WebRTC 1.0
浏览器间实时通讯

W3C 推荐标准

网易云信

30分钟构建流畅清晰的视频、音频通话

1天实现稳定的APP私聊、群聊、聊天室功能

网易云信核心能力

IM云

单聊
群聊
多人大群
聊天室
自定义消息扩展

音视频通话

音频通话
视频通话
点对点通话
多对多通话
录制

音视频通话

视频采集
短视频采集编辑
视频播放
CDN分发
频道管理
录制美颜、伴音等

点播

媒体存储
断点续传
视频转码
播放器
CDN加速
视频管理
短视频秒开管理

短信码通道

短信码验证
模板短信
营销短信
多通道切换
短信统计

专线电话

多人通话
多方通话
分钟统计
拨打查询

WebRTC1.0: 浏览器间的实时通信

1.介绍

本规范涵盖了对等通信（peer-to-peer communications）和网页视频会议的多个方面：

- 利用ICE，STUN，TURN等NAT穿透技术连接至远程对等终端。
- 将本地产生的媒体流发送至对端并接收对端产生的媒体流。
- 直接向远程对等终端发送任意数据。

本文档针对这些特性定义了一些API。本规范是在IETF RTCWEB工作组的协议规范，和Media Capture Task Force工作组的访问本地媒体设备API规范[GEtUSERMEDIA]，两者共同催动下发展出来的。系统的概览可以在引用列表中的RTCWEB-OVERVIW和RTCWEB-SECURITY找到。

2.一致性

关键词`MAY, MUST, MUST NOT, SHALL, SHOULD`的语义解释在RFC2119中都有描述。 本规范定义了适用于某个产品的一致性标准：**用户代理**应实现规范包含的接口。 一致性的要求可以被表述为一些算法，或被实现为任意行为的特定步骤，只要它们最终的结果是等效的。（特别地，规范中定义的算法更易理解，但性能也许不尽如人意。）本规范中定义的API，必须（MUST）以WEBIDL中规定的行为一致的ECMAScript绑定的方式实现，毕竟我们使用了它们的规范与术语。

3.术语

EventHandler接口代表了一个事件回调，ErrorEvent接口定义在HTML51
任务入队（queue a task）和网络任务源（networking task source）的概念定义在HTML51。
构造事件（fire an event）的概念定义在DOM。
事件（event），事件句柄（event handler）和事件句柄类型（event handler event types）定义在HTML51。
performance.timeOrigin和performance.now()定义在HIGHRES-TIME。
可序列化对象（serializable objects），序列化步骤（serialization step），反序列化步骤（deserialization steps）定义在HTML。
媒体流（MediaStream），**媒体流轨**（MediaStreamTrack），**媒体流约束**（MediaStreamConstraints）定义在GETUSERMEDIA。
Blob 定义在FILEAPI。
媒体描述（media description）定义在RFC4566。
媒体传输（media transport）定义在RFC7656。
地址（generation）定义在TRICKLE-ICE的第二节。
RTCStatsType，stats object和monitored object 定义在WEBRTC-STATS。
当引入异常时，WEBIDL-1中定义了 **throw和create**。
"throw"作为INFRA中的规定来使用：它会终止目前正在运行的操作。
`Promises`的上下文中使用的 **fulfilled, rejected, resolved, pending和settled** 在ECMASCRIPT-6.0中定义。
捆绑（bundle），**只捆绑**（bundle-only）和**捆绑策略**（bundle-policy）在JSEP中定义。
OAuth客户端（OAuth Client）和**授权服务**（Authorization Server）在RFC6749的1.1节被定义。
隔离流（isolated stream），**对等身份**（peer identity），**身份声明请求**（request an identity assertion）和**身份认证**（validate the identity）在WEBRTC-IDENTITY中定义。

> 注意： 通常使用Javascript API的原则包括：`同步运行`和`数据独立`，它们都在API-DESIGN-PRINCIPLES中定义了。也就是说，当一个任务正在运行时，任何外部事件都不会影响Javascript应用的可见性。例如，当Javascript执行时，缓存在数据通道里的数据数量将会随着"send"的调用而增长，并且直到任务的检查点之后，由于发送数据包导致的减少才被应用可见。 用户代理负责确保呈现给应用程序的的数据是一致的——例如`getContributingSources()`（同步调用）会返回当前所有被检测的数据源的值。

4. 对等连接

4.1 介绍

一个RTCPeerConnection实例允许与另一个浏览器，或实现了制定协议的终端中的`RTCPeerConnection`实例建立对等通信。双方在信令通道中通过控制消息（即自定义的信令协议）协商会话，信号通道并没有明确的制定，但通常是服务页面中的一段脚本，例如XMLHttpRequest，也可以是WebSockets。

4.2 配置

4.2.1 `RTCConfiguration`字典

`RTCConfiguration`定义了一系列用于配置如何通过`RTCPeerConnection`建立/重建对等通信的参数。

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

`RTCConfiguration`字典成员变量：

- sequence类型的`iceServers`：描述可供ICE使用的服务对象数组，例如STUN服务和TURN服务。
- RTCIceTransportPolicy类型的`iceTransportPolicy`，缺省值为"all"：指示哪个候选`ICE Agent`可用。
- RTCBundlePolicy类型的`bundlePolicy`，缺省值为"balanced"：当收集候选ICE时指示使用什么媒体捆绑策略。
- RTCRtcpMuxPolicy类型的`rtcMuxPolicy`，缺省值为"require"：当收集候选ICE时指示使用什么RTCP复用策略。
- DOMString类型的`peerIdentity`：为RTCPeerConnection设置目标对等终端的身份。只有成功地对身份进行鉴权，RTCPeerConnection才能与远程对等终端建立起连接。
- sequence类型的`certificates`：RTCPeerConnection鉴权时所需的一系列证书。
  此参数的合法值通过调用`generateCertificate`函数得到。
  尽管任意给定的DTLS连接只会使用一份证书，但这一属性使得调用方可以供应多种证书以支持不同的算法。在DTLS连接的握手阶段，它会最终选择一份允许范围内的证书。RTCPeerConnection的具体实现中完成了对给定连接的证书选择过程，但证书是如何选择的并不在本规范的讨论范围之内。
  如果值为空，则每个RTCPeerConnection实例都会生成默认的证书集合。
  此选项还使得应用的密钥连续性成为可能。一个`RTCCertificate`可以被持久化存储在INDEXEDDB中并被复

用。持久化和复用避免了密钥重复生成的开销。
此配置选项的值在初始化阶段被选择后就不能再被改变。

- `octet` 类型的 `iceCandidatePoolSize` ，缺省值为 `0` ：预先获取的ICE池的大小在JSEP的第3.5.4节和4.1.1节被定义。

4.2.2 `RTCIceCredentialType` 枚举值

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

枚举值简述：

- password：此凭据是依托于用户名和密码的长期认证方式，RFC5389的10.2节有详细描述
- oauth：一个基于OAuth2.0的认证方法，在RFC7635有描述。
  对于OAuth认证，需要向ICE Agent供应3份证书信息： `kid` （用于RTCIceServer成员变量username），`macKey` 和 `accessToken` （存在于RTCOAuthCredential字典类型内）。
  注意：本规范并没有定义应用（起OAuth Client的作用）是如何从 `Authorization Server` 获取 `accessToken, kid, macKey` 这些证书的，因为WebRTC只处理ICE Agent与TURN Server之间的交互。例如，应用可能使用PoP（Proof-of-Possession）的Token证书类型，使用OAuth 2.0隐式授权类型。RFC的附录B中有此示例。
  OAuth Client应用，负责刷新证书信息，并且在 `accessToken` 失效前利用新的证书信息更新ICE Agent。OAuth Client可以利用RTCPeerConnection的setConfiguration方法来周期性的刷新TURN证书。
  HMAC密钥（RTCOAuthCredential.macKey）的长度应是一个大于20字节的整数（160位）。
  注意：根据RFC7635 4.1节，HMAC密钥必须是对称密钥，但对称密钥会生成□大型的访问令牌，可能和单个STUN信息不兼容。
  注意：目前的STUN/TURN协议只是用了SHA-1/SHA-2族哈希算法来保证消息完整性，这在[RFC5389]的15.3节和[STUN-BIS]的14.6节作了定义。

4.2.3 `RTCOAuthCredential` 字典

`RTCOAuthCredential` 字典被STUN/TURN客户端（内置于ICE Agent内）用于描述OAuth的鉴权证书信息，对STUN/TURN服务器进行身份认证，RFC7635有相关描述。注意 `kid` 参数并不在此字典类型中，而在 `RTCIceServer` 的 `username` 成员变量中。

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

`RTCOAuthCredential` 字典的成员变量：

- DOMString类型的 `macKey` ，非空："mac_key"是一串base64-url格式的编码，在RFC7635的6.2节有相关描述。它被用在STUN的消息完整性哈希计算中（密码使用的则是基于密码的认证方式）。注意，OAuth响应里的"key"参数是一个JSON Web Key（JWK）或JWK编码后JWE格式的消息。同样注意，这是OAuth中唯一一个不被直接使用的参数，它只能从JWK的"k"参数中提取出来，"k"参数包含了需要的base-64编码的"mac_key"。
- DOMString类型的 `accessToken` ，非空："access_token"是一串base64格式的编码，在RFC7635的6.2节有相关描述。这是一个自持有的令牌，应用不可见。认证加密被用于消息的加密和完整性保护。访问令牌包括了一

个未加密的nonce值，供认证服务生成唯一的`mac_key`。令牌的第二部分由认证加密服务保护着，包括mac_key，时间戳和生存时间。时间戳和生存时间共同组成了过期信息，过期信息描述了令牌证书合法且能被TURN服务接受的时间窗口。

