

            i. 让收发器成为用于创建媒体描述的 RTCRtpTransceiver。

            ii. 将收发器的中间值设置为媒体描述的中间值。

            iii. 如果收发器的[[Stopped]]为true，则中止这些子步骤。

            iv. 如果根据[BUNDLE]将媒体描述指示为使用现有媒体传输，则让 transport 和 rtcpTransport 分别为表示该传输的 RTP 和 RTCP 组件的R TCDtlsTransport 对象。

            v. 否则，让 transport 和 rtcpTransport 成为新创建的 RTCDtlsTransport 对象，每个对象都有一个新的底层 RTCIceTransport。虽然如果根据[RFC5761]协商 RTCP 多路复用，或者如果需要 connection 的 RTCRtcpMuxPolicy，则不要创建任何特定于 RTCP 的传输对象，而是让 rtcpTransport 等于传输。

vi. 设置transceiver.[[Sender]].[[SenderTransport]]进行传输。

vii. 将transceiver.[[Sender]].[[SenderRtcpTransport]]设置为 rtcpTransport。

viii. 设置transceiver.[[接收器]].[[ReceiverTransport]]进行传输。

ix. 将transceiver.[[Receiver]].[[ReceiverRtcpTransport]]设置为 rtcpTransport。

ii. 让收发器成为与媒体描述相关联的 RTCRtpTransceiver。

iii. 如果收发器的[[Stopped]]值为true，则中止这些子步骤。

iv. 设方向是 RTCRtpTransceiverDirection 值，表示媒体描述的方向。

v. 如果 direction 是“sendrecv”或“recvonly”，则将收发器的[[Receptive]]值设置为true，否则将其设置为false。

vi. 如果 description 的类型为“answer”或“pranswer”，则执行以下步骤：

i. 如果 direction 是“sendonly”或“inactive”，并且收发器的[[FiredDirection]]值是“sendrecv”或“recvonly”，则执行以下步骤：

i. 给收发器设置相关的transceiver.[[Receiver]]，空列表，另一个空列表和removeList。

ii. 在给定收发器和静音轨道的情况下，处理媒体描述的远程轨道的移除。

ii. 将收发器的[[CurrentDirection]]和[[FiredDirection]]值设置为方向。

viii. 如果将 description 设置为远程描述，则运行以下步骤：

i. 对 description 中的每个媒体描述运行以下步骤：

i. 设方向是 RTCRtpTransceiverDirection 类型的值，表示来自媒体描述的方向，但发送方式和接收方向相反，以表示此对等方的视图。

ii. 如[JSEP]（第5.10节）所述，尝试查找现有的 RTCRtpTransceiver 对象，收发器，以表示媒体描述。

iii. 如果未找到合适的收发器（未设置收发器），请执行以下步骤：

i. 从媒体描述创建 RTCRtpSender 对象作为 sender。

ii. 从媒体描述创建RTCRtpReceiver 对象作为 receiver。

iii. 使用 sender、 receiver 和 值为“recvonly”的 RTCRtpTransceiverDirection 创建一个 RTCRtpTransceiver，并让收发器成为结果。

iv. 将收发器的中间值设置为相应媒体描述的中间值。如果媒体描述没有 MID，并且未设置收发器的 mid，则生成随机值，如[JSEP]（第5.10节）中所述。

v. 如果 direction 是 “sendrecv” 或 “recvonly”，则让 msids 成为媒体描述指示收发器的 MSID 列表。[[Receiver]]。[[ReceiverTrack]]将与之关联。否则，让msids为空列表。

vi. 给定 transceiver.[[Receiver]]，msids，addList 和 removeList 的相关远程流。

vii. 如果上一步增加了 addList 的长度，或者收发器的 [[FiredDirection]]值既不是 “sendrecv” 也不是 “recvonly”，则在给定收发器和 trackEventInits 的情况下添加媒体描述的远程轨道。

viii. 如果direction是“sendonly”或“inactive”，则将收发器的 [[Receptive]]值设置为 false。

ix. 如果direction是“sendonly”或“inactive”，并且收发器的 [[FiredDirection]]值是“sendrecv”或“recvonly”，则在给定收发器和muteTracks的情况下移除媒体描述的远程轨道。

x. 将收发器的[[FiredDirection]]值设置为方向。

xi. 如果描述的类型为 “answer” 或 “pranswer”，则执行以下步骤：

i. 将收发器的[[CurrentDirection]]和[[Direction]]值设置为方向。

ii. 根据[BUNDLE]，让 transport 和 rtcpTransport 成为 RTCDtlsTransport 对象，表示收发器相关媒体描述所使用的媒体传输的 RTP 和 RTCP 组件。

iii. 设置收发器.[[Sender]].[[SenderTransport]]进行传输。

iv. 将收发器.[[Sender]].[[SenderRtcpTransport]]设置为 rtcpTransport。

v. 设置收发器.[[接收器]].[[ReceiverTransport]]进行传输。

vi. 将收发器.[[Receiver]].[[ReceiverRtcpTransport]]设置为 rtcpTransport。

xii. 如果媒体描述被拒绝，并且收发器尚未停止，请停止 RTCRtpTransceiver 收发器。

ix. 如果 description 是 “rollback” 类型，则运行以下步骤：

i. 如果正在回滚的 RTCSessionDescription 将 RTCRtpTransceiver 的中间值设置为非空值，请将该收发器的中间值设置为 null，如 [JSEP]（第4.1.8.2节）所述。

ii. 如果通过应用正在回滚的 RTCSessionDescription 创建了RTCRtpTransceiver，并且没有通过addTrack将轨道连接到它，则从 connection 的收发器集中删除该收发器，如[JSEP]（第4.1.8.2节）所述。

iii. 对于保持 connection 的 RTCRtpTransceivers，对正在回滚的RTCSessionDescription 应用程序所做的[[CurrentDirection]]和[[Receptive]]内部值的任何更改进行还原。

iv. 将 connection 的[[SctpTransport]]的值恢复为上次稳定信令状态下的值。

x. 如果 connection 的信令状态发生变化，则在连接时触发名为signalingstatechange 的事件。

xi. 对于 muteTracks 中的每个轨道，将轨道的静音状态设置为值true。

xii. 对于 removeList 中的每个流和轨道对，从流中删除轨道轨道。

xiii. 对于 addList 中的每个流和轨道对，将轨道轨道添加到流。

xiv. 对于 trackEventInits 中的每个条目，使用 RTCTrackEvent 接口触发名为track的事件，其 receiver 属性初始化为 entry.receiver，其 track 属性初始化为entry.track，其 streams 属性初始化为entry.streams，其 receiver 属性初始化为entry .transceiver 在 connection 对象。

xv. 如果 connection 的信令状态现在是“stable”，则更新需要协商的标志。如果此更新之前和之后连接的[[NegotiationNeeded]]值均为true，则对运行以下步骤的任务进行排队：

i. 如果connection的[[IsClosed]]值为true，则中止这些步骤

ii. 如果连接的[[NegotiationNeeded]]值为false，则中止这些步骤。

iii. 在连接时触发名为 negotiationneeded 的事件。

xvi. 将 p 设为值为 undefined 的 Resolve 状态。

3. 返回p。

4.4.1.7 设置配置

要设置配置，请运行以下步骤：

1. 让配置成为要处理的 RTCConfiguration 字典。
2. 让connection 成为目标 RTCPeerConnection 对象。
3. 如果设置了 configuration.peerIdentity 且其值与目标的 peer identity 不同，则抛出 InvalidModificationError。
4. 如果设置了 configuration.certificates 且证书集与构造连接时使用的证书集不同，则抛出 InvalidModificationError。
5. 如果设置了 configuration.bundlePolicy 的值且其值与连接的捆绑策略不同，则抛出 InvalidModificationError。
6. 如果设置了 configuration.rtcpMuxPolicy 的值且其值与连接的 rtcpMux 策略不同，则抛出 InvalidModificationError。如果值为“negotiate”且用户代理未实现非复用的 RTCP，则抛出 NotSupportedError。
7. 如果设置了 configuration.iceCandidatePoolSize 的值并且其值与连接的先前设置的 iceCandidatePoolSize 不同，并且已经调用了 setLocalDescription，则抛出 InvalidModificationError。
8. 将 ICE 代理的 ICE 传输设置设置为 configuration.iceTransportPolicy 的值。如[JSEP]（第4.1.16节）中所定义，如果新的 ICE 传输设置更改了现有设置，则在下一个收集阶段之前不会采取任何操作。如果脚本希望立即发生这种情况，则应该重新启动 ICE。
9. 将[JSEP]（第3.5.4节和第4.1.1节中）中定义的 ICE 代理的预取ICE候选池大小设置为 configuration.iceCandidatePoolSize 的值。如果新的 ICE 候选池大小改变了现有设置，则可能导致立即收集新候选池里的候选者，或丢弃现有的池里的候选者，如[JSEP]（第4.1.16节）中所定义。
10. 让 validatedServers 为空列表。
11. 如果定义了 configuration.iceServers，则对每个元素运行以下步骤：
    i. 让 server 成为当前的列表元素。
    ii. 让 urls 为 server.urls。
    iii. 如果 url 是一个字符串，请将 url 设置为仅包含该字符串的列表。
    iv. 如果 url 为空，则抛出一个 SyntaxError。
    v. 对于 urls 中的每个 url，请执行以下步骤：
        i. 使用[RFC3986]中定义的通用 URI 语法解析 URL 并获取方案名称。如果基于[RFC3986]中定义的语法的解析失败，则抛出一个 SyntaxError。如果浏览器未实现方案名称，则抛出 NotSupportedError。如果方案名称是 turn 或 turns，并且使用[RFC7064]中定义的语法解析 URL 失败，则抛出一个 SyntaxError。如果方案名称是 stun 或 stuns，并且使用[RFC7065]中定义的语法解析url失败，则抛出一个 SyntaxError。

ii. 如果方案名称是 turn 或 turns ，并且省略了 server.username 或 server.credential，则抛出 InvalidAccessError。

iii. 如果方案名称是turn或turn，并且 server.credentialType 是 “password”，而 server.credential 不是 DOMString，则抛出 InvalidAccessError。

iv. 如果方案名称是 turn 或 turns ，server.credentialType 是 “oauth”，而 server.credential 不是 RTCOAuthCredential，则抛出 InvalidAccessError。

vi. 将 server 附加到validatedServers。

让 validatedServers 成为 ICE 代理的 ICE服务器列表。

如[JSEP]（第4.1.16节）中所定义，如果新的服务器列表替换ICE代理的现有ICE服务器列表，则在下一个收集阶段之前不会采取任何操作。如果脚本希望立即发生这种情况，则应该重新启动ICE。但是，如果ICE候选池具有非零大小，则将丢弃任何现有池化候选者，并且将从新服务器收集新候选者。

12. 将配置存储在[[Configuration]]值中。

4.4.2 接口定义

本节中介绍的 RTCPeerConnection 接口通过本规范中的几个部分接口进行了扩展。值得注意的是，RTP Media API部分添加了 API 以发送和接收 MediaStreamTrack 对象。

```
[Constructor(optional RTCConfiguration configuration),
  Exposed=Window]
interface RTCPeerConnection : EventTarget {
  Promise<RTCSessionDescriptionInit> createOffer(optional RTCOfferOptions options);
  Promise<RTCSessionDescriptionInit> createAnswer(optional RTCAnswerOptions options);
  Promise<void> setLocalDescription(RTCSessionDescriptionInit description);
  readonly attribute RTCSessionDescription? localDescription;
  readonly attribute RTCSessionDescription? currentLocalDescription;
  readonly attribute RTCSessionDescription? pendingLocalDescription;
  Promise<void> setRemoteDescription(RTCSessionDescriptionInit description);
  readonly attribute RTCSessionDescription? remoteDescription;
  readonly attribute RTCSessionDescription? currentRemoteDescription;
  readonly attribute RTCSessionDescription? pendingRemoteDescription;
  Promise<void> addIceCandidate(RTCIceCandidateInit candidate);
  readonly attribute RTCSignalingState signalingState;
  readonly attribute RTCIceGatheringState iceGatheringState;
  readonly attribute RTCIceConnectionState iceConnectionState;
  readonly attribute RTCPeerConnectionState connectionState;
  readonly attribute boolean? canTrickleIceCandidates;
  static sequence<RTCIceServer> getDefaultIceServers();
  RTCConfiguration getConfiguration();
  void setConfiguration(RTCConfiguration configuration);
  void close();
  attribute EventHandler onnegotiationneeded;
  attribute EventHandler onicecandidate;
  attribute EventHandler onicecandidateerror;
  attribute EventHandler onsignalingstatechange;
  attribute EventHandler oniceconnectionstatechange;
  attribute EventHandler onicegatheringstatechange;
  attribute EventHandler onconnectionstatechange;
};
```

构造函数

RTCPeerConnection

请参阅 RTCPeerConnection 构造函数算法。

属性

localDescription 类型 RTCSessionDescription，只读，不能为空：

如果 localDescription 属性不为 null，则它必须返回[[PendingLocalDescription]]，否则它必须返回[[CurrentLocalDescription]]。

注意[[CurrentLocalDescription]].sdp和[[PendingLocalDescription]].sdp不需要与传递给相应 setLocalDescription 调用的 SDP 值完全相同（即 SDP 可以被解析和重新格式化，并且可以添加 ICE 候选者）。

currentLocalDescription 类型为 RTCSessionDescription，只读，不能为空：

currentLocalDescription 属性必须返回[[CurrentLocalDescription]]。

它表示上次 RTCPeerConnection 转换为稳定状态时成功协商的本地描述，以及自创建 offer 或 answer 以来 ICE 代理生成的任何本地候选项。

`pendingLocalDescription` 类型为 RTCSessionDescription，只读，不能为空：

pendingLocalDescription 属性必须返回[[PendingLocalDescription]]。

它表示正在协商的本地描述以及自创建 offer 或 answer 以来 ICE 代理生成的任何本地候选人。如果 RTCPeerConnection 处于稳定状态，则该值为空。

`remoteDescription` 类型 RTCSessionDescription，只读，不能为空：

remoteDescription 属性必须返回[[PendingRemoteDescription]]，如果它不为 null，否则它必须返回[[CurrentRemoteDescription]]。

注意[[CurrentRemoteDescription]].sdp和[[PendingRemoteDescription]].sdp 不需要与传递给相应的 setRemoteDescription 调用的 SDP 值完全相同（即 SDP 可以被解析和重新格式化，并且可以添加 ICE 候选者） 。

`currentRemoteDescription` 类型为RTCSessionDescription，只读，不能为空：

currentRemoteDescription 属性必须返回[[CurrentRemoteDescription]]。

它表示上次 RTCPeerConnection 转换为稳定状态时成功协商的最后一个远程描述，以及自创建 offer 或 answer 以来通过addIceCandidate（）提供的任何远程候选。

`pendingRemoteDescription` 类型RTCSessionDescription，只读，不能为空：

pendingRemoteDescription 属性必须返回[[PendingRemoteDescription]]。

它表示正在协商过程中的远程描述，包括自创建 offer 或 answer 以来通过 addIceCandidate（）提供的任何远程候选项。如果 RTCPeerConnection 处于稳定状态，则该值为空。

TRANSPCS 类型为 RTCSignalingState 的 signalingState， 只读:

signalingState 属性必须返回 RTCPeerConnection 对象的信令状态。

`iceGatheringState` 类型为RTCIceGatheringState，只读

iceGatheringState 属性必须返回 RTCPeerConnection 实例的 ICE 收集状态。

`RTCIceConnectionState` 类型的iceConnectionState，只读

iceConnectionState 属性必须返回 RTCPeerConnection 实例的 ICE 连接状态。

类型为RTCPeerConnectionState的connectionState，只读：

connectionState 属性必须返回 RTCPeerConnection 实例的连接状态。

canTrickleIceCandidates类型为boolean，只读，不能为空：

The canTrickleIceCandidates attribute indicates whether the remote peer is able to accept trickled ICE candidates [TRICKLE-ICE]. The value is determined based on whether a remote description indicates support for trickle ICE, as defined in [JSEP] (section 4.1.15.). Prior to the completion of setRemoteDescription, this value is null.

canTrickleIceCandidates 属性指示远程对等方是否能够接受 trickle ICE 候选[TRICKLE-ICE]。该值是根据远程描述是否表示支持 trickle ICE 确定的，如[JSEP]（第4.1.15节）中所定义。在完成 setRemoteDescription 之前，此值为 null。

onnegotiationneeded 类型 EventHandler

此事件处理程序的事件类型是 negotiationneeded。

eventHandler 类型的 onicecandidate

此事件处理程序的事件类型是 icecandidate。

eventHandler 类型的 onicecandidateerror

此事件处理程序的事件类型是 icecandidateerror。

onSignalingstate 类型 EventHandler 的更改

此事件处理程序的事件类型是 signalingstatechange。

eventHandler 类型的 oniceconnectionstatechange

此事件处理程序的事件类型是 iceconnectionstatechange

eventHandler 类型的 onicegatheringstatechange

此事件处理程序的事件类型是 icegatheringstatechange。

eventHandler 类型的 onconnectionstatechange

此事件处理程序的事件类型是 connectionstatechange。

方法

`createOffer`

createOffer 方法生成一个 SDP blob，其中包含 RFC 3264 offer，其中包含会话支持的配置，包括附加到此 RTCPeerConnection 的本地 MediaStreamTracks 的说明，此实现支持的编解码器/ RTP / RTCP 功能以及 ICE 的参数代理和 DTLS 连接。可以提供选项参数以提供对所生成的 offer 的额外控制。

如果系统具有有限的资源（例如，有限数量的解码器），则 createOffer 需要返回反映系统当前状态的 offer，以便 setLocalDescription 在尝试获取这些资源时成功。会话描述必须仍然可以通过 setLocalDescription 保持可用，而不会导致错误，直到至少返回的 promise 的履行回调结束。

创建 SDP 必须遵循适当的流程来生成[JSEP]中描述的 offer。作为要约，生成的 SDP 将包含会话支持或优选的全套编解码器/ RTP / RTCP功能（而不是 answer，其将仅包括要使用的特定协商子集）。如果在建立会话后调用 createOffer，createOffer 将生成与当前会话兼容的 offer，其中包含自上次完成 offer - 应答交换以来对会话所做的任何更改，例如添加或删除轨道。如果未进行任何更改，则要约将包括当前本地描述的功能以及可在更新的要约中协商的任何其他功能。

生成 SDP还将包含 ICE 代理的 usernameFragment，密码和 ICE 选项（在[ICE]，第14节中定义），还可以包含代理收集的任何本地候选。

RTCPeerConnection 的配置中的证书值提供了应用程序为 RTCPeerConnection 配置的证书。这些证书以及任何默认证书用于生成一组证书指纹。这些证书指纹用于构建 SDP 和作为身份断言请求的输入。

如果 RTCPeerConnection 配置为通过调用 setIdentityProvider 生成 Identity 断言，则会话描述应包含适当的断言。

生成 SDP 的过程暴露了底层系统的媒体功能的子集，其在设备上提供通常持久的跨源信息。因此，它增加了应用的指纹表面。在隐私敏感的上下文中，浏览器可以考虑缓解，例如仅生成 SDP 匹配功能的公共子集。

调用该方法时，用户代理必须执行以下步骤：

让connection 成为调用方法的 RTCPeerConnection 对象。

如果connection的[[IsClosed]] 值为 true，则返回使用新创建的 InvalidStateError 拒绝的 promise。

如果使用身份提供程序配置连接，则在尚未开始的情况下开始身份声明请求过程。

将以下步骤的结果返回到连接的操作队列：

让p成为新的Promise。

同时，在给定 p 的情况下，开始创建 offer 的步骤。

返回p。

在给定承诺 p 的情况下创建要约的步骤如下：

如果未使用一组证书构建连接，并且尚未生成一个证书，请等待生成。

如果已配置提供者，则提供者是连接的当前配置身份提供者，否则为 null。

如果 provider 为非 null，则等待身份断言请求过程完成。

如果提供程序无法生成标识声明，请使用新创建的 NotReadableError 拒绝 p 并中止这些步骤。

检查系统状态以确定生成要约所需的当前可用资源，如[JSEP]（第4.1.6节）中所述。

如果此检查因任何原因失败，请使用新创建的 OperationError 拒绝p并中止这些步骤。

在给定 p 的情况下，对运行最终步骤以创建 offer 的任务进行排队。

给出承诺 p 创建要约的最后步骤如下：

如果connection的[[IsClosed]] 值为true，则中止这些步骤。

如果以这样的方式修改连接，即需要对系统状态进行额外检查，或者如果其配置的 indentity 提供程序不再是提供者，则并行开始执行再次创建 offer 的步骤，给定 p，并中止这些步骤。

NOTE：如果仅在连接中添加了音频RTCRtpTransceiver时调用了createOffer，但在执行并行创建 offer的步骤时，可能需要添加视频RTCRtpTransceiver，这需要额外检查视频系统资源。

给定从先前检查获得的信息，当前连接状态及其 RTCRtpTransceivers，以及来自提供者的身份断言（如果非空），生成 SDP offer sdpString，如[JSEP]（第5.2节）中所述。

如[捆绑]（第7节）中所述，如果使用捆绑（请参阅RTCBundlePolicy），则必须选择标记为 m = section 的提议者才能协商 BUNDLE 组。用户代理必须选择 m = 部分，该部分对应于收发器集合中的第一个不停止的收发器，作为提供者标记的 m = 部分。这允许远程端点预测哪个收发器是提供者标记的 m = 部分而不必解析 SDP。

m = 段的相关收发器的编解码器首选项被称为 RTCRtpTranceiver 的 [[PreferredCodecs]]的值，应用了以下过滤（或者如果[[PreferredCodecs]]为空则表示不设置）：

如果方向是“sendrecv”，则排除 RTCRtpSender.getCapabilities（kind）.codecs 和 RTCRtpReceiver.getCapabilities（kind）.codecs 交集中未包含的任何编解码器。

如果方向是“sendonly”，则排除 RTCRtpSender.getCapabilities（kind）.codecs 中未包含的任何编解码器。

如果方向是“recvonly”，则排除 RTCRtpReceiver.getCapabilities（kind）.codecs 中未包含的任何编解码器。

过滤绝不能改变编解码器首选项的顺序。

让 offer 成为一个新创建的 RTCSessionDescriptionInit 字典，其类型成员初始化为字符串“offer”，其 sdp 成员初始化为 sdpString。

将[[LastOffer]]内部值设置为 sdpString。

用 Resolve 解决 p。

`createAnswer`

createAnswer 方法使用支持的会话配置生成[SDP] answer，该配置与远程配置中的参数兼容。与 createOffer 一样，返回的 SDP blob 包含附加到此 RTCPeerConnection 的本地 MediaStreamTracks 的描述，为此会话协商的编解码器/ RTP / RTCP 选项，以及 ICE 代理收集的任何候选项。可以提供选项参数以提供对生成的 answer 的附加控制。

与 createOffer 一样，返回的描述应该反映系统的当前状态。会话描述必须仍然可以通过 setLocalDescription 保持可用，而不会导致错误，直到至少返回的 promise 的履行回调结束。

作为 answer，生成的SDP将包含特定的编解码器/ RTP / RTCP 配置，该配置与相应的 offer 一起指定应如何建立媒体平面。 SDP 的生成必须遵循生成[JSEP]中描述的 answer 的适当过程。

生成的 SDP 还将包含 ICE 代理的 usernameFragment，密码和 ICE 选项（在[ICE]，第14节中定义），还可以包含代理收集的任何本地候选。

RTCPeerConnection 的配置中的证书值提供了应用程序为 RTCPeerConnection 配置的证书。这些证书以及任何默认证书用于生成一组证书指纹。这些证书指纹用于构建 SDP 和作为身份断言请求的输入。

通过将类型设置为“pranswer”，可以将 answer 标记为临时，如[JSEP]（第4.1.8.1节）中所述。

如果RTCPeerConnection配置为通过调用 setIdentityProvider 生成 Identity 断言，则会话描述应包含适当的断言。

调用该方法时，用户代理必须执行以下步骤：

让 connection 成为调用方法的 RTCPeerConnection 对象。

如果connection的[[IsClosed]]值为 true，则返回使用新创建的 InvalidStateError 拒绝的 promise。

如果使用身份提供程序配置连接，则在尚未开始的情况下开始身份声明请求过程。

将以下步骤的结果返回到连接的操作队列：

如果连接的信令状态既不是“have-remote-offer”也不是“have-local-pranswer”，则返回使用新创建的 InvalidStateError 拒绝的 promise。

让 p 成为新的 Promise。

同时，在给定 p 的情况下，开始创建 answer 的步骤。

返回p。

给出 p 创建 answer 的步骤如下：

如果未使用一组证书构建连接，并且尚未生成一个证书，请等待生成。

如果已配置提供者，则提供者是连接的当前配置身份提供者，否则为 null。

如果 provider 为非 null，则等待身份断言请求过程完成。

如果提供程序无法生成标识声明，请使用新创建的 NotReadableError 拒绝 p 并中止这些步骤。

检查系统状态以确定生成 answer 所需的当前可用资源，如[JSEP]（第4.1.7节）中所述。

如果此检查因任何原因失败，请使用新创建的 OperationError 拒绝 p 并中止这些步骤。

在给定 p 的情况下，对运行最终步骤以创建 answer 的任务进行排队。

给出承诺 p 创建 answer 的最后步骤如下：

如果connection的[[IsClosed]] 值为 true，则中止这些步骤。

如果以这样的方式修改连接，即需要对系统状态进行额外检查，或者如果其配置的 indentity 提供程序不再是提供者，那么并行开始步骤再次创建 answer，给定 p，并中止这些步骤。

```
NOTE：如果在 RTCRtpTransceiver 的方向是“recvonly”时调用 createAnswer，则可能需要这样做，但
给定从先前检查获得的信息以及当前连接状态及其 RTCRtpTransceivers，以及来自提供者的身份断言（如
m =段的相关收发器的编解码器首选项被称为 RTCRtpTranceiver的[[PreferredCodecs]]的值，应用了以
```

如果方向是“sendrecv”，则排除 RTCRtpSender.getCapabilities（kind）.codecs 和 RTCRtpReceiver.getCapabilities（kind）.codecs交集中未包含的任何编解码器。

如果方向是“sendonly”，则排除 RTCRtpSender.getCapabilities（kind）.codecs 中未包含的任何编解码器。

如果方向是“recvonly”，则排除 RTCRtpReceiver.getCapabilities（kind）.codecs 中未包含的任何编解码器。

过滤绝不能改变编解码器首选项的顺序。

让我们回答一个新创建的 RTCSessionDescriptionInit 字典，其类型成员初始化为字符串“answer”，其 sdp 成员初始化为 sdpString。

将[[LastAnswer]]值设置为sdpString。

Resolve p 并返回answer。

`setLocalDescription`

setLocalDescription 方法指示 RTCPeerConnection 将提供的 RTCSessionDescriptionInit 应用为本地描述。

此 API 更改本地媒体状态。为了成功处理应用程序想要提供的从一种媒体格式更改为不同的不兼容格式的场景，RTCPeerConnection 必须能够同时支持使用当前和未决的本地描述（例如，支持两者中存在的编解码器）描述）直到收到最终 answer，此时 RTCPeerConnection 可以完全采用待处理的本地描述，或者如果远程端拒绝更改，则回滚到当前描述。

如[JSEP]（第5.4节）中所述，从 createOffer 或 createAnswer 返回的 SDP 在传递给 setLocalDescription 之前不得更改。因此，当调用该方法时，用户代理必须运行以下步骤：

让 description 成为 setLocalDescription 的第一个参数。

如果 description.sdp 为空字符串且 description.type 为“answer” 或 “pranswer”，则将 description.sdp 设置为connection [[LastAnswer]]的值。

如果 description.sdp 为空字符串且 description.type 为 “offer”，则将 description.sdp 设置为 connection [[LastOffer]]的值。

返回设置描述所指示的 RTCSessionDescription 的结果。

如[JSEP]（第5.9节）所述，调用此方法可能会触发 ICE 代理的 ICE 候选人收集过程。

`setRemoteDescription`

setRemoteDescription 方法指示 RTCPeerConnection 将提供的 RTCSessionDescriptionInit 应用为远程提供或 answer。此 API 更改本地媒体状态。

调用该方法时，用户代理必须返回设置方法的第一个参数指示的 RTCSessionDescription 的结果。

此外，处理远程描述以确定和验证对等体的身份。

如果会话描述中存在 a = identity 属性，则浏览器验证标识声明。

如果 peerIdentity 配置应用于 RTCPeerConnection，则会建立所提供值的目标对等体标识。或者，如果 RTCPeerConnection 先前已经验证了对等体的身份（即，解析了peerIdentity promise），那么这也建立了目标对等体身份。

设置后，目标对等身份不能更改。

如果设置了目标对等体标识，则必须在声明返回 setRemoteDescription 解析之前完成身份验证。如果身份验证失败，则会拒绝 setRemoteDescription 返回的承诺。

如果没有目标对等体标识，则 setRemoteDescription 不会等待完成身份验证。

`addIceCandidate`

addIceCandidate 方法为 ICE 代理提供远程候选。当用候选成员的空字符串调用时，此方法还可用于指示远程候选者的结束。此方法使用的参数的唯一成员是 candidate，sdpMid，sdpMLineIndex 和 usernameFragment;其余的都被忽略了。调用该方法时，用户代理必须运行以下步骤：

让候选人成为方法的参数。

让 connection 成为调用方法的 RTCPeerConnection 对象。

如果 candidate.candidate 不是空字符串且 candidate.sdpMid 和 candidate.sdpMLineIndex 都为 null，则返回使用新创建的 TypeError拒绝的 promise。

将以下步骤的结果返回到连接的操作队列：

如果 remoteDescription 为 null，则返回使用新创建的 InvalidStateError 拒绝的承诺。

让p成为新的Promise。

如果candidate.sdpMid不为null，请运行以下步骤：

如果 candidate.sdpMid 不等于 remoteDescription 中任何媒体描述的中间部分，则使用新创建的 OperationError拒绝 p 并中止这些步骤。

否则，如果candidate.sdpMLineIndex不为null，请运行以下步骤：

如果 candidate.sdpMLineIndex 等于或大于 remoteDescription 中的媒体描述数，则使用新创建的 OperationError 拒绝 p 并中止这些步骤。

如果 candidate.usernameFragment 既不是未定义也不是 null，并且不等于应用的远程描述的相应媒体描述中存在的任何用户名片段，则使用新创建的 OperationError 拒绝 p 并中止这些步骤。