RTCOAuthCredential字典的一个例子：

```
// EXAMPLE 1
{
  macKey: 'WmtzanB3ZW9peFhtdm42NzUzNGO=',
  accessToken:
'AAwg3kPHWPfvk9bDFL936wYvkoctMADzQ5VhNDgeMR3+ZlZ35byg972fW8QjpEl7bx91YLBPFsIhsxloWcXPhA==
'
}
```

4.2.4 `RTCIceServer`字典

`RTCIceServer`字典被ICE Agent用来描述和对等终端建立连接的STUN/TURN服务器信息。

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

`RTCIceServer`字典的成员变量：

- DOMString或sequence类型的`urls`，非空：[RFC7064]和[RFC7065]中定义的STUN/TURN的URI(s)，或其他的URI类型。
- DOMString类型的`username`：如果`RTCIceServer`代表了一个TURN服务器，且`credentialType`是`"password"`，那么这一属性指定的是TURN服务器使用的用户名。
  如果`RTCIceServer`代表了一个TURN服务器，且`credentialType`是`"oauth"`，那么这一属性指定的是TURN服务器和认证服务器之间共享的对称密钥的密钥id，RFC7635有相关描述。这是一个短暂且唯一的密钥标识符。`kid`允许TURN服务器选择合适的密钥材料对访问令牌进行解密，因此以`kid`为代表的密钥标识符被用于"access_token"的加密。`kid`值和OAuth响应中的"kid"参数相同，这被定义在RFC7515的4.1.4节。
- DOMString或RTCOAuthCredential类型的`credential`：如果`RTCIceServer`代表了一个TURN服务器，那么这一属性指定的是TURN服务器使用的证书。
  如果`credentialType`是`"password"`，那么`credential`是`DOMString`类型，代表了长期使用的认证密码，这在RFC5389的10.2节有相关描述。
  如果`credentialType`是`"oauth"`，那么`credential`是`RTCOAuthCredential`类型，包含了OAuth访问令牌和MAC值。
- RTCIceCredentialType类型的`credentialType`，默认值为"password"：如果`RTCIceServer`代表了一个TURN服务器，那么这一属性在TURN服务器需要认证客户端时使用。

一个RTCIceServer对象数组的例子：

```
[
  {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: 'WmtzanB3ZW9peFhtdm42NzUzNGO=',
      accessToken:
'AAwg3kPHWPfvk9bDFL936wYvkoctMADzQ5VhNDgeMR3+ZlZ35byg972fW8QjpEl7bx91YLBPFsIhsxloWcXPhA==
'
    },
    credentialType: 'oauth'}
];
```

4.2.5 `RTCIceTransportPolicy`枚举值

如JSEP4.1.1节所定义，如果`RTCConfiguration`的`iceTransportPolicy`成员被指定，它将指示浏览器获取ICE候选地址的策略，定义在JSEP 3.5.3节，浏览器只会收集特定的候选地址用于连接性检查。

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

枚举值的非规范描述：

- relay：ICE Agent仅获取媒体中继ice候选地址，例如通过TURN服务器传递的ice候选地址。**注意：该配置表明只收集中继服务器分配的ice候选地址，这可以在某些特定场景下防止远程终端获取该用户的真实IP地址。例如，在一个基于"调用"的应用中，应用可能想防止某个未知的调用者获得被调用方得IP地址，除非被调用方以某些同意。**
- all：当被指定为"all"时，ICE Agent可以使用任意类型的候选地址。**注意：在具体实现中，仍然可以使用自己的候选地址过渡策略来限制暴露给应用的IP地址，这在RTCIceCandidate.address中有提到。**

4.2.6 `RTCBundlePolicy`枚举值

如JSEP 4.1.1节提到，如果远程端点不支持捆绑，则捆绑策略会影响哪些媒体轨参与协商，以及哪些ICE候选地址被收集。如果远程端点支持捆绑，所有媒体轨和数据通道都会被捆绑到同一传输路径上。

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

枚举值的非规范描述：

- balanced：为所有正在使用中的媒体类型（音频，视频和数据）收集ICE候选地址。如果远程端点不支持捆绑，则只会为每个独立的传输协商一个音频或视频。
- max-compat：为每个流媒体轨收集ICE候选地址。如果远程端点不支持捆绑，为每个独立传输协商所有的媒体轨。
- max-bundle：只为一个媒体轨收集ICE候选地址。如果远程端点不支持捆绑，只协商一个媒体轨。

4.2.7 `RTCRtcpMuxPolicy`枚举值

如JSEP 4.1.1节中描述的，RtcpMuxPolicy会影响ICE候选地址收集哪些内容以支持非多路复用RTCP。

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

枚举值的非规范描述：

- negotiate：同时收集RTP候选地址和RTCP候选地址。如果远程端点能够复用RTCP，则在RTP候选地址之上复用RTCP。如果不能，独立地使用RTP和RTCP候选地址。注意，JSEP 4.1.1节提到，用户代理可能没有实现不复用的RTCP，在这种情况下所有试图以`negotiate`策略构造`RTCPeerConnection`的操作都会被拒绝。
- require：只收集RTC候选地址和在RTP基础上复用了RTCP的候选地址。如果远程端点不支持rtcp复用，那么会话协商将失败。

> 风险特征：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，因为实现者没有明确的承诺。包括：1. 对于`negotiate`值，实现者没有明确承诺与此相关的行为。 2.在`RTCRtpSender`和`RTCRtpReceiver`之内支持`rtcpTransport`属性。

4.2.8 邀请/应答选项

这些字典类型描述了可用于邀请/应答创建过程的选项。

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

`RTCOfferAnswerOptions`成员变量：

- boolean类型的`voiceActivityDetection`，缺省值为"true"：很多编解码器和系统都能够检测到“静音”，并改变它们的行为，例如不传输任何媒体信息。在很多场景下，例如处理紧急呼叫或不仅仅人声之外的语音时，我们希望能够关闭这个选项。这个选项允许应用提供关于是否希望开启或关闭这类处理的信息。

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

`RTCOfferOptions`成员变量：

- boolean类型的`iceRestart`，缺省值为"false"：当此值为true时，会生成与当前证书（在`localDescription`属性的SDP中可见）不同的ICE证书。应用此描述将重启ICE，具体描述在ICE的9.1.1.1节。
  当此值为false，`localDescription`属性具有有效的ICE证书，生成的描述将和当前的`localDescription`属性一致。 **注意：当`iceConnectionState`转换为"failed"时，建议执行ICE重启。应用也可能额外监听`iceConnectionState`到"disconnected"的变化，然后使用其他信息来源（比如使用`getStats`测量接下来几秒内发送或接收的字节数是否增加）确定是否应该重启ICE。**

`RTCAnswerOptions`字典描述了指定`answer`类型会话的选项。

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

4.3 状态定义

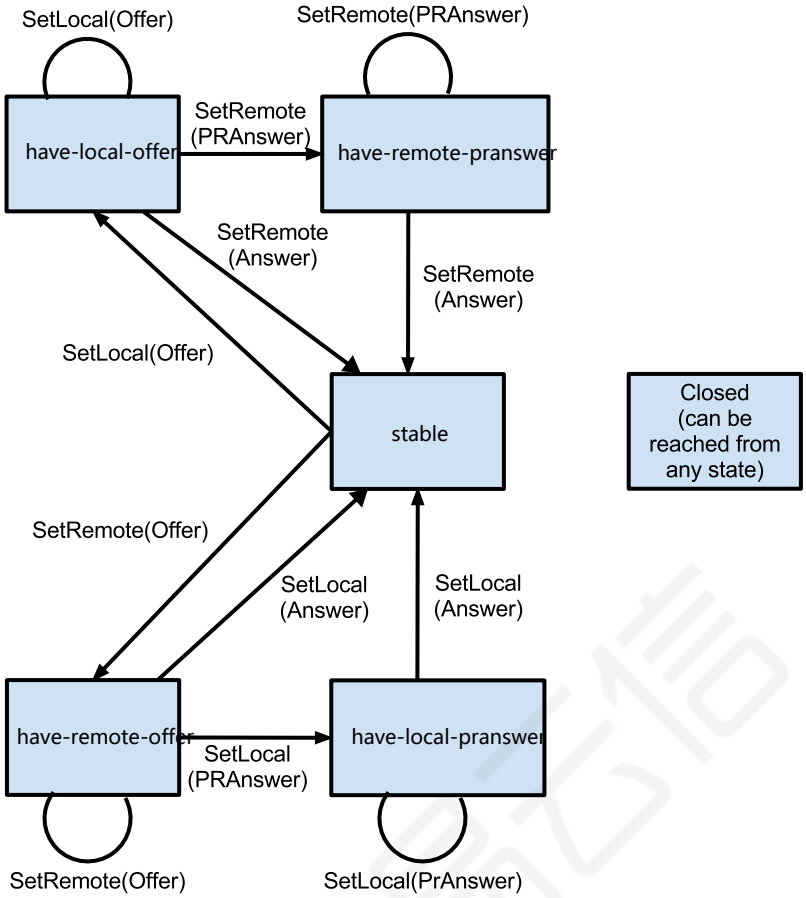
4.3.1 `RTCSignalingState`枚举值

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

枚举类型描述：

- stable：过程中无邀请/应答的交换。这也是初始状态，本地和远程描述都是空的。
- have-local-offer：本地的`邀请`类型的描述已经被成功应用了。
- have-remote-offer：远程的`邀请`类型的描述已经被成功应用了。
- have-local-pranswer：远程的`邀请`类型的描述已经被成功应用且本地的`对端应答`类型的描述已被成功应用。
- have-remote-pranswer：本地的`邀请`类型的描述已经被成功应用且远程的`对端应答`类型的描述已被成功应用。
- closed：`RTCPeerConnection`已经被关闭，其[IsClosed]槽值变为true。

信号状态转移图



一个状态转移的例子： 调用方的转移：

- 创建新RTCPeerConnection(): `stable`
- setLocalDescription(offer): `have-local-offer`
- setRemoteDescription(pranswer): `have-remote-pranswer`
- setRemoteDescription(answer): `stable`

被调用方的转移：

- 创建新RTCPeerConnection(): `stable`
- setRemoteDescription(offer): `have-remote-offer`
- setLocalDescription(pranswer): `have-local-pranswer`
- setLocalDescription(answer): `stable`

4.3.2 `RTCIceGatheringState`枚举值

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

枚举类型描述：

- new：所有`RTCIceTransports`都在"new"的收集状态，没有任何一个处于"gathering"状态，或当前还没有传输。

- gathering：所有`RTCIceTransports`都在"gathering"的收集状态
- complete：至少有一个`RTCIceTransports`存在，且都处于"completed"状态。

4.3.3 RTCPeerConnectionState枚举值

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

枚举类型描述：

- new：所有`RTCIceTransport`和`RTCDtlsTranport`都在"new"状态，且没有任何一个处于"connecting"，"checking"，"failed"，"disconnected"状态，也可以是所有传输都处于"closed"状态，或当前还没有传输。
- connecting：所有`RTCIceTransport`和`RTCDtlsTranport`都在"connecting"或"checking"状态，且没有一个处于"failed"状态。
- connected：所有`RTCIceTransport`和`RTCDtlsTranport`都在"connected"，"completed"或"closed"状态，且其中至少有一个处于"connected"或"completed"状态。
- disconnected：所有`RTCIceTransport`和`RTCDtlsTranport`都在"disconnected"状态，且没有一个处于"failed"，"connecting"或"checking"状态。
- failed：所有`RTCIceTransport`和`RTCDtlsTranport`都在"failed"状态。
- closed：`RTCPeerConnection`对象的[IsClosed]槽为值true。

4.3.4 `RTCIceConnectionState`枚举值

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

枚举类型描述：

- new：所有`RTCIceTransport`都在"new"状态，且没有任何一个处于"disconnected"，"checking"，"failed"，"disconnected"状态，也可以是所有`RTCIceTransports`都处于"closed"状态，或当前还没有传输。
- checking：所有`RTCIceTransport`都在"checking"状态且没任何一个处于"disconnected"或"failed"状态。
- connected：所有`RTCIceTransport`都在"connected"，"completed"或"closed"状态，且其中至少有一个处于"connected"状态。
- completed：所有`RTCIceTransport`都在"completed"或"closed"状态，且其中至少有一个处于"completed"状态。

- disconnected：所有`RTCIceTransport`都在"disconnected"状态，且没有一个处于"failed"状态。
- failed：所有`RTCIceTransport`都在"failed"状态。
- closed：`RTCPeerConnection`对象的[IsClosed]槽为值true。

值得注意的是，如果`RTCIceTransport`由于信令的存在而被丢弃（如RTCP复用或执行捆绑），或被信令创建（如增加新的媒体描述），则状态可以从某一状态跳变到另一状态。

4.4 `RTCPeerConnection`接口

JSEP规范从整体介绍了`RTCPeerConnection`的运作细节。下文会适时供应对[JSEP]特定小节的参考。

4.4.1 操作

调用`new RTCPeerConnection(configuration)`创建一个`RTCPeerConnection`对象。 `configuration.servers`包含了ICE用以探测并访问服务器的相关信息。应用可以为同一类型的服务供应多个实例，并且任何TURN服务器也可以用作STUN服务器，用于收集服务器自反候选地址。 一个`RTCPeerConnection`对象持有 **信令状态，连接状态，ICE收集状态和ICE连接状态** 四个状态。它们在对象创建时被初始化。 `RTCPeerConnection`的ICE协议实现部分用 **ICE agent** 来表示。`RTCPeerConnection`中与ICE Agent交互的方法被分别命名为：`addIceCandidate`, `setConfiguration`, `setLocalDescription`, `setRemoteDescription`和`close`。与这些交互相关的小节都被记录在JSEP文档中。ICE Agent同样向用户代理指示了代表其内部的`RTCIceTransport`状态何时发生变化，这在5.6 RTCIceTransport Interface。本节中列举的任务源即网络任务源networking task source

4.4.1.1 构造函数

当`RTCPeerConnection`的构造函数被调用了，用户代理 **必须** 按照以下步骤运行：

1. 如果以下任何一个步骤出现了未知错误，都会抛出`UnkownError`错误，并在"message"域设置相应的错误描述。
2. *connection*应是最新创建的`RTCPeerConnection`对象。
3. 如果 *configuration* 中的证书域`certificates`非空，则将来要对每个值检查是否过期。如果证书已过期或证书内部的[origin]插槽与当前证书的插槽不匹配，则会抛出`InvalidAccessError`，否则保存此证书。如果没有指定`certificates`的值，一个或多个新`RTCCertificates`实例将生成供`RTCPeerConnection`实例使用。以上步骤可能是 *异步* 发生的，因此在步骤子序列运行过程中，`certificates`的值可能还是未定义的。如RTCWEB-SECURITY 4.3.2.3所说，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，同时不需要额外的证书检查。
4. 初始化ICE Agent的 *connection*。
5. 填充 *connection* 内部的[Configuration] 槽。设置配置的规则由 *configuration* 指定。
6. 填充 *connection* 内部的[IsClosed] 槽，初始化为`false`。
7. 填充 *connection* 内部的[NegotiationNeeded] 槽，初始化为`false`。
8. 填充 *connection* 内部的[SctpTransport] 槽，初始化为`null`。
9. 填充 *connection* 内部的[Operations] 槽，代表一个操作队列，初始化为空列表。
10. 填充 *connection* 内部的[LastOffer] 槽，初始化为空字符串。
11. 填充 *connection* 内部的[LastAnswer] 槽，初始化为空字符串。
12. 设置 *connection* 的信令状态为`"stable"`。
13. 设置 *connection* 的ICE连接状态为`"new"`。
14. 设置 *connection* 的ICE收集状态为`"new"`。
15. 设置 *connection* 的连接状态为`"new"`。
16. 填充 *connection* 内部的[PendingLocalDescrtiption] 槽，初始化为`null`。
17. 填充 *connection* 内部的[CurrentLocalDescrtiption] 槽，初始化为`null`。

18. 填充 *connection* 内部的[PendingRemoteDescrtiption] 槽，初始化为 `null`。
19. 填充 *connection* 内部的[CurrentRemoteDescrtiption] 槽，初始化为 `null`。
20. 返回 *connection* 。

4.4.1.2 操作入队

一个 `RTCPeerConnection` 对象持有一个 **操作队列（operations queue）**，槽名[Operations]，它保证了队列中只有一个操作能异步并发地执行。如果后续的调用在之前的 `promise` 对象有执行结果之前产生了，它们会被加入队列中直到之前的 `promise` 有了结果才会被依次调用。 让某操作进入 `RTCPeerConnection` 对象的队列中，需按照以下步骤执行：

1. *connection* 即 `RTCPeerConnection` 对象。
2. 如果 *connection* 的[IsClosed]槽为 `"true"`，用 `promise` 包装一个新创建的 `InvalidStateError` 并返回。
3. 让 *connection* 成为即将入队的那一项。
4. 创建新 `promise`，名为 `p`。
5. 将 *connection* 添加至[Operations]队尾。
6. 如果[Operations]的长度为1，则执行该操作。
7. 在履行或拒绝该操作返回的 `promise` 之后，运行以下步骤：
    1. 如果 *connection* 的[IsClosed]槽为 `true`，终止以下步骤。
    2. 如果操作返回的 `promise` 履行并伴随了执行结果，把结果赋给 `p`。
    3. 如果操作返回的 `pormose` 拒绝并伴随了错误结果，把结果赋给 `p`。
    4. 根据 `p` 值，执行以下步骤：
    5. 如果 *connection* 的[IsClosed]槽为 `true`，终止以下步骤。
    6. 移除[Operations]队列中的第一个元素。
    7. 如果[Operations]队列非空，执行队列中的第一个操作。
8. 返回 `p`。

4.4.1.3 更新连接状态

`RTCPeerConnection` 集成了连接状态（connection state)。当 `RTCDtlsTransport` 或 `RTCIceTransport` 状态转移或[IsClosed]槽值为 `true` 时，用户代理必须将包含以下步骤的任务入队以 **更新连接状态** ：

1. *connection* 即 `RTCPeerConnection` 对象。
2. `newState` 变量值即 `RTCPeerConnectionState` 枚举值中派生的新状态值。
3. 如果 *connection* 的连接状态与 `newState` 值相同，终止以下步骤。
4. 将 *connection* 的 连接状态设置为 `newState`。
5. 触发此 *connection* 的 `connectionstatechange` 事件。

4.4.1.4 更新CE收集状态

为了**更新** `RTCPeerConnection` 实例的 **ICE收集状态**，用户代理必须将包含以下步骤的任务入队：

1. 如果 *connection* 的[IsClosed]槽值为 `true`，终止以下步骤。
2. `newState` 变量值即 `RTCIceGatheringState` 枚举值中派生的新状态值。
3. 如果 *connection* 的ICE收集状态与 `newState` 值相同，终止以下步骤。
4. 触发此 *connection* 的 `icegatheringstatechange` 事件。
5. 如果 `newState` 值为 `"completed"`，使用 `RTCPeerConnectionIceEvent` 接口触发名为 `icecandidate` 的事件，其候选地址属性设为 `null`。