如果 candidate.usernameFragment 既不是未定义也不是 null，并且不等于应用的远程描述的相应媒体描述中存在的任何用户名片段，则使用新创建的 OperationError 拒绝p并中止这些步骤。

同时，按照[JSEP]（第4.1.17节）中的描述添加 ICE 候选候选者。使用 candidate.usernameFragment 来标识 ICE 生成;如果 usernameFragment 为 null，则处理最近 ICE 生成的候选项。如果 candidate.candidate 是空字符串，则将候选处理作为对应的媒体描述和 ICE 候选生成的候选结束指示。如果 candidate.sdpMid 和 candidate.sdpMLineIndex 都为 null，则这适用于所有媒体描述。

```
zh:如果无法成功添加候选者，则用户代理必须对运行以下步骤的任务进行排队：
```

如果 connection的[[IsClosed]] 的值为true，则中止这些步骤。

使用新创建的 OperationError 拒绝 p 并中止这些步骤。

```
zh:如果候选成功应用，则用户代理必须对运行以下步骤的任务进行排队：
```

如果 connection的[[IsClosed]] 的值为 true，则中止这些步骤。

如果 connection.[[PendingRemoteDescription]] 不为 null，并且表示处理候选的 ICE 生成，则将候选添加到连接.[[PendingRemoteDescription]].sdp。

如果 connection.[[CurrentRemoteDescription]]不为 null，并且表示处理候选的 ICE 生成，则将候选添加到connection.[[CurrentRemoteDescription]].sdp。

Resolve p 并返回 undefined。

返回p。

NOTE:由于 WebIDL 处理，addIceCandidate（null） 被解释为具有默认字典的调用，在上述算法中，该字典指示所有媒体描述和 ICE 候选生成的候选结束。这是出于遗留原因的设计。

`getDefaultIceServers`

返回配置到浏览器中的 ICE 服务器列表。浏览器可能配置为使用本地或私有 STUN 或 TURN 服务器。此方法允许应用程序了解这些服务器并可选择使用它们。

此列表可能是持久的，并且在起源之间是相同的。因此，它增加了浏览器的指纹表面。在隐私敏感的上下文中，浏览器可以考虑缓解，例如仅将此数据提供给白名单来源（或根本不提供）。

NOTE:由于此信息的使用由应用程序开发人员自行决定，因此使用这些默认值配置用户代理本身并不会增加用户限制其 IP 地址暴露的能力。

`getConfiguration`

返回表示此 RTCPeerConnection 对象的当前配置的 RTCConfiguration 对象。

调用此方法时，用户代理必须返回存储在[[Configuration]]内部值中的 RTCConfiguration 对象。

`setConfiguration`

setConfiguration 方法更新此 RTCPeerConnection 对象的配置。这包括更改 ICE 代理的配置。如[JSEP]（第3.5.1节）中所述，当 ICE 配置以需要新收集阶段的方式更改时，需要重新启动 ICE。

调用 setConfiguration 方法时，用户代理必须运行以下步骤：

让 connection 成为调用方法的 RTCPeerConnection。

如果 connection的[[IsClosed]]的值为 true，则抛出 InvalidStateError。

设置配置指定的配置。

`close`

当调用 close 方法时，用户代理必须运行以下步骤：

让 connection 成为调用方法的 RTCPeerConnection 对象。

如果 connection的[[IsClosed]]的值为 true，则中止这些步骤。

将connection.[[IsClosed]]插槽设置为 true。

将连接的信令状态设置为“closed”。

让收发器成为执行 CollectTransceivers 算法的结果。对于收发器中的每个 RTCRtpTransceiver 收发器，请运行以下步骤：

如果收发器的[[Stoped]]的值为真，则中止这些步骤。

让发送者成为收发器的[[Sender]]。

让接收器成为收发器的[[Receiver]]。

停止向发件人发送媒体。

按照[RFC3550]中的规定，为发送方发送的每个 RTP 流发送 RTCP BYE。

停止使用接收器接收媒体。

将接收者的[[ReceiverTrack]]的readyState设置为“ends”。

将收发器的[[Stopped]]的值设置为 true。

将每个连接的RTCDataChannels的[[ReadyState]]的值设置为“closed”。

RTCDataChannels将突然关闭，并且不会调用关闭过程。

如果connection.[[SctpTransport]] 不为 null，则通过发送 SCTP ABORT 块并将[[SctpTransportState]]设置为“closed”来拆除底层 SCTP 关联。

将每个连接的 RTCDtlsTransports.[[DtlsTransportState]]的值设置为“closed”。

销毁连接的 ICE 代理，突然结束任何活动的 ICE 处理并释放任何相关资源（例如 TURN 权限）。

将每个连接的 RTCIceTransports 的[[IceTransportState]]插槽设置为“closed”。

将连接的 ICE 连接状态设置为“closed”。

将连接的连接状态设置为“closed”。

4.4.3 旧接口扩展

NOTE 出于可读性考虑，本节内容已经过时。在这里考虑部分接口是它们的主要对应物的一部分，因为它们使现有方法重载。

本节中支持的方法是可选的。但是，如果支持这些方法，则必须根据此处指定的方法实现。

NOTE

以前存在于RTCPeerConnection上的addStream方法很容易polyfill为：

```
RTCPeerConnection.prototype.addStream = function(stream) {
  stream.getTracks().forEach((track) => this.addTrack(track, stream));
};
```

4.4.3.1方法扩展

```
partial interface RTCPeerConnection {
    Promise<void> createOffer(RTCSessionDescriptionCallback successCallback,
                              RTCPeerConnectionErrorCallback failureCallback,
                              optional RTCOfferOptions options);
    Promise<void> setLocalDescription(RTCSessionDescriptionInit description,
                                      VoidFunction successCallback,
                                      RTCPeerConnectionErrorCallback failureCallback);
    Promise<void> createAnswer(RTCSessionDescriptionCallback successCallback,
                               RTCPeerConnectionErrorCallback failureCallback);
    Promise<void> setRemoteDescription(RTCSessionDescriptionInit description,
                                       VoidFunction successCallback,
                                       RTCPeerConnectionErrorCallback failureCallback)
    Promise<void> addIceCandidate(RTCIceCandidateInit candidate,
                                  VoidFunction successCallback,
                                  RTCPeerConnectionErrorCallback failureCallback);
};
```

方法

`createOffer`

调用createOffer方法时，用户代理必须执行以下步骤：

1. 让successCallback成为方法的第一个参数。
2. 让failureCallback成为方法第二个参数指示的回调。
3. 让options成为方法第三个参数指示的回调。
4. 运行RTCPeerConnection的createOffer（）方法指定的步骤，将options作为唯一参数，并将p作为生成的promise。
5. 当p完成并返回 offer 时，以 offer 作为参数调用 successCallback。
6. 当p被拒绝并返回原因 r 时，以 r 作为参数调用 failureCallback。
7. 返回使用 undefined 作为 resolved 的 promise。

`setLocalDescription`

zh:调用 setLocalDescription 方法时，用户代理必须运行以下步骤：

1. 让 description 成为方法的第一个参数。
2. 让 successCallback 成为方法第二个参数指示的回调。
3. 让 failureCallback 成为方法第三个参数指示的回调。
4. 运行 RTCPeerConnection 的 setLocalDescription 方法指定的步骤，将 description 作为唯一参数，并将p作为生成的promise。
5. 完成p后，使用undefined作为参数调用 successCallback。
6. 当p被拒绝并返回原因 r 时，以r作为参数调用 failureCallback。
7. 返回使用 undefined 作为 resolved 的 promise。

`createAnswer`

NOTE 旧版 createAnswer 方法不接受 RTCAnswerOptions 参数，因为没有任何已知的旧版 createAnswer 实现支持它。

调用createAnswer方法时，用户代理必须运行以下步骤：

1. 让successCallback成为方法的第一个参数。
2. 让failureCallback成为方法第二个参数指示的回调。
3. 运行RTCPeerConnection的createAnswer（）方法指定的步骤，不带参数，让p为结果的promise。
4. 当p完成并返回 answer 时，调用 successCallback 并将 answer 作为参数。
5. 当p被拒绝并返回原因 r 时时，以 r 作为参数调用 failureCallback。
6. 返回使用 undefined 作为 resolved 的 promise。

`setRemoteDescription`

调用setRemoteDescription方法时，用户代理必须执行以下步骤：

1. 让描述成为方法的第一个参数。
2. 让 successCallback 成为方法第二个参数指示的回调。
3. 让 failureCallback 成为方法第三个参数指示的回调。
4. 运行 RTCPeerConnection 的 setRemoteDescription 方法指定的步骤，将 description 作为唯一参数，并将 p 作为生成的 promise。
5. 完成 p 后，使用 undefined 作为参数调用 successCallback。
6. 当 p 被拒绝并返回原因 r 时，以 r 作为参数调用failureCallback。
7. 返回使用 undefined 作为 resolved 的 promise。

`addIceCandidate`

调用 addIceCandidate 方法时，用户代理必须执行以下步骤：

1. 让候选人成为方法的第一个参数。
2. 让 successCallback 成为方法第二个参数指示的回调。
3. 让 failureCallback 成为方法第三个参数指示的回调。
4. 运行 RTCPeerConnection 的 addIceCandidate（）方法指定的步骤，候选者作为唯一参数，让 p 为结果承诺。
5. 完成 p 后，使用 undefined 作为参数调用 successCallback。
6. 当 p 被拒绝并返回原因 r 时时，以 r 作为参数调用 failureCallback。
7. 返回使用 undefined 作为 resolved 的 promise。

Callback Definitions

这些回调仅用于旧版API。

`RTCPeerConnectionErrorCallback`

```
callback RTCPeerConnectionErrorCallback = void (DOMException error);
```

Callback RTCPeerConnectionErrorCallback Parameters

error of type DOMException An error object encapsulating information about what went wrong.

`RTCSessionDescriptionCallback`

```
callback RTCSessionDescriptionCallback = void (RTCSessionDescriptionInit description);
```

Callback RTCSessionDescriptionCallback Parameters description of type RTCSessionDescriptionInit The object containing the SDP [SDP].

4.4.3.2 旧版配置扩展

除了添加到RTCPeerConnection的媒体之外，本节还介绍了一组可用于影响 offer 创建方式的旧版扩展。鼓励开发人员使用 RTCRtpTransceiver API。

使用本节中指定的任何旧版选项调用 createOffer 时，请运行以下步骤而不是常规 createOffer 步骤：

1. 让 options 成为方法的第一个参数。
2. 让connection成为当前的RTCPeerConnection对象。
3. 对于 options 中的每个"offerToReceive"成员，以及它的类别 kind ，执行以下步骤：
    i. 如果字典成员的值为false，
        i. 对每个未停止的"sendrecv"类的收发器 transceiver ，把 transceiver 中的[Direction] 设置为"sendonly"。
        ii. 对每个未停止的"recvonly"类别的收发器 transceiver ，把 transceiver 中的[Direction]值 设置为"inactive"。 如果有下一选项，继续此步骤

    继续下一个选项（如果有）。
    i. 如果连接具有任何不停止的“sendrecv”或“recvonly”类型的 transceiver，则继续下一个选项（如果有）。
    ii. 让调用connection.addTransceiver（kind）的结果赋值给 transceiver，除了这个操作绝不能更新需要协商的标志。
    iii. 如果由于先前的操作引发错误而未设置收发器，则中止这些步骤。
    iv. 将transceiver的[[Direction]]值设置为“recvonly”。
4. 运行 createOffer 指定的步骤以创建 offer。

```
partial dictionary RTCOfferOptions {
    boolean offerToReceiveAudio;
    boolean offerToReceiveVideo;
};
```

Attributes zh:属性

offerToReceiveAudio 类型为 boolean

此设置提供对音频方向性的额外控制。例如，无论是否发送音频，它都可用于确保接收音频。

offerToReceive 类型为 boolean

此设置提供对视频方向性的额外控制。例如，无论是否发送视频，它都可用于确保接收视频。

4.4.4 垃圾回收

只要有任何可能在对象上触发事件处理器的事件存在，RTCPeerConnection对象就不能被垃圾回收。当对象的[[IsClosed]]内部插槽为true时，就没有事件处理器可以被触发了，因此可以安全地执行垃圾回收。

连接到RTCPeerConnection的所有RTCDataChannel和MediaStreamTrack对象都具有对RTCPeerConnection对象的强引用。

4.5 异常处理

4.5.1 通用原则

所有返回 Promise 的方法都受 Promise 的异常错误处理规则的约束。不返回 Promise 的方法可能会抛出异常来显示错误。

4.6 会话描述模型

4.6.1 `RTCSdpType` 枚举

RTCSdpType 枚举描述了RTCSessionDescriptionInit或RTCSessionDescription实例的类型。

```
enum RTCSdpType {
    "offer",
    "pranswer",
    "answer",
    "rollback"
};
```

枚举描述	
offer	一个RTCSdpType的`offer`表示必须将一个描述视为一个[SDP]提供。
pranswer	一个RTCSdpType的`pranswer`表示必须将一个描述视为[SDP]应答，但不是最终应答。一个描述可以作为一个 SDP offer 的应答响应，或对先前发送的 SDP pranswer 的更新。
answer	一个RTCSdpType的`answer`表示必须将一个描述视为[SDP]的最终应答，并且必须认为 offer-answer 交换完成。一个描述可以用于响应 SDP offer 的最终应答或者可作为对先前发送的 SDP pranswer的更新。
rollback	一个RTCSdpType的`rollback`表示必须将一个描述视为取消当前SDP协商并回退SDP [SDP]提供和最终应答到它先前的稳定状态下。请注意，如果还没有成功进行 offer - answer 协商，则先前稳定状态的本地或远程SDP的描述可能为空。

4.6.2 RTCSessionDescription 类

RTCPeerConnection通过RTCSessionDescription类来公开本地和远程会话描述。

```
[Constructor(RTCSessionDescriptionInit descriptionInitDict),
 Exposed=Window]
interface RTCSessionDescription {
    readonly attribute RTCSdpType type;
    readonly attribute DOMString  sdp;
    [Default] object toJSON();
};
```

构造函数

`RTCSessionDescription 类`

RTCSessionDescription() 构造函数接受一个字典参数 `descriptionInitDict`，用于初始化新的 `RTCSessionDescription` 对象。这个构造函数已废弃，不推荐使用此构造函数;它的存在仅用于历史兼容。

属性

`type` 类型为 `RTCSdpType`，只读项

```
这个RTCSessionDescription的类型。
```

`sdp` 类型为 `DOMString`，只读项

SDP [SDP]的字符串表示形式。

方法

`toJSON()`

调用时，运行[WEBIDL]的默认 `toJSON` 操作。

```
dictionary RTCSessionDescriptionInit {
    required RTCSdpType type;
             DOMString  sdp = "";
};
```

字典RTCSessionDescriptionInit成员

`type` 类型为RTCSdpType，必需项 DOMString sdp `sdp` 类型为 `DOMString` SDP [SDP]的字符串表示形式。如果 `type` 为 `rollback`，则该成员未使用。

4.7 会话协商模型

对RTCPeerConnection状态的许多改变将需要通过信令信道与远程侧通信，以便具有期望的效果。通过收听协商需要的事件，可以通知应用程序何时需要进行信令。根据连接的需要协商标志的状态触发这个事件，该标志由[[NegotiationNeeded]]内部插槽表示。

4.7.1 设置Negotiation-Needed

本节不具有规范性。

如果在需要信令的RTCPeerConnection上执行操作，则该连接将被标记为需要协商。这种操作的示例包括添加或停止RTCRtpTransceiver，或添加第一个RTCDataChannel。

实现中的内部更改还可能导致连接被标记为需要协商。

请注意，更新需要协商的标志的确切过程如下所示。

4.7.2 清除Negotiation-Needed

本节不具有规范性。

当应用类型为“answer”的RTCSessionDescription时，将清除 `negotiation-needed` 的标志，并且提供的描述与RTCPeerConnection上当前存在的RTCRtpTransceivers和RTCDataChannel的状态相匹配。具体而言，所有未停止的收发器在本地描述中具有匹配属性的相关部分，并且，如果已经创建了任何数据信道，则本地描述中存在数据部分。

NOTE：更新Negotiation-Needed的标志的明确过程如下所示。

4.7.3 更新Negotiation-Needed标志

以下的过程在本文档的其他地方引用。它也可能由于实施中影响谈判的内部变化而发生。如果发生此类更改，则用户代理必须对任务进行排队以更新协商需要的标志。

要更新连接协商需要的标志，请运行以下步骤：

1. 如果connection的[[IsClosed]]插槽为true，则中止这些步骤。
2. 如果连接的信令状态不是“stable”，则中止这些步骤。

   NOTE：一旦状态转换为“稳定”，将更新协商需要的标志，作为设置RTCSessionDescription的步骤的一部分。

3. 如果检查是否需要协商的结果为false，则通过将connection的[[NegotiationNeeded]]槽设置为false来清除需要协商的标志，并中止这些步骤。
4. 如果连接的[[NegotiationNeeded]]插槽已经为真，则中止这些步骤。
5. 将连接的[[NegotiationNeeded]]插槽设置为true。
6. 对运行以下步骤的任务进行排队：

   i. 如果connection的[[IsClosed]]插槽为true，则中止这些步骤。
   ii. 如果连接的[[NegotiationNeeded]]插槽为false，则中止这些步骤。
   iii. 在连接时触发名为 `negotiationneeded` 的事件。

   NOTE：这种排队可防止在一次性对连接进行多次修改的常见情况下过早触发协商。

要检查连接是否需要协商，请执行以下检查：

1. 如果需要任何特定于实现的协商，如本节开头所述，则应返回true。
2. 让描述成为连接。[[CurrentLocalDescription]]。
3. 如果连接已创建任何RTCDataChannel，并且尚未为数据协商描述中的 `m= section` ，则返回true。
4. 对于连接的一组收发器中的每个收发器，执行以下检查：

   i. 如果收发器未停止且尚未与描述中的 `m= section` 关联，则返回true。

   ii. 如果收发器没有停止并且与描述中的 `m= section` 相关联，则执行以下检查：

      i. 如果收发器。[[Direction]]是“sendrecv”或“sendonly”，并且描述中关联的m =部分不包含单个“a = msid”行，或者来自“a = msid”的MSID数量“此m =节中的行或MSID值本身与transceiver.sender中的行不同。[[AssociatedMediaStreamIds]]，返回true。

      ii. 如果描述的类型为 `offer` ，并且关联 `m= section` 的方向在两个连接中都没有。[[CurrentLocalDescription]]也没有连接。[[CurrentRemoteDescription]]匹配收发器。[[Direction]]，返回true。

iii. 如果描述的类型为 `answer` ，并且描述中关联的 `m= section` 的方向与收发器不匹配。[[Direction]]与提供的方向相交（如[JSEP]（第5.3.1节）中所述） ），返回true。

iii. 如果收发器已停止且与 `m= section` 相关联，但关联的 `m= section` 尚未在连接中被拒绝[[CurrentLocalDescription]]或连接。[[CurrentRemoteDescription]]，则返回true。

5. 如果执行了所有前面的检查并且未返回true，则无需进行任何协商;则返回false。

4.8 连接建立接口

4.8.1 RTCIceCandidate接口

该接口描述了ICE候选人，在[ICE]第2节中描述。除了 `candidate` ， `sdpMid` ， `sdpMLineIndex` 和 `usernameFragment` 之外，其余属性解析来自于 `candidateInitDict` 中的 `candidate` 成员，如果它是格式良好的。

```
[Constructor(optional RTCIceCandidateInit candidateInitDict),
 Exposed=Window]
interface RTCIceCandidate {
    readonly attribute DOMString               candidate;
    readonly attribute DOMString?              sdpMid;
    readonly attribute unsigned short?         sdpMLineIndex;
    readonly attribute DOMString?              foundation;
    readonly attribute RTCIceComponent?        component;
    readonly attribute unsigned long?          priority;
    readonly attribute DOMString?              address;
    readonly attribute RTCIceProtocol?         protocol;
    readonly attribute unsigned short?         port;
    readonly attribute RTCIceCandidateType?    type;
    readonly attribute RTCIceTcpCandidateType? tcpType;
    readonly attribute DOMString?              relatedAddress;
    readonly attribute unsigned short?         relatedPort;
    readonly attribute DOMString?              usernameFragment;
    RTCIceCandidateInit toJSON();
};
```

构造函数

`RTCIceCandidate` 接口

RTCIceCandidate() 构造函数接受一个字典参数candidateInitDict，该参数用于初始化新的RTCIceCandidate对象。

调用时，请执行以下步骤：

1. 如果 `candidateInitDict` 中的 `sdpMid` 和 `sdpMLineIndex` 字典成员都为 `null` ，则抛出 TypeError 。
2. 让 `iceCandidate` 成为新创建的 `RTCIceCandidate` 对象。
3. 将 `iceCandidate` 的以下属性初始化为 `null` ： `foundation` ， `component` ， `priority` ， `address` ， `protocol` ， `port` ， `type` ， `tcpType` ， `relatedAddress`和 `relatedPort` 。
4. Set the candidate, sdpMid, sdpMLineIndex, usernameFragment attributes of iceCandidate with the corresponding dictionary member values of candidateInitDict. zh:使用candidateInitDict的相应字典成员值设置iceCandidate的候选者，sdpMid，sdpMLineIndex，usernameFragment属性。
5. 将候选人设置为 `candidateInitDict` 的 `candidate` 成员。如果设置的候选者不是空字符串，请运行以下步骤：
    i. 使用候选属性语法解析候选者。

ii. 如果候选属性语法解析失败，则中止这些步骤。

iii. 如果解析结果中的任何字段表示iceCandidate中相应属性的无效值，则中止这些步骤。

iv. 将 `iceCandidate` 中的相应属性设置为已解析结果的字段值。

6. 返回 `iceCandidate` 。

NOTE： `RTCIceCandidate` 的构造函数仅对 `candidateInitDict` 中的字典成员进行基本解析和类型检查。在将RTCIceCandidate对象传递给 addIceCandidate() 时，会完成 `candidate` , `sdpMid` , `sdpMLineIndex` , `usernameFragment` 以及相应会话描述的良构性的详细验证。

属性

以下大多数属性在[ICE]的第15.1节中定义。

`candidate` 的类型为 `DOMString` 只读项

```
`candidate-attribute`在[ICE]第15.1节中有详细说明。如果此`RTCIceCandidate表示候选者结束指示
```

`sdpMid` 的类型为 `DOMString` 只读项，可为空

```
如果不为`null`，则其包含[RFC5888]中定义的媒体流“标识标签”，用于与该候选者相关联的媒体组件。
```

`sdpMLineIndex` 的类型为 `unsigned short` 只读项，可为空

```
如果不为空，则这指示该候选与之关联的SDP中的媒体描述的索引（从零开始）。
```

`foundation` 的类型为 `DOMString` 只读项，可为空 zh:DOMString类型的基础，只读，可空

```
一个唯一标识符，允许ICE关联出现在多个 RTCIceTransports 上的候选项。
```

`component` 的类型为 `RTCIceComponent` 只读，可为空

```
候选人指定的网络组件（rtp或rtcp）。这对应于 `candidate-attribute` 中的 `component-id` 字段
```

`priority` 的类型为 `unsigned long` 只读，可为空

```
候选人指定的优先顺序。
```

`address` 的类型为 `DOMString` 只读，可为空

候选的地址，允许IPv4地址，IPv6地址和完全限定的域名（FQDN）。这对应于候选属性中的连接地址字段。

> NOTE：通过ICE收集并在RTCIceCandidate实例中对应用程序进行访问的候选者中公开的地址可以揭示有关设备和用户的更多信息（例如，位置，本地网络拓扑），而不是用户在非WebRTC启用的浏览器中可能预期的信息。
>
> 这些地址始终暴露给应用程序，并且可能暴露给通信方，并且可以在没有任何特定用户同意的情况下暴露（例如，用于与数据信道一起使用的对等连接，或仅用于接收媒体）。
>
> 这些地址也可以用作临时或持久的跨源状态，因此有助于设备的指纹表面。
>
> 应用程序可以通过强制ICE代理仅通过RTCConfiguration的iceTransportPolicy成员报告中继候选者来避免暂时或永久地向通信方公开地址。

>

> 为了限制暴露给应用程序本身的地址，浏览器可以为其用户提供有关共享本地地址的不同策略，如[RTCWEB-IP-HANDLING]中所定义。

`protocol` 的类型为 `RTCIceProtocol`，只读项，可为空

候选协议可以为udp和tcp。对应于候选属性中的传输字段。

`port` 的类型为 `unsigned short`，只读项，可为空

候选人的端口。

`type` 的类型为 `RTCIceCandidateType`，只读项，可为空

候选人的类型。对应于候选属性中的传输字段。

`tcpType` 的类型为 `RTCIceTcpCandidateType`，只读项，可为空

如果 protocol 是 tcp ，则 tcpType 表示 TCP 候选的类型。否则，tcpType为null。这对应于candid

`relatedAddress` 的类型为 `DOMString`，只读项，可为空

对于从另一个派生的候选者（例如中继或自反候选者），relatedAddress是从其派生的候选者的IP地址。

`relatedPort` 的类型为 `unsigned short`，只读项，可为空

对于从另一个派生的候选者（例如中继或反身候选者），`relatedPort` 是从其派生的候选者的端口。对

`usernameFragment` 的类型为 `DOMString`，只读项，可为空 详细描述参照[ICE]第15.4节中定义的ufrag。

方法

`toJSON()` toJSON()

1. 把json成为一个新的RTCIceCandidateInit字典。
2. 要调用RTCIceCandidate接口的toJSON()操作，请运行以下步骤：让json成为新的RTCIceCandidateInit字典。对于«“候选者”，“sdpMid”，“sdpMLineIndex”，“usernameFragment”中的每个属性标识符attr»:
    i. 在给定此RTCIceCandidate对象的情况下，将value设置为获取由attr标识的属性的基础值的结果。
    ii. 将json [attr]设置为value。
3. 返回json。

```
dictionary RTCIceCandidateInit {
    DOMString       candidate = "";
    DOMString?      sdpMid = null;
    unsigned short? sdpMLineIndex = null;
    DOMString       usernameFragment;
};
```

字典RTCIceCandidateInit成员

`candidate` of type DOMString, defaulting to "": 详细描述参照[ICE]第15.1节中定义的候选属性。如果这表示 `candidate` 结束指示，则 `candidate` 是空字符串。

`sdpMid` 的类型为 `DOMString`, 可为空，默认为null 如果不为空，则其包含[RFC5888]中定义的媒体流“标识标签”，用于与该 `candidate` 相关联的媒体组件。

`sdpMLineIndex` 的类型为 `unsigned short`, 可为空，默认为空 如果不为空，则这指示该候选与之关联的SDP中的媒体描述的索引（从零开始）。

`usernameFragment` 的类型为 `DOMString` 只读，可为空 详细描述参照[ICE]第15.4节中有 `ufrag` 的定义。

4.8.1.1 候选属性语法

候选属性语法用于在RTCIceCandidate() 构造函数中解析candidateInitDict的候选成员。

候选属性主要语法在[ICE]的第15.1节中定义。此外，浏览器必须支持ICE TCP的语法扩展，如[RFC6544]第4.5节中所定义。

浏览器可以支持其他RFC中定义的候选属性的其他语法扩展。

4.8.1.2 RTCIceProtocol枚举

`RTCIceProtocol` 表示ICE候选者的协议。

```
enum RTCIceProtocol {
    "udp",
    "tcp"
};
```

枚举描述	
udp	表示一个基于UDP协议的候选人，详细描述参考 [ICE].
tcp	表示一个基于tcp协议的候选人，详细描述参考 [RFC6544].

4.8.1.3 RTCIceTcpCandidateType枚举

`RTCIceTcpCandidateType` 表示ICE TCP候选的类型，如[RFC6544]中所定义。

```
enum RTCIceTcpCandidateType {
    "active",
    "passive",
    "so"
};
```

枚举描述	
active	表示一个`active`状态的TCP候选人，这将会尝试打开一个出站连接，但不会接收传入连接的请求。
passive	表示一个`passive`状态的TCP候选人，这将会接收传入连接的请求，但不会尝试打开一个出站连接。
so	表示一个`so`状态的候选人，这将会尝试与对端同时打开一个连接。

NOTE：用户代理通常只收集活动的ICE TCP候选者。

4.8.1.4 RTCIceCandidateType枚举

RTCIceCandidateType表示ICE候选的类型，如[ICE]第15.1节中所定义。

```
enum RTCIceCandidateType {
    "host",
    "srflx",
    "prflx",
    "relay"
};
```

枚举描述	
host	表示一个主持人候选人, 详细描述参考 4.1.1.1 节 [ICE].
srflx	表示一个服务端反身候选人, 详细描述参考 4.1.1.2 节 [ICE].
prflx	表示一个对端反身候选人，详细描述参考 4.1.1.2 节 [ICE].
relay	表示一个中继候选人, 详细描述参考 7.1.3.2.1 节 [ICE].