> 注意：触发候选地址属性为`null`的事件是为了确保传统兼容性。新代码应该监控收集`RTCIceTransport`或`RTCPeerConnection`的状态。

4.4.1.5 更新CE连接状态

为了**更新** `RTCPeerConnection`实例的 **ICE连接状态**，用户代理必须将包含以下步骤的任务入队：

1. 如果 *connection* 的[IsClosed]槽值为`true`，终止以下步骤。
2. `newState`变量值即`RTCIceConnectionState`枚举值中派生的新状态值。
3. 如果 *connection* 的ICE连接状态与`newState`值相同，终止以下步骤。
4. 将 *connection* 的 ICE连接状态设置为`newState`。
5. 触发此连接的`iceconnectionstatechange`事件。

4.4.1.6 设置RTCSessionDescription

为了设置`RTCPeerConnection`对象的`RTCSessionDescription`，将以下步骤加入 *connection* 的操作队列：

1. 变量`p`为新的`promise`对象。
2. 并行启动进程应用JSEP 5.5&5.6中的 *description*。
    1. 如果应用描述的进程因为某个原因异常退出了，用户代理必须将包含以下步骤的任务入队：
        1. 如果 *connection* 的[IsClosed]槽值为`true`，则终止以下步骤。
        2. 如果 *description* 类型对于当前的连接信令状态是非法的，如JESP 5.5&5.6中提到的，则拒绝此`promise`并创建一个新的`InvalidStateError`错误然后终止步骤。
        3. 如果 *description* 被设为本地描述，且如果`description.type`是`offer`，`description.sdp`与连接的[LastOffer]槽值不同，则拒绝此`promise`并创建一个新的`InvalidModifcationError`错误然后终止步骤。
        4. 如果 *description* 被设为本地描述，且如果`description.type`是`rollback`，信令状态是`"stable"`，则拒绝此`promise`并创建一个新的`InvalidStateError`错误然后终止步骤。
        5. 如果 *description* 被设为本地描述，且如果`description.type`是`answer`或`pranswer`，`description.sdp`与连接的[LastAnswer]槽值不同，则拒绝此`promise`并创建一个新的`InvalidModifcationError`错误然后终止步骤。
        6. 如果 *description* 的内容不合SDP语法，则以RTCError（`errorDetail`被设置为"sdp-syntax-error"并把`sdpLineNumber`设置为检测到的SDP内容中非法语法所在行）拒绝此`promise`并终止步骤。
        7. 如果 *description* 被设为远程描述，则`RTCRtcpMuxPolicy`是必须项，若远程描述并没有使用RTCP复用，则拒绝此`promise`并创建一个新的`InvalidAccessError`错误然后终止步骤。
        8. 如果 *description* 中的内容非法，则拒绝此`promise`并创建一个新的`InvalidAccessError`错误然后终止步骤。
        9. 对于其他所有错误，拒绝`promise`并创建一个`OperationError`。
    2. 如果 *description* 被成功应用，用户代理必须将包含以下步骤的任务入队：
        1. 如果 *connection* 的[IsClosed]槽值为`true`，则终止以下步骤。
        2. 如果 *description* 被设为本地描述，则运行以下步骤中的某一个：
            - 如果 *description* 的类型为`"offer"`，设置连接的[PendingLocalDescription]槽为一个以 *description* 为依据构造的新`RTCSessionDescription`对象，并把信令状态设置为`"have-local-offer"`。
            - 如果 *description* 的类型为`"answer"`，则它完成了一次提供或应答的协商。将 *connection* 的[CurrentLocalDescription]槽设置为一个以 *description* 为依据构造的新

`RTCSessionDescription`对象，并把[CurrentRemoteDescription]设置为[PendingRemoteDescription]。把[PendingRemoteDescription]和[PendingLocalDescription]都设为`null`。最后将 *connection* 的信令状态设为`"stable"`。

- 如果 *description* 类型为`"rollback"`，则这是一个回滚操作。将 *connection* 的[PendingLocalDescription]槽设为`"null"`，并把信令状态设为`"stable"`。
- 如果 *description* 类型为`"pranswer"`，则把 *connection* 的[PendingLocalDescription]槽设置为一个以 *description* 为依据构造的新`RTCSessionDescription`对象，并把信令状态设为`"have-local-pranswer"`。

3. 否则，如果 *description* 被设为远程描述，则运行以下步骤中的某一个：
   - 如果 *description* 类型为`"rollback"`且信令状态为`"stable"`，则拒绝此`promise`并创建一个新的`InvalidStateError`错误然后终止步骤。
   - 如果 *description* 的类型为`"offer"`，设置连接的[PendingRemoteDescription]槽为一个以 *description* 为依据构造的新`RTCSessionDescription`对象，并把信令状态设置为`"have-remote-offer"`。
   - 如果 *description* 的类型为`"answer"`，则它完成了一次提供或应答的协商。将 *connection* 的[CurrentRemoteDescription]槽设置为一个以 *description* 为依据构造的新`RTCSessionDescription`对象，并把[CurrentLocalDescription]设置为[PendingLocalDescription]。把[PendingRemoteDescription]和[PendingLocalDescription]都设为`null`。最后将 *connection* 的信令状态设为`"stable"`。
   - 如果 *description* 类型为`"rollback"`，则这是一个回滚操作。将 *connection* 的[PendingRemoteDescription]槽设为`"null"`，并把信令状态设为`"stable"`。
   - 如果 *description* 类型为`"pranswer"`，则把 *connection* 的[PendingRemoteDescription]槽设置为一个以 *description* 为依据构造的新`RTCSessionDescription`对象，并把信令状态设为`"have-remote-pranswer"`。
4. 如果 *description* 类型为`"answer"`，启动一个与现有SCTP关联的闭包，如SCTP-SDP的10.3节和10.4节中所定义，并把 *connection* 的[SctpTransport]槽值设为`null`。
5. 如果 *description* 类型为`"answer"`或`"pranswer"`，则运行以下步骤：
   1. 如果 *description* 发起与一个新的SCTP建立关联，如SCTP-SDP的10.3节和10.4节中所定义，则以`"connecting"`为初始状态，创建一个新的`RTCSctpTransport`实例，并赋给[SctpTransport]槽。
   2. 否则，如果SCTP关联已创建完毕，而SDP属性的"max-message-size"也被更新了，则对 *connection* 的[SctpTransport]槽的最大消息长度的数据进行更新。
   3. 如果 *description* 对SCTP传输中的DTLS角色进行了协商，且存在一个`id`为`null`的`RTCDataChannel`，则根据RTCWEB-DATA-PROTOCAL生成一个ID。如果没有可用的ID，则运行以下步骤：
      1. *channel* 即当前无可用ID的`RTCDataChannel`对象。
      2. 将 *channel* 的[ReadyState]槽值设为`"closed"`。
      3. 在当前 *channel* 使用`RTCErrorEvent`接口触发名为`"error"`，且`errorDetail`属性被设为"data-channel-failure"的事件。
      4. 在当前 *channel* 触发名为`close`的事件。
6. 将 *trackEventInits, muteTracks, addList, removeList* 置空。
7. 如果 *description* 被设为本地描述，则运行以下步骤：
   1. 为 *description* 中的每个媒体描述执行：
      1. 如果媒体描述尚未与一个`RTCRtpTransceiver`对象关联，运行以下步骤：

1. *transceiver* 即创建媒体描述的`RTCRtpTransceiver`对象。
2. 将 *transceiver* 的`mid`值设为媒体描述中的对应值。
3. 如果 *transceiver* 的[Stopped]槽值为`true`，停止此子步骤。
4. 如果根据[Bundle]将媒体描述表示为“使用现有媒体传输”，并使transport和rtcpTransport变量成为分别表示传输中的RTP组件和RTCP组件的`RTCDtlsTransport`对象。
5. 否则，使transport和rtcpTransport变量成为新创建的`RTCDtlsTransport`对象，每个都持有一个新创建的底层`RTCIceTransport`。如果根据RFC5761协商RTCP多路复用，或者 *connection* 的`RTCRtcpMuxPolicy`为`require`，则不要创建任何特定的RTCP传输对象，而是让rtcpTransport等于transport变量。
6. *transceiver.[Sender].[SenderTransport]* 设为transport。
7. *transceiver.[Sender].[SenderRtcpTransport]* 设为rtcpTransport。
8. *transceiver.[Receiver].[ReceiverTransport]* 设为transport。
9. *transceiver.[Receiver].[ReceiverRtcpTransport]* rtcpTransport。

2. 设 *transceiver* 为已与媒体描述关联的`RTCRtpTransceiver`。
3. 如果 *transceiver* 的[Stopped]槽值为`true`，则终止以下子步骤。
4. 设 *direction* 为表示媒体收发方向的`RTCRtpTransceiverDirection`值。
5. 如果 *direction* 值为`sendrecv`或`recvonly`，则将 *transceiver* 的[Receptive]槽的值设为`true`，否则设为`false`。
6. 如果 *description* 的类型为`answer`或`pranswer`，则运行以下步骤：
    1. 如果 *direction* 值为`sendonly`或`inactive`，且 *transceiver* 的[FiredDirection]槽值为`sendrecv`或`recvonly`，则继续运行以下步骤：
        1. 给定transceiver.[Receiver]，2个空列表和removeList，设置相关联的远程媒体流。
        2. 在给定 *transceiver* 和 *muteTracks（静音轨）* 的情况下，处理媒体描述的远程媒体轨的移除。
    2. 将 *transceiver* 的[CurrentDirection]槽和[FiredDirection]槽设为 *direction* 。