4.8.2 RTCPeerConnectionIceEvent接口

`RTCPeerConnection` 的 `icecandidate` 事件使用 `RTCPeerConnectionIceEvent` 接口。

当触发包含 `RTCIceCandidate` 对象的 `RTCPeerConnectionIceEvent` 事件时，它必须包含 `sdpMid` 和 `sdpMLineIndex` 的值。如果RTCIceCandidate的类型为srflx或类型为relay，则事件的url属性必须设置为从中获取候选者的ICE服务器的URL。

NOTE

The icecandidate event is used for three different types of indications:
zh:icecandidate事件用于三种不同类型的指示：

- A candidate has been gathered. The candidate member of the event will be populated normally. It should be signaled to the remote peer and passed into addIceCandidate. zh: 候选人已经聚集。该活动的候选成员将正常填充。它应该发信号通知远程对等体并传递给addIceCandidate。
- An RTCIceTransport has finished gathering a generation of candidates, and is providing an end-of-candidates indication as defined by Section 8.2 of [TRICKLE-ICE]. This is indicated by candidate.candidate being set to an empty string. The candidate object should be signaled to the remote peer and passed into addIceCandidate like a typical ICE candidate, in order to provide the end-of-candidates indication to the remote peer. zh: RTCIceTransport已完成收集一代候选者，并提供[TRICKLE-ICE]第8.2节定义的候选终结指示。这由candidate.candidate设置为空字符串表示。候选对象应该被发信号通知远程对等体并像典型的ICE候选者一样被传递到addIceCandidate，以便向远程对等体提供候选终止指示。
- All RTCIceTransports have finished gathering candidates, and the RTCPeerConnection's RTCIceGatheringState has transitioned to "complete". This is indicated by the candidate member of the event being set to null. This only exists for backwards compatibility, and this event does not need to be signaled to the remote peer. It's equivalent to an "icegatheringstatechange" event with the "complete" state. zh: 所有RTCIceTransports都已完成收集候选者，RTCPeerConnection的RTCIceGatheringState已转换为“完成”。这由事件的候选成员设置为null来指示。这仅用于向后兼容，并且不需要向远程对等方发信号通知此事件。它相当于具有“完整”状态的“icegatheringstatechange”事件。

```
[Constructor(DOMString type, optional RTCPeerConnectionIceEventInit eventInitDict),
 Exposed=Window]
interface RTCPeerConnectionIceEvent : Event {
    readonly attribute RTCIceCandidate? candidate;
    readonly attribute DOMString?       url;
};
```

Constructors zh:构造函数

`RTCPeerConnectionIceEvent`

Attributes zh:属性

candidate of type RTCIceCandidate, readonly, nullable: zh:RTCIceCandidate类型的候选者，只读，可以为空

The candidate attribute is the RTCIceCandidate object with the new ICE candidate that caused the event.

zh:候选属性是RTCIceCandidate对象，其中包含导致该事件的新ICE候选对象。

This attribute is set to null when an event is generated to indicate the end of candidate gathering.

zh:生成事件以指示候选收集结束时，此属性设置为null。

NOTE Even where there are multiple media components, only one event containing a null candidate is fired.

zh:即使存在多个媒体组件，也只会触发一个包含空候选的事件。

url of type DOMString, readonly, nullable: zh:DOMString类型的url，只读，可以为空

The url attribute is the STUN or TURN URL that identifies the STUN or TURN server used to gather this candidate. If the candidate was not gathered from a STUN or TURN server, this parameter will be set to null.

zh:url属性是STUN或TURN URL，用于标识用于收集此候选项的STUN或TURN服务器。如果未从STUN或TURN服务器收集候选项，则此参数将设置为null。

```
dictionary RTCPeerConnectionIceEventInit : EventInit {
    RTCIceCandidate? candidate;
    DOMString?       url;
};
```

Dictionary RTCPeerConnectionIceEventInit Members zh:字典RTCPeerConnectionIceEventInit成员

candidate of type RTCIceCandidate, nullable: zh:RTCIceCandidate类型的候选者，可以为空

See the candidate attribute of the RTCPeerConnectionIceEvent interface.

zh:请参阅RTCPeerConnectionIceEvent接口的候选属性。

url of type DOMString, nullable: zh:DOMString类型的url，可以为空

The url attribute is the STUN or TURN URL that identifies the STUN or TURN server used to gather this candidate.

zh:url属性是STUN或TURN URL，用于标识用于收集此候选项的STUN或TURN服务器。

4.8.3 RTCPeerConnectionIceErrorEvent接口

`RTCPeerConnection` 的 `icecandidateerror` 事件使用 `RTCPeerConnectionIceErrorEvent` 接口。

```
[Constructor(DOMString type, RTCPeerConnectionIceErrorEventInit eventInitDict),
 Exposed=Window]
interface RTCPeerConnectionIceErrorEvent : Event {
    readonly attribute DOMString      hostCandidate;
    readonly attribute DOMString      url;
    readonly attribute unsigned short errorCode;
    readonly attribute USVString      errorText;
};
```

构造函数

`RTCPeerConnectionIceErrorEvent`

属性

`hostCandidate` 的类型为 `DOMString`，只读项:

`hostCandidate` 属性是用于与STUN或TURN服务器通信的本地IP地址和端口。

在多宿主系统上，可以使用多个接口来联系服务器，该属性允许应用程序确定故障发生在哪一个上。

如果出于隐私原因禁止使用多个接口，则此属性将根据需要设置为0.0.0.0:0或[::]:0。

`url` 的类型为 `DOMString`，只读项:

`url` 属性是 STUN 或 TURN URL，用于标识发生故障的STUN或TURN服务器。

`errorCode` 的类型为 `unsigned short`，只读项:

`errorCode` 属性是 STUN 或 TURN 服务器 [STUN-PARAMETERS] 返回的数字 STUN 错误代码。

如果没有主机候选者可以到达服务器，则 `errorCode` 将被设置为超出 STUN 错误代码范围的值701。在“收集”的 `RTCIceGatheringState` 中，每个服务器 URL 仅触发一次此错误。

`errorText` 的类型为 `USVString`，只读项:

`errorText` 属性是 STUN 或 TURN 服务器 [STUN-PARAMETERS] 返回的 STUN 原因文本。

如果无法访问服务器，则 `errorText` 将设置为特定于实现的值，提供有关错误的详细信息。

```
dictionary RTCPeerConnectionIceErrorEventInit : EventInit {
             DOMString      hostCandidate;
             DOMString      url;
    required unsigned short errorCode;
             USVString      statusText;
};
```

字典**RTCPeerConnectionIceErrorEventInit**的成员

`hostCandidate` 的类型为 `DOMString`

用于与 STUN 或 TURN 服务器通信的本地地址和端口。

`url` 的类型为 `DOMString`：

STUN 或 TURN URL，用于标识发生故障的STUN或TURN服务器。

`errorCode` 的类型为unsigned short，必需项：

STUN 或 TURN 服务器返回的数字 STUN 错误代码。

`statusText` 的类型为 `USVString`：

STUN 或 TURN 服务器返回的 STUN 原因文本。

4.9 优先级和QoS模型

许多应用程序中包含多路相同数据类型的媒体流，并且当中的某些媒体流相比于其他媒体流质量更为重要。WebRTC使用了在[RTCWEB-TRANSPORT]和[TSVWG-RTCWEB-QOS]中描述的优先级和服务质量（QoS）框架，来帮助在某些网络环境中标记特定类型数据包的优先级以及差分服务代码点（DSCP），以此来达到提升服务质量的目的。优先级设定可以用来标识不同媒体流的相对优先级。通过调用优先级设置API可以让JavaScript应用程序更改RTCRtpEncodingParameters对象的中priority属性来告诉浏览器特定媒体流的优先级是高，中，低还是非常低，priority属性可以设置为以下值。

4.9.1 RTCPriorityType枚举类型

```
enum RTCPriorityType {
    "very-low",
    "low",
    "medium",
    "high"
};
```

枚举类型描述	
very-low	参看[RTCWEB-TRANSPORT], 第4.1和4.2小节. 等同于定义在[RTCWEB-DATA]中的"below normal".
low	参看[RTCWEB-TRANSPORT], 第4.1和4.2小节. 等同于定义在[RTCWEB-DATA]中的"normal".
medium	参看[RTCWEB-TRANSPORT], 第4.1和4.2小节. 等同于定义在[RTCWEB-DATA]中的"high".
high	参看[RTCWEB-TRANSPORT], 第4.1和4.2小节. 等同于定义在[RTCWEB-DATA]中的"extra high".

当在应用程序中使用该API时，开发者需要意识到更好的整体用户体验通常来自于主动降低某些不重要媒体流的优先级而不是一味地去提高某些重要媒体流的优先级。

4.10证书管理

RTCPeerConnection实例用于与对等方进行身份验证的证书使用RTCCertificate接口。这些对象可以由使用generateCertificate方法的应用程序显式生成，并且可以在构造新的RTCPeerConnection实例时在RTCConfiguration中提供。

此处提供的显式证书管理功能是可选的。如果应用程序在构造RTCPeerConnection时未提供证书配置选项，则必须由用户代理生成一组新证书。该集必须包括一个ECDSA证书，在P-256曲线上有一个私钥，一个签名带有SHA-256哈希。

```
partial interface RTCPeerConnection {
    static Promise<RTCCertificate> generateCertificate(AlgorithmIdentifier keygenAlgor
};
```

方法

generateCertificate, static

generateCertificate函数使用户代理创建并存储X.509证书X509V3和相应的私钥。以RTCCertificate接口的形式提供信息句柄。返回的RTCCertificate可用于控制由RTCPeerConnection建立的DTLS会话中提供的证书。

keygenAlgorithm参数用于控制如何生成与证书关联的私钥。 keygenAlgorithm参数使用WebCrypto WebCryptoAPI AlgorithmIdentifier类型。 keygenAlgorithm值必须是window.crypto.subtle.generateKey的有效参数;也就是说，当根据WebCrypto算法规范化过程[WebCryptoAPI]进行规范化时，该值必须产生非错误结果，其中操作名称为generateKey，并且[[supportedAlgorithms]]值特定于RTCPeerConnection的证书生成。如果算法规范化过程产生错误，则必须拒绝对generateCertificate的调用。

生成的密钥生成的签名用于验证DTLS连接。所识别的算法（由标准化AlgorithmIdentifier的名称标识）必须是可用于产生签名的非对称算法。

此过程生成的证书还包含签名。此签名的有效性仅与兼容性原因相关。RTCPeerConnection仅使用公钥和生成的证书指纹，但如果证书格式正确，则更有可能接受证书。浏览器选择用于签署证书的算法;如果需要哈希算法，浏览器应该选择SHA-256 FIPS-180-4。

生成的证书不得包含可链接到用户或用户代理的信息。应该使用可分辨名称和序列号的随机值。

如果keygenAlgorithm参数标识用户代理不能或不会用于为RTCPeerConnection生成证书的算法，则用户代理必须拒绝使用类型为NotSupportedError的DOMException调用generateCertificate（）。

用户代理必须支持以下值：{name：“RSASSA-PKCS1-v1_5”，modulusLength：2048，publicExponent：new Uint8Array（[1,0,1]），hash：“SHA-256”}，以及{name：“ECDSA”，namedCurve：“P-256”}。

NOTE

预计用户代理将具有它将接受的小的或甚至固定的值集。

4.10.1 RTCCertificateExpiration词典

RTCCertificateExpiration用于设置generateCertificate生成的证书的到期日期。

```
dictionary RTCCertificateExpiration {
    [EnforceRange]
    DOMTimeStamp expires;
};
```

expires

可选的expires属性可以添加到传递给generateCertificate的算法的定义中。如果此参数存在，则表示RTCCertificate相对于当前时间有效的最长时间。

当使用object参数调用generateCertificate时，用户代理会尝试将对象转换为RTCCertificateExpiration。如果这不成功，立即返回一个被新创建的TypeError和中止处理拒绝的promise。

用户代理生成的证书的到期日期设置为当前时间加上expires属性的值。返回的RTCCertificate的expires属性设置为证书的到期时间。用户代理可以选择限制expires属性的值。

4.10.2 RTCCertificate Interface zh:4.10.2 RTCCertificate接口

The RTCCertificate interface represents a certificate used to authenticate WebRTC communications. In addition to the visible properties, internal slots contain a handle to the generated private keying materal ([[KeyingMaterial]]), a certificate ([[Certificate]]) that RTCPeerConnection uses to authenticate with a peer, and the origin ([[Origin]]) that created the object.

zh:RTCCertificate接口表示用于验证WebRTC通信的证书。除了可见属性之外，内部插槽还包含生成的私有密钥子集（[[KeyingMaterial]]）的句柄，RTCPeerConnection用于与对等体进行身份验证的证书（[[Certificate]]）和原点（[[[[Certificate]]） Origin]]）创建了对象。

```
[Exposed=Window,
 Serializable]
interface RTCCertificate {
    readonly attribute DOMTimeStamp expires;
    static sequence<AlgorithmIdentifier> getSupportedAlgorithms();
    sequence<RTCDtlsFingerprint>         getFingerprints();
};
```

Attributes zh:属性

expires of type DOMTimeStamp, readonly

he expires attribute indicates the date and time in milliseconds relative to 1970-01-01T00:00:00Z after which the certificate will be considered invalid by the browser. After this time, attempts to construct an RTCPeerConnection using this certificate fail.

zh:expires属性指示相对于1970-01-01T00：00：00Z的日期和时间（以毫秒为单位），之后浏览器将认为证书无效。在此之后，尝试使用此证书构造RTCPeerConnection失败。

Note that this value might not be reflected in a notAfter parameter in the certificate itself.

zh:请注意，此值可能不会反映在证书本身的notAfter参数中。

Methods zh:方法

getSupportedAlgorithms

Returns a sequence providing a representative set of supported certificate algorithms. At least one algorithm MUST be returned.

zh:返回提供一组有代表性的支持证书算法的序列。必须返回至少一个算法。

For example, the "RSASSA-PKCS1-v1_5" algorithm dictionary, RsaHashedKeyGenParams, contains fields for the modulus length, public exponent, and hash algorithm. Implementations are likely to support a wide range of modulus lengths and exponents, but a finite number of hash algorithms. So in this case, it would be reasonable for the implementation to return one

AlgorithmIdentifier for each supported hash algorithm that can be used with RSA, using default/recommended values for modulusLength and publicExponent (such as 1024 and 65537, respectively).

zh:例如，“RSASSA-PKCS1-v1_5”算法字典RsaHashedKeyGenParams包含模数长度，公共指数和散列算法的字段。实现可能支持各种模数长度和指数，但是有限数量的散列算法。因此，在这种情况下，对于每个支持的可与RSA一起使用的哈希算法，使用modulusLength和publicExponent的默认/推荐值（分别为1024和65537），为实现返回一个AlgorithmIdentifier是合理的。

getFingerprints

Returns the list of certificate fingerprints, one of which is computed with the digest algorithm used in the certificate signature.

zh:返回证书指纹列表，其中一个是使用证书签名中使用的摘要算法计算的。

For the purposes of this API, the [[Certificate]] slot contains unstructured binary data. No mechanism is provided for applications to access the [[KeyingMaterial]] internal slot. Implementations MUST support applications storing and retrieving RTCCertificate objects from persistent storage. In implementations where an RTCCertificate might not directly hold private keying material (it might be stored in a secure module), a reference to the private key can be held in the [[KeyingMaterial]] internal slot, allowing the private key to be stored and used.

zh:出于此API的目的，[[Certificate]]插槽包含非结构化二进制数据。没有为应用程序提供访问[[KeyingMaterial]]内部插槽的机制。实现必须支持从持久存储中存储和检索RTCCertificate对象的应用程序。在RTCCertificate可能不直接保存私人密钥资料（可能存储在安全模块中）的实现中，私钥的引用可以保存在[[KeyingMaterial]]内部插槽中，允许私钥存储和用过的。

RTCCertificate objects are serializable objects [HTML]. Their serialization steps, given value and serialized, are:

zh:RTCCertificate对象是可序列化的对象[HTML]。它们的序列化步骤，给定值和序列化，是：

1. Set serialized.[[Expires]] to the value of value's expires attribute. zh:将序列化。[[Expires]]设置为value的expires属性的值。
2. Set serialized.[[Certificate]] to a copy of the unstructured binary data in value's [[Certificate]] slot. zh:将序列化。[[Certificate]]设置为值[[Certificate]]插槽中非结构化二进制数据的副本。
3. Set serialized.[[Origin]] to a copy of the unstructured binary data in value's [[Origin]] slot. zh:将序列化。[[Origin]]设置为值[[Origin]]插槽中非结构化二进制数据的副本。
4. Set serialized.[[KeyingMaterial]] to a serialization of the private keying material represented by value's [[KeyingMaterial]] slot. zh:将序列化。[[KeyingMaterial]]设置为由值[[KeyingMaterial]]槽表示的私有密钥材料的序列化。

Their deserialization steps, given serialized and value, are:

zh:根据序列化和价值，他们的反序列化步骤是：

1. Initialize value's expires attribute to contain serialized.[[Expires]]. zh:初始化value的expires属性以包含serialized。[[Expires]]。
2. Set value's [[Certificate]] slot to a copy of serialized.[[Certificate]]. zh:将值的[[Certificate]]插槽设置为序列化的副本。[[Certificate]]。
3. Set value's [[Origin]] slot to a copy of serialized.[[Origin]]. zh:将值的[[Origin]]插槽设置为序列化的副本。[[Origin]]。
4. Set value's [[KeyingMaterial]] slot to the private key material resulting from deserializing serialized.[[KeyingMaterial]] zh:将值的[[KeyingMaterial]]槽设置为序列化反序列化产生的私钥材料。[[KeyingMaterial]]

Supporting structured cloning in this manner allows RTCCertificate instances to be persisted to stores. It also allows instances to be passed to other origins using APIs like postMessage [webmessaging]. However, the object cannot be used by any other origin than the one that originally created it.

zh:以这种方式支持结构化克隆允许将RTCCertificate实例持久化到商店。它还允许使用postMessage [webmessaging]等API将实例传递给其他来源。但是，该对象不能由最初创建它的任何其他来源使用。

5. RTP Media API

RTP媒体API允许Web应用程序通过p2p连接发送和接收MediaStreamTracks。当(音视频)轨道添加到RTCPeerConnection时会产生信令消息;当该消息 被转发到远程对端时，会在远端创建相应的(音视频)轨道。

NOTE:

一个RTCPeerConnection发送的轨道与另一个RTCPeerConnection接收的轨道之间没有确切的1：1对应关系。例如，发送的轨道的ID没有映射到接>收的轨道ID。此外，replaceTrack更改RTCRtpSender发送的轨道，而不会在接收方创建新的轨道;相应的RTCRtpReceiver只有一个轨道，可能代表缝>合在一起的多个媒体源。addTransceiver和replaceTrack接口都可以用于使用相同的轨道多次发送，这将在接收器侧被观察为多个接收器，每个接>收器具有其自己的独立轨道。因此，考虑发送侧的RTCRtpSender与接受侧的RTCRtpReceiver的轨道之间的1：1关系更为准确，如果需要，使用RTCRtpTransceiver的mid匹配发送者和接收者。

发送媒体时，发送方可能需要对媒体数据进行缩放或重采样处理，以满足包括SDP协商在内的各种要求。

遵循[JSEP]（第3.6节）中的规则，可以缩小视频尺寸以适应SDP约束。媒体数据不得通过拉伸来创建未在输入源中出现的伪数据;除非为满足像素 计数约束，否则不得裁剪媒体，并且不能更改视频宽高比。

NOTE:

WebRTC工作组正在寻求关于需求和时间表的实施反馈，以便更全面地处理这种情况。 GitHub issue 1283中讨论了一些可能的设计。

当视频被重新缩放时，例如对于宽度或高度的某些组合以及根据scaleResolutionDownBy数值的缩放，可能出现所得宽度或高度不是整数的情况。在这种情况下，用户代理必须使用结果的整数部分。如果缩放宽度或高度的整数部分为零，传输行为则可依据特定实现。

MediaStreamTracks的实际编码和传输是通过名为RTCRtpSenders的对象来管理的。同样，MediaStreamTracks的接收和解码通过称为RTCRtpReceivers的对象进行管理。每个RTCRtpSender与至多一个轨道相关联，并且要接收的每个轨道仅与一个RTCRtpReceiver相关联。

应该对每个MediaStreamTrack进行编码和传输，使其特性（视频轨道的宽度，高度和帧率;音频轨道的音量，采样大小，采样率和声道数）与在远 端创建的轨道保持合理的程度。在某些情况下，如果不适用，可能会在端上或网络中出现资源限制，或者可能会应用RTCRtpSender设置来指示实施采 取不同的行动。

RTCPeerConnection对象包含一组RTCRtpTransceivers，表示配对的发送者和接收者具有某种共享状态。创建RTCPeerConnection对象时，此集初 始化为空集。RTCRtpSenders和RTCRtpReceivers始终与RTCRtpTransceiver同时创建，它们将在其生命周期内保持连接状态。当应用程序通过addTrack方法将MediaStreamTrack附加到RTCPeerConnection时，或者在应用程序使用addTransceiver方法时显式地创建RTCRtpTransceivers。当远端SDP中 加入新的

媒体描述时，也会创建它们。此外，当远端SDP指示端上有要发送的媒体时，相关的MediaStreamTrack和RTCRtpReceiver会通过(音视频)轨道的事件添加到应用程序。

5.1 RTCPeerConnection Interface Extensions

RTP媒体API扩展了RTCPeerConnection接口，如下所述。

```
partial interface RTCPeerConnection {
    sequence<RTCRtpSender>      getSenders ();
    sequence<RTCRtpReceiver>    getReceivers ();
    sequence<RTCRtpTransceiver> getTransceivers ();
    RTCRtpSender                addTrack (MediaStreamTrack track, MediaStream... strea
    void                        removeTrack (RTCRtpSender sender);
    RTCRtpTransceiver           addTransceiver ((MediaStreamTrack or DOMString) trackO
                    attribute EventHandler ontrack;
};
```

属性

ontrack 数据类型为EventHandler:

该事件句柄的事件类型是track。

方法

`getSenders`

该方法返回一系列RTCRtpSender对象，这些对象是RTP数据的发送者，从属于未停止的RTCRtpTransceiver对象，并且当前被附加到此RTCPeerConnection中。

当调用getSenders方法时，用户代理必须返回执行CollectSenders算法的结果。

CollectSenders算法定义如下：

1. 执行CollectTransceivers算法，结果得到一个收发器序列。
2. 将发送者序列置为空。
3. 遍历收发器序列中的每个收发器，
    i. 如果收发器的[[Stopped]]为false，则将该收发器的[[Sender]]添加到发送者序列。
4. 返回发送者序列。

`getReceivers`

该方法返回一系列RTCRtpReceiver对象，这些对象是RTP数据的接收者，从属于未停止的RTCRtpTransceiver对象，并且当前被附加到此RTCPeerConnection中。

当调用getReceivers方法时，用户代理必须运行以下步骤：

1. 执行CollectTransceivers算法，结果得到一个收发器序列。
2. 将接收者序列置为空。
3. 遍历收发器序列中的每个收发器，
    i. 如果收发器的[[Stopped]]为false，则将该收发器的[[Receiver]]添加到接收者序列。
4. 返回接收者序列。

`getTransceivers`

该方法返回一系列RTCRtpTransceiver对象，这些对象是RTP数据的收发器，且当前被附加到此RTCPeerConnection中。

getTransceivers方法必须返回执行CollectTransceivers算法的结果。

CollectTransceivers算法定义如下：

1. 将收发器序列置为空，由该RTCPeerConnection对象收集所有RTCRtpTransceiver对象并插入收发器序列，按插入顺序排列。
2. 返回收发器序列。

`addTrack`

向RTCPeerConnection添加新轨道，并表明它包含在指定的MediaStream中。

当调用addTrack方法时，用户代理必须运行以下步骤：

1. 让connection成为调用此方法的RTCPeerConnection对象。
2. 让track成为方法第一个参数指示的MediaStreamTrack对象。
3. 让kind成为track.kind。
4. 剩余参数为MediaStream对象列表，如果使用单个参数调用该方法，则该列表为空。
5. 如果connection的slot为true，则抛出InvalidStateError。
6. 发送者列表应该是执行CollectSenders算法的结果，如果track中的RTCRtpSender已经存在，则抛出InvalidAccessError。
7. 以下步骤描述了如何确定是否可以重用现有发送者对象。这样做可以在将来调用createOffer和createAnswer时能够将相应的media描述标记为sendrecv或sendonly，并添加发送者流的MSID，如[JSEP]（第5.2.2节和第5.3.2节）中所定义。

   如果发送者序列中的任何RTCRtpSender对象符合以下所有条件，则让该发送者成为可重用的对象，否则为null：

   - 发送者的track为空。
   - 发送者和与其相关联的收发器的kind类型相匹配。
   - 收发器没有停止。更确切地说，与发送者关联的收发器的[[Stopped]]为false。
   - 发送者从未被用来发送。更确切地说，与发送者关联的收发器的[[CurrentDirection]]从未具有sendrecv或sendonly的值。
8. 如果发送者不为空，请执行以下步骤以使用该发送者：
   i. 将发送者设置到track里面。
   ii. 将发送者的[[AssociatedMediaStreamIds]]设置为空集。
   iii. 对于流列表中的每个流，如果stream.id在[[AssociatedMediaStreamIds]]中不存在则将id添加进去。
   iv. 让收发器成为与发送者关联的RTCRtpTransceiver对象。

v. 如果收发器的[[Direction]]值是recvonly，请将收发器的[[Direction]]值设置为sendrecv。

vi. 如果收发器的[[Direction]]为无效值，则将收发器的[[Direction]]设置为sendonly。

9. 如果发送者为空，请执行以下步骤：

i. 创建一个包含track，kind和streams的发送者。

ii. 创建一个包含kind的接收器。

iii. 创建一个包含之前创建的发送者，接收者并且RTCRtpTransceiverDirection值为sendrecv的收发器。

iv. 将收发器添加到connection的收发器集

10. track中可能包含应用程序无法访问的内容。这可能是由于track被标记了peerIdentity选项或任何会使track CORS交叉起源的选项。这些track可以提供给addTrack方法，并为它们创建一个RTCRtpSender对象，但内容绝不能被传输，除非它们也标记为peerIdentity并且它们符合发送要求（参见隔离流）。应用程序无法访问的所有其他tracks不得发送给对端，取而代之的是发送静音（音频），黑帧（视频）或等效的内容。请注意，此属性可能会随时间而变化。

11. 更新连接所需的协商标志。

12. 返回发送者对象。

`removeTrack`

发送者停止发送媒体数据，但getSenders仍然可以获取RTCRtpSender。这样将来调用createOffer时可以将相应收发器的媒体描述标记为recvonly或inactive，如[JSEP]（第5.2.2节）中所定义。

当另一端用这种方式停止发送一个track，这个track将从最初在track事件中显示的任何远端MediaStream中移除，并且如果MediaStreamTrack尚未静音，则在轨道处触发静音事件。

当调用removeTrack方法时，用户代理必须执行以下步骤：

1. 让sender成为removeTrack的参数。

2. 让connection成为调用该方法的RTCPeerConnection对象。

3. 如果connection的[[IsClosed]]值为true，则抛出InvalidStateError异常。

4. 如果sender不是connection创建的，则抛出InvalidAccessError异常。

5. 执行CollectSenders算法返回所有的sender。

6. 如果sender不在sender列表中（表示由于设置了“回滚”类型的RTCSessionDescription而将其删除），则中止这些步骤。

7. 如果sender的[[SenderTrack]]值为空，则中止这些步骤。

8. 将sender的[[SenderTrack]]值设置为null。

9. 将收发器设置为与发送者对应的RTCRtpTransceiver对象。

10. 如果收发器的[[Direction]]值是sendrecv，请将收发器的[[Direction]]值设置为recvonly。
11. 如果收发器的[[Direction]]值是sendonly，则将收发器的[[Direction]]值设置为inactive。
12. 更新连接所需的协商标志。

`addTransceiver`

创建一个新的RTCRtpTransceiver并将其添加到收发器序列。

添加一个收发器，则将来调用createOffer会为相应的收发器添加媒体描述，如[JSEP]（第5.2.2节）中所定义。

mid的初始值为null，设置新的RTCSessionDescription可能会将其更改为非空值，如[JSEP]（第5.5节和第5.6节）中所定义。

sendEncodings参数可用于指定提供的联播编码的数量，以及可选的RID和编码参数。

当调用此方法时，用户代理必须执行以下步骤：

1. 让init成为第二个参数。
2. 让streams成为init的streams成员。
3. 让sendEncodings成为init的sendEncodings成员。
4. 让direction成为init的direction成员。
5. 如果第一个参数是一个字符串，那就让它成为kind，然后运行以下步骤：
    i. 如果kind不是合法的MediaStreamTrack类型，则抛出TypeError。
    ii. 将track置为空。
6. 如果第一个参数是MediaStreamTrack，那么让它成为track并将kind赋值track.kind。
7. 如果connection的[[IsClosed]]值为true，则抛出InvalidStateError异常。
8. 通过运行以下步骤验证sendEncodings合法性：
    i. 验证sendEncodings中的每个rid值仅由字母数字字符（a-z，A-Z，0-9）组成，最多16个字符。如果其中一个RID不满足这些要求，则抛出TypeError异常。
    ii. 如果sendEncodings中的任何RTCRtpEncodingParameters字典包含除rid之外的只读参数，则抛出InvalidAccessError异常。
    iii. 验证sendEncodings中的每个scaleResolutionDownBy值是否大于或等于1.0。如果scaleResolutionDownBy值之一不满足此要求，则抛出RangeError异常。
    iv. 验证sendEncodings中的每个maxFramerate值是否大于或等于0.0。如果其中一个maxFramerate值不满足此要求，则抛出RangeError异常。
    v. 令maxN为用户代理可以支持的最大同时编码的数量，至少为1.这应该是一个乐观的数字，因为要使用的编解码器还不知道。

vi. 如果存储在sendEncodings中的RTCRtpEncodingParameters的数量超过maxN，则从尾部修剪sendEncodings直到其长度为maxN。

vii. 如果现在存储在sendEncodings中的RTCRtpEncodingParameters的数量为1，则从单个条目中删除任何rid成员。

注意： 如果在sendEncodings中只提供单个默认RTCRtpEncodingParameters，则允许应用程序随后使用setParameters方法设置编码参数，即使未使用联播也是如此。

9. 创建一个包含track，kind，streams和sendEncodings的RTCRtpSender。

如果设置了sendEncodings，则后续调用createOffer时将被配置为发送多个RTP编码，定义在[JSEP]（第5.2.2节和第5.2.1节）。当使用能够接收多个RTP编码，定义在[JSEP]（第3.7节），的相应远程描述调用setRemoteDescription方法时，RTCRtpSender可以发送多个RTP编码，并且通过收发器的sender.getParameters()检索的参数可以反映出来协商的编码格式。

10. 创建一个带有kind的RTCRtpReceiver。

11. 创建一个带sender，receiver和direction的RTCRtpTransceiver。

12. 将收发器添加到connection的收发器序列

13. 更新connection所需的协商标志。

14. 返回收发器。

```
dictionary RTCRtpTransceiverInit {
             RTCRtpTransceiverDirection         direction = "sendrecv";
             sequence<MediaStream>              streams = [];
             sequence<RTCRtpEncodingParameters> sendEncodings = [];
};
```

字典 `RTCRtpTransceiverInit` 的参数列表

direction 类型为RTCRtpTransceiverDirection，默认为“sendrecv”

该参数属于RTCRtpTransceiver的成员。

streams 类型为序列

当远程PeerConnection的track事件触发与正在添加的RTCRtpReceiver相对应时，这些streams将被放入事件中。

sendEncodings 类型为序列

包含用于发送媒体的RTP编码的参数的序列。

```
enum RTCRtpTransceiverDirection {
    "sendrecv",
    "sendonly",
    "recvonly",
    "inactive"
};
```