2. 如果 *description* 被设为远程描述，则运行以下步骤：
    1. 为 *description* 中的每个媒体描述执行：
        1. 设 *direction* 为代表媒体流收发方向的`RTCRtpTransceiverDirection`值，但在对等连接的角度看来，发送和接受的方向是相反的。
        2. 如JESP 5.10所述，尝试找到现有的`RTCRtpTransceiver`对象，即 *transceiver* ，以代表媒体描述。
        3. 如果没有找到合适的收发器（ *transceiver* 为空），则运行以下步骤：
            1. 从媒体描述创建`RTCRtpSender`对象 *sender* 。
            2. 从媒体描述创建`RTCRtpReceiver`对象 *receiver* 。
            3. 根据 *sender, receiver* 以及一个值为`recvonly`的`RTCRtpTransceiverDirection`创建一个`RTCRtpTransceiver`，即成为 *transceiver* 。
        4. 将 *transceiver* 的`mid`值设为媒体描述中的对应值。如果媒体描述没有MID，且 *transceiver* 的`mid`未定义，则生成一个随机值，在JSEP 5.10中有相关描述。
        5. 如果 *direction* 值为`sendrecv`或`recvonly`，则使 *msids* 为媒体描述指示的 *transceiver.[Receiver].[ReceiverTrack]* 相关联的MSID列表，否则 *msids* 为空。

6. 给定 *transceiver.[Receiver], msids, addList, removeList* ，设置关联的远程媒体流。
7. 给定 *transceiver, trackEventInits*， 如果前一步骤使 *addList* 的长度增长了，或 *transceiver* 的[FireDirection]槽为 `sendrecv` 或 `recvonly`，则为媒体描添加一个远程媒体轨。
8. 如果 *direction* 值为 `sendonly` 或 `inactive`，则将 *transceiver* 的[Receptive]槽值设为 `false`。
9. 如果 *direction* 值为 `sendonly` 或 `inactive`，且 *transceiver* 的[FiredDirection]槽值为 `sendrecv` 或 `recvonly`，则给定 *transceiver, muteTracks* ， 则为媒体描述移除一个远程媒体轨。
10. 把 *direction* 赋给 *transceiver* 的[FiredDirection]槽。
11. 如果 *description* 的类型为 `answer` 或 `pranswer`，则运行以下步骤：
    1. 把 *direction* 赋给 *transceiver* 的[CurrentDirection]和[Direction]槽。
    2. 根据BUNDLE，让 *transport, rtcpTransport* 成为代表与 *transceiver* 相关联的 RTP, RTCP媒体传输组件的 `RTCDtlsTransport` 对象。
    3. 设 *transceiver.[Sender].[SenderTransport]* 为 *transport* 。
    4. 设 *transceiver.[Sender].[SenderRtcpTransport]* 为 *rtcpTransport* 。
    5. 设 *transceiver.[Receiver].[ReceiverTransport]* 为 *transport* 。
    6. 设 *transceiver.[Receiver].[ReceiverRtcpTransport]* 为 *rtcpTransport* 。
12. 如果媒体描述被拒绝，且 *transceiver* 未准备好停止，则将它停止。

3. 如果 *description* 的类型为 `rollback`，则运行以下步骤：
    1. 如果 `RTCRtpTransceiver` 的 `mid` 值被即将回滚的 `RTCSessionDescription` 对象设为一个非空值，则将收发器的 `mid` 值设为空（null）。
    2. 如果 `RTCRtpTransceiver` 是通过即将回滚的 `RTCSessionDescription` 创建的，且媒体轨没有通过调用 `addTrack` 附加到 `RCTRtpTransceiver`，则从 *connection* 的transceiver列表移除该transceiver。
    3. 对于那些留在 *connection* 中的 `RTCRtpTransceiver` 对象，将即将回滚的 `RTCSessionDescription` 所在应用造成的[CurrentDirection]和[Receptive]两个槽的所有改动都复原。
    4. 将 *connection* 的[SctpTransport]槽值重置为上次信令状态为 `stable` 时的值。
4. 如果 *connection* 的信令状态改变了，触发一个名为 `signalingstatechange` 的事件。
5. 对 *muteTracks* 中的每个 *track* ，将其静音状态设为 `true`。
6. 对 *removeList* 中的每个媒体流（stream）和媒体轨（track），从媒体流中移除媒体轨。
7. 对 *addList* 中的每个媒体流（stream）和媒体轨（track），将媒体轨添加至媒体流。
8. 对于 *trackEventInits* 中的每个入口entry，使用 `RTCTrackEvent` 接口触发名为 `track` 的事件，其 `receiver` 属性初始化为 `entry.receiver`， `track` 属性初始化为 `entry.track`， `streams` 属性初始化为 `entry.streams`， `transceiver` 属性初始化为 `entry.transceiver`。
9. 如果当前 *connection* 的信令状态为 `stable`，则更新是否需要协商的标志位。如果更新前后 *connection* 的[NegotiationNeeded]槽值一直为 `true`，则将包含以下步骤的任务加入队列：
    1. 若 *connection* 的[IsClosed]槽值为 `true`，则终止后续步骤。
    2. 若 *connection* 的[NegotiationNeeded]槽值为 `false`，则终止后续步骤。
    3. 触发名为 `negotiationneeded` 事件。
10. 解析未定义的 `p`。

3. 返回变量`p`。

4.4.1.7 设置配置

为了 **设置配置** ，运行以下步骤：

1. *configuration* 即要被处理的`RTCConfiguration`字典。
2. *connection* 即目标`RTCPeerConnection`对象。
3. 如果`configuration.peerIdentity`已被设置，且其值与目标对等连接的对应值不相同，则抛出一个`InvalidModificationError`。
4. 如果`configuration.certificates`已被设置，且其值与连接建立时使用的证书列表不同，则抛出一个`InvalidModificationError`。
5. 如果`configuration.bundlePolicy`已被设置，且其值与连接的捆绑策略不同，则抛出一个`InvalidModificationError`。
6. 如果`configuration.rtcpMuxPolicy`已被设置，且其值与连接的rtcpMux策略不同，则抛出一个`InvalidModificationError`。如果策略为`negotiate`且用户代理没有实现非多路复用的RTCP，则抛出一个`NotSupportedError`。
7. 如果`configuration.iceCandidatePoolSize`已被设置，其值与连接先前使用的对应值不同，且`setLocalDescription`方法已被调用了，则抛出一个`InvalidModificationError`。
8. 将ICE代理的 **ICE传输设置** 值设为`configuration.iceTransportPolicy`。JSEP 4.1.16中定义，如果新的ICE传输设置改变了现有的设置，则在下一收集阶段之前都不会有新的操作执行。如果某段脚本希望立即被执行，则应该先重启ICE。
9. JSEP 3.5.4 & 4.1.1中定义，将ICE代理预先获取的ICE候选池大小设为`configuration.iceCandidatePoolSize`的值。如果新的ICE候选池大小改变了现有的设置，可能会导致为候选池立刻开始收集新的候选地址，或忽略池中现有的候选地址。
10. 将 *validatedServers* 设为一个空列表。
11. 如果`configuration.iceServers`已被定义，则对其的每个元素执行以下步骤：
    1. *server* 即当前列表中的元素。
    2. *urls*即`server.urls`。
    3. 如果 *urls* 是一个字符串，则将 *urls* 设为由此字符串组成的列表。
    4. 如果 *urls* 为空，抛出一个`SyntaxError`。
    5. 对于 *urls* 中的每个 *url* ，执行以下步骤：
        1. 利用RFC3986中定义的通用URI格式解析此url，并获得 *模式名*。如果解析失败，则抛出`SyntaxError`。如果提取出的模式没有被浏览器实现，则抛出`NotSupportedError`。如果模式名为`turn`或`turns`，且用RFC7064定义的语法也无法解析url，则抛出`SyntaxError`。如果模式名为`stun`或`stuns`，且用RFC7065定义的语法也无法解析url，则抛出`SyntaxError`。
        2. 若模式名为`turn`或`turns`，且`server.username`和`server.credential`都为空，则抛出`InvalidAccessError`。
        3. 若模式名为`turn`或`turns`，且`server.credentialType`为`password`，`server.credential`不是一个DOMString，则抛出`InvalidAccessError`。
        4. 如果模式名为`turn`或`turns`，且`server.credentialType`为`oauth`，`server.credential`不是一个`RTCOAuthCredential`对象，则抛出`InvalidAccessError`。
    6. 将 *server* 追加到 *validatedServers* 。

使 *validatedServers* 成为ICE代理的 **ICE服务器列表** 。 如JSEP 4.1.16定义的，如果一个新的服务器列表取代了当前ICE代理的服务器列表，下一收集阶段之前都不会有动作执行。如果某段脚本希望立即被执行，则应该先重启ICE。无论如何，如果ICE候选池的大小非零，所有现有的池内候选地址都会被忽略，新的候选地址会从新服务器中收集。