RTCRtpTransceiverDirection 枚举值描述	
sendrecv	RTCRtpTransceiver的类型为RTCRtpSender的sender将提供去发送RTP，并且如果远程对等方接受并且sender.getParameters().encodings[i].active对于任何i的值为true，则将发送RTP数据。 RTCRtpTransceiver的RTCRtpReceiver将提供去接收RTP，并在对端接受时接收RTP。
sendonly	RTCRtpTransceiver的类型为RTCRtpSender的sender将提供去发送RTP，并且如果远程对等方接受并且sender.getParameters().encodings[i].active对于任何i的值为true，则将发送RTP。 RTCRtpTransceiver的RTCRtpReceiver不会提供去接收RTP，也不会接收RTP。
recvonly	RTCRtpTransceiver的RTCRtpSender不会提供去发送RTP，也不会发送RTP。 RTCRtpTransceiver的RTCRtpReceiver将提供去接收RTP，并在对端接受时接收RTP。
inactive	RTCRtpTransceiver的RTCRtpSender不会提供去发送RTP，也不会发送RTP。 RTCRtpTransceiver的RTCRtpReceiver不会提供去接收RTP，也不会接收RTP。

5.1.1 处理远程 MediaStreamTracks

应用程序可以通过调用RTCRtpTransceiver.stop()停止收发direction来拒绝传入的媒体描述，或者将收发器的direction设置为“sendonly”以仅拒绝对端接入。

要在给定RTCRtpTransceiver收发器和trackEventInits的情况下为传入媒体描述[JSEP]（第5.10节）添加remote track，用户代理必须执行以下步骤：

1. 让receiver成为收发器的[[receiver]]。
2. 让track成为receiver的[[ReceiverTrack]]。
3. 让streams成为receiver的[[AssociatedRemoteMediaStreams]]。
4. 创建一个新的RTCTrackEventInit字典，其中包含receiver，track，streams和收发器作为成员，并将其添加到trackEventInits。

要在给定RTCRtpTransceiver收发器和muteTracks的情况下处理传入媒体描述[JSEP]（第5.10节）的remote track，用户代理必须执行以下步骤：

1. 让receiver成为收发器的[[Receiver]]。
2. 让track成为receiver的[[ReceiverTrack]]。
3. 如果track.muted为false，则将track添加到muteTracks。

要在给定RTCRtpReceiver类型的receiver，msids，addList和removeList的情况下设置关联的remote streams，用户代理必须执行以下步骤：

1. 让connection成为与receiver关联的RTCPeerConnection对象。
2. 对于msids中的每个MSID都创建MediaStream对象，除非先前已使用该连接的id创建了MediaStream对象。
3. 让stream成为上一步创建的MediaStream对象的列表。
4. 让track成为receiver的[[ReceiverTrack]]。
5. 对于streams中不存在于receiver的[[AssociatedRemoteMediaStreams]]中的每个stream，将其和track作为一对添加到removeList中。
6. 对于streams中不存在于receiver的[[AssociatedRemoteMediaStreams]]中的每个stream，将其和track作为一对添加到addList。
7. 将receiver的[[AssociatedRemoteMediaStreams]]值设置为streams。

5.2 `RTCRtpSender` Interface

`RTCRtpSender` 接口允许应用程序控制给定 `MediaStreamTrack` 的编码方式并将其传输到远程对等方。在 `RTCRtpSender` 对象上调用 `setParameters` 时，编码会相应更改。

要使用 `MediaStreamTrack` 创建**RTCRtpSender**,跟踪，字符串，种类， `MediaStream` 对象列表，流以及可选的 `RTCRtpEncodingParameters` 对象列表，**sendEncodings**,请运行以下步骤：

1. 让sender成为新的 `RTCRtpSender` 对象。
2. 让sender具有初始化为track的**[[SenderTrack]]**内部插槽。
3. 让sender具有初始化为null的**[[SenderTransport]]**内部插槽。
4. 让sender具有初始化为null的**[[Dtmf]]**内部插槽。
5. 如果种类是 `“audio”` ，则创建一个 `RTCDTMFSender` **dtmf**并将[[Dtmf]]内部插槽设置为**dtmf**。
6. 让sender具有初始化为null的**[[SenderRtcpTransport]]**内部插槽。
7. 让sender具有一个**[[AssociatedMediaStreamsIds]]**内部插槽，表示此发件人要关联的 `MediaStream` 对象的Id列表。如[JSEP] (第5.2.1节)中所述，当发送方在SDP中表示时，使用 `[[AssociatedMediaStreamIds]]` 槽。
8. 设置sender的 `[[AssociatedMediaStreamIds]]` 为空集合。
9. 对于stream中的所有流，如果不存在则向 `[[AssociatedMediaStreamIds]]` 添加stream.id。
10. 让sender具有**[[SendEncodings]]**内部插槽，表示 `RTCRtpEncodingParameters` 字典列表。
11. 如果sendEncodings作为算法输入，并且非空，设置[[SendEncodings]]插槽为sendEncodings。否则，设置它为一个列表，包含单一 `RTCRtpEncodingParameters` ，并且 `active` 设置为 `true` 。
12. 让sender具有**[[SendCodecs]]**内部插槽，表示 `RTCRtpCodecParameters` 字典列表，并且初始化为空列表。
13. 让sender具有**[[LastReturnedParameters]]**内部插槽，它被用来匹配 `getParameters` 和 `setParameters` 操作。
14. 返回sender。

```
[Exposed=Window] interface RTCRtpSender {
    readonly        attribute MediaStreamTrack? track;
    readonly        attribute RTCDtlsTransport?  transport;
    readonly        attribute RTCDtlsTransport? rtcpTransport;
    static RTCRtpCapabilities? getCapabilities (DOMString kind);
    Promise<void>             setParameters (RTCRtpSendParameters parameters);
    RTCRtpSendParameters          getParameters ();
    Promise<void>             replaceTrack (MediaStreamTrack? withTrack);
    void                            setStreams(MediaStream... streams);
    Promise<RTCStatsReport>   getStats();
};
```

属性

MediaStreamTrack类型的 `track` ，只读，可以为null：

`track` 属性是与此 `RTCRtpSender` 对象关联的轨道。如果 `track` 结束，或者输出被禁用，即track被禁用和/或静音，则 `RTCRtpSender` 必须发送静音(音频)，黑帧(视频)，或零信息内容等效物。在视频的情况下， `RTCRtpSender` 应该每秒发送一个黑帧。如果 `track` 为null，则 `RTCRtpSender` 不发送。获取时，属性必须返回[[SenderTrack]]插槽的值。

`RTCDtlsTransport` 类型的 `transport` ，只读，可以为null：

`transport` 属性是一个传输，在此之上来自 `track` 的媒体通过RTP数据包的形式被发送。在 `RTCDtlsTransport` 对象构造之前， `transport` 属性将会为null。当使用了绑定，多个 `RTCRtpSender` 对象共享一个 `transport` 并且都会在相同的传输上发送RTP和RTCP。

获取时，属性必须返回[[SenderTransport]]插槽的值。

`RTCDtlsTransport` 类型的 `rtcpTransport` ，只读，可以为null:

`rtcpTransport` 属性是一个传输，在此之上RTCP被发送和接收。
在 `RTCDtlsTransport` 对象构造之前， `rtcpTransport` 属性将会是null。当多路RTCP复用时， `rtcpTransport` 将会为null，并且RTP和RTCP流都会进入 `transport` 描述的传输。

获取时，属性必须返回[[SenderRtcpTransport]]插槽的值。

方法

`getCapabilities` , static

`getCapabilities()` 方法返回用于发送指定媒体的系统功能的最优化视图。它不会保存任何资源，端口，或者其它状态，但旨在提供一种方法来发现浏览器的功能类型，包括可能支持的编解码器。用户代理必须支持 `"audio"` 和 `"video"` 的类型值。
如果系统没有与此类型参数值对应的功能， `getCapabilities` 返回 `null` 。

这些功能通常在设备上提供持久的跨源信息，从而增加了应用程序的指纹表面。在隐私敏感的上下文中，浏览器可能考虑缓解，例如仅报告功能的公共子集。

`setParameters`

`setParameters` 方法更新 `track` 如何编码并传输到远程对等端。

当调用 `setParameters` 方法时，用户代理必须运行以下步骤：

1. 让parameters成为方法的第一个参数。
2. 让sender成为调用 `setParameters` 的 `RTCRtpSender` 对象。
3. 让transceiver成为与sender关联的 `RTCRtpTransceiver` 对象、
4. 如果transceiver的[[Stopped]]插槽为 `true` ，返回一个promise，reject为一个新创建的 `InvalidStateError` 。
5. 如果sender的[[LastReturnedParameters]]内部插槽为 `null` ，返回一个promise，reject为新创建的 `InvalidStateError` 。
6. 通过以下步骤验证parameter：
    i. 让encodings为 `parameters.encodings` 。
    ii. 让codecs为 `parameters.codecs` 。
    iii. 让N为存储在sender的内部插槽[[SendEncodings]]中的 `RTCRtpEncodingParameters` 数量。
    iv. 如果符合以下条件任意之一，返回一个promise，reject为新创建的 `InvalidModificationError` ：

i. `encodings.length` 与N不同。
ii. encodings已经被重新排序。
iii. parameters中的任何参数被标记为只读参数(例如RID)，并且具有与相应sender的[[LastReturnedParameters]]内部插槽不同的值。注意这对transactionId同样有效。

v. 验证encodings中的每个 `scaleResolutionDownBy` 值大于等于1.0。如果任意一个值不满足条件，返回一个promise，reject为新创建的 `RangeError` 。
vi. 验证encodings中的每个 `maxFramerate` 值大于等于0.0。如果其中一个值不满足条件，返回一个承诺，reject为新创建的 `RangeError` 。

7. 让p成为一个新的promise。
8. 同时，配置媒体栈来使用parameters传输sender的[[SenderTrack]]。
   i. 如果使用parameters成功配置了媒体栈，对任务进行排序，允许下列步骤：
      i. 让sender的内部[[LastReturnedParameters]]插槽为null。
      ii. 让sender的内部[[SendEncodings]]插槽为 `parameters.encodings` 。
      iii. 解析p为undefined。
   ii. 如果配置媒体栈的时候出现了error，排序任务运行以下步骤：
      i. 如果由于硬件资源不可用引发error，reject p为新创建的 `RTCError` ,它的 `errorDetail` 被设置为"硬件编码器不可用"，并且中断这些步骤。
      ii. 如果由于硬件编码器不支持parameters引发error，reject p为新创建的 `RTCError` ,它的 `errorDetail` 被设置为"硬件编码器错误"并且中断这些步骤。
      iii. 其它错误，reject p为新创建的 `OperationError` 。
9. 返回p。

`setParameters` 不会导致SDP重新协商，只能被用于更改媒体栈在由Offer/Answer协商的信封中发送或接收的内容。 `RTCRtpSendParameters` 字典中的属性旨在不启用此属性，因此像 `cname` 之类无法更改的属性是只读的。另外，像比特率，
由 `maxBitrate` 之类的界限控制，此处用户代理需要确保没有超出 `maxBitrate` 指定的最大比特率，同时确保它满足其它例如SDP的地方指定的对比特率的限制。

`getParameters`

`getParameters()` 方法返回 `RTCRtpSender` 对象当前参数，此参数表示 `track` 是如何编码并传输到远程 `RTCRtpReceiver` 。

当调用 `getParameters` 时， `RTCRtpSendParameters` 字典构造如下：

- `transactionId` 被设置为新的独特Id,被用来将此次 `getParameters` 调用与将来可能会出现的 `setParameters` 调用匹配。
- `encodings` 被设置为[[SendEncodings]]内部插槽的值。
- `headerExtensions` 序列根据已协商发送的标头扩展名进行填充。
- `codecs` 被设置为[[SendCodecs]]内部插槽的值。
- `rtcp.cname` 被设置为关联 `RTCPeerConnection` 的CNAME。 `rtcp.reducedSize` 被设置为 `true` ，如果reduced-size RTCP已经被协商发送，否则为 `false` 。
- `degradationPreference` 被设置为传入 `setParameters` 的最后一个值，如果 `setParameters` 还未调用，则设置为默认值"balanced"。

返回的 `RTCRtpSendParameters` 字典必须存入 `RTCRtpSender` 对象的[[LastReturnedParameters]]内部插槽。

getParameters可能与setParameters并用，更改参数：

```
async function updateParameters() {
  try {
    const params = sender.getParameters();
    // ... make changes to parameters
    params.encodings[0].active = false;
    await sender.setParameters(params);
  } catch (err) {
    console.error(err);
  }
}
```

在对 `setParameters` 的完全调用后，后续对 `getParameters` 调用将会返回修改后的参数集合。

`replaceTrack`

试图将 `RTCRtpSender` 当前的 `track` 替换为指定的track，而不需要重新协商。

当 `replaceTrack` 方法被调用时，用户代理必须运行以下步骤：

1. 让sender成为 `RTCRtpSender` 对象，并在此对象上调用 `replaceTrack` 。
2. 让transceiver成为与sender关联的 `RTCRtpTransceiver` 对象。
3. 让connection成为与sender关联的 `RTCPeerConnection` 对象。
4. 让withTrack成为方法参数。
5. 如果 `withTrack` 不为null，并且 `withTrack.kind` 与transceiver的transceiver类别不同，返回一个promise，reject为新创建的 `TypeError` .
6. 返回将下列步骤加入connection的运行队列之后的结果：
    i. 如果transceiver的[[Stopped]]插槽为true，返回一个promise，reject为新创建的 `InvalidStateError` 。
    ii. 让p成为新的promise。
    iii. 让sending为 `true` ，如果transceiver的[[CurrentDirection]]是 `"sendrecv"` 或 `"sendonly"` ，否则为 `false` 。
    iv. 并行运行以下步骤：
        i. 如果sending为 `true` ，并且withTrack为 `null` ，使sender停止发送。
        ii. 如果sending为 `true` ，withTrack不为 `null` ，确定withTrack是否可以由发送方立即发送而不违反发送方协商好的信封，如果不能，则reject p为新创建的 `InvalidModificationError` ，并中断这些步骤。
        iii. 如果sending为 `true` ，withTrack不为 `null` ，让发送方无缝切换到传输withTrack，而不是sender现存的track。
        iv. 对任务进行排序，运行以下步骤：
            i. 如果connection的[[IsClosed]]插槽为 `true` ，中断下列步骤。
            ii. 设置sender的 `track` 属性为withTrack。
            iii. 用 `undefined` 解析p。
    v. 返回p。

> NOTE:改变尺寸和/或帧率可能不需要协商。可能需要协商的情形包括:
>
> 1. 将分辨率改为超出协商好的imageattr边界的值，如[RFC6236]中所述。
> 2. 将帧率改为导致编码器block rate超出的值。
> 3. 视频轨道与原始格式和预编码格式不同。
> 4. 音轨具有不同数量的通道数。
> 5. 同样编码的源(通常是硬件编码器)可能无法生成协商的编解码器；类似的，软件源可能不会实现为编码源协商的编解码器。

`setStreams`

设置 `MediaStreams` 与sender的track相关联。

当 `setStreams` 方法被调用后，用户代理必须运行以下步骤：

1. 让sender成为调用方法的 `RTCRtpSender` 对象。
2. 让connection成为调用方法的 `RTCPeerConnection` 对象。
3. 如果connection的[[IsClosed]]插槽为 `true` ，抛出 `InvalidStateError` 。
4. 让streams成为通过方法参数构建的MediaStream对象列表，如果方法没有传入参数被调用，则设置为空列表。
5. 设置sender的[[AssociatedMediaStreamIds]]为空集合。
6. 对于streams中的流，如果未存在则向[[AssocaitedMediaStreamIds]]中添加stream.id。
7. 更新连接的negotiation-needed 标记。

`getStats`

仅收集发送方的统计信息，并异步报告结果。

当调用 `getStats()` 后，用户代理必须运行以下步骤：

1. 让selector成为调用方法的 `RTCRtpSender` 对象。
2. 让p成为新的promise，并行运行以下步骤：
    i. 根据统计选择算法收集selector指定的统计信息。
    ii. 使用 `RTCStatsReport` 对象解析p，包含收集到的统计信息。
3. 返回p。

5.2.1 `RTCRtpParameters` 字典

```
dictionary RTCRtpParameters {
            required sequence<RTCRtpHeaderExtensionParameters> headerExtensions;
            required RTCRtcpParameters                         rtcp;
            required sequence<RTCRtpCodecParameters>           codecs;
};
```

字典 `RTCRtpParameters` 成员

序列类型的headerExtensions:

包含RTP标头扩展参数的序列。只读参数。

`RTCRtcpParameters` 类型的 `rtcp`：

用于RTCP的参数。只读参数。

序列 `<RTCRtpCodeParameters>` 类型的 `codecs`：

包含 `RTCRtpSender` 将选择的媒体编码器序列，以及RTX，RED和FEC机制的条目。对应于通过RTX进行重新传输的每个媒体编解码器，在 `codecs[]` 中将有一个条目，其中 `mimeType` 属性表示通过"audio/rtx"或"video/rtx"重新传输，以及 `sdpFmtpLine` 属性(提供"apt"和"rtx-time"参数)。只读参数。

5.2.10 `RTCRtpHeaderExtensionParameters` 字典

```
dictionary RTCRtpHeaderExtensionParameters {
             required DOMString      uri;
             required unsigned short id;
             boolean                 encrypted = false;
};
```

字典 `RTCRtpHeaderExtensionParameters` 成员

DOMString类型的 `uri` ：

RTP标头扩展的URI，[RFC5285]中定义。只读参数。

unsigned short类型的 `id` ：

放入RTP数据包用于验证标头扩展的值。只读参数。

Boolean类型的 `encrypted` ：

标头是否加密。只读参数。

> NOTE
>
> `RTCRtpHeaderExtensionParameters` 字典启用应用程序来确定是否配置了标头扩展以在 `RTCRtpSender` 或 `RTCRtpReceiver` 中使用。对于一个 `RTCRtpTransceiver` 收发器，应用程序可以在不解析SDP的情况下确定标头扩展的"direction"参数([RFC5285]第五章定义)，如下：
>
> 1. sendonly:标头扩展只能包含在transceiver.sender.getParameters().headerExtensions中。
> 2. recvonly:标头扩展只能包含在transceiver.receiver.getParameters().headerExtensions中。
> 3. sendrecv:标头扩展可以包含在transceiver.sender.getParameters().headerExtensions和transceiver.receiver.getParameters().headerExtensions中。
> 4. inactive:标头扩展既不包含在transceiver.sender.getParameters().headerExtensions中也不包含在transceiver.receiver.getParameters().headerExtensions中。

5.2.11 `RTCRtpCodecParameters` 字典

```
dictionary RTCRtpCodecParameters {
             required octet          payloadType;
             required DOMString      mimeType;
             required unsigned long  clockRate;
             unsigned short channels;
             DOMString      sdpFmtpLine;
};
```

字典 `RTCRtpCodecParameters` 成员

octet类型的 `payloadType`：

用于验证编解码器的RTP负载类型。只读参数。

DOMString类型的mimeType：

编解码器MIME媒体类型/子类型。有效的媒体类型和子类型都在[IANA-RTP-2]中列出。只读参数。

unsigned long类型的 `clockRate`：

编解码器的时钟速率，以赫兹为单位。只读参数。

unsigned short类型的 `channels`：

存在时，表示通道数量(mono=1, stereo=2)。只读参数。

DOMString类型的 `sdpFmtpLine`：

SDP中对应于编解码器的"a=fmtp"行中的"格式特定参数"字段(如果存在)，如[JSEP]所定义(第5.8节)。对于 `RTCRtpSender` ,这些参数来自远程描述，对于 `RTCRtpReceiver` ，它们来自本地描述。只读参数。

5.2.12 `RTCRtpCapabilities` 字典

```
dictionary RTCRtpCapabilities {
            required sequence<RTCRtpCodecCapability>           codecs;
            required sequence<RTCRtpHeaderExtensionCapability> headerExtensions;
};
```

字典 `RTCRtpCapabilities` 成员

序列 `<RTCRtpCodecCapability>` 类型的 `codecs` :

支持的媒体编解码器以及RTX,RED和FEC机制的条目。在 `codecs[]` 中只有一个条目用于通过RTX重传，并且 `sdpFmtpLine` 不存在。

序列 `<RTCRtpHeaderExtensionCapability>` 类型的 `headerExtensions` :

支持的RTP标头扩展。

5.2.13 `RTCRtpCodecCapability` 字典

```
dictionary RTCRtpCodecCapability {
             required DOMString mimeType;
             required unsigned long  clockRate;
             unsigned short channels;
             DOMString      sdpFmtpLine;
};
```

字典 `RTCRtpCodecCapability` 成员

`RTCRtpCodecCapability` 字典提供有关编解码器功能的信息。仅提供将在生成SDP请求中利用不同有效载荷类型的capability组合。例如：

1. 两个H.264/AVC编解码器，分别用于两个支持的分组模式值。
2. 具有不同始终速率的两个CN编解码器。

DOMString类型的 `mimeType`：

编解码器MIME媒体类型/子类型。[IANA-RTP-2]中列出了有效的媒体类型和子类型。

unsigned long 类型的 `clockRate`：

以赫兹为单位的编解码器的时钟速率。

unsigned short类型的 `channels`：

如果存在，表示最大通道数量(mono=1，stereo=2)。

DOMString类型的 `sdpFmtpLine`：

来自SDP中与编解码器对应的"a=fmtp"行的"格式特定参数"字段(如果存在)。

5.2.14 `RTCRtpHeaderExtensionCapability` 字典

```
dictionary RTCRtpHeaderExtensionCapability {
            DOMString uri;
};
```

字典 `RTCRtpHeaderExtensionCapability` 成员

DOMString类型的 `uri`：

[RFC5285]中定义的RTP标头扩展的URI。

5.2.2 `RTCRtpSendParameters` 字典

```
dictionary RTCRtpSendParameters : RTCRtpParameters {
             required DOMString             transactionId;
             required sequence<RTCRtpEncodingParameters>  encodings;
             RTCDegradationPreference       degradationPreference = "balanced";
             RTCPriorityType                priority = "low";
};
```

字典 `RTCRtpSendParameters` 成员

DOMString类型的 `transactionId` ：

应用的最后一组参数的唯一标识符。确保只能基于先前的getParameters调用setParameters，并且没有干预更改。只读参数。

序列类型的 `encodings` ：

包含媒体RTP编码参数的序列。

`RTCDegradationPreference` 类型的 `degradationPreference` ,默认为 `"balanced"`:

当带宽受到限制， `RTCRtpSender` 需要在降低分辨率和降低帧率之间做出选择， `degradationPreference` 指定倾向的选择。

`RTCPriorityType` 类型的 `priority` ，默认为 `"low"` :

表示 `RTCRtpSender` 的优先级，它影响 `RTCRtpSender` 对象之间的带宽分配。它在[RTCWEB-TRANSPORT]第四节中指定。用户代理可以在 `RTCRtpSender` 的编码之间自由地分配带宽。

5.2.3 `RTCRtpReceiveParameters` 字典

```
dictionary RTCRtpReceiveParameters : RTCRtpParameters {
            required sequence<RTCRtpDecodingParameters>  encodings;
};
```

字典 `RTCRtpReceiveParameters` 成员

序列类型的encodings：

包含入向媒体RTP编码信息的序列。

> (FEATURE AT RISK) ISSUE 3
>
> 对 `RTCRtpReceiveParameters` 的 `encodings` 属性的支持被标记为存在危险的特性，因为实现者没有明确的承诺。

5.2.4 `RTCRtpCodingParameters` 字典

```
dictionary RTCRtpCodingParameters {
            DOMString           rid;
};
```

字典RTCRtpCodingParameters成员

DOMString类型的 `rid`：

如果设置，RTP编码将与[JSEP] (第5.2.1节)定义的RID头扩展一起发送。RID不能通过 `setParameters` 修改。它只能在发送端的 `addTransceiver` 中设置或修改。只读参数。

5.2.5 `RTCRtpDecodingParameters` 字典

```
dictionary RTCRtpDecodingParameters : RTCRtpCodingParameters {
};
```

5.2.6 RTCRtpEncodingParameters Dictionary

```
dictionary RTCRtpEncodingParameters : RTCRtpCodingParameters {
             octet               codecPayloadType;
             RTCDtxStatus        dtx;
             boolean             active = true;
             unsigned long       ptime;
             unsigned long       maxBitrate;
             double              maxFramerate;
             double              scaleResolutionDownBy;
};
```

Dictionary RTCRtpEncodingParameters Members

codecPayloadType of type octet: zh:octet类型的codecPayloadType

References a payload type from the codecs member of RTCRtpParameters. Read-only parameter.

zh:从RTCRtpParameters的编解码器成员引用有效内容类型。只读参数。

dtx of type RTCDtxStatus: zh:RTCDtxStatus类型的dtx

This member is only used if the sender's kind is "audio". It indicates whether discontinuous transmission will be used. Setting it to disabled causes discontinuous transmission to be turned off. Setting it to enabled causes discontinuous transmission to be turned on if it was negotiated (either via a codec-specific parameter or via negotiation of the CN codec); if it was not negotiated (such as when setting voiceActivityDetection to false), then discontinuous operation will be turned off regardless of the value of dtx, and media will be sent even when silence is detected.

zh:仅当发件人的类型为“音频”时才使用此成员。它表示是否使用不连续传输。将其设置为禁用会导致关闭不连续传输。将其设置为启用会导致在协商时（通过特定于编解码器的参数或通过协商CN编解码器）打开不连续传输;如果没有协商（例如将voiceActivityDetection设置为false），则无论dtx的值如何，都将关闭不连续操作，即使检测到静音也会发送媒体。

active of type boolean, defaulting to true: zh:boolean类型的活动，默认为true

Indicates that this encoding is actively being sent. Setting it to false causes this encoding to no longer be sent. Setting it to true causes this encoding to be sent.

zh:表示正在积极发送此编码。将其设置为false会导致不再发送此编码。将其设置为true会导致发送此编码。

ptime of type unsigned long: zh:类型为unsigned long的ptime

When present, indicates the preferred duration of media represented by a packet in milliseconds for this encoding. Typically, this is only relevant for audio encoding. The user agent MUST use this duration if possible, and otherwise use the closest available duration. This value MUST take precedence over any "ptime" attribute in the remote description, whose processing is described in [JSEP] (section 5.10.). Note that the user agent MUST still respect the limit imposed by any "maxptime" attribute, as defined in [RFC4566], Section 6.

zh:如果存在，则表示此编码的数据包所代表的媒体的首选持续时间（以毫秒为单位）。通常，这仅与音频编码有关。如果可能，用户代理必须使用此持续时间，否则使用最接近的可用持续时间。该值必须优先于远程描述中的任何“ptime”属性，其处理在[JSEP]（第5.10节）中描述。请注意，用户代理必须仍然遵守任何“maxptime”属性所施加的限制，如[RFC4566]第6节中所定义。

maxBitrate of type unsigned long: zh:maxBitrate类型为unsigned long

When present, indicates the maximum bitrate that can be used to send this encoding. The encoding may also be further constrained by other limits (such as maxFramerate or per-transport or per-session bandwidth limits) below the maximum specified here. maxBitrate is computed the same way as the Transport Independent Application Specific Maximum (TIAS) bandwidth defined in [RFC3890] Section 6.2.2, which is the maximum bandwidth needed without counting IP or other transport layers like TCP or UDP.

zh:存在时，表示可用于发送此编码的最大比特率。编码还可能受到低于此处指定的最大值的其他限制（例如maxFramerate或每传输或每会话带宽限制）的限制。maxBitrate的计算方法与[RFC3890]第6.2.2节中定义的传输独立应用程序特定最大值（TIAS）带宽相同，后者是不计算IP或TCP或UDP等其他传输层所需的最大带宽。

maxFramerate of type double: zh:double类型的maxFramerate

When present, indicates the maximum framerate that can be used to send this encoding, in frames per second.

zh:存在时，表示可用于发送此编码的最大帧速率，以每秒帧数为单位。

If changed with setParameters, the new framerate takes effect after the current picture is completed; setting the max framerate to zero thus has the effect of freezing the video on the next frame.

zh: 如果使用setParameters更改，则新帧速率在当前图片完成后生效;将最大帧速率设置为零因此具有在下一帧上冻结视频的效果。

scaleResolutionDownBy of type double: zh:scaleResolutionDown类型为double

This member is only present if the sender's kind is "video". The video's resolution will be scaled down in each dimension by the given value before sending. For example, if the value is 2.0, the video will be scaled down by a factor of 2 in each dimension, resulting in sending a video of one quarter the size. If the value is 1.0, the video will not be affected. The value must be greater than or equal to 1.0. By default, the sender will not apply any scaling, (i.e., scaleResolutionDownBy will be 1.0).

zh:此会员仅在发件人的类型为“视频”时才会出现。在发送之前，视频的分辨率将按给定值按比例缩小。例如，如果值为2.0，则视频将在每个维度中按比例缩小2倍，从而发送大小为四分之一的视频。如果值为1.0，则视频不会受到影响。该值必须大于或等于1.0。默认情况下，发件人不会应用任何缩放（即scaleResolutionDownBy将为1.0）。

5.2.7 `RTCDtxStatus` 枚举

```
enum RTCDtxStatus {
         "disabled",
         "enabled"
         };
```

RTCDtxStatus枚举描述	
`disabled`	不连续传输被禁用。
`enabled`	如果协商成功，不连续传输被启用。

5.2.8 `RTCDegradationPreference` 枚举

```
enum RTCDegradationPreference {
    "maintain-framerate",
    "maintain-resolution",
    "balanced"
};
```

RTCDegradationPreference枚举描述	
`maintain-framerate`	降低分辨率以便维持帧率。
`maintain-resolution`	降低帧率以便维持分辨率。
`balanced`	在降低帧率和分辨率之间保持平衡。

5.2.9 `RTCRtpParameters` 字典

```
dictionary RTCRtcpParameters {
             DOMString cname;
             boolean   reducedSize;
};
```

字典 `RTCRtcpParameters` 成员

DOMString类型的 `cname`：

RTCP使用的规范名称(CNAME) (例如，在SDES消息中)。只读参数。

boolean类型的 `reducedSize`：

是否已配置缩小的RTCP[RFC5506] (如果为true)或[RFC3550]中指定的复合RTCP(如果为false)。只读参数。

5.3 RTCRtpReceiver 接口

`RTCRtpReceiver` 接口允许应用程序检查 `MediaStreamTrack` 的接收。

要使用字符串 `kind` 来创建 `RTCRtpReceiver`，请运行以下步骤：

1. 让接收器成为新的 `RTCRtpReceiver` 对象。
2. 让 `track` 成为一个新的MediaStreamTrack对象[GETUSERMEDIA]。轨道源是接收器提供的远程源。请注意， `track.id` 由用户代理生成，不会映射到远程端的任何轨道ID。
3. 将 `track.kind` 初始化为 `kind` 。
4. 将 `track.label` 初始化为将字符串“remote”与 `kind` 连接的结果。
5. 将 `track.readyState` 初始化为 `live` 。
6. 将 `track.muted` 初始化为 `true` 。请参阅 `MediaStreamTrack` 部分，了解在 `MediaStreamTrack` 接收媒体数据或者没有接收的情况下，静音属性如何反映。
7. 让接收器将[[ReceiverTrack]]内部插槽初始化为 `track` 。
8. 让接收器将[[ReceiverTransport]]内部插槽初始化为 `null` 。
9. 让接收器将[[ReceiverRtcpTransport]]内部插槽初始化为 `null` 。
10. 让接收器具有[[AssociatedRemoteMediaStreams]]内部槽，表示与该接收器的 `MediaStreamTrack` 对象相关联的 `MediaStream` 对象的列表，并初始化为空列表。
11. 让接收器有一个[[ReceiveCodecs]]内部插槽，表示 `RTCRtpCodecParameters` 字典列表，并初始化为空列表。
12. 返回接收器。

```
[Exposed=Window] interface RTCRtpReceiver {
    readonly        attribute MediaStreamTrack  track;
    readonly        attribute RTCDtlsTransport? transport;
    readonly        attribute RTCDtlsTransport? rtcpTransport;
    static RTCRtpCapabilities?         getCapabilities (DOMString kind);
    RTCRtpReceiveParameters            getParameters ();
    sequence<RTCRtpContributingSource>    getContributingSources ();
    sequence<RTCRtpSynchronizationSource> getSynchronizationSources ();
    Promise<RTCStatsReport>   getStats();
};
```

属性 *track* 类型 `MediaStreamTrack` ，只读 `track` 属性是与此 `RTCRtpReceiver` 对象接收器关联的轨道。

请注意， `track.stop()` 是最终的，尽管克隆不受影响。由于 `receiver.track.stop()` 不会隐式停止接收器，因此继续发送 `Receiver Reports` 。获取时，属性必须返回[[ReceiverTrack]]槽的值。

transport `RTCDtlsTransport` 类型的传输，只读，可空

传输属性是以RTP分组的形式接收接收者的轨道的媒体的传输。在构造 `RTCDtlsTransport` 对象之前，transport属性将为null。使用捆绑时，多个 `RTCRtpReceiver` 对象将共享一个传输，并将通过同一传输接收 RTP 和 RTCP。

获取时，属性必须返回[[ReceiverTransport]]槽的值。

`rtcpTransport` 类型为 `RTCDtlsTransport` ，只读，可空

`rtcpTransport` 属性是发送和接收 RTCP 的传输。在构造 `RTCDtlsTransport` 对象之前， `rtcpTransport` 属性将为 `null` 。当使用 RTCP mux（或捆绑，强制执行 RTCP mux）时， `rtcpTransport` 将为 `null` ，并且 RTP 和 RTCP 流量都将通过传输流动。

获取时，属性必须返回[[ReceiverRtcpTransport]]槽的值。

方法 `getCapabilities` ，静态 `getCapabilities（）` 方法返回系统功能的最乐观视图，用于接收给定类型的媒体。它不保留任何资源，端口或其他状态，但旨在提供一种方法来发现浏览器的功能类型，包括可能支持的编解码器。用户代理必须支持“音频”和“视频”的类型值。如果系统没有与 `kind` 参数的值相对应的功能，则 `getCapabilities` 返回 `null` 。

这些功能通常在设备上提供持久的跨源信息，从而增加了应用程序的指纹表面。在隐私敏感的上下文中，浏览器可以考虑缓解，例如仅报告功能的公共子集。

`getParameters` `getParameters（）` 方法返回 `RTCRtpReceiver` 对象的当前参数，用于解码轨道的方式。 调用 `getParameters` 时， `RTCRtpReceiveParameters` 字典构造如下： 根据当前远程描述中存在的 RID 填充编码。

- `headerExtensions` 序列基于接收器当前准备接收的头扩展来填充。
- 编解码器序列基于接收器当前准备接收的编解码器来填充。 注意本地和远程描述可能会影响此编解码器列表。例如，如果提供了三个编解码器，接收器将准备好接收它们中的每一个，并将从 `getParameters` 返回它们。但是如果远程端点只回答两个，则 `getParameters` 将不再返回缺少的编解码器，因为接收器不再需要准备接收它。
- 如果接收器当前准备接收缩小的RTCP数据包，则 `rtcp.reducedSize` 设置为true，否则设置为 `false` 。 `rtcp.cname` 被省略了。

`getContributingSources` 返回此RTCRtpReceiver在过去10秒内收到的每个唯一CSRC标识符的RTCRtpContributingSource。

`getSynchronizationSources` 返回此 `RTCRtpReceiver` 在过去10秒内收到的每个唯一SSRC 标识符的 `RTCRtpContributingSource`

`getStats` 仅收集此接收器的统计信息并异步报告结果。 当调用 `getStats（）` 方法时，用户代理必须运行以下步骤：

1. 让 `selector` 成为调用该方法的 `RTCRtpReceiver` 对象。
2. 让 `p` 成为新的承诺，并行执行以下步骤：
    i. 根据统计选择算法收集选择器指示的统计数据。
    ii. 使用生成的RTCStatsReport对象解析p，其中包含收集的统计信息。
3. 返回p。

`RTCRtpContributingSource` 和 `RTCRtpSynchronizationSource` 字典分别包含有关给定贡献源（CSRC）或同步源（SSRC）的信息，包括源所贡献的数据包的最近时间。浏览器必须保留前 10 秒内收到的 RTP 数据包的信息。当包含在 RTP 数据包中的第一个音频或视频帧被传送到 `RTCRtpReceiver` 的 `MediaStreamTrack` 进行播出时，用户代理必须排队任务，以根据数据包的内容更新 `RTCRtpContributingSource` 和 `RTCRtpSynchronizationSource` 字典的相关信息。每次更

新与 SSRC 标识符对应的 `RTCRtpSynchronizationSource` 字典相关的信息，并且如果 RTP 分组包含 CSRC 标识符，则还更新与那些 CSRC 标识符对应的 `RTCRtpContributingSource` 字典相关的信息。

NOTE 如一致性部分所述，只要最终结果是等效的，可以以任何方式实现作为算法的要求。因此，实现不需要为每个数据包逐字排队任务，只要最终结果是在单个事件循环任务执行中，所有返回的特定 `RTCRtpReceiver` 的 `RTCRtpSynchronizationSource` 和 `RTCRtpContributingSource` 字典都包含来自 RTP 流中单个点的信息。

```
dictionary RTCRtpContributingSource {
    required DOMHighResTimeStamp timestamp;
    required unsigned long       source;
             double              audioLevel;
};
```

字典 RTCRtpContributingSource 成员

timestamp 类型为DOMHighResTimeStamp， 必需 DOMHighResTimeStamp [HIGHRES-TIME]类型的时间戳，指示到达源自此源的RTP数据包的媒体播放的最近时间。时间戳在播出时定义为 `performance.timeOrigin + performance.now（）` 。

source 类型为unsigned，必需 贡献或同步源的 CSRC 或 SSRC 标识符。

audioLevel 类型为double 仅适用于音频接收器。这是 0..1（线性）之间的值，其中 1.0 表示 0 dBov，0 表示静音，0.5 表示声压级从 0 dBov变化约 6 dBSPL。 对于 CSRC，如果存在 RFC 6465 标头扩展，则必须从[RFC6465]中定义的级别值转换，否则该成员必须不存在。 对于 SSRC，如果存在 RFC 6464 标头扩展，则必须从[RFC6464]中定义的级别值转换，否则用户代理必须从音频数据计算值（音频接收器不得缺少该成员）。 两个 RFC 将级别定义为 0 到 127 的整数值，表示相对于系统可能编码的最大信号的负分贝的音频电平。因此，0 代表系统可能编码的最响亮信号，127 代表静音。 要将这些值转换为线性 0..1 范围，将值 127 转换为 0，并使用以下公式转换所有其他值： `10 ^（ - rfc_level / 20）` 。

```
dictionary RTCRtpSynchronizationSource : RTCRtpContributingSource {
    boolean voiceActivityFlag;
};
```

字典 RTCRtpSynchronizationSource 成员

voiceActivityFlag boolean类型 仅适用于音频接收器。从该源播放的最后一个RTP数据包是否包含语音活动（true）或不包含语音活动（false）。如果 RFC 6464 扩展头不存在，或者对等体通过将“vad”扩展属性设置为“off”来表示它没有使用 V bit，如[RFC6464]第 4 节中所述，则 `voiceActivityFlag` 将是缺席。

5.4 RTCRtpTransceiver 接口

`RTCRtpTransceiver` 接口表示共享公共mid的 `RTCRtpSender` 和 `RTCRtpReceiver` 的组合。如[JSEP]（第3.4.1节）中所定义的，如果 `RTCRtpTransceiver` 的 `mid` 属性为非 `null`，则称其与媒体描述相关联;否则据说是不相关的。从概念上讲，相关联的收发器是在最后应用的会话描述中表示的收发器。 `RTCRtpTransceiver` 的收发器类型由关联的 `RTCRtpReceiver` 的 `MediaStreamTrack` 对象的类型定义。 要使用 `RTCRtpReceiver` 对象，接收方， `RTCRtpSender` 对象，发送方和 `RTCRtpTransceiverDirection` 值，方向创建 `RTCRtpTransceiver`，请运行以下步骤：

1. 让收发器成为新的 `RTCRtpTransceiver` 对象。
2. 让收发器有一个[[Sender]]内部插槽，初始化为发送方。
3. 让收发器有一个[[Receiver]]内部插槽，初始化为接收器。
4. 让收发器有一个[[Stopped]]内部插槽，初始化为 `false` 。
5. 让收发器有一个[[Direction]]内部插槽，初始化为方向。
6. 让收发器有一个[[Receptive]]内部插槽，初始化为 `false` 。
7. 让收发器具有[[CurrentDirection]]内部插槽，初始化为 `null` 。
8. 让收发器有一个[[FiredDirection]]内部插槽，初始化为 `null` 。
9. 让收发器具有[[PreferredCodecs]]内部插槽，初始化为空列表。
10. 返回收发器。

> NOTE
>
> 创建收发器不会创建基于 `RTCDtlsTransport` 和 `RTCIceTransport` 的对象。这只会在设置 `RTCSessionDescription` 的过程中发生。

```
[Exposed=Window] interface RTCRtpTransceiver {
    readonly        attribute DOMString?                mid;
    [SameObject]
    readonly        attribute RTCRtpSender              sender;
    [SameObject]
    readonly        attribute RTCRtpReceiver            receiver;
    readonly        attribute boolean                   stopped;
                    attribute RTCRtpTransceiverDirection  direction;
    readonly        attribute RTCRtpTransceiverDirection? currentDirection;
    void stop ();
    void setCodecPreferences (sequence<RTCRtpCodecCapability> codecs);
};
```

属性

mid DOMString类型，只读，可以为空 `mid` 属性是中间协商的，并且存在于[JSEP]（第 5.2.1 节和第 5.3.1 节）中定义的本地和远程描述中。在协商完成之前，中间值可能为空。回滚后，该值可能会从非 `null` 值更改为 `null` 。

sender RTCRtpSender类型，只读 `sender` 属性公开了可能以 `mid = mid` 发送的RTP 媒体相对应的 `RTCRtpSender` 。获取时，属性必须返回[[Sender]]槽的值。

receiver RTCRtpReceiver类型，只读 接收器属性是对应于可以用 `mid = mid` 接收的RTP媒体的 `RTCRtpReceiver` 。获取属性时必须返回[[Receiver]]插槽的值。

stopped 类型boolean，只读 `stopped` 属性表示此收发器的发送方将不再发送，接收方将不再接收。如果已调用 `stop` 或设置本地或远程描述导致 `RTCRtpTransceiver` 停止，则为 `true` 。获取时，此属性必须返回[[Stopped]]插槽的值。

direction RTCRtpTransceiverDirection类型 如[JSEP]（第 4.2.4 节）中所定义， `direction` 属性指示此收发器的首选方向，该方向将用于调用 `createOffer`和 `createAnswer` 。方向性更新不会立即生效。相反，将来调用 `createOffer` 和 `createAnswer` 会将相应的媒体描述标记为 `sendSearchv` ， `sendonly` ， `recvonly` 或 `inactive` ，如[JSEP]中所定义（第 5.2.2 节和第 5.3.2 节）。 获取时，此属性必须返回[[Direction]]插槽的值。 在设置时，用户代理必须执行以下步骤：

1. 让收发器成为调用设置者的 `RTCRtpTransceiver` 对象。
2. 让连接成为与收发器关联的 `RTCPeerConnection` 对象。
3. 如果连接的[[IsClosed]]槽为 `true` ，则抛出 `InvalidStateError` 。
4. 如果收发器的[[Stopped]]插槽为 `true` ，则抛出 `InvalidStateError` 。
5. 让 `newDirection` 成为设置者的参数。
6. 如果 `newDirection` 等于收发器的[[Direction]]插槽，则中止这些步骤。
7. 将收发器的[[Direction]]插槽设置为 `newDirection` 。
8. 更新连接所需的协商标志。

currentDirection 类型为RTCRtpTransceiverDirection，只读，可为空 如[JSEP]（第4.2.5节）中所定义， `currentDirection` 属性指示为此收发器协商的当前方向。 `currentDirection` 的值独立于 `RTCRtpEncodingParameters.active` 的值，因为无法从另一个推导出。如果此收发器从未在提供/应答交换中表示，或者收发器已停止，则该值为空。获取时，该属性必须返回[[CurrentDirection]]槽的值。

方法

`stop` 不可逆转地停止 `RTCRtpTransceiver` 。此收发器的发送器将不再发送，接收器将不再接收。调用 `stop（）` 会更新 `RTCRtpTransceiver` 关联的RTCPeerConnection的需要协商标志。 停止收发器将导致将来调用 `createOffer` 或 `createAnswer` 以在相应收发器的媒体描述中生成零端口，如[JSEP]（第 4.2.1 节）中所定义。

> NOTE
>
> 如果在应用远程报价和创建答案之间调用此方法，并且收发器与[BUNDLE]中定义的“提供者标记”媒体描述相关联，则这将导致捆绑组中的所有其他收发器停止为好。为避免这种情况，当 `signalingState` 为“稳定”并执行后续的提议/应答交换时，可以只停止当前收发器。

当调用stop方法时，用户代理必须运行以下步骤：

1. 让收发器成为调用该方法的 `RTCRtpTransceiver` 对象。
2. 让连接成为与收发器关联的 `RTCPeerConnection` 对象。
3. 如果连接的[[IsClosed]]槽为 `true` ，则抛出 `InvalidStateError` 。
4. 如果收发器的[[已停止]]插槽为 `true` ，则中止这些步骤。
5. 停止收发器指定的 `RTCRtpTransceiver` 。
6. 更新连接所需的协商标志。

给定收发器的 `RTCRtpTransceiver` 停止算法如下：

1. 让发送者成为收发器的[[发件人]]。

2. 让接收器成为收发器的[[接收器]]。
3. 停止向发件人发送媒体。
4. 按照[RFC3550]中的规定，为发送方发送的每个 RTP 流发送 RTCP BYE。
5. 停止使用接收器接收媒体。
6. 执行接收方[[ReceiverTrack]]的结束步骤。
7. 将收发器的[[Stopped]]插槽设置为 `true` 。
8. 将收发器的[[Receptive]]插槽设置为 `false` 。
9. 将收发器的[[CurrentDirection]]插槽设置为空。

`setCodecPreferences` `setCodecPreferences` 方法会覆盖用户代理使用的默认编解码器首选项。当使用 `createOffer` 或 `createAnswer` 生成会话描述时，用户代理必须按照编解码参数中指定的顺序使用指示的编解码器，用于与此 `RTCRtpTransceiver` 对应的媒体部分。 此方法允许应用程序禁用特定编解码器的协商。它还允许应用程序使远程对等方优先选择列表中首先出现的编解码器。 对于包含此 `RTCRtpTransceiver` 的 `createOffer` 和 `createAnswer` 的所有调用，编解码器首选项仍然有效，直到再次调用此方法。将编解码器设置为空序列会将编解码器首选项重置为任何默认值。 传递给 `setCodecPreferences` 的编解码器序列只能包含由 `RTCRtpSender.getCapabilities（kind）` 或 `RTCRtpReceiver.getCapabilities（kind）` 返回的编解码器，其中 `kind` 是调用该方法的 `RTCRtpTransceiver` 的类型。此外，不能修改 `RTCRtpCodecCapability` 字典成员。如果编解码器不满足这些要求，则用户代理必须抛出 `InvalidAccessError` 。

> NOTE
>
> 由于[SDP]中的建议，对 `createAnswer` 的调用应该只使用编解码器首选项的公共子集和商品中出现的编解码器。例如，如果编解码器首选项是“C，B，A”，但仅提供编解码器“A，B”，则答案应仅包含编解码器“B，A”。但是，[JSEP]（第 5.3.1 节）允许添加不在商品中的编解码器，因此实现的行为可能不同。

当调用 `setCodecPreferences（）` 时，用户代理必须运行以下步骤：

1. 让收发器成为调用此方法的 `RTCRtpTransceiver` 对象。
2. 让编解码器成为第一个参数。
3. 如果编解码器是空列表，请将收发器的[[PreferredCodecs]]插槽设置为编解码器并中止这些步骤。
4. 删除编解码器中的任何重复值。从列表的后面开始，以保持编解码器的优先级;列表中第一次出现编解码器的索引在此步骤之前和之后是相同的。
5. 让我们成为收发器的收发器类型。
6. 如果编解码器和 `RTCRtpSender.getCapabilities（kind）.codecs` 之间的交集或编解码器与 `RTCRtpReceiver.getCapabilities（kind）.codecs` 之间的交集是一个空集，则抛出 `InvalidModificationError` 。这确保了无论收发器方向如何，我们总能提供一些东西。
7. 让 `codecCapabilities` 成为 `RTCRtpSender.getCapabilities（kind）.codecs` 和 `RTCRtpReceiver.getCapabilities（kind）.codecs` 的联合。
8. 对于编解码器中的每个编解码器，
    i. 如果编解码器不在 `codecCapabilities` 中，则抛出 `InvalidModificationError` 。
9. 将收发器的[[PreferredCodecs]]插槽设置为编解码器。

5.4.1 联播功能

通过 `RTCPeerConnection` 对象的 `addTransceiver` 方法和 `RTCRtpSender` 对象的 `setParameters` 方法提供联播功能。 `addTransceiver` 方法建立联播信封，其中包括可以发送的最大联播流数量，以及编码的顺序。虽然可以使用 `setParameters` 方法修改单个联播流的特征，但不能更改联播信封。此模型的一个含义是 `addTrack` 方法无法提供联播功能，因为它不将 `sendEncodings` 作为参数，因此无法配置 `RTCRtpTransceiver` 来发送联播。 虽然 `setParameters` 无法修改联播信封，但仍可以控制发送的流的数量和这些流的特征。使用 `setParameters` ，可以通过将 `active` 属性设置为 `false` 来使联播流处于非活动状态，或者可以通过将 `active` 属性设置为 `true` 来重新激活联播流。使用 `setParameters` ，可以通过修改 `maxBitrate` 和 `maxFramerate` 等属性来更改流特性。

此规范未定义如何配置 `createOffer` 以接收多个 RTP 编码。但是，当使用能够发送[JSEP]中定义的多个 RTP 编码的相应远程描述调用 `setRemoteDescription` 时， `RTCRtpReceiver` 可以接收多个 RTP 编码，并且通过收发器的 `receiver.getParameters()` 检索的参数将反映协商的编码。

> NOTE
>
> 在选择性转发单元（SFU）在从用户代理接收的联播流之间切换的情况下， `RTCRtpReceiver` 可以接收多个RTP流。如果 SFU 不重写 RTP 报头以便在转发之前将切换流安排到单个 RTP 流中，则 RTCRtpReceiver 将接收来自不同 RTP 流的分组，每个 RTP 流具有其自己的 SSRC 和序列号空间。虽然 SFU 可能仅在任何给定时间转发单个 RTP 流，但是由于重新排序，来自多个 RTP 流的分组可能在接收器处混合。因此，配备用于接收多个 RTP 流的 RTCRtpReceiver 将需要能够正确地对接收的分组进行排序，识别潜在的丢失事件并对它们作出反应。在这种情况下的正确操作是非常重要的，因此对于本规范的实现是可选的。

5.4.1.1 编码参数示例

本节不具有规范性。

使用编码参数实现的联播场景示例：

```
EXAMPLE 4
// Example of 3-layer spatial simulcast with all but the lowest resolution layer disab
var encodings = [
  {rid: 'f', active: false},
  {rid: 'h', active: false, scaleResolutionDownBy: 2.0},
  {rid: 'q', active: true, scaleResolutionDownBy: 4.0}
];

// Example of 3-layer framerate simulcast with the middle layer disabled
var encodings = [
  {rid: 'f', active: true, maxFramerate: 60},
  {rid: 'h', active: false, maxFramerate: 30},
  {rid: 'q', active: true, maxFramerate: 15}
];
```

5.5 RTCDtlsTransport接口

`RTCDtlsTransport` 接口允许应用程序获取数据报传输层安全性协议(DTLS)传输的信息，通过此协议，RTP与RTCP数据包被 `RTCRtpSender` 和 `RTCRtpReceiver` 对象发送和接收，还有数据通道发送和接收的其它数据，例如STCP数据包。特别是DTLS为底层传输添加了安全性，并且 `RTCDtlsTransport` 接口允许获取底层传输和安全性的信息。因为需要调用 `setLocalDescription()` 和 `setRemoteDescription()` 函数， `RTCDtlsTransport` 对象被创建。每个 `RTCDtlsTransport` 对象代表一个特定的 `RTCRtpTransceiver` 的RTP或RTCP组件的DTLS传输层，或是一组已经通过BUNDLE协商好的 `RTCRtpTransceivers` 。

> NOTE:现存RTCRtpTransceiver的一个新DTLS关联将会由一个现存RTCDtlsTransport对象代表，它的状态将会对应更新，而不是换做一个新的对象来表示。

一个 `RTCDtlsTransport` 有一个被初始化为 `new` 的[[DtlsTransportState]]内部插槽，和一个被初始化为空列表的[[RemoteCertificates]]的插槽。

当底层DTLS传输产生错误时，例如证书失效，或错误警报，用户代理必须对运行下列步骤的任务排序：

1. 让transport成为 `RTCDtlsTransport` 对象，用来接收状态更新和错误提示。
2. 如果transport的状态已经为failed，中断这些步骤。
3. 设置transport的[[DtlsTransportState]]插槽为 `failed` 。
4. 使用RTCErrorEvent接口和它的或者为dtlsfailure，或者为fingerprint-failure的errorDetail属性，发起一个名为error的事件，并且其它fields都按照RTCErrorDetailType枚举的那样恰当设置，在transport。
5. 在transport发起一个名为statechange的事件。

当底层DTLS传输由于任何其它原因需要更新相应RTCDtlsTransport对象的状态时，用户代理必须对运行下列步骤的任务排序：

1. 让transport成为RTCDtlsTransport对象来接收状态更新。
2. 让newState成为新状态。
3. 设置transport的[[DtlsTransportState]]插槽为newState。
4. 如果newState连接，那么让newRemoteCertificates成为远端使用的证书链，每个证书都使用二进制可辨别编码规则(DER)[X690]进行编码，饼设置transport的[[RemoteCertificates]]插槽为newRemoteCertificates。
5. 在transport发起一个名为statechange的事件。

```
[Exposed=Window] interface RTCDtlsTransport : EventTarget {
    [SameObject]
    readonly        attribute RTCIceTransport       iceTransport;
    readonly        attribute RTCDtlsTransportState state;
    sequence<ArrayBuffer> getRemoteCertificates ();
                    attribute EventHandler          onstatechange;
                    attribute EventHandler          onerror;
};
```

属性

`iceTransport` of type `RTCIceTransport`, readonly: iceTransport属性是用来发送接收数据包的底层传输。底层传输在多个活跃的RTCDtlsTransport对象间可能不会被共享。

`state` of type `RTCDtlsTransportState`, readonly: 当需要state属性时，它必须返回[[DtlsTransportState]]插槽的值。

`onstatechange` of type `EventHandler`: 此eventhandler的事件类型是statechange。

`onerror` of type `EventHandler`: 此eventhandler的事件类型是error。

方法

`getRemoteCertificates` 返回[[RemoteCertificates]]的值。

`RTCDtlsTransportState` 枚举

```
enum RTCDtlsTransportState {
    "new",
    "connecting",
    "connected",
    "closed",
    "failed"
};
```

枚举描述	
`new`	DTLS还没有开始协商。
`connecing`	DTLS正在协商一个安全连接，并验证远端指纹。
`connected` 测试：1	DTLS已经完成安全连接的协商，并已经确认远端指纹。
`closed` 测试：1	transport已经关闭，由于收到close_notify警报，或调用close()。
`failed`	由于产生了错误，transport已经失败(例如接收到错误警报或未能验证远端指纹)。

5.5.1 RTCDtlsFingerprint字典

`RTCDtlsFingerprint` 字典包含[RFC4572]中描述的哈希函数算法和证书指纹。

```
dictionary RTCDtlsFingerprint {
             DOMString algorithm;
             DOMString value;
};
```

字典RTCDtlsFingerprint成员

DOMString类型的 `算法` :它是哈希函数文本名称注册表[IANA-HASH-FUNCTION]中定义的一个哈希函数算法。

DOMString类型的 `值` :使用[RFC4572]第五节中“指纹”语法表示的小写十六进制字符串中的指纹证书的值。

5.6 RTCIceTransport 接口

RTCIceTransport 接口允许应用程序获取有关用来发送接收数据包的ICE传输的信息。特别的，ICE管理对等连接，它牵涉到应用程序可能获取的状态。由于需要调用 setLocalDescription() 和 setRemoteDescription() ，RTCIceTransport 对象被创建。底层ICE状态由ICE代理管理，所以当ICE代理向用户代理提供指示时， RTCIceTransport 的状态就会改变。每个 RTCIceTransport 对象代表一个特定 RTCRtpTransceiver 的RTP或RTCP组件的ICE传输层，或者是一组已经通过[BUNDLE]协商的 RTCRtpTransceiver 。

> NOTE:现存 RTCRtpTranceiver 的ICE重启将会由现存 RTCIceTransport 对象表示，它的状态将会被对应更新，而不是由新对象表示。

当ICE代理表示它将为一个 RTCIceTransport 收集一系列的候选者时，用户代理必须对运行以下步骤的任务进行排序:

1. 让connection成为与这个ICE代理关联的 RTCPeerConnection 对象。
2. 如果connection的[[IsClosed]]插槽为 true ，中断这些步骤。
3. 让transport成为RTCIceTransport。
4. 设置transport的[[IceGathererState]]插槽为 gathering 。
5. 在transport发起一个名为 gatheringstatechange 的事件。
6. 更新connection的ICE收集状态。

当ICE代理表示它已经成功收集了一个 RTCIceTransport 的一系列候选者时，用户代理必须对运行以下步骤的任务进行排序：

1. 让connection成为与ICE代理关联的 RTCPeerConnection 对象。
2. 如果connection的[[IsClosed]]插槽为 true ，中断这些步骤。
3. 让transport成为 RTCIceTransport 。
4. 让 newCandidate 成为创建一个RTCIceCandidate的结果，伴随一个新字典，它的 sdpMid 和 sdpMLineIndex 都被设置为与此 RTCIceTransport 相关联的值， usernameFrangment 被设置为收集完成的candidates的用户名片段， candidate 被设置为空字符串。
5. 使用 RTCPeerConnectionIceEvent 接口发起一个名为 icecandidate 的事件，并且candidate属性在connection被设置为 newCandidate 。
6. 如果另一代candidates还在被收集，中断这些步骤。

   > NOTE:这可能会出现，如果在ICE代理还在收集上一代的candidates时，开始ICE重启，可能会发生这种情况。

7. 设置transport的[[IceGathererState]]插槽为complete。
8. 在transport发起一个名为 gatheringstatechange 的事件。
9. 更新connection的ICE收集状态。

当用户代理表示新的ICE candidate可用于 RTCIceTransport ，或者从ICE候选者池中选择一个，或者从头开始收集它，用户代理必须对运行以下步骤的任务进行排序：

1. 让 candidate 成为可用的ICE候选者。

2. 让connection成为与这个ICE代理相关联的 `RTCPeerConnection` 对象。
3. 如果connection的[[IsClosed]]插槽为 `true` ，中断这些步骤。
4. 让transport成为可供候选者使用的 `RTCIceTransport` 。
5. 如果connection.[[PendingLocalDescription]]不是 `null` ，并且表示收集候选者的ICE generation，向connection.[[PendingLocalDesciption]].sdp中添加候选者。
6. 如果connection.[[CurrentLocalDescription]]不是 `null` ，并且表示收集候选者的ICE generation，向connection.[[CurrentLocalDescription]].sdp中添加候选者。
7. 让 `newCandidate` 成为创建一个RTCIceCandidate的结果，伴随一个新字典，它的 `sdpMid` 和 `sdpMLineIndex` 都被设置为与此 `RTCIceTransport` 相关联的值， `usernameFrangment` 被设置为候选者的用户名片段，并且 `candidate` 被设置为使用 `candidate-attribute` 语法编码的字符串来代表 `candidate` 。
8. 将 `newCandidate` 添加到transport的本地候选者集合中。
9. 使用 `RTCPeerConnectionIceEvent` 接口发起一个名为 `icecandidate` 的事件，并且候选者属性在 `connection` 被设置为 `newCandidate` 。

当ICE代理表示 `RTCIceTransport` 的 `RTCIceTransportState` 已经改变时，用户代理必须对运行以下步骤的任务进行排序:

1. 让 `connection` 成为与这个ICE代理相关联的 `RTCPeerConnection` 对象。
2. 如果 `connection` 的[[IsClosed]]插槽为 `true` ，中断这些步骤。
3. 让 `transport` 成为状态正在改变的 `RTCIceTransport` 。
4. 让 `newState` 成为新的被指示的 `RTCIceTransportState` 。
5. 设置 `transport` 的[[IceTransportState]]插槽为 `newState` 。
6. 在 `transport` 发起一个名为 `statechange` 的事件。
7. 更新 `connection` 的ICE连接状态。
8. 更新 `connection` 的连接状态。