12. 将当前配置保存至内部的[Configuration]槽。

4.4.2 接口定义

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

```
[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** ：参阅RTCPeerConnection构造算法。

属性：

- RTCSessionDescription类型的`localDescription`，只读，可空：如果[PendingLocalDescription]槽非空，则`localDescription`属性必须返回它，否则返回[CurrentLocalDescription]。
  注意，[CurrentLocalDescription].sdp和[PendingLocalDescription].sdp与传入`setLocalDescription`的SDP值不必是字符串值相等的（例如，SDP可能被解析后又格式化了，或ICE候选地址有新增）。
- RTCSessionDescription类型的`currentLocalDescription`，只读，可空：`currentLocalDescription`属性必须返回[CurrentLocalDescription]槽的内容。
  它代表了上次`RTCPeerConnection`转化为稳定状态时成功协商好的本地描述，以及创建邀请/应答以来ICE代理生成的所有本地候选地址。
- RTCSessionDescription类型的`pendingLocalDescription`，只读，可空：`pendingLocalDescription`属性必须返回[PendingLocalDescription]槽的内容。
  它代表了正在协商过程中的本地描述，以及创建邀请/应答以来ICE代理生成的所有本地候选地址。如果`RTCPeerConnection`正处于稳定状态，则此值为`null`。
- `RTCSessionDescription`类型的`remoteDescription`，只读，可空：如果[PendingRemoteDescription]槽非空，则`remoteDescription`属性必须返回它，否则返回[CurrentRemoteDescription]。
  注意，[CurrentRemoteDescription].sdp和[PendingRemoteDescription].sdp与传入`setRemoteDescription`的SDP值不必是字符串值相等的（例如，SDP可能被解析后又格式化了，或ICE候选地址有新增）。
- RTCSessionDescription类型的`currentRemoteDescription`，只读，可空：它代表了上次`RTCPeerConnection`转化为稳定状态时成功协商好的远程描述，以及创建邀请/应答以来通过`addIceCandidate()`方法提供的所有远程候选地址。
- RTCSessionDescription类型的`pendingRemoteDescription`，只读，可空：`pendingRetmoteDescription`属性必须返回[PendingRemoteDescription]槽的内容。
  它代表了正在协商过程中的远程描述，以及创建邀请/应答以来通过`addIceCandidate()`方法提供的所有远程候选地址。。如果`RTCPeerConnection`正处于稳定状态，则此值为`null`。
- RTCSignalingState类型的`signalingState`，只读：`signalingState`属性必须返回`RTCPeerConnection`对象的信令状态。
- RTCIceGatheringState类型的`iceGatheringState`，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。
- RTCPeerConnectionState类型的`connectionState`，只读：`connectionState`属性必须返回`RTCPeerConnection`实例的连接状态。
- boolean类型的`canTrickleIceCandidates`，只读，可空：`canTrickleIceCandidates`属性指示了远程对等连接是否能够接受递增式的ICE候选地址TRICKLE-ICE。这个值根据远程描述是否支持递增式ICE来确定，JSEP 4.1.15有相关描述。在`setRemoteDescription`调用完成之前，此值为`null`。
- EventHandler类型的`onnegotiationneeded`：此事件处理器的事件类型为`negotiationneeded`。
- EventHnadler类型的`onicecandidate`：此事件处理器的事件类型为`icecandidate`。
- EventHnadler类型的`onicecandidateerror`：此事件处理器的事件类型为`icecandidateerror`。
- EventHnadler类型的`onsignalingstatechange`：此事件处理器的事件类型为`signalingstatechange`。
- EventHnadler类型的`oniceconnectionstatechange`：此事件处理器的事件类型为`iceconnectionstatechange`。
- EventHnadler类型的`onicegatheringstatechange`：此事件处理器的事件类型为`icegatheringstatechange`。
- EventHnadler类型的`onconnectionstatechange`：此事件处理器的事件类型为`connectionstatechange`。

方法：

- **createOffer** ：`createOffer`方法生成一个包含符合[RFC 3264]邀请规范的SDP blob对象，附带会话支持的配置，包括附加到本`RTCPeerConnection`的本地`MediaStreamTrack`对象的描述，本实现支持的编解码器/RTP/RTCP功能，ICE代理的参数以及DTLS连接。`options`参数也许会用于在邀请生成后施加额外的控制。如果系统对资源作了限制（例如有限个数的解码器），`createOffer`需要返回反映当前系统状态的一个邀请，这样当它尝试获取对应资源的时候`setLocalDescription`方法可以调用成功。会话描述必须保证至少在`promise`对象的回调函数返回前`setLocalDescription`调用不会抛出错误，在此期间一直保持可用。
  为了生成[JSEP]中定义的邀请，创建SDP必须遵循一套合适的流程。对于一个邀请，生成的SDP包含会话支持的编解码器/RTP/RTCP全套功能（对应的应答只包含一个特定的子集）。在会话建立后的`createOffer`调用事件中，`createOffer`将生成一个兼容当前会话的邀请，包含自上次完整的邀约-答复以来对会话所做的所有更改，例如媒体轨的增加或删除。如果没有更改发生，邀请将包含当前本地描述的功能以及未来可以通过协商达成的附加功能。
  生成的SDP同样包含ICE代理的`usernameFragment, password`及ICE选项（ICE 14节中定义），也可能包含代理收集的任何本地候选地址。
  `RTCPeerConnection`对象 *configuration* 中的`certificates`值提供了应用配置的证书。这些证书和其他默认证书一起生成证书指纹集合。这些证书指纹将被用于SDP的构造以及请求身份声明时的输入。
  如果`RTCPeerConnection`被配置用于调用`setIdentityProvider`生成身份声明，则会话描述 *SHALL* 将包含一个合适的断言。
  SDP的创建过程暴露了底层系统的一部分媒体功能，它在设备上能提供持久的跨源信息。因此，它增加了应用的指纹表面。在隐私敏感的上下文中，浏览器可以考虑放缓，例如仅生成与SDP匹配的公共功能子集。（这是指纹向量）。
  当此方法被调用，用户代理必须运行以下步骤：
  1. *connection* 即调用此方法的`RTCPeerConnection`对象。
  2. 如果 *connection* 的[IsClosed]槽为`true`，则返回一个用新创建的`InvalidStateErrror`拒绝的`promise`对象。
  3. 如果 *connection* 配置了身份提供方，且连接没有正式建立，则开启身份声明请求。
  4. 将以下操作加入 *connection* 的操作队列，并返回结果：
     1. *p* 即`promise`对象。
     2. 并行开启创建邀请。
     3. 返回 *p* 。 给定`promise`对象 *p* ，**创建邀请** 的步骤如下：
  5. 如果 *connection* 没有通过证书集合创建，或某个证书还没被生成，则等待直到生成完毕。
  6. 若 *connection* 配置了身份提供方，则命名为 *provider* ，否则 *provider* 为`null`。
  7. 若 *provider* 非空，等待身份声明过程结束。
  8. 如果 *provider* 身份声明失败，则以一个新创建的`NotReadableError`拒绝 *p* 并终止后续步骤。
  9. 检查系统状态，确定当前可用的资源足够生成邀请，这在JSEP 4.1.6有定义。
  10. 如果因为任何原因检查失败，以一个新创建的`OperationError`拒绝 *p* 并终止后续步骤。
  11. 将包含创建邀请的最终步骤的任务加入队列。 promise *p* 中 **创建邀请的最终步骤** 包含以下：
  12. 如果 *connection* 的[IsClosed]槽为`true`，则终止以下步骤。
  13. 如果以某种方式修改了 *connection* ，则需要额外检查系统状态，或者如果连接配置的身份提供方不再是 *provider* ，则在 *p* 中重新开始创建邀请的过程，并终止以下步骤。 **注意：这一步可能很重要，例如，当连接中只有一个音频`RTCRtpTransceiver`对象时，`createOffer`被调用了，但当并行执行创建邀请的过程中，一个视频`RTCRtpTransceiver`对象被附加到连接上，这时候就需要检查视频系统资源。**

14. 从先前的检查中获取信息，包括 *connection* 的当前状态及其`RTCRtpTranceiver`列表， 来自 *provider* （若非空）的身份声明，然后生成一个SDP邀请， *sdpString* ，这在JSEP 5.2中又定义。
15. 设 *offer* 为新创建的`RTCSessionDescriptionInit`字典，其`type`成员被初始化为`"offer"`字符串，`sdp`成员被初始化为 *sdpString* 。
16. 将内部的[LastOffer]槽设为 *sdpString* 。
17. 用 *offer* 解析 *p* 。

- **createAnswer**：`createAnswer`方法生成了一个包含与远程配置中的参数兼容的会话配置的[SDP]应答。就像`createOffer`，返回的SDP blob对象包含了附加到本`RTCPeerConnection`的本地`MediaStreamTracks`描述，与本会话协商好的编解码器/RTP/RTCP选项以及ICE代理收集到的所有候选地址。`options`参数也许会用于在应答生成后施加额外的控制。
  就像`createOffer`，返回的描述应该能反应当前的系统状态。会话描述必须保证至少在`promise`对象的回调函数返回前`setLocalDescription`调用不会抛出错误，在此期间一直保持可用。
  对于一个应答，生成的SDP应该包含特定的编解码器/RTP/RTCP配置，以及对应的邀请，邀请指定了如何建立媒体平面。生成的SDP必须按照JSEP中定义的过程来生成应答。
  生成的SDP同样包含ICE代理的`usernameFragment, password`及ICE选项（ICE 14节中定义），也可能包含代理收集的任何本地候选地址。
  `RTCPeerConnection`对象 *configuration* 中的`certificates`值提供了应用配置的证书。这些证书和其他默认证书一起生成证书指纹集合。这些证书指纹将被用于SDP的构造以及请求身份声明时的输入。
  如JSEP 4.1.8.1定义，应答可以通过设置`type`成员为`pranswer`来标记为临时的。
  如果`RTCPeerConnection`被配置用于调用`setIdentityProvider`生成身份声明，则会话描述 *SHALL* 将包含一个合适的断言。
  当方法被调用，用户代理必须运行以下步骤：
  1. *connection* 即调用此方法的`RTCPeerConnection`对象。
  2. 如果 *connection* 的[IsClosed]槽为`true`，则返回一个用新创建的`InvalidStateErrror`拒绝的`promise`对象。
  3. 如果 *connection* 配置了身份提供方，且连接没有正式建立，则开启身份声明请求。