当ICE代理表示选定的一对 `RTCIceTransport` 候选者已经改变时，用户代理必须对运行以下步骤的任务进行排序：

1. 让 `connection` 成为与这个ICE代理相关联的 `RTCPeerConnection` 对象。
2. 如果 `connection` 的[[IsClosed]]插槽为 `true` ，中断这些步骤。
3. 让 `transport` 成为选定候选者对正在改变的 `RTCIceTransport` 。
4. 让 `newCandidatePair` 成为新常见的 `RTCIceCandidatePair` ，如果选定了一个，表示选择正确，否则为null。
5. 设置 `transport` 的[[SelectedCandidatePair]]插槽为 `newCandidatePair` 。
6. 在 `transport` 发起一个名为 `selectedcandidatepairchange` 的事件。

一个 `RTCIceTransport` 对象具有下列内部插槽：

- [[IceTransportState]]初始化为 `new`
- [[IceGathererState]]初始化为 `new`
- [[SelectedCandidatePair]]初始化为 `null`

```
[Exposed=Window] interface RTCIceTransport : EventTarget {
    readonly        attribute RTCIceRole           role;
    readonly        attribute RTCIceComponent      component;
    readonly        attribute RTCIceTransportState state;
    readonly        attribute RTCIceGathererState gatheringState;
    sequence<RTCIceCandidate> getLocalCandidates ();
    sequence<RTCIceCandidate> getRemoteCandidates ();
    RTCIceCandidatePair?      getSelectedCandidatePair ();
    RTCIceParameters?         getLocalParameters ();
    RTCIceParameters?         getRemoteParameters ();
                    attribute EventHandler         onstatechange;
                    attribute EventHandler         ongatheringstatechange;
                    attribute EventHandler         onselectedcandidatepairchange;
};
```

属性

`RTCIceRole` 类型的 `role`，只读： `role` 属性必须返回transport的ICE role。

`RTCIceComponent` 类型的 `component`，只读： `component` 属性必须返回 `transport` 的ICE组件。当RTCP mux被使用时，单一的 `RTCIceTransport` 同时传输RTP和RTCP，并且 `component` 被设置为 `RTP` 。

`RTCIceTransportState` 类型的 `state`，只读：当需要获得 `state` 属性时，它必须返回[[IceTransportState]]插槽的值。

`RTCIceGathererState` 类型的 `gatheringState`，只读：当获取 `gathering state` 属性时，它必须返回[[IceGathererState]]插槽的值。

`EventHandler` 类型的 `onstatechange` :此event handler，当 `RTCIceTransportstate` 类型改变时，必须启动。

`EventHandler` 类型的 `ongatheringstatechange` ：此event handler，当 `RTCIceTransportgatheringstate` 改变时，必须启动。

`EventHandler` 类型的 `onselectedcandidatepairchange` ：此event handler，当 `RTCIceTransport` 选定的候选者对改变时，必须启动。

方法

`getLocalCandidates` :返回一个序列，描述为 `RTCIceTransport` 收集并在 `onicecandidate` 中发送的本地候选者。

`getRemoteCandidates` :返回一个序列，描述通过 `addIceCandidate()` ，由 `RTCIceTransport` 接收的ICE候选者。

> NOTE: `getRemoteCandidates` 不会暴露peer reflexive candidates，因为它们不是通过 `addIceCandidate()` 接收的。

`getSelectedCandidatePair` :返回用来发送数据包的选定候选者对。此方法必须返回[[SelectedCandidatePair]]插槽的值。

`getLocalParameters` :返回通过 `setLocalDescription` 由 `RTCIceTransport` 接收的本地ICE参数，如果参数未被接收，则为 `null` 。

`getRemoteParameters` :返回通过 `setRemoteDescription` ，由 `RTCIceTransport` 接收的ICE远程参数，如果参数未被接收，则为 `null` 。

5.6.1 `RTCIceParameters` 字典

```
dictionary RTCIceParameters {
             DOMString usernameFragment;
             DOMString password;
};
```

字典 `RTCIceParameters` 成员

`DOMString` 类型的 `usernameFragment` :在[ICE],章节7.1.2.3中定义的ICE用户名片段。

`DOMString` 类型的 `password` :在[ICE],章节7.1.2.3中定义的ICE密码。

5.6.2 `RTCIceCandidatePair` 字典

```
dictionary RTCIceCandidatePair {
             RTCIceCandidate local;
             RTCIceCandidate remote;
};
```

字典 `RTCIceCandidatePair` 成员

`RTCIceCandidate` 类型的 `local` ： 本地ICE 候选者。

`RTCIceCandidate` 类型的 `remote` ： 远程ICE 候选者。

5.6.3 `RTCIceGathererState` 枚举

```
enum RTCIceGathererState {
    "new",
    "gathering",
    "complete"
};
```

RTCIceGathererState 枚举描述	
`new`	`RTCIceTransport` 刚刚被创建，还未开始收集候选者。
`gathering`	`RTCIceTransport` 正在收集候选者。
`complete`	`RTCIceTransport` 已经完成收集，并已发送此传输的候选者终止指示。在ICE重启导致它重启之前，它不会再次收集候选者。

5.6.4 RTCIceTransportState枚举

```
enum RTCIceTransportState {
    "new",
    "checking",
    "connected",
    "completed",
    "disconnected",
    "failed",
    "closed"
};
```

`RTCIceTransportState` 枚举描述	
`new`	`RTCIceTransport` 正在收集候选者并/或等待被提供远程候选者，并且还未开始校验。
`checking`	`RTCIceTransport` 已经接收至少一个远程候选者并且正在校验候选者对，或是未发现连接，或是对之前成功的候选者对的校验已经失败。除此之外，它还可能继续收集。
`connected`	`RTCIceTransport` 已经发现一个可用的连接，但是仍在校验其它候选者对试图找到一个更好的连接。它还可能继续收集并且/或等待另外的远程候选者。如果对正在使用的连接的consent check[RFC7675]失败，并且不存在其它成功的候选者对，那么状态转变为"checking"(如果还存在需要校验的候选者对)，或"disconnected"(如果没有候选者对需要校验，但是对等体仍在收集并且/或等待其它的远程候选者)。
`completed`	`RTCIceTransport` 已经完成收集，接收到 没有更多远程候选者的指示，完成校验所有候选者对，并且已经发现一个连接。如果对于所有成功的候选者对的consent checks[RFC7675]接连失败，状态转变为" `failed` ".
`disconnected`	ICE代理已经确定当前 `RTCIceTransport` 的连接丢失。这是一个暂态，在片段网络中可能会间歇触发。确定状态的方式与实现方式相关。例如:丢失正在使用的连接的网络接口，或是接收对STUN请求的响应反复失败。另外， `RTCIceTransport` 已经完成校验所有现存候选者对，还没有发现一个连接，但是它仍在收集或等待额外的远程候选者。
`failed`	RTCIceTransport已经完成收集，接收了没有更多远程候选者的指示，完成了对所有候选者对的校验，这是最终状态。
`closed`	RTCIceTransport已经断开，不再对STUN请求响应。

ICE重启导致候选收集和连接检查重新开始，如果在 `completed` 状态下开始，则转换为 `connected` 。如果在暂态 `disconnected` 开始，则会导致转变为 `checking` ，从而有效的忘记之前丢失的连接。

`failed` 和 `completed` 状态需要一个不再有额外远程候选者的指示。这可以通过调用 `addIceCandidate` 来实现，其中候选者值的 `candidate` 属性被设置为空字符串或者通过 `canTrickleIceCandidates` 被设置为 `false` 。

一些状态转变的例子包括：

- (`RTCIceTransport` 首次创建，作为 `setLocalDescription` 或 `setRemoteDescription` 的结果)： `new`
- (`new` ,接收远程候选者)： `checking`
- (`checking` ，发现可用连接)： `connected`
- (`checking` ,校验失败，仍在收集过程)： `disconnected`
- (`checking` ,放弃)： `failed`
- (`disconnected` ,新的本地候选者)： `checking`
- (`connected` ,完成所有校验)： `completed`
- (`completed` ,丢失连接)： `disconnected`
- (`disconnected` 或 `failed` ，出现ICE重启)： `checking`
- (`completed` ,出现ICE重启)： `connected`
- `RTCPeerConnection.close()`：`closed`

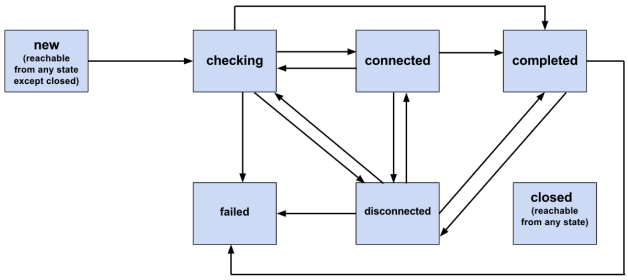


Figure 2 Non-normative ICE transport state transition diagram

5.6.5 RTCIceRole枚举

```
enum RTCIceRole {
    "controlling",
    "controlled"
};
```

RTCIceRole枚举描述	
`controlling`	一个由[ICE]，第三节中定义的 `controlling agent` 。
`controlled`	一个由[ICE]，第三节中定义的 `controlled agent` 。

5.6.6 RTCIceComponent枚举

```
enum RTCIceComponent {
    "rtp",
    "rtcp"
};
```

RTCIceComponent 枚举描述	
`rtp`	ICE传输被用到[ICE],4.1.1.1中定义的 `RTP` (或 `RTCP` 复用)。与RTP复用的协议(例如数据通道)共享其组件ID。这表示在候选属性中编码时 `component-id` 值为1。
`rtcp`	ICE传输用于RTCP，如[ICE]第4.1.1.1节中所定义。这表示在候选属性中编码时的 `component-id` 值2。

5.7 RTCTrackEvent

`track` 事件使用 `RTCTrackEvent` 接口。

```
        [ Constructor (DOMString type, RTCTrackEventInit eventInitDict), Exposed=Windo
interface RTCTrackEvent : Event {
    readonly        attribute RTCRtpReceiver           receiver;
    readonly        attribute MediaStreamTrack         track;
    [SameObject]
    readonly        attribute FrozenArray<MediaStream> streams;
    readonly        attribute RTCRtpTransceiver        transceiver;
};
```

构造函数

`RTCTrackEvent` (https://github.com/web-platform-tests/wpt/blob/master/webrtc/RTCTrackEvent-constructor.html)

属性

`RTCRtpReceiver` 类型的 `receiver` ， 只读： `receiver` 属性表示与事件关联的 `RTCRtpReceiver` 对象。

`MediaStreamTrack` 类型的track， 只读： `track` 属性表示与 `RTCRtpReceiver` 关联的由 `receiver` 验证的 `MediaStreamTrack` 对象。

`FrozenArray<MediaStream>` 类型的streams,只读： `streams` 属性返回 `MediaStream` 对象的数组，表示此事件的track是 `MediaStreams` 的一部分。

`RTCRtpTransceiver` 类型的 `transceiver` ， 只读： `transceiver` 属性表示与此事件关联的 `RTCRtpTransceiver` 对象。

```
dictionary RTCTrackEventInit : EventInit {
    required RTCRtpReceiver        receiver;
    required MediaStreamTrack      track;
             sequence<MediaStream> streams = [];
    required RTCRtpTransceiver     transceiver;
};
```

字典 **`RTCTrackEventInit`** 成员

`RTCRtpReceiver` 类型的 `receiver`

`MediaStreamTrack` 类型的 `track`

`sequence<MediaStream>` 类型的 `streams`

`RTCRtpTransceiver` 类型的 `transceiver`

6 对等数据API

对等数据API允许网络应用点对点发送接收通用应用数据。发送接收数据的API模拟 `WebSockets` [WEBSOCKETS-API]的行为。

6.1.1 `RTCSctpTransport` 接口

`RTCSctpTransport` 接口允许应用获取SCTP绑定到特定SCTP关联的数据通道的信息。

6.1.1.1创建一个实例

为了创建一个可选初始状态的 `RTCSctpTransport` ， `initialState` ，运行下列步骤。

1.让transport成为一个新的 `RTCSctpTransport` 对象。

2.让transport具有一个[SctpTransportState]内部插槽，初始化为 `initialState` ，如果提供，否则为 `"new"` 。

3.让transport具有[MaxMessageSize]内部插槽，运行标记为 `update the data max message size` 的标签对其初始化。

4.让transport具有[MaxChannels]内部插槽初始化为 `null` 。

5.返回transport。

6.1.1.2 更新最大信息容量

为了更新 `RTCSCTPTransport` 的数据最大信息容量，运行下列步骤：

1.让transport成为用来更新的 `RTCSCTPTransport` 对象。

2.让 `remoeteMaxMessageSize` 成为从远程描述中获取的"max-message-size"SDP属性的值，如[SCTP-SDP]中所述，或65536如果属性缺失。

3.让 `canSendSize` 成为客户端可发送的字节数(本地发送buffer的大小)，如果可以处理任意大小的信息，则设置为0。

4.如果 `remoteMaxMessageSize` 和 `canSendSize` 都为0，设置[MaxMessageSize]为正无穷。

5.否则，如果 `remoteMaxMessageSize` 与 `canSendSize` 有一个为0，设置[MaxMessageSize]为两者较大值。

6.否则，设置[MaxMessageSize]为两者较小值。

6.1.1.3 连接步骤

`一旦SCTP传输建立连接`，这意味着`RTCSctpTransport`的SCTP关联已经被建立，运行下列步骤：

1.让transport成为`RTCSCTPTransport`对象。

2.让connection成为与transport关联的`RTCPeerConnection`对象。

3.设置[MaxChannels]为协商好的输入输出SCTP流的最小值。

4.在transport发起一个名为`statechange`的事件。

5.对于connection的每一个`RTCDataChannel`：

1. 让channel成为RTCDataChannel对象。
2. 如果channel的[DataChannelId]插槽大于等于transport的[MaxChannels]插槽，由于失败关闭channel。否则，声明channel为open。

```
[Exposed=Window] interface RTCSctpTransport : EventTarget {
    readonly        attribute RTCDtlsTransport transport;
    readonly        attribute RTCSctpTransportState state;
    readonly        attribute unrestricted double maxMessageSize;
    readonly        attribute unsigned short? maxChannels;
                    attribute EventHandler     onstatechange;
};
```

属性

`RTCDtlsTransport`类型的`transport`，只读：数据通道的所有SCTP数据包通过此传输发送和接收。测试2

`RTCSctpTransportState`类型的`state`，只读：SCTP传输的目前状态。当获取时，此属性必须返回[SctpTransportState]插槽的值。

无限制double类型的`maxMessageSize`，只读：可以被传入`RTCDataChannel`的`send()`方法的最大数据量。此属性，返回[MaxMessageSize]插槽的值。

无符号short类型的`maxChannels`,只读，可以为null：可以被同时使用的`RTCDataChannel`的最大量。此属性必须返回[MaxChannels]插槽的值。

> NOTE:此属性的值将会为null，直到SCTP传输进入connected状态。

EventHadnler类型的`onstatechange`:此event handler的事件类型是`statechange`。

6.1.2 RTCSctpTransportState枚举

RTCSctpTransportState表示SCTP传输状态。

```
enum RTCSctpTransportState {
    "connecting",
    "connected",
    "closed"
};
```

枚举描述	
`connecting`	`RTCSctpTransport` 正在协商一个关联。这是当 `RTCSctpTransport` 被创建时[SctpTransportState]插槽的初始状态。
`connected`	当协商完成后，会出现一个更新[SctpTransportState]插槽为 “`connected`” 的任务。
`closed`	当接收到SHUTDOWN或者ABORT时，或是SCTP关联被有意关闭，例如通过关闭对等连接或应用拒绝数据，改变SCTP端口的远程描述，[SctpTransportState]插槽的值会被更新为 “`closed`” 。

6.2 RTCDataChannel

RTCDataChannel 接口表示两个对等体之间的双向数据信道。 RTCDataChannel 是通过 RTCPeerConnection 对象上的工厂方法创建的。浏览器之间发送的消息在[RTCWEB-DATA]和[RTCWEB-DATA-PROTOCOL]中描述。

有两种方法可以与 RTCDataChannel 建立连接。第一种方法是在其中一个对等体上创建一个 RTCDataChannel ，并且协商的 RTCDataChannelInit 字典成员为未设置或被设置为默认值false。这将在带内公布新通道，并在另一个对等体上触发带有相应 RTCDataChannel 对象的 RTCDataChannelEvent 。第二种方法是让应用程序协商 RTCDataChannel 。为此，创建一个 RTCDataChannel 对象，将协商的 RTCDataChannelInit 字典成员设置为true，并通过带外信号（例如通过Web服务器）向另一方发出信号，它应该创建一个带有协商的 RTCDataChannelInit 字典成员集的相应 RTCDataChannel ，字典成员被设置为true，并且具有相同 id 。这将连接两个单独创建的 RTCDataChannel 对象。第二种方法可以创建具有非对称属性的通道，并通过指定匹配的ID以声明方式创建通道。

每个 RTCDataChannel 都有一个关联的底层数据传输，用于将实际数据传输到另一个对等端。在SCTP数据信道利用 RTCSctpTransport （表示SCTP关联的状态）的情况下，底层数据传输是SCTP流对。底层数据传输的传输属性（例如有序发送设置和可靠性模式）由对等方在创建通道时配置。创建通道后，通道的属性不会更改。对等体之间的实际有线协议由WebRTC DataChannel协议规范[RTCWEB-DATA]指定。

可以将 RTCDataChannel 配置为在不同的依赖模式下操作。可靠的信道确保通过重新传输在另一个对等体上传达数据。不可靠信道被配置为限制重传次数（ maxRetransmits ）或设置允许传输（包括重传）的时间（ maxPacketLifeTime ）。这些属性不能同时使用，这样做会导致错误。不设置任何这些属性会得到可靠的通道。

使用 createDataChannel 创建或通过 RTCDataChannelEvent 调度的 RTCDataChannel 必须最初处于 connecing 状态。当 RTCDataChannel 对象的底层数据传输准备就绪时，用户代理必须宣布 RTCDataChannel 为open。

要创建 RTCDataChannel ，请运行以下步骤:

1. 让channel成为新创建的 RTCDataChannel 对象。
2. 让通道将[[ReadyState]]内部插槽初始化为 “connecting” 。测试1
3. 让通道将[[BufferedAmount]]内部插槽初始化为 0 。
4. 让通道具有内部插槽，名为[[DataChannelLabel]]，[[Ordered]]，[[MaxPacketLifeTime]]，[[MaxRetransmits]]，[[DataChannelProtocol]]，[[Negotiated]]，[[DataChannelId]]和[[DataChannelPriority]。
5. 返回通道。

当用户代理宣布 RTCDataChannel 为open时，用户代理必须对任务进行排队,运行以下步骤：

1. 如果关联的 RTCPeerConnection 对象的[[IsClosed]]插槽为true，则中止这些步骤。
2. 让channel成为要宣布的 RTCDataChannel 对象。
3. 如果通道的[[ReadyState]]为 closing 或 closed ，请中止这些步骤。

4. 将通道的[[ReadyState]]插槽设置为 `open` 。
5. 在通道上触发一个名为open的事件。测试1

当要宣布底层数据传输时（另一个对等方创建一个negotiated为未设置或设置为false的通道），未启动创建过程的对等方的用户代理必须对任务排序，以运行以下步骤：

1. 如果关联的 `RTCPeerConnection` 对象的[[IsClosed]]插槽为true，则中止这些步骤。
2. 创建一个 `RTCDataChannel` ，通道。
3. 让配置成为从另一个对等体接收的信息包，作为建立由WebRTC数据通道协议规范[RTCWEB-DATA-PROTOCOL]描述的底层数据传输的过程的一部分。
4. 将通道的[[DataChannelLabel]]，[[Ordered]]，[[MaxPacketLifeTime]]，[[MaxRetransmits]]，[[DataChannelProtocol]]和[[DataChannelId]]内部插槽初始化为配置中的相应值。
5. 将通道的[[Negotiated]]内部插槽初始化为 `false` 。
6. 根据配置中的整数优先级值初始化通道的[[DataChannelPriority]]内部插槽，根据以下映射：

   | configuration priority value | RTCPriorityType value | | ---------- | ---------- | | 0 to 128 | very-low | | 129 to 256 | low | | 257 to 512 | medium | | 513 and greater | high |

7. 将通道的[[ReadyState]]设置为 `open` （但不要触发 `open` 事件）。

   > NOTE:这允许在触发open事件之前开始在 `datachannel` 事件处理程序内发送消息。

8. 使用 `RTCDataChannelEvent` 接口触发名为 `datachannel` 的事件，并将channel属性设置为 `RTCPeerConnection` 对象的channel。
9. 宣布数据通道处于open状态。

通过运行关闭过程，可以以非突然的方式拆除 `RTCDataChannel` 对象的底层数据传输。当发生这种情况时，用户代理必须排队任务以运行以下步骤：

1. 让channel成为其传输已关闭的 `RTCDataChannel` 对象。
2. 除非通过通道的 `close` 方法启动该过程，否则将通道的[[ReadyState]]插槽设置为 `closing` 。
3. 并行运行以下步骤:
   i. 完成发送channel的所有当前待处理消息。
   ii. 遵循为通道的底层传输定义的关闭过程：
      i. 如果是基于SCTP的传输，请按照[RTCWEB-DATA]的6.7节进行操作。
   iii. 按照相关步骤渲染通道的数据传输。

当 `RTCDataChannel` 对象的基础数据传输已关闭时，用户代理必须对任务进行排队以运行以下步骤：

1. 让channel成为其传输已关闭的 `RTCDataChannel` 对象。
2. 将通道的[[ReadyState]]插槽设置为 `closed` 。

3. 如果传输因错误而关闭，则使用 `RTCErrorEvent` 接口触发名为 `error` 的事件，并在通道中将其 `errorDetail` 属性设置为 `“sctp-failure”` 。
4. 在通道内发起一个名为 `close` 的事件。

在某些情况下，用户代理可能无法创建 `RTCDataChannel` 的基础数据传输。例如，数据通道的 `id` 可能超出SCTP握手中[RTCWEB-DATA]实现协商的范围。当用户代理确定无法创建 `RTCDataChannel` 的基础数据传输时，用户代理必须排队任务以运行以下步骤：

1. 令channel为 `RTCDataChannel` 对象，用户代理无法为其创建底层数据传输。
2. 将通道的[[ReadyState]]插槽设置为 `closed` 。
3. 使用 `RTCErrorEvent` 接口触发名为 `error` 的事件，并在通道中将 `errorDetail` 属性设置为“data-channel-failure”。
4. 在通道上发起一个名为 `close` 的事件。

当通过类型 `type` 和数据 `rawData` 的底层数据传输接收到 `RTCDataChannel` 消息时，用户代理必须排队任务以运行以下步骤：

1. 令channel为用户代理已收到消息的 `RTCDataChannel` 对象。
2. 如果通道的[[ReadyState]]插槽不为 `open` ，则中止这些步骤并丢弃 `rawData` 。
3. 通过打开 `type` 和 `channel` 的 `binaryType` 来执行子步骤：
    - 如果type表示rawData是一个字符串:

      令数据为DOMString，表示将 `rawData` 解码为UTF-8的结果。

      测试1
    - 如果type表示rawData是二进制而 `binaryType` 是 `“blob”` ：

      让data成为包含rawData作为其原始数据源的新 `Blob` 对象。
    - 如果type表示rawData是二进制，而 `binaryType` 是 `“arraybuffer”` ：

      让data成为一个新的 `ArrayBuffer` 对象，包含rawData作为原始数据源。
4. 使用MessageEvent接口触发名为 `message` 的事件，其 `origin` 属性初始化为创建通道关联的 `RTCPeerConnection` 的文档的原点，并且 `data` 属性初始化为通道上的数据。

```
[Exposed=Window] interface RTCDataChannel : EventTarget {
    readonly        attribute USVString           label;
    readonly        attribute boolean             ordered;
    readonly        attribute unsigned short?     maxPacketLifeTime;
    readonly        attribute unsigned short?     maxRetransmits;
    readonly        attribute USVString           protocol;
    readonly        attribute boolean             negotiated;
    readonly        attribute unsigned short?     id;
    readonly        attribute RTCPriorityType     priority;
    readonly        attribute RTCDataChannelState readyState;
    readonly        attribute unsigned long       bufferedAmount;
                    [EnforceRange]
                    attribute unsigned long       bufferedAmountLowThreshold;
                    attribute EventHandler        onopen;
                    attribute EventHandler        onbufferedamountlow;
                    attribute EventHandler        onerror;
                    attribute EventHandler        onclose;
    void close ();
                    attribute EventHandler        onmessage;
                    attribute DOMString           binaryType;
    void send (USVString data);
    void send (Blob data);
    void send (ArrayBuffer data);
    void send (ArrayBufferView data);
};
```

属性

USVString类型的 `label` ，只读:label属性表示可用于将此 `RTCDataChannel` 对象与其他 `RTCDataChannel` 对象区分开的标签。允许脚本使用相同的标签创建多个 `RTCDataChannel` 对象。获取时，属性必须返回[[DataChannelLabel]]槽的值。测试1

boolean类型 `ordered` ，只读：如果 `RTCDataChannel` 有序，则 `ordered` 属性返回true，如果允许无序传递，则返回false。获取时，属性必须返回[[Ordered]]槽的值。

unsigned short类型的 `maxPacketLifeTime` ，只读的，可以为null： `maxPacketLifeTime` 属性返回在不可靠模式下可能发生传输和重传的时间窗口的长度（以毫秒为单位）。获取时，属性必须返回[[MaxPacketLifeTime]]槽的值。

unsigned short类型的 `maxRetransmits` ，只读的，可以为null： `maxRetransmits` 属性返回在不可靠模式下尝试的最大重新传输次数。获取时，属性必须返回[[MaxRetransmits]]槽的值。

USVString类型的 `protocol` ，只读的： `protocol` 属性返回与此 `RTCDataChannel` 一起使用的子协议的名称。获取时，属性必须返回[[DataChannelProtocol]]槽的值。

boolean类型的 `negotiated` ，只读的：如果此 `RTCDataChannel` 由应用程序协商，则 `negotiated` 属性返回true，否则返回false。获取时，属性必须返回[[Negotiated]]插槽的值。

unsigned short类型的 `id` ，只读的，可以为null： `id` 属性返回此 `RTCDataChannel` 的ID。该值初始为null，如果在创建通道时未提供ID，则返回该值，并且尚未协商SCTP传输的DTLS角色。否则，它将返回由脚本选择的ID或由用户代理根据[RTCWEB-DATA-PROTOCOL]生成的ID。将ID设置为非空值后，它不会更改。获取时，属性必须返回[[DataChannelId]]槽的值。测试2

RTCPriorityType类型的 `priority` ，只读: `priority` 属性返回此 `RTCDataChannel` 的优先级。优先级由用户代理在通道创建时分配。获取时，属性必须返回[[DataChannelPriority]]槽的值。

`RTCDataChannelState` 类型的 `readyState` ，只读的： `readyState` 属性表示 `RTCDataChannel` 对象的状态。获取时，属性必须返回[[ReadyState]]槽的值。

unsigned long类型的 `bufferedAmount` ，只读的：获取时， `bufferedAmount` 属性必须返回[[BufferedAmount]]槽的值。该属性公开使用 `send()` 排队的应用程序数据（UTF-8文本和二进制数据）的字节数。即使数据传输可以并行发生，在当前任务返回事件循环以防止race condition之前，不得减小返回值。该值不包括协议产生的帧开销，或操作系统或网络硬件完成的缓冲。只要[[ReadyState]]插槽打开，[[BufferedAmount]]插槽的值只会随着每次调用send（）方法而增加;但是，一旦通道关闭，插槽不会重置为零。当底层数据传输从其队列发送数据时，用户代理必须排队一个任务，该任务随着发送的字节数减少[[BufferedAmount]]。

unsigned long类型的 `bufferedAmountLowThreshold`：`bufferedAmountLowThreshold` 属性设置 `bufferedAmount` 被视为低的阈值。当 `bufferedAmount` 从此阈值以上减小到等于或低于此阈值时，将触发 `bufferedamountlow` 事件。 `bufferedAmountLowThreshold` 在每个新的 `RTCDataChannel` 上最初为零，但应用程序可能随时更改其值。

EventHandler类型的 `onopen` :此事件处理程序的事件类型为 `open` 。

eventHandler类型的 `onbufferedamountlow` :此事件处理程序的事件类型为 `bufferedamountlow` 。

eventHandler类型的 `onerror` :此事件处理程序的事件类型是 `RTCErrorEvent` 。 `errorDetail` 包含“sctp-failure”， `sctpCauseCode` 包含SCTP Cause Code值，并且 `message` 包含SCTP Cause-Specific-Information，可能包含其他文本。

EventHandler类型的 `onclose` :此事件处理程序的事件类型为 `close` 。

eventHandler类型的 `onmessage` :此事件处理程序的事件类型是 `message` 。

DOMString类型的 `binaryType` :获取时， `binaryType` 属性必须返回上次设置的值。在设置时，如果新值是字符串 `“blob”` 或字符串 `“arraybuffer”` ，则将IDL属性设置为此新值。否则，抛出一个 `SyntaxError` 。创建 `RTCDataChannel` 对象时，必须将 `binaryType` 属性初始化为字符串 `“blob”` 。

此属性控制二进制数据如何向脚本公开。有关更多信息，请参阅[WEBSOCKETS-API]。

方法

`close`：

关闭 `RTCDataChannel` 。无论 `RTCDataChannel` 对象是由对等方还是远程对等方创建，都可以调用它。

调用close方法时，用户代理必须执行以下步骤：

1. 让channel成为即将关闭的 `RTCDataChannel` 对象。
2. 如果通道的[[ReadyState]]插槽为 `closing` 或 `closed` ，则中止这些步骤。
3. 将通道的[[ReadyState]]插槽设置为 `closing` 。
4. 如果关闭程序尚未开始，请启动它。

`send`

使用参数类型 `string` 对象运行 `send()` 算法描述的步骤。测试1

`send`

使用参数类型 `Blob` 对象运行 `send()` 算法描述的步骤。

`send`

使用参数类型 `ArrayBuffer` 对象运行 `send()` 算法描述的步骤。

`send`

使用参数类型 `ArrayBufferView` 对象运行 `send()` 算法描述的步骤。

`send()` 方法被重载，用来处理不同数据参数类型。当该方法的任何版本被调用时，用户代理必须运行下列步骤：

1. 让channel成为将要发送数据的 `RTCDataChannel` 对象。
2. 如果channel的[ReadyState]插槽不为 `open` ，抛出 `InvalidStateError` 。
3. 执行对应方法参数类型的子步骤：
    - `string` 对象:

      让data成为byte buffer，表示将方法参数编码为UTF-8的结果。
    - `Blob` 对象:

      让data成为由 `Blob` 对象表示的原始数据。
    - `ArrayBuffer` 对象:

      让data成为由 `ArrayBuffer` 对象描述的存在buffer中的数据。
    - `ArrayBufferView` 对象：

      让data成为 `ArrayBufferView` 对象提到的 `ArrayBuffer` 对象描述的buffer中存储的数据。

    > NOTE:任何该方法没有重载的数据类型将会导致 `TypeError` 。这包括null和undefined。
4. 如果data的大小超过了channel的关联 `RTCSctpTransport` 上的 `maxMessageSize` 的值，抛出 `TypeError` 。
5. 对在channel的底层数据传输的data进行排队。

    > NOTE:实际数据传输是并行的。如果发送数据导致SCTP层级的错误，应用程序将会被通过 `onerror` 异步通知。
6. 增加[BufferedAmount]插槽的值，增加量为data的大小。

```
dictionary RTCDataChannelInit {
             boolean         ordered = true;
             [EnforceRange]
             unsigned short  maxPacketLifeTime;
             [EnforceRange]
             unsigned short  maxRetransmits;
             USVString       protocol = "";
             boolean         negotiated = false;
             [EnforceRange]
             unsigned short  id;
             RTCPriorityType priority = "low";
};
```

字典 `RTCDataChannelInit` 成员

boolean类型的 `ordered` ，默认为true:如果设置为false，则允许数据不按顺序传送。默认值为true，保证数据按顺序传递。

unsigned short类型的 `maxPacketLifeTime` :限制通道在未确认的情况下传输或重新传输数据的时间（以毫秒为单位）。如果该值超过用户代理支持的最大值，则可以限制该值。测试1

unsigned short类型的 `maxRetransmits` :如果未成功传递，则限制通道重新传输数据的次数。如果该值超过用户代理支持的最大值，则可以限制该值。

USVString类型的 `protocol` ，默认为 “” :用于此通道的子协议名称。

boolean类型的 `negotiated` ，默认为 `false` :默认值false指示用户代理在带内通告通道并指示另一个对等方分派相应的 `RTCDataChannel` 对象。如果设置为true，则由应用程序协商通道并在另一个对等方创建具有相同ID的 `RTCDataChannel` 对象。

> NOTE:如果设置为true，则应用程序还必须注意不要发送消息，直到另一个对等方创建了一个数据通道来接收它。在没有关联数据通道的SCTP流上接收消息是未定义的行为，可能会以静默方式丢弃。只要两个端点在第一个提议/应答交换完成之前创建其数据通道，就不可能实现这一点。

unsigned short类型的 `id` :重写此通道的默认ID选择。

RTCPriorityType类型的 `priority` ，默认为 `low` :此通道的优先级。

```
enum RTCDataChannelState {
    "connecting",
    "open",
    "closing",
    "closed"
};
```

RTCDataChannelState 枚举描述	
`connecting`	用户代理正在尝试建立底层数据传输。这是 `RTCDataChannel` 对象的初始状态，无论是使用 `createDataChannel` 创建，还是作为 `RTCDataChannelEvent` 的一部分调度。
`open`	建立基础数据传输并且可以进行通信。
`closing`	关闭底层数据传输的过程已经开始。
`closed`	底层数据传输已关闭或无法建立。

6.3 RTCDataChannelEvent

使用 `RTCDataChannelEvent` 接口的 `datachannel` 事件。测试1

```
        [ Constructor (DOMString type, RTCDataChannelEventInit eventInitDict), Exposed
interface RTCDataChannelEvent : Event {
    readonly        attribute RTCDataChannel channel;
};
```

构造函数

`RTCDataChannelEvent`

测试1

属性

`RTCDataChannel` 类型的channel，只读：channel属性表示与事件关联的 `RTCDataChannel` 对象。

```
dictionary RTCDataChannelEventInit : EventInit {
             required RTCDataChannel channel;
};
```

字典 `RTCDataChannelEventInit` 成员

`RTCDataChannel` 类型的channel，required：将要被事件宣布的 `RTCDataChannel` 对象。

6.4垃圾收集

`RTCDataChannel`对象必须不能被垃圾回收如果它的

- [ReadyState]插槽为`connecting`并且至少一个事件听众
  以`open`，`message`，`error`，`close`事件登记。
- [ReadyState]插槽为`open`并且至少一个事件听众
  以`message`，`error`，`close`事件登记。
- [ReadyState]插槽为`closing`并且至少一个事件听众以`error`，`close`事件登记。
- 底层数据传输被建立，并且数据已经排序准备传输。

7 点对点DTMF

本节介绍 `RTCRtpSender` 上的一个接口，用于在 `RTCPeerConnection` 上发送DTMF（电话键盘）值。有关如何将DTMF发送到其他对等方的详细信息，请参见[RTCWEB-AUDIO]。

7.1 RTCRtpSender接口扩展

点对点DTMF API对 `RTCRtpSender` 接口的扩展如下。

```
partial interface RTCRtpSender {
    readonly        attribute RTCDTMFSender? dtmf;
};
```

属性

RTCDTMFSender类型的dtmf,只读的，可以为null：当获取时，dtmf属性返回[Dtmf]内部插槽的值，它代表一个可用来发送DTMF的 `RTCDTMFSender` ,如果未设置，则为null。当RTCRtpSender的[SenderTrack]为“audio”，[Dtmf]内部插槽被设置。

7.2 RTCDTMFSender

为了创建RTCDTMFSender，用户代理必须运行以下步骤：

1. 让dtmf成为最新创建的 `RTCDTMFSender` 对象。
2. 让dtmf具有[Duration]内部插槽。
3. 让dtmf具有[InterToneGap]内部插槽。
4. 让dtmf具有[ToneBuffer]内部插槽。

```
[Exposed=Window] interface RTCDTMFSender : EventTarget {
    void insertDTMF (DOMString tones, optional unsigned long duration = 100, optional
                     attribute EventHandler ontonechange;
    readonly         attribute boolean      canInsertDTMF;
    readonly         attribute DOMString    toneBuffer;
};
```

属性

事件处理程序类型的 `ontonechange` :此事件处理程序的事件类型时候tonechange。测试1

boolean类型的 `canInsertDTMF` ,只读：不管 `RTCDTMFSender` dtmfSender是否能发送DTMF。当获取时，用户代理必须返回结果，确定是否可以为dtmfSender发送DTMF。

DOMString类型的 `toneBuffer` ，只读： `toneBuffer` 属性必须返回剩余需要播放的tone列表。关于此列表的语法，内容，解释，查看 `insertDTMF` 。

方法

`insertDTMF` : `RTCDTMFSender` 对象的 `insertDTMF` 方法用于发送DTMF音调。

tone参数被视为一系列字符。字符0到9，A到D， `#` 和 `*` 生成相关的DTMF音调。字符a到d必须在输入时标准化为大写，并且等同于A到D.如[RTCWEB-AUDIO]第3节所述，需要支持字符0到9，A到D，＃和*。必须支持字符','，并指示在处理tone参数中的下一个字符之前的2秒延迟。所有其他字符（以及仅其他字符）必须被视为无法识别。测试1

持续时间参数指示用于在音调参数中传递的每个字符的持续时间（以毫秒为单位）。持续时间不能超过6000毫秒或小于40毫秒。每种音调的默认持续时间为100 ms。

interToneGap参数表示以ms为单位的音调间隙。用户代理将其钳位至少30毫秒，最多6000毫秒。默认值为70毫秒。

浏览器可以增加持续时间和interToneGap时间，以使DTMF开始和停止的时间与RTP数据包的边界对齐，但它不能增加它们中的任何一个超过单个RTP音频数据包的持续时间。

当调用 `insertDTMF()` 方法时，用户代理必须运行以下步骤：

1. 让sender成为用来发送DTMF的 `RTCRtpSender` 。
2. 让transceiver成为与sender关联的 `RTCRtpTransceiver` 对象。

3. 如果transceiver的[Stopped]插槽为 `true` ，抛出 `InvalidStateError` 。
4. 如果transceiver的[CurrentDirection]插槽为 `recvonly` 或者 `inactive` ，抛出 `InvalidStateError` 。
5. 让dtmf成为与sender关联的 `RTCDTMFSender` 对象。
6. 如果决定是否可以为drmf发送DTMF返回 `false` ，抛出 `InvalidStateError` 。
7. 让tones成为方法的第一个参数。
8. 如果tones包含任意未识别的字符，抛出 `InvalidCharacterError` 。
9. 设置对象的[ToneBuffer]插槽为tones。
10. 设置dtmf的[Duration]插槽为 `duration` 参数的值。
11. 设置dtmf的[InterToneGap]插槽为 `interToneGap` 参数的值。
12. 如果 `duration` 参数的值小于40毫秒，设置dtmf的[Duration]插槽为40毫秒。
13. 如果 `duration` 参数的值大于6000 ms，请将dtmf的[[Duration]]插槽设置为6000 ms。
14. 如果 `interToneGap` 参数的值小于30 ms，则将dtmf的[[InterToneGap]]插槽设置为30 ms。
15. 如果 `interToneGap` 参数的值大于6000 ms，请将dtmf的[[InterToneGap]]插槽设置为6000 ms。
16. 如果[[ToneBuffer]]插槽为空字符串，则中止这些步骤。
17. 如果计划运行Playout任务，则中止这些步骤;否则运行以下步骤的任务排队（播出任务）：
    i. 如果收发器的[[Stopped]]插槽为真，则中止这些步骤。
    ii. 如果收发器的[[CurrentDirection]]插槽是 `recvonly` 或 `inactive` ，则中止这些步骤。
    iii. 如果[[ToneBuffer]]槽包含空字符串，则使用RTCDTMFToneChangeEvent接口触发名为 `tonechange` 的事件，并将 `tone` 属性设置为RTCDTMFSender对象的空字符串并中止这些步骤。
    iv. 从[[ToneBuffer]]插槽中删除第一个字符，然后让该字符变为tone。
    v. 如果音调是“，”延迟在相关的RTP媒体流上发送tones 2000毫秒，并为2000毫秒后开始执行的任务排队，此任务运行标记为播出任务的步骤。
    vi. 如果音调不是“，”使用适当的编解码器在相关的RTP媒体流上开始播放[[持续时间]] ms的音调，然后将要在[[持续时间]] + [[InterToneGap]] ms中执行的任务排队，此任务运行标记为播出任务的步骤。
    vii. 使用 `RTCDTMFToneChangeEvent` 接口触发名为 `tonechange` 的事件，并将 `tone` 属性设置为 `RTCDTMFSender` 对象的tone。

由于 `insertDTMF` 替换了音调缓冲区，为了添加正在播放的DTMF音调，必须使用包含剩余音调（存储在[[ToneBuffer]]插槽中）和附加在一起的新音调的字符串调用 `insertDTMF` 。使用空音调参数调用 `insertDTMF` 可用于取消在当前播放音调之后排队等待播放的所有音调。

7.3 canInsertDTMF算法

为了确定是否可以为RTCDTMFSender实例dtmfSender发送DTMF,用户代理必须对运行以下步骤的任务排队：

1. 让sender成为与dtmfSender关联的 `RTCRtpSender` 。
2. 让收发器成为与sender关联的 `RTCRtpTransceiver` 。
3. 让connection成为与收发器关联的 `RTCPeerConnection` 。
4. 如果connection的 `RTCPeerConnectionState` 不为 `connected` ， 返回 `false` 。
5. 如果sender的[SenderTrack]为 `null` ， 返回 `false` 。
6. 如果收发器的[CurrentDirection]既不是 `sendrecv` 也不是 `sendonly` ,返回 `false` 。
7. 如果发送者的[[SendEncodings]] `[0] .active` 为 `false` ， 则返回 `false` 。
8. 如果没有协商mimetype“audio / telephone-event”的编解码器与此发件人一起发送，则返回 `false` 。
9. 否则，返回 `true` 。

7.4 RTCDTMFToneChangeEvent

`tonechange` 事件使用 `RTCDTMFToneChangeEvent` 接口。

```
       [ Constructor (DOMString type, RTCDTMFToneChangeEventInit eventInitDict), Expos
interface RTCDTMFToneChangeEvent : Event {
    readonly        attribute DOMString tone;
};
```

构造函数

`RTCDTMFToneChangeEvent`

属性

DOMString类型的 `tone` ，只读：：`tone` 属性包含刚刚开始播放的音调（包括“，”）的字符（参见 `insertDTMF` ）。如果该值为空字符串，则表示[[ToneBuffer]]插槽为空字符串，并且前一个音调已完成播放。

```
dictionary RTCDTMFToneChangeEventInit : EventInit {
            required DOMString tone;
};
```

字典 `RTCDTMFToneChangeEventInit` 成员

DOMString类型的 `tone`：`tone` 属性包含刚刚开始播放的音调（包括“，”）的字符（参见 `insertDTMF` ）。如果该值为空字符串，则表示[[ToneBuffer]]插槽为空字符串，并且前一个音调已完成播放。

8 统计模型

8.1简介

基本的统计模型是浏览器以统计对象的形式维护一系列受监控对象的统计信息。

选择器可以引用一组相关对象。例如，选择器可以是 `MediaStreamTrack` 。要使track成为有效的选择器，它必须是由发出统计请求的 `RTCPeerConnection` 对象发送或接收的 `MediaStreamTrack` 。调用Web应用程序为getStats（）方法提供选择器，浏览器根据统计信息选择算法发出（在JavaScript中）与选择器相关的一组统计信息。请注意，该算法采用选择器的发送方或接收方。

stats对象中返回的统计信息的设计方式是重复查询可以由 `RTCStats id` 字典成员链接。因此，Web应用程序可以通过在该时段的开始和结束请求测量来在给定时间段内进行测量。

除少数例外情况外，受监控对象一旦创建，就会在其关联的 `RTCPeerConnection` 期间存在。这样可以确保getStats（）的结果中的统计信息在关闭的关联对等连接之后可用。

只有少数受监控对象的生命周期较短。对于这些对象，它们的生命周期在被算法删除时结束。在删除时，将在包含RTCStatsReport对象的单个 `statsended` 事件中发出其统计信息的记录，该对象包含同时删除的所有对象的统计信息。后续getStats（）结果中不再提供这些对象的统计信息。 [WEBRTC-STATS]中的对象描述描述了何时删除这些受监视对象。

8.2 RTCPeerConnection接口扩展

统计API对RTCPeerConnection接口的扩展如下。

```
partial interface RTCPeerConnection {
    Promise<RTCStatsReport> getStats (optional MediaStreamTrack? selector = null);
    attribute EventHandler onstatsended;
          };
```

属性

事件处理程序类型的 `onstatsended`：

此事件处理程序的事件类型是 `statsended`。

为了删除与一个 `RTCPeerConnection`，connection相关联的一系列被监控对象的统计信息，用户代理必须并行运行以下步骤：

1. 收集将要删除的一系列被监控对象的统计信息。这些信息必须代表删除时的最终值。这些信息不能出现在接下来对getStats()的调用中。
2. 对运行以下步骤的任务进行排队：
    i. 让report成为新的 `RTCStatsReport` 对象。
    ii. 对于每个受监视对象，使用上面针对该受监视对象收集的统计信息创建新的相关统计信息对象对象，并将其添加到报告中。
    iii. 使用 `RTCStatsEvent` 接口发起一个名为 `statsended` 的事件，在connection将 `report` 属性设置为report。

方法

`getStats`

收集给定选择器的统计信息并异步报告结果。

当调用 `getStats()` 方法时，用户代理必须运行以下步骤：

1. 让selectorArg成为方法的第一个参数。
2. 让connection成为调用方法的 `RTCPeerConnection` 对象。
3. 如果selectorArg为 `null`，则让selector为 `null`。
4. 如果selectorArg是MediaStreamTrack，让选择器成为connection上的 `RTCRtpSender` 或 `RTCRtpReceiver`，并且connection的 `track` 成员匹配selectorArg。如果不存在此类发件人或收件人，或者如果多个发件人或收件人符合此条件，则返回承诺，拒绝条件是新创建 `InvalidAccessError`。
5. 让p成为新的承诺。
6. 并行运行以下步骤：
    i. 根据统计选择算法收集选择器指示的统计数据。
    ii. 使用生成的 `RTCStatsReport` 对象解析p，其中包含收集的统计信息。
7. 返回p。

8.3 `RTCStatsReport` 对象

getStats（）方法以 `RTCStatsReport` 对象的形式传达成功的结果。 `RTCStatsReport` 对象是标识被检查对象（ `RTCStats` 实例中的 `id` 属性）的字符串之间的映射，以及它们对应的 `RTCStats` 派生的字典。

`RTCStatsReport` 可以由几个 `RTCStats` 派生的字典组成，每个字典报告实现认为与选择器相关的一个底层对象的统计信息。通过对某种类型的所有统计量求和，可以实现选择器的总和;例如，如果 `RTCRtpSender` 使用多个SSRC通过网络传输其轨迹，则每个SSRC中 `RTCStatsReport` 可以包含一个 `RTCStats` 派生的字典（可以通过“ssrc” stats属性的值来区分）。

```
[Exposed=Window] interface RTCStatsReport {
    readonly maplike<DOMString, object>;
};
```

这个接口有“entries”，“forEach”，“get”，“has”，“keys”，“values”，@@ iterator方法和 `readonly maplike` 带来的“size”getter。

使用这些来检索此统计报告由 `RTCStats` 组成的各种词典。支持的属性名称[WEBIDL-1]的集合被定义为为此统计信息报告生成的所有 `RTCStats` 派生词典的ID。

8.4 `RTCStats` 字典

`RTCStats` 字典表示通过检查特定受监视对象构造的stats对象。 `RTCStats` 字典是一种基本类型，它指定一组默认属性，例如时间戳和类型。通过扩展 `RTCStats` 字典添加特定的统计信息。

请注意，虽然统计信息名称是标准化的，但任何给定的实现都可能使用Web应用程序尚不知道的值或实验值。因此，应用程序必须准备好处理未知的统计数据。

统计需要彼此同步才能产生合理的计算值;例如，如果同时报告“bytesSent”和“packetsSent”，则需要在相同的时间间隔内报告它们，这样“平均数据包大小”可以计算为“字节/数据包” - 如果间隔不同，则会产生错误。因此，实现必须返回 `RTCStats` 派生字典中所有统计信息的同步值。

```
dictionary RTCStats {
             required DOMHighResTimeStamp timestamp;
             required RTCStatsType        type;
             required DOMString           id;
};
```

字典 `RTCStats` 成员

DOMHighResTimeStamp类型的 `timestamp` :与此对象关联的DOMHighResTimeStamp [HIGHRES-TIME]类型的时间戳。时间相对于UNIX纪元（1970年1月1日，UTC）。对于来自远程源（例如，来自接收的RTCP数据包）的统计，时间戳表示信息到达本地端点的时间。如果适用，可以在 `RTCStats` 派生的字典中的附加字段中找到远程时间戳。

RTCStatsType类型的 `type` :

此对象的类型。

`type` 属性必须初始化为此 `RTCStats` 字典所代表的最具体类型的名称。

DOMString类型的 `id` :与检查生成此 `RTCStats` 对象的对象关联的唯一ID。从两个不同的 `RTCStatsReport` 对象中提取的两个 `RTCStats` 对象，如果它们是通过检查相同的底层对象生成的，则必须具有相同的id。用户代理可以自由选择id的任何格式，只要它符合上述要求即可。

RTBRStatsType的有效值集以及它们指示的RTCStats派生的字典记录在[WEBRTC-STATS]中。

8.5 RTCStatsEvent

`statsended` 事件使用 `RTCStatsEvent` 。

```
[ Constructor (DOMString type, RTCStatsEventInit
          eventInitDict), Exposed=Window]
          interface RTCStatsEvent : Event {
            readonly attribute RTCStatsReport report;
          };
```

构造函数

`RTCStatsEvent`

属性

RTCStatsReport类型的 `report`：`report` 属性包含RTCStats对象的相应子类的stats对象，给出生命周期结束时受监视对象的统计信息的值。

```
dictionary RTCStatsEventInit : EventInit {
            required RTCStatsReport report;
          };
```

字典**RTCStatsEventInit**成员

RTCStatsReport类型的 `report` ，必需的:

包含 `RTCStats` 对象，为生命周期结束的对象提供统计信息。

8.6 统计选择算法

统计选择算法如下：

1. 设result为空RTCStatsReport。
2. 如果selector为 `null` ，则收集整个连接的统计信息，将它们添加到result，返回结果并中止这些步骤。
3. 如果selector是 `RTCRtpSender` ，则收集统计信息并将以下对象添加到结果:
   - 表示由选择器发送的代表RTP流的所有 `RTCOutboundRTPStreamStats` 对象。
   - 添加了由 `RTCOutboundRTPStreamStats` 对象直接或间接引用的所有stats对象。
4. 如果selector是 `RTCRtpReceiver` ，则收集stats并将以下对象添加到result：
   - 表示由选择器接收的代表RTP流的所有 `RTCInboundRTPStreamStats` 对象。
   - 添加了由 `RTCInboundRTPStreamStats` 直接或间接引用的所有stats对象。
5. 返回结果。

8.7 强制实施物理数据

[WEBRTC-STATS]中列出的统计数据被用来涵盖广泛的用例。并非所有WebRTC实现都必须实现它们。

当PeerConnection上存在相应的对象时，实现必须支持生成以下类型的统计信息，并使用所列的对该对象有效时的属性：

- RTCRTPStreamStats，具有ssrc，kind，transportId，codecId，nackCount属性
- RTCReceivedRTPStreamStats，包含继承的字典中的所有必需属性，还包括packetsReceived，packetsLost，jitter，packetsDiscarded属性
- RTCInboundRTPStreamStats，包含继承的字典中的所有必需属性，还包括bytesReceived，trackId，receiverId，remoteId，framesDecoded属性
- RTCRemoteInboundRTPStreamStats，包含继承词典的所有必需属性，以及属性localId，roundTripTime
- RTCSentRTPStreamStats，包含继承的字典中的所有必需属性，以及属性packetsSent，bytesSent
- RTCOutboundRTPStreamStats，包含继承词典的所有必需属性，还包括trackId，senderId，remoteId，framesEncoded属性
- RTCRemoteOutboundRTPStreamStats，包含继承的字典中的所有必需属性，以及localId，remoteTimestamp属性
- RTCPeerConnectionStats，具有dataChannelsOpened，dataChannelsClosed属性
- RTCDataChannelStats，带有属性标签，协议，dataChannelIdentifier，state，messagesSent，bytesSent，messagesReceived，bytesReceived
- RTCMediaStreamStats，具有属性streamIdentifer，trackIds
- RTCMediaStreamTrackStats，属性detached
- RTCMediaHandlerStats具有属性trackIdentifier，remoteSource，ended
- 带有属性audioLevel的RTCAudioHandlerStats
- RTCVideoHandlerStats具有属性frameWidth，frameHeight，framesPerSecond
- 带有属性framesSent的RTCVideoSenderStats
- 带有属性framesReceived，framesDecoded，framesDropped，framesCorrupted的RTCVideoReceiverStats
- RTCCodecStats，具有属性payloadType，codec，clockRate，channels，sdpFmtpLine
- RTCTransportStats，具有bytesSent，bytesReceived，rtcpTransportStatsId，selectedCandidatePairId，localCertificateId，remoteCertificateId属性
- RTCIceCandidatePairStats，具有属性transportId，localCandidateId，remoteCandidateId，state，priority，nominated，bytesSent，bytesReceived，totalRoundTripTime，currentRoundTripTime
- RTCIceCandidateStats，具有属性地址，端口，协议，候选类型，URL
- RTCCertificateStats，具有属性fingerprint，fingerprintAlgorithm，base64Certificate，issuerCertificateId

实现可以支持生成[WEBRTC-STATS]中定义的任何其他统计信息，并且可以生成未记录的统计信息。

8.8 GetStats示例

考虑用户遇到不良声音并且应用程序想要确定其原因是否是丢包的情形。可能使用以下示例代码：

```
async function gatherStats() {
  try {
    const sender = pc.getSenders()[0];
    const baselineReport = await sender.getStats();
    await new Promise((resolve) => setTimeout(resolve, aBit)); // ... wait a bit
    const currentReport = await sender.getStats();

    // compare the elements from the current report with the baseline
    for (let now of currentReport.values()) {
      if (now.type != 'outbound-rtp') continue;

      // get the corresponding stats from the baseline report
      const base = baselineReport.get(now.id);

      if (base) {
        const remoteNow = currentReport.get(now.remoteId);
        const remoteBase = baselineReport.get(base.remoteId);

        const packetsSent = now.packetsSent - base.packetsSent;
        const packetsReceived = remoteNow.packetsReceived - remoteBase.packetsReceived

        const fractionLost = (packetsSent - packetsReceived) / packetsSent;
        if (fractionLost > 0.3) {
          // if fractionLost is > 0.3, we have probably found the culprit
        }
      }
    }
  } catch (err) {
    console.error(err);
  }
}
```

9 网络用途的媒体流API扩展

9.1 简介

`MediaStreamTrack` 接口（如[GETUSERMEDIA]规范中所定义）通常表示音频或视频的数据流。可以在 `MediaStream` 中收集一个或多个 `MediaStreamTracks` （严格来说，[GETUSERMEDIA]中定义的 `MediaStream` 可能包含零个或多个 `MediaStreamTrack` 对象）。

可以扩展 `MediaStreamTrack` 以表示来自或被发送到远程对等体（例如，不仅仅是本地相机）的媒体流。此部分将介绍在 `MediaStreamTrack` 对象上启用此功能所需的扩展。在[RTCWEB-RTP]，[RTCWEB-AUDIO]和[RTCWEB-TRANSPORT]中描述了如何将媒体传输到对等体。

发送给另一个对等方的 `MediaStreamTrack` 将作为一个且唯一一个MediaStreamTrack向收件人显示。对等体被定义为支持该规范的用户代理。此外，发送方应用程序可以指示 `MediaStreamTrack` 所属的 `MediaStream` 对象。接收端相应的 `MediaStream` 对象将会被创建(如果不是已经存在)并且对应补充。

正如本文档之前所述，应用程序可以使用对象 `RTCRtpSender` 和 `RTCRtpReceiver` 来对 `MediaStreamTracks` 的传输和接收进行更细粒度的控制。

通道是 `MediaStream` 规范中考虑的最小单位。信道旨在被编码在一起以便传输，例如，RTP有效载荷类型。编解码器需要联合编码的所有通道必须位于同一个 `MediaStreamTrack` 中，编解码器应该能够编码或丢弃轨道中的所有通道。

对于给定 `MediaStreamTrack` 的输入和输出的概念也适用于通过网络传输的 `MediaStreamTrack` 对象。由 `RTCPeerConnection` 对象创建的MediaStreamTrack（如本文档前面所述）将把从远程对等方接收的数据作为输入。类似地，来自本地源的 `MediaStreamTrack` （例如通过[GETUSERMEDIA]的摄像机）将具有输出，如果该对象与RTCPeerConnection对象一起使用,该输出表示传输到远程对等端的内容。

[GETUSERMEDIA]中描述的复制 `MediaStream` 和 `MediaStreamTrack` 对象的概念也适用于此处。例如，该功能可用于视频会议场景中，以便在本地监视器中显示来自用户摄像头和麦克风的本地视频，同时仅将音频发送到远程对等端（例如，响应用户使用“视频静音“功能”。在某些情况下，将不同的 `MediaStreamTrack` 对象组合到新的 `MediaStream` 对象中非常有用。

> NOTE:在本文档中，我们仅指定与 `RTCPeerConnection` 一起使用时相关的以下对象的各个方面。有关使用 `MediaStream` 和 `MediaStreamTrack` 的一般信息，请参阅[GETUSERMEDIA]文档中对象的原始定义。

9.2 媒体流

9.2.1 id

`MediaStream` 指定的id属性返回了对流唯一的id，这样 `RTCPeerConnection API` 的远程端就可以识别媒体流。

当媒体流被创建用来表示从远程对等体获得的流时，id属性是通过远程源提供的信息初始化的。

> NOTE:MediaStream对象的id对于流的源是唯一的，但这并不意味着不可能以重复项结束。例如，本地生成的流的轨道可以使用 `RTCPeerConnection` 从一个用户代理发送到远程对等体，然后以相同的方式发送回原始用户代理，在这种情况下，原始用户代理将得到具有相同id的多条流（本地生成的id和从远程对等体发送的id）。

9.3.1 MediaTrackSupportedConstraints, MediaTrackCapabilities, MediaTrackConstraints 和 MediaTrackSettings

[GETUSERMEDIA]概述

了 `MediaTrackSupportedConstraints` ， `MediaTrackCapabilites` ， `MediaTrackConstraints` 和 `MediaTrackSettings` 的基础知识。然而，由 `RTCPeerConnection` 提供的 `MediaStreamTrack` 的 `MediaTrackSettings` 只会填充成员，以便通过由 `setRemoteDescription` 和实际RTP数据应用的远程 `RTCSessionDescription` 提供数据。这意味着某些成员（例如 `facingMode` ， `echoCancellation` ， `latency` ， `deviceId` 和 `groupId` ）将始终缺失。

9.3 媒体流轨道

`MediaStreamTrack` 对象在非本地媒体源案例中对其 `MediaStream` 的引用（RTP源，每个 `RTCRtpReceiver` 关联一个 `MediaStreamTrack` 的情况）总是很强。

每当 `RTCRtpReceiver` 在相应的MediaStreamTrack被静音的RTP源上接收数据，并且包含 `RTCRtpReceiver` 的 `RTCRtpTraceceiver` 对象的[[Receptive]]插槽为 `true` 时，它必须对任务排序以设置相应MediaStreamTrack的静音状态为 `false` 。

当RTCRtpReceiver接收到的RTP源媒体流的SSRC之一由于接收到BYE或超时而被移除时，它必须对任务排序以将相应MediaStreamTrack的静音状态设置为 `true` 。注意， `setRemoteDescription` 还可以将 `track` 的静音状态设置为值 `true` 。