  4. 将以下操作加入 *connection* 的操作队列，并返回结果：
     1. 如果 *connection* 的信令状态并非`"have-remote-offer"`或`"have-local-pranswer"`，返回一个用新创建的`InvalidStateError`拒绝的`promise`对象。
     2. *p* 即`promise`对象。
     3. 并行开启创建应答。
     4. 返回 *p* 。 给定`promise`对象 *p* ，**创建应答** 的步骤如下：
  5. 如果 *connection* 没有通过证书集合创建，或某个证书还没被生成，则等待直到生成完毕。
  6. 若 *connection* 配置了身份提供方，则命名为 *provider* ，否则 *provider* 为`null`。
  7. 若 *provider* 非空，等待身份声明过程结束。
  8. 如果 *provider* 身份声明失败，则以一个新创建的`NotReadableError`拒绝 *p* 并终止后续步骤。
  9. 检查系统状态，确定当前可用的资源足够生成邀请，这在JSEP 4.1.7有定义。
  10. 如果因为任何原因检查失败，以一个新创建的`OperationError`拒绝 *p* 并终止后续步骤。
  11. 将包含创建应答的最终步骤的任务加入队列。 promise *p* 中 **创建应答的最终步骤** 包含以下：
  12. 如果 *connection* 的[IsClosed]槽为`true`，则终止以下步骤。

13. 如果以某种方式修改了 *connection* ，则需要额外检查系统状态，或者如果连接配置的身份提供方不再是 *provider* ，则在 *p* 中重新开始创建应答的过程，并终止以下步骤。 **注意：这一步可能很重要，例如，当一个`RTCRtpTransceiver`的方向为`recvonly`时，`createAnswer`被调用了，但当并行执行创建邀请的过程中，方向又变为`sendrecv`了，这时候就需要检查视频编码资源。**
14. 从先前的检查中获取信息，包括 *connection* 的当前状态及其`RTCRtpTranceiver`列表， 来自 *provider* （若非空）的身份声明，然后生成一个SDP邀请， *sdpString* ，这在JSEP 5.2中又定义。
15. 设 *offer* 为新创建的`RTCSessionDescriptionInit`字典，其`type`成员被初始化为`"answer"`字符串，`sdp`成员被初始化为 *sdpString* 。
16. 将内部的[LastAnswer]槽设为 *sdpString* 。
17. 用 *offer* 解析 *p* 。

- **setLocalDescription**：`setLocalDescription`方法命令`RTCPeerConnection`将提供的`RTCSessionDescriptionInit`作为本地描述。
  这个API改变了本地媒体的状态。为了成功处理应用想要提供的从一种媒体格式更改为另一种不兼容格式的场景，`RTCPeerConnection`必须能够同时支持使用当前已有的和正在准备中的的本地描述（例如支持存在于两种描述中的编解码器）直到收到最终的应答，此时，`RTCPeerConnection`可以完全采用准备中的本地描述，或者如果远程端拒绝更改，则回滚至当前描述。
  如JSEP 5.4中提到的，`createOffer`和`createAnswer`返回的SDP在传入`setLocalDescription`之前一定不能被更改。最终，当本方法被调用，用户代理必须按以下步骤运行：
  1. *description* 即`setLocalDescription`得第一个参数。
  2. 如果`description.sdp`为空字符串且`description.type`为`answer`或`pranswer`，将 *connection* 得[LastAnswer]槽的值赋给`description.sdp`。
  3. 如果`description.sdp`为空字符串且`description.type`为`offer`，将 *connection* 得[LastOffer]槽的值赋给`description.sdp`。
  4. 将用`description`表示的设置RTCSessionDescription的结果返回。 **注意：如JSEP 5.9中提到的，调用此方法可能触发ICE代理收集ICE候选地址。**
- **setRemoteDescription**：`setRemoteDescription`方法命令`RTCPeerConnection`将提供的`RTCSessionDescriptionInit`作为远程邀请或应答。
  这个API改变了本地媒体的状态。
  当方法被调用，用户代理必须返回以方法的第一个参数表示的设置RTCSessionDescription的结果。
  除此之外，一个远程描述被用来确定并确认对等连接的身份。
  如果`a=identity`属性存在于会话描述中，浏览器会验证身份声明。
  如果`peerIdentity`配置被应用于`RTCPeerConnection`，将建立起所提供值的 **目标对等身份**。另外，如果`RTCPeerConnection`之前的身份已被认证（也就是`peerIdentity`的promise对象被解析），则同样会建立起目标对等身份。
  目标对等身份一旦被设置就不能被改变。
  如果目标等对身份被设置，那么身份的合法性验证必须在`setRemoteDescription`返回的promise被解析前完成。如果身份合法性验证失败了，则拒绝`setRemoteDescription`返回的promise。
  若无目标对等身份，`setRemoteDescription`无需等待身份合法性验证的完成。
- **addIceCandidate**：`addIceCandidate`方法向ICE代理提供了一个远程候选地址。当以一个空字符串表示`candidate`成员调用本方法时，同样可用于表示远程候选地址的终端。 本方法使用的参数仅包括`candidate`, `sdpMid`, `sdpMLineIndex`和`usernameFragment`，其余都被忽略。当方法被调用，用户代理必须按以下步骤运行：
  1. *candidate* 即方法的参数。

2. *connection* 即调用此方法的`RTCPeerConnetion`对象。如果`sdp`和`sdpMLineIndex`都为`null`，用一个新创建的`TypeError`拒绝promise并返回。
3. 将包含以下步骤的任务加入 *connection* 的操作队列，并将结果返回：
    1. 如果`remoteDescription`为`null`，用一个新创建的`InvalidStateError`拒绝promise并返回。
    2. *p* 即新的promise。
    3. 若 *candidate.sdpMid* 非空，运行以下步骤：
        1. 如果 *candidate.sdpMid* 不等于任何媒体描述`remoteDescription`的mid值，用一个新创建的`OperationError`拒绝p并返回，中止这些步骤。
    4. 否则，如果 *candidate.sdpMLineIndex* 非空，运行以下步骤：
        1. 如果 *candidate.sdpMLineIndex* 大于等于`remoteDescription`中媒体描述的数量，用一个新创建的`OperationError`拒绝p并返回，中止这些步骤。
    5. 若`candidate.usernameFragment`值不是`undefined`或`null`，且已应用的远程描述中相应媒体描述存在的任何用户名片段，用一个新创建的`OperationError`拒绝p并返回，中止这些步骤。
    6. 并行地，按照JSEP 4.1.17，添加ICE候选地址。用`candidate.usernameFragment`识别ICE生成过程；如果`usernameFragment`为空，将 *candidate* 用于最近一次ICE生成过程。如果`candidate.candidate`是空字符串，则将 *candidate* 作为相应媒体描述和ICE候选地址生成过程地终止指示。
        1. 若 *candidate* 没有被成功添加，用户代理必须将包含以下步骤地任务加入队列：
            1. 如果 *connection* 的[IsClosed]槽为`true`，则终止步骤。
            2. 用一个新创建的`OperationError`拒绝promise并返回。
        2. 如果 *candidate* 被成功应用了，用户代理必须将包含以下步骤地任务加入队列：
            1. 如果 *connection* 的[IsClosed]槽为`true`，则终止步骤。
            2. 如果 *connection.[PendingRemoteDescription]* 非`null`，且代表了被处理的 *candidate* 的ICE生成过程，则将 *candidate* 加入到 *connection.[PendingRemoteDescription].sdp* 。
            3. 如果 *connection.[CurrentRemoteDescription]* 非`null`，且代表了被处理的 *candidate* 的ICE生成过程，则将 *candidate* 加入到 *connection.[CurrentRemoteDescription].sdp* 。
            4. 用`undefined`解析p。
    7. 返回p。

- **getDefaultIceServers**：返回配置入浏览器的ICE服务器列表。浏览器可被配置使用本地或私有的STUN/TURN服务器。本方法允许应用了解这些服务器并有选择地使用它们。
  这个列表可能是持久且跨源的。它同样增加了浏览器的指纹表面。在隐私敏感的上下文中，浏览器可以考虑暂缓，例如仅将此数据提供给列入白名单的数据源（或彻底不提供）（这是一个指纹向量）。
  注意：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**：返回一个代表当前`RTCPeerConnection`配置的`RTCConfiguration`对象。
  当本方法被调用，用户代理必须必须返回存储在[Configuration]槽中的`RTCConfiguration`对象。
- **setConfiguration**：`setConfiguration`方法会更新`RTCPeerConnection`对象的配置。包括改变ICE Servers的配置。JSEP 3.5.1提到，当ICE相关配置更新需要重新收集ice候选地址时，ICE应该重启。
  当`setConfiguration`被调用，用户代理必须按以下步骤运行：
    1. *connection* 即调用此方法的`RTCPeerConnection`对象。
    2. 如果 *connection* 的[IsClosed]槽值为`true`，抛出一个`InvalidStateError`。
    3. 将配置设为指定的 *configuration* 。

- **close**：当`close`方法被调用，用户代理必须按以下步骤运行：
    1. *connection* 即调用此方法的`RTCPeerConnection`对象。
    2. 如果 *connection* 的[IsClosed]槽值为`true`，终止步骤。
    3. 将 *connection* 的[IsClosed]槽设为`true`。
    4. 将 *connection* 的信令状态设为`closed`。
    5. 设 *transceivers* 为收集收发器算法的执行结果。对于 *transceivers* 中的每个`RTCRtpTransceiver`对象，执行以下：
        1. 如果 *transceiver* 的[Stopped]槽值为`true`，终止步骤。
        2. 设 *sender* 为 *transceiver* 的[Sender]槽内容。
        3. 设 *receiver* 为 *transceiver* 的[Receiver]槽内容。
        4. 停止 *sender* 发送媒体数据。
        5. 根据RFC3550定义， *sender* 向每个RTP流发送RTCP BYE信号。
        6. 停止 *receiver* 接收媒体数据。
        7. 将 *receiver* 的[ReceiverTrack]的`readyState`为`ended`。
        8. 设 *transceiver * 的[Stopped]槽为`true`。
    6. 将每个连接的`RTCDataChannel`对象的[ReadyState]槽值设为`closed`。
       注意：`RTCDataChannel`将被突然关闭，关闭程序不会被调用