在[GETUSERMEDIA]中指定了添加track，删除track和设置track静音状态的步骤。

当 `RTCRtpReceiver` 接收器生成的MediaStreamTrack轨道已经结束[GETUSERMEDIA]时（例如通过调用 `receiver.track.stop` ），用户代理可以选择释放为输入流分配的资源，例如通过关闭接收端解码器。

10 示例和呼叫流程

本章节不具有规范性。

10.1 单一点对点示例

当两个对等方决定彼此建立连接时，它们都会执行这些步骤。 STUN / TURN服务器配置描述了可用于获取公共IP地址或设置NAT穿透的服务器。他们还必须使用相同的带外机制相互发送信令信道的数据，此机制用于确定他们首先要进行通信。

```
const signaling = new SignalingChannel(); // handles JSON.stringify/parse
const constraints = {audio: true, video: true};
const configuration = {iceServers: [{urls: 'stuns:stun.example.org'}]};
const pc = new RTCPeerConnection(configuration);

// send any ice candidates to the other peer
pc.onicecandidate = ({candidate}) => signaling.send({candidate});

// let the "negotiationneeded" event trigger offer generation
pc.onnegotiationneeded = async () => {
  try {
    await pc.setLocalDescription(await pc.createOffer());
    // send the offer to the other peer
    signaling.send({desc: pc.localDescription});
  } catch (err) {
    console.error(err);
  }
};

// once media for a remote track arrives, show it in the remote video element
pc.ontrack = (event) => {
  // don't set srcObject again if it is already set.
  if (remoteView.srcObject) return;
  remoteView.srcObject = event.streams[0];
};

// call start() to initiate
async function start() {
  try {
    // get a local stream, show it in a self-view and add it to be sent
    const stream = await navigator.mediaDevices.getUserMedia(constraints);
    stream.getTracks().forEach((track) => pc.addTrack(track, stream));
    selfView.srcObject = stream;
  } catch (err) {
    console.error(err);
  }
}

signaling.onmessage = async ({desc, candidate}) => {
  try {
    if (desc) {
      // if we get an offer, we need to reply with an answer
      if (desc.type == 'offer') {
        await pc.setRemoteDescription(desc);
        const stream = await navigator.mediaDevices.getUserMedia(constraints);
        stream.getTracks().forEach((track) => pc.addTrack(track, stream));
        await pc.setLocalDescription(await pc.createAnswer());
        signaling.send({desc: pc.localDescription});
      } else if (desc.type == 'answer') {
        await pc.setRemoteDescription(desc);
      } else {
        console.log('Unsupported SDP type. Your code may differ here.');
      }
    } else if (candidate) {
      await pc.addIceCandidate(candidate);
    }
  } catch (err) {
    console.error(err);
  }
};
```

10.2 具有预热的高级对等示例

当两个对等方决定彼此建立连接并希望让ICE，DTLS和媒体连接“热身”以便他们准备立即发送和接收媒体时，他们都会完成这些步骤。

```
const signaling = new SignalingChannel();
const configuration = {iceServers: [{urls: 'stuns:stun.example.org'}]};
const audio = null;
const audioSendTrack = null;
const video = null;
const videoSendTrack = null;
const started = false;
let pc;

// Call warmup() to warm-up ICE, DTLS, and media, but not send media yet.
async function warmup(isAnswerer) {
  pc = new RTCPeerConnection(configuration);
  if (!isAnswerer) {
    audio = pc.addTransceiver('audio');
    video = pc.addTransceiver('video');
  }

  // send any ice candidates to the other peer
  pc.onicecandidate = (event) => {
    signaling.send(JSON.stringify({candidate: event.candidate}));
  };

  // let the "negotiationneeded" event trigger offer generation
  pc.onnegotiationneeded = async () => {
    try {
      await pc.setLocalDescription(await pc.createOffer());
      // send the offer to the other peer
      signaling.send(JSON.stringify({desc: pc.localDescription}));
    } catch (err) {
      console.error(err);
    }
  };

  // once media for the remote track arrives, show it in the remote video element
  pc.ontrack = async (event) => {
    try {
      if (event.track.kind == 'audio') {
        if (isAnswerer) {
          audio = event.transceiver;
          audio.direction = 'sendrecv';
          if (started && audioSendTrack) {
            await audio.sender.replaceTrack(audioSendTrack);
          }
        }
      } else if (event.track.kind == 'video') {
        if (isAnswerer) {
          video = event.transceiver;
          video.direction = 'sendrecv';
          if (started && videoSendTrack) {
            await video.sender.replaceTrack(videoSendTrack);
          }
        }
      }

      // don't set srcObject again if it is already set.
      if (remoteView.srcObject) return;
      remoteView.srcObject = event.streams[0];
    } catch (err) {
      console.error(err);
    }
  };

  try {
    // get a local stream, show it in a self-view and add it to be sent
    const stream = await navigator.mediaDevices.getUserMedia({audio: true, video: true
```

```
    selfView.srcObject = stream;
    audioSendTrack = stream.getAudioTracks()[0];
    if (started) {
      await audio.sender.replaceTrack(audioSendTrack);
    }
    videoSendTrack = stream.getVideoTracks()[0];
    if (started) {
      await video.sender.replaceTrack(videoSendTrack);
    }
  } catch (err) {
    console.erro(err);
  }
}

// Call start() to start sending media.
function start() {
  started = true;
  signaling.send(JSON.stringify({start: true}));
}

signaling.onmessage = async (event) => {
  if (!pc) warmup(true);

  try {
    const message = JSON.parse(event.data);
    if (message.desc) {
      const desc = message.desc;

      // if we get an offer, we need to reply with an answer
      if (desc.type == 'offer') {
        await pc.setRemoteDescription(desc);
        await pc.setLocalDescription(await pc.createAnswer());
        signaling.send(JSON.stringify({desc: pc.localDescription}));
      } else {
        await pc.setRemoteDescription(desc);
      }
    } else if (message.start) {
      started = true;
      if (audio && audioSendTrack) {
        await audio.sender.replaceTrack(audioSendTrack);
      }
      if (video && videoSendTrack) {
        await video.sender.replaceTrack(videoSendTrack);
      }
    } else {
      await pc.addIceCandidate(message.candidate);
    }
  } catch (err) {
    console.error(err);
  }
};
```

10.3 发信前的对等媒体示例

应答者可能希望发送媒体与发送答案并行进行，并且提议者可能希望在应答到来之前呈现媒体。

```
const signaling = new SignalingChannel();
const configuration = {iceServers: [{urls: 'stuns:stun.example.org'}]};
let pc;

// call start() to initiate
async function start() {
  pc = new RTCPeerConnection(configuration);

  // send any ice candidates to the other peer
  pc.onicecandidate = (event) => {
    signaling.send(JSON.stringify({candidate: event.candidate}));
  };

  // let the "negotiationneeded" event trigger offer generation
  pc.onnegotiationneeded = async () => {
    try {
      await pc.setLocalDescription(await pc.createOffer());
      // send the offer to the other peer
      signaling.send(JSON.stringify({desc: pc.localDescription}));
    } catch (err) {
      console.error(err);
    }
  };

  try {
    // get a local stream, show it in a self-view and add it to be sent
    const stream = await navigator.mediaDevices.getUserMedia({audio: true, video: true
    selfView.srcObject = stream;
    // Render the media even before ontrack fires.
    remoteView.srcObject = new MediaStream(pc.getReceivers().map((r) => r.track));
  } catch (err) {
    console.error(err);
  }
};

signaling.onmessage = async (event) => {
  if (!pc) start();

  try {
    const message = JSON.parse(event.data);
    if (message.desc) {
      const desc = message.desc;

      // if we get an offer, we need to reply with an answer
      if (desc.type == 'offer') {
        await pc.setRemoteDescription(desc);
        await pc.setLocalDescription(await pc.createAnswer());
        signaling.send(JSON.stringify({desc: pc.localDescription}));
      } else {
        await pc.setRemoteDescription(desc);
      }
    } else {
      await pc.addIceCandidate(message.candidate);
    }
  } catch (err) {
    console.error(err);
  }
};
```

10.4 同时联播示例

客户端想要向服务器发送多个RTP编码（同时广播）。

```
const signaling = new SignalingChannel();
const configuration = {'iceServers': [{'urls': 'stuns:stun.example.org'}]};
let pc;

// call start() to initiate
async function start() {
  pc = new RTCPeerConnection(configuration);

  // let the "negotiationneeded" event trigger offer generation
  pc.onnegotiationneeded = async () => {
    try {
      await pc.setLocalDescription(await pc.createOffer());
      // send the offer to the other peer
      signaling.send(JSON.stringify({desc: pc.localDescription}));
    } catch (err) {
      console.error(err);
    }
  };

  try {
    // get a local stream, show it in a self-view and add it to be sent
    const stream = await navigator.mediaDevices.getUserMedia({audio: true, video: true
    selfView.srcObject = stream;
    pc.addTransceiver(stream.getAudioTracks()[0], {direction: 'sendonly'});
    pc.addTransceiver(stream.getVideoTracks()[0], {
      direction: 'sendonly',
      sendEncodings: [
        {rid: 'f'},
        {rid: 'h', scaleResolutionDownBy: 2.0},
        {rid: 'q', scaleResolutionDownBy: 4.0}
      ]
    });
  } catch (err) {
    console.error(err);
  }
}

signaling.onmessage = async (event) => {
  try {
    const message = JSON.parse(event.data);
    if (message.desc) {
      await pc.setRemoteDescription(message.desc);
    } else {
      await pc.addIceCandidate(message.candidate);
    }
  } catch (err) {
    console.error(err);
  }
};
```

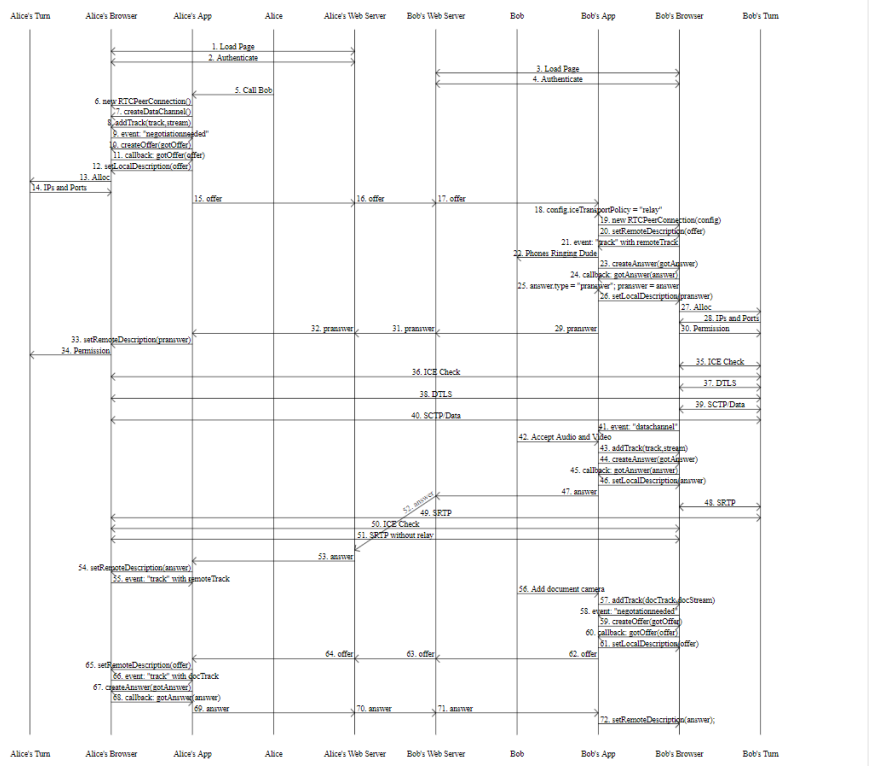
10.5 点对点数据示例

此示例显示如何创建 `RTCDataChannel` 对象并执行将通道连接到其他对等方所需的提供/应答交换。 `RTCDataChannel` 用于简单聊天应用程序的上下文中，当通道准备就绪，收到消息以及关闭通道时，监听器将连接到监视器。

```
const signaling = new SignalingChannel(); // handles JSON.stringify/parse
const configuration = {iceServers: [{urls: 'stuns:stun.example.org'}]};
let pc;
let channel;

// call start(true) to initiate
function start(isInitiator) {
  pc = new RTCPeerConnection(configuration);

  // send any ice candidates to the other peer
  pc.onicecandidate = (candidate) => {
    signaling.send({candidate});
  };

  // let the "negotiationneeded" event trigger offer generation
  pc.onnegotiationneeded = async () => {
    try {
      await pc.setLocalDescription(await pc.createOffer());
      // send the offer to the other peer
      signaling.send({desc: pc.localDescription});
    } catch (err) {
      console.error(err);
    }
  };

  if (isInitiator) {
    // create data channel and setup chat
    channel = pc.createDataChannel('chat');
    setupChat();
  } else {
    // setup chat on incoming data channel
    pc.ondatachannel = (event) => {
      channel = event.channel;
      setupChat();
    };
  }
}

signaling.onmessage = async ({desc, candidate}) => {
  if (!pc) start(false);

  try {
    if (desc) {
      // if we get an offer, we need to reply with an answer
      if (desc.type == 'offer') {
        await pc.setRemoteDescription(desc);
        await pc.setLocalDescription(await pc.createAnswer());
        signaling.send({desc: pc.localDescription});
      } else {
        await pc.setRemoteDescription(desc);
      }
    } else {
      await pc.addIceCandidate(candidate);
    }
  } catch (err) {
    console.error(err);
  }
};

function setupChat() {
  // e.g. enable send button
  channel.onopen = () => enableChat(channel);
  channel.onmessage = (event) => showChatMessage(event.data);
}
```

10.6 两个浏览器之间的呼叫流程

这显示了两个浏览器之间可能的一个呼叫流程的示例。这不会展示访问本地媒体或每个被触发的回调的过程，而是尝试将其减少为仅显示关键事件和消息。



10.7 DTMF示例

示例假设发送方是 `RTCRtpSender` 。

发送DTMF信号“1234”，每个tone持续时间为500毫秒：

```
if (sender.dtmf.canInsertDTMF) {
  const duration = 500;
  sender.dtmf.insertDTMF('1234', duration);
} else {
  console.log('DTMF function not available');
}
```

发送DTMF信号“123”并在发送“2”后中止。

```
async function sendDTMF() {
  if (sender.dtmf.canInsertDTMF) {
    sender.dtmf.insertDTMF('123');
    await new Promise((r) => sender.dtmf.ontonechange = (e) => e.tone == '2' && r());
    // empty the buffer to not play any tone after "2"
    sender.dtmf.insertDTMF('');
  } else {
    console.log('DTMF function not available');
  }
}
```

发送DTMF信号“1234”，并在播放音调时使用lightKey（键）点亮活动键（假设lightKey（“”）将使所有键变暗）：

```
const wait = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

if (sender.dtmf.canInsertDTMF) {
  const duration = 500;
  sender.dtmf.insertDTMF(sender.dtmf.toneBuffer + '1234', duration);
  sender.dtmf.ontonechange = async (event) => {
    if (!event.tone) return;
    lightKey(event.tone); // light up the key when playout starts
    await wait(duration);
    lightKey(''); // turn off the light after tone duration
  };
} else {
  console.log('DTMF function not available');
}
```

附加到音调缓冲区始终是安全的。此示例在任何音色播放开始之前以及播放期间附加。

```
if (sender.dtmf.canInsertDTMF) {
  sender.dtmf.insertDTMF('123');
  // append more tones to the tone buffer before playout has begun
  sender.dtmf.insertDTMF(sender.dtmf.toneBuffer + '456');

  sender.dtmf.ontonechange = (event) => {
    if (event.tone == '1') {
      // append more tones when playout has begun
      sender.dtmf.insertDTMF(sender.dtmf.toneBuffer + '789');
    }
  };
} else {
  console.log('DTMF function not available');
}
```

发送1秒“1”音，然后发出2秒“2”音：

```
if (sender.dtmf.canInsertDTMF) {
  sender.dtmf.ontonechange = (event) => {
    if (event.tone == '1') {
      sender.dtmf.insertDTMF(sender.dtmf.toneBuffer + '2', 2000);
    }
  };
  sender.dtmf.insertDTMF(sender.dtmf.toneBuffer + '1', 1000);
} else {
  console.log('DTMF function not available');
}
```

11 错误处理

某些操作抛出或引起 `RTCError` 。这是 `DOMException` 的扩展，包含了额外的WebRTC信息。

11.1 `RTCError` 接口

```
[
    Exposed=Window,
    Constructor(RTCErrorInit init, optional DOMString message = "")
] interface RTCError /*: DOMException*/ {
    readonly attribute RTCErrorDetailType errorDetail;
    readonly attribute long? sdpLineNumber;
    readonly attribute long? httpRequestStatusCode;
    readonly attribute long? sctpCauseCode;
    readonly attribute unsigned long? receivedAlert;
    readonly attribute unsigned long? sentAlert;
};
```

11.1.1 构造函数

`RTCError`

运行下列步骤：

1. 让init成为构造函数的第一个参数。
2. 让message成为构造函数的第二个参数。
3. 让e成为新的 `RTCError` 对象。
4. 将 `message` 参数设置为message， `name` 参数设置为 `“RTCError”` ，触发e的 `DOMException` 构造函数。

   > NOTE:这个name不具有对legacy的映射，因此e的 `code` 属性返回0。

5. 设置e的所有 `RTCError` 属性为init中的对应属性的值，如果存在，否则设置为 `null` 。测试：1
6. 返回e。

11.1.2 属性

RTCErrorDetailType类型的 `errorDetail` ，只读：针对出现错误类型的WebRTC特定错误代码。

测试：2

测试：2

long类型的 `sdpLineNumber` ,只读，可以为null：如果 `errorDetail` 为 `“sdp-syntax-error”` ,这是错误被检测到的行号(第一行行号为1)。

long类型的 `httpRequestStatusCode` ,只读，可以为null：如果 `errorDetail` 为 `“idp-load-failure”` ，这是IdP URI回应的HTTP状态码。

long类型的 `sctpCauseCode` ,只读，可以为null：如果 `errorDetail` 是 `“actp-failure”` ，这是SCTP协商失败的SCTP原因代码。

unsigned long类型的 `receiveAlert` ,只读，可以为null：如果 `errorDetail` 是 `“dtls-failure”`，并且接收到致命的DTLS警报，这是接收到的DTLS警报的值。

unsigned long类型的 `sentAlert` ，只读，可以为null：如果 `errorDetail` 是 `"dtls-failure"`，并且发送了 致命的DTLS警报，这是发送的DTLS警报值。

RTCErrorInit字典

```
dictionary RTCErrorInit {
    required RTCErrorDetailType errorDetail;
    long sdpLineNumber;
    long httpRequestStatusCode;
    long sctpCauseCode;
    unsigned long receivedAlert;
    unsigned long sentAlert;
};
```

`RTCErrorInit` 的 `errorDetail` , `sdpLineNumber` , `httpRequestStatusCode` , `sctpCauseCode` , `receivedAlert` 和 `sentAlert` 成员与 `RTCError` 的同名属性具有相同的定义。

11.1.3 字典 `RTCError` 成员

RTCErrorDetailType类型的 `errorDetail` ， 必需的：查看 `RTCError` 的 `errorDetail` 。

long类型的 `sdpLineNumber` :查看 `RTCError` 的 `sdpLineNumber` 。

long类型的 `httpRequestStatusCode` :查看 `RTCError` 的 `httpRequestStatusCode` 。

long类型的 `sctpCauseCode` :查看 `RTCError` 的 `sctpCauseCode` 。

unsigned long类型的 `receivedAlert` :查看 `RTCError` 的 `receivedAlert` 。

unsigned long类型的 `sentAlert` ： 查看 `RTCError` 的 `sentAlert` 。

11.1.4 `RTCErrorDetailType` 枚举

```
enum RTCErrorDetailType {
              "data-channel-failure",
              "dtls-failure",
              "fingerprint-failure",
              "idp-bad-script-failure",
              "idp-execution-failure",
              "idp-load-failure",
              "idp-need-login",
              "idp-timeout",
              "idp-tls-failure",
              "idp-token-expired",
              "idp-token-invalid",
              "sctp-failure",
              "sdp-syntax-error",
              "hardware-encoder-not-available",
              "hardware-encoder-error"
          };
```

名称以“idp”为前缀的错误详细信息类型由[WEBRTC-IDENTIFY]规范使用。这里描述了它们，因为WebIDL枚举必须只能在一个地方描述。

枚举描述	
`data-channel-failure`	数据通道已经失败。
`dtls-failure`	DTLS协商已经失败，或者连接由于致命错误被中断。 `message` 包含关于error的信息。如果接收到致命DTLS警报， `receivedAlert` 属性被设置为接收到的DTLS警报。如果发送了致命TLS警报， `sentAlert` 属性被设置为发送的DTLS警报的值。
`fingerprint-failure`	`RTCDtlsTransport` 的远程证书与SDP提供的指纹不匹配。如果远程对等体不能将本地证书与提供的指纹匹配，则不会生成此错误。可能会从远程对等体接收到一个"bad-certificate" DTLS警报，这会导致“dtls-failure”。
`idp-bad-script-failure`	从身份提供方加载的脚本不是有效的JavaScript或没有实现正确的接口。
`idp-execution-failure`	身份提供方抛出异常或返回了rejected的承诺。
`idp-load-failure`	Idp URI的加载已经失败。 `httpRequestStatusCode` 属性被设置为回应的HTTP状态码。
`idp-need-login`	身份提供方需要用户登录。 `idpLoginUrl` 属性被设置为可以被用来登录的URL。
`idp-timeout`	Idp timer已经失效。
`idp-tls-failure`	用户Idp HTTPS连接的TLS证书不受信任。
`idp-token-expired`	Idp token已经失效。
`idp-token-invalid`	Idp token是无效的。
`sctp-failure`	SCTP协商已经失败，或者连接由于致命错误而被终止。 `sctpCauseCode` 属性被设置为SCTP原因码。
`sdp-syntax-error` 测试:1	SDP语法无效。 `sdpLineNumber` 属性被设置为SDP中检测到语法错误的行号。
`hardware-encoder-not-available`	请求的操作所需的硬件编码器资源不可用。
`hardware-encoder-error`	硬件编码器不支持提供的参数。

11.2 `RTCErrprEvent` 接口

`RTCErrorEvent` 接口是针对将 `RTCError` 作为事件引发的情况定义的:

```
[
    Exposed=Window,
    Constructor (DOMString type, RTCErrorEventInit eventInitDict)
] interface RTCErrorEvent : Event {
    [SameObject] readonly attribute RTCError error;
};
```

11.2.1 构造函数

`RTCErrorEvent` :测试：1

构造一个新的 `RTCErrorEvent` 。

11.2.2 属性

`RTCError` 类型的error，只读，可以为null：

描述触发事件的错误的 `RTCError` 。

11.3 `RTCErrorEventInit` 字典

```
dictionary RTCErrorEventInit : EventInit {
     required RTCError error;
};
```

11.3.1 字典RTCErrorEventInit成员

`RTCError` 类型的 `error` ,默认为 `null`：

描述与事件(如果存在)关联的错误的 `RTCError` 。

12 事件总结

本章节不具有规范性。

下列事件触发 `RTCDataChannel` 对象：

事件名称	接口	触发当...
`open`	Event	`RTCDataChannel` 对象的数据传输已经建立(或重立)。
`message`	MessageEvent[webmessaging]	成功接收到一条信息
`bufferedamountlow`	Event	`RTCDataChannel` 对象的 `bufferedAmount` 从它的 `bufferedAmountLowT` 值降低到小于等于它的 `bufferedAmountLowT` 值。
`error`	`RTCErrorEvent`	数据通道上产生了错
`close`	Event	`RTCDataChannel` 对象的数据传输已经关闭。

下列事件触发 `RTCPeerConnection` 对象：

事件名称	接口	触发当
`track`	`RTCTrackEvent`	新进入媒体已经 体的 `RTCRtpRece` 收方的track已经 到任何相关联的 流中。
`negotiationneeded`	Event	浏览器希望通知 需要协商(换句话 createOffer调用 用setLocalDesc
`signalingstatechange`	Event	发信状态已经改 由于触发 了 `setLocalDescr` 者 `setRemoteDesc`
`iceconnectionstatechange`	Event	`RTCPeerConnecti` 连接状态已经改
`icegatheringstatechange`	Event	`RTCPeerConnecti` 收集状态已经改
`icecandidate`	`RTCPeerConnectionIceEvent`	一个新 的 `RTCIceCandida` 可用。
`connectionstatechange`	Event	`RTCPeerConnecti` 态已经改变。
`icecandidateerror`	`RTCPeerConnectionIceErrorEvent`	收集ICE候选者 失败。
`datachannel`	`RTCDataChannelEvent`	新的 `RTCDataChan` 配给脚本，作为 体创建通道的回
`isolationchange`	Event	当 `MediaStreamTr` isolated属性改 事件被分配给脚
`statsended`	`RTCStatsEvent`	新的 `RTCStatsEve` 配给脚本，作为 一个或多个受控 应。

下列事件触发 `RTCDTMFSender` 对象：

事件名称	接口	触发当...
`tonechange`	`RTCDTMFToneChangeEvent`	`RTCDTMFSender` 对象或是刚刚开始播放tone(作为tone属性返回)，或是刚刚结束对 `toneBuffer` 中的tone的播放(在 `tone` 属性中作为空值返回)。

下列事件触发 `RTCIceTransport` 对象：

事件名称	接口	触发当...
`statechange`	Event	`RTCIceTransport` 状态改变。
`gatheringstatechange`	Event	`RTCIceTransport` 收集状态改变。
`selectedcandidatepairchange`	Event	`RTCIceTransport` 选定的候选者对发生改变。

下列事件触发 `RTCDtlsTransport` 对象：

事件名称	接口	触发当...
`statechange`	Event	`RTCDtlsTransport` 状态改变。
`error`	`RTCErrorEvent`	`RTCDtlsTransport` 上发生了一个错误(dtls-error或是fingerprint-failure)。

下列事件触发 `RTCSctpTransport` 对象：

事件名称	接口	触发当...
`statechange`	Event	`RTCSctpTransport` 状态改变。

13 隐私安全考虑

此章节不具有规范性。

本章不具有规范性；它没有介绍新的行为，但是总结了规范的其它部分已经存在的信息。[RTCWEB-SECURITY-ARCH]中描述了对于一般在WebRTC中使用的API和协议的安全性考虑。

13.1 同源政策的影响

本文档扩展了Web平台，能够在浏览器和其它设备之间(包括其它浏览器)建立实时，直接的通信。

这意味着数据和媒体可以在运行在不同的浏览器中的应用程序之间共享，也可以在运行在同一浏览器中的应用程序和不是浏览器的应用程序之间共享。

WebRTC规范对于通信不提供用户提示或Chrome指示。它假设一旦允许网页访问媒体，就可以自由的与其它实体共享该媒体。因此可以在没有任何用户明确同意或参与的情况下进行数据查看WebRTC数据通道的对等交换，类似于以服务器为媒介的交换(例如，通过Web Sockets)可以在没有用户参与情况下发生。

peeridentity机制从充当身份提供者的第三方服务器加载运行JavaScript代码。该代码在单独的JS领域中运行，不会影响同一原始策略提供的保护。

13.2 揭示IP地址

即使没有WebRTC，提供Web应用程序的Web服务器也会知道应用程序交付的公共IP地址。设置通讯向Web应用公开关于浏览器网络环境的额外信息，并且可能包括浏览器可用于WebRTC的一组IP地址(可能是私有的)。必须将其中一些信息传递给相应方，以便建立通信会话。

揭示IP地址可能会泄露位置和连接方式；这是敏感的。根据网络环境，这还会增加增加指纹表面并创建持久的跨源状态，用户无法轻易清除。

连接将始终显示建议用于与对等方通信的IP地址。应用程序可以通过选择不使用某些地址，这些地址使用 `RTCIceTransportPolicy` 字典公开的设置，以及使用中继(例如，TURN服务器)而不是参与者之间的直接连接，来限制这种暴露。通常会假设TURN服务器的IP地址不是敏感信息。例如，这些选择可以由应用程序基于用户是否已经同意开始与另一方媒体连接来作出。

减少将IP地址暴露给应用程序本身需要限制可以使用的IP地址，这会影响端点之间的最直接路径上进行通信的能力。浏览器被鼓励基于用户所需的安全性提供适当的控制来决定哪些IP地址可供应用程序使用。选择公开哪个地址是由本地策略控制的(有关详细信息，请参阅[RTCWEB-IP-HANDLING])。

13.3 对本地网络的影响

由于浏览器是在受信任的网络环境(防火墙内)执行的活动平台，因此限制浏览器可以对本地网络中的其它元素造成的损害非常重要，保护数据免受拦截，操纵非常重要，和被不受信任的参与者更改。

措施如下：

- 用户代理将始终请求相应的用户代理的许可使用ICE进行通信。这确保用户代理只能向具有共享证书的合作方发送数据。
- 用户代理将始终使用ICE continued consent持续请求许可，以继续发送数据，这确保接受者可以撤回同意接收的操作。
- 用户代理将始终使用强大的per-session密钥(DTLS-SRTP)对数据加密。
- 用户代理将始终使用拥塞控制。这可以确保WebRTC不能被用于堵塞网络。

这些方法在相关IETF文件中详细列出。

13.4 通信的保密性

通信正在发生的事实不能从可以观察网络的对手那里隐藏，因此必须将其视为公共信息。

一种叫peerIdentity的机制，为JavaScript提供了请求相同JS无法访问的媒体的选择，但只能发送给其它特定实体。

13.5 WebRTC公开的持久信息

如上所述，WebRTC API公开的IP地址列表可以用作持久的跨源状态。

除IP地址以外，WebRTC API通过 `RTCRtpSender.getCapabilities` 和 `RTCRtpReceiver.getCapabilities` 方法公开有关底层媒体系统的信息，包括系统可以生成和使用的编解码器的详细和有序信息。该信息的一部分可能在会话协商期间生成，公开和传输的SDP会话描述中表示。该信息大多数情况下，跨时间和跨源是持久的，并且增加了给定设备的指纹表面。

如果设置，由 `RTCPeerConnection` 实例上的 `getDefaultIceServers` 公开的配置的默认ICE服务器也提供跨时间跨源信息，这增加了给定浏览器的指纹表面。

建立DTLS连接时，WebRTC API可以生成可由应用程序持久保存的证书(例如在IndexedDB中)。这些证书不是跨源共享的，当该源的持久存储被清除时，它同时被清除。