    7. 如果 *connection* 的[SctpTransport]槽非`null`，通过发送一个SCTP ABORT数据块且设[SctpTransportState]槽为`closed`来与底层的SCTP解除关联。
    8. 将 *connection* 的每个`RTCDtlsTransport`对象的[DtlsTransportState]设为`closed`。
    9. 销毁 *connection* 的ICE代理，结束所有活跃的ICE过程，释放所有相关资源（例如TRUN权限）。
    10. 将 *connection* 的每个`RTCIceTransport`对象的[IceTransportState]设为`closed`。
    11. 设 *connection* 的ICE连接状态为`closed`。
    12. 设 *connection* 的连接状态为`closed`。

4.4.3 旧版接口扩展

> 注意：出于可读性考虑，本节已被拆解。 将部分接口视为其主要对应部分的一部分，因为它们会重载现有的方法。

是否支持本节中的方法是可选的。但如果决定支持这些方法，则必须根据此处指定的方法实现。

> 注意：`RTCPeerConnection`中的`addStream`方法可以很容易被填充为：
>
> ```
> RTCPeerConnection.prototype.addStream = function(stream) {
> stream.getTracks().forEach((track) => this.addTrack(track, stream));
> };
> ```

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**：当`createOffer`方法被调用，用户代理必须按以下步骤运行：
    1. 设 *successCallback* 为方法的第一个参数。
    2. 设 *failureCallback* 为方法的第二个参数，表示一个回调。
    3. 设 *options* 为方法的第三个参数，表示一个回调。
    4. 将 *options* 作为单独的参数，执行`RTCPeerConnection`的`createOffer()`中指定的步骤，设 *p* 为返回的promise。
    5. 完成后p含有值 *offer* ，将 *offer* 作为参数调用 *successCallback* 。
    6. 被拒绝的话p会附带原因 *r* ，将 *r* 作为参数调用 *failureCallback* 。
    7. 用`undefine`解析一个promise并返回。
- **setLocalDescription**：当`setLocalDescription`方法被调用，用户代理必须按以下步骤运行：
    1. 设 *description* 为方法的第一个参数。
    2. 设 *successCallback* 为方法的第二个参数。
    3. 设 *failureCallback* 为方法的第三个参数，表示一个回调。
    4. 将 *description* 作为单独的参数，执行`RTCPeerConnection`的`setLocalDescription()`中指定的步骤，设 *p* 为返回的promise。
    5. p完成后，将`undefined`作为参数调用 *successCallback* 。
    6. 被拒绝的话p会附带原因 *r* ，将 *r* 作为参数调用 *failureCallback* 。
    7. 用`undefine`解析一个promise并返回。
- **createAnswer**：**注意：旧版的`createAnswer`不接受`RTCAnswerOptions`作为参数，因为没有任何旧版的`createAnswer`实现支持。** 当`createAnswer`方法被调用，用户代理必须按以下步骤运行：
    1. 设 *successCallback* 为方法的第一个参数。
    2. 设 *failureCallback* 为方法的第二个参数，表示一个回调。
    3. 不传入任何参数，执行`RTCPeerConnection`的`createAnswer()`中指定的步骤，设 *p* 为返回的promise。
    4. 完成后p含有值 *answer*，将`answer`作为参数调用 *successCallback* 。
    5. 被拒绝的话p会附带原因 *r* ，将 *r* 作为参数调用 *failureCallback* 。
    6. 用`undefine`解析一个promise并返回。
- **setRemoteDescription**：当`setRemoteDescription`方法被调用，用户代理必须按以下步骤运行：
    1. 设 *description* 为方法的第一个参数。
    2. 设 *successCallback* 为方法的第二个参数，表示一个回调。
    3. 设 *failureCallback* 为方法的第三个参数，表示一个回调。
    4. 将 *description* 作为单独的参数，执行`RTCPeerConnection`的`setRemoteDescription()`中指定的步骤，设 *p* 为返回的promise。

5. p完成后，将`undefined`作为参数调用 *successCallback* 。
6. 被拒绝的话p会附带原因 *r* ，将 *r* 作为参数调用 *failureCallback* 。
7. 用`undefine`解析一个promise并返回。

- **addIceCandidate**：当`addIceCandidate`方法被调用，用户代理必须按以下步骤运行：

1. 设 *candidate* 为方法的第一个参数。
2. 设 *successCallback* 为方法的第二个参数，表示一个回调。
3. 设 *failureCallback* 为方法的第三个参数，表示一个回调。
4. 将 *candidate* 作为单独的参数，执行`RTCPeerConnection`的`addIceCandidate()`中指定的步骤，设 *p* 为返回的promise。
5. p完成后，将`undefined`作为参数调用 *successCallback* 。
6. 被拒绝的话p会附带原因 *r* ，将 *r* 作为参数调用 *failureCallback* 。
7. 用`undefine`解析一个promise并返回。

回调定义

这些回调只被用于旧版API中。

`RTCPeerConnectionErrorCallback`：

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

`RTCPeerConnectionErrorCallback`回调参数： DOMException类型的`error`：封装了出错信息的错误对象。

`RTCSessionDescriptionCallback`：

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

`RTCSessionDescriptionCallback`回调参数： RTCSessionDescriptionInit类型的`description`：一个包含SDP的对象。

旧版配置扩展

除了被添加至`RTCPeerConnection`的媒体数据外，本节描述了一些可能会被用于影响邀请创建行为的旧版扩展。我们鼓励开发者使用`RTCRtpTransceiver`的API。 当`createOffer`被任何本节中指定的旧版选项调用时，执行以下步骤而不是常规的`createOffer`步骤：

1. 设 *options* 为方法的第一个参数。
2. 设 *connection* 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：
    1. 如果字典成员的值为false：
        1. 对于每个未停止的"sendrecv"类别的收发器 *transceiver* ，设 *transceiver* 的[Direction]槽为"sendonly"。
        2. 对于每个未停止的"recvonly"类别的收发器 *transceiver* ，设 *transceiver* 的[Direction]槽为"inactive"。 如果有下一选项，继续此步骤。
    2. 如果 *connection* 有任何为停止的"sendrecv"或"recvonly"类别的收发器 *transceiver* ，继续下一个选项。
    3. 设 *transceiver* 为调用`connection.addTransceiver(kind)`的结果，这个操作绝不能更改协商所必须的标记位。
    4. 如果因为前面的步骤抛出了错误，使得 *transceiver* 未被设置，则终止步骤。

5. 设 *transceiver* 的[Direction]槽为"recvonly"。

4. 运行`createOffer`中指定的步骤来创建邀请。

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

属性

- boolean类型的`offerToReceiveAudio`：此设置提供对音频方向的额外控制。 例如，无论是否发送音频，它都可用于确保可以接收音频。
- boolean类型的`offerToReceiveVideo`：此设置提供对视频方向的额外控制。 例如，无论是否发送视频，它都可用于确保可以接收视频。

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。当对象内部的[IsClosed]槽值为`true`，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。 所有`RTCDataChannel`和`MediaStreamTrack`都是以强引用地形式连接到`RTCPeerConnection`对象上的。

4.5 错误处理

4.5.1 通用原则

所有返回promise的方法都由promise的标准错误处理规则接管。不返回promise的方法可能会抛出异常来表示错误。

4.6 会话描述模型

4.6.1 RTCSdpType

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

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

枚举值描述：

- offer：`RTCSdpType`类型的`offer`表示该描述必须被视作一个SDP邀请。
- pranswer：`RTCSdpType`类型的`pranswer`表示该描述必须被视作一个SDP应答，但不是一个最终应答。一个SDP `pranswer`的描述可以应用作为SDP邀请的响应，或作为先前发送的SDP `pranswer`的更新。
- answer：`RTCSdpType`类型的`answer`表示该描述必须被视作一个SDP最终应答，并且邀请-应答的交换过程被视作已结束了。一个SDP `answer`的描述可以应用作为SDP邀请的响应，或作为先前发送的SDP `pranswer`的更新。

- rollback：`RTCSdpType`类型的`rollback`表示该描述必须被视作取消当前SDP协商，移动SDP邀请并回复先前稳定的状态。注意如果当前还没有达成邀请-应答的协商，则先前稳定状态中的本地或远程SDP描述可能是空的。

4.6.2 RTCSessionDescription类

`RTCSessionDescription`类被`RTCPeerConnection`用于暴露本地或远程会话描述。

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

构造函数：

- **RTCSessionDescription** ：`RTCSessionDescription()`构造函数接收一个字典参数， *descriptionInitDict* ，其内容被用来初始化一个新`RTCSessionDescription`对象。本构造函数已被弃用，它的存在只是出于后向兼容性的考虑。

属性：

- RTCSdpType类型的`type`，只读：本`RTCSessionDescription`的类型。
- DOMString类型的`sdp`，只读：代表SDP的字符串。

方法：

- *toJSON()* ：被调用时执行WEBIDL的默认toJSON操作。

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

`RTCSessionDescriptionInit`字典成员：

- RTCSdpType类型的`type`，必须项：DOMString sdp
- DOMString类型的`sdp`：代表SDP的字符串。如果`type`为`rollback`，则不会使用此成员。

4.7 会话协商模型

为了达到预期效果，`RTCPeerConnection`的很多状态改变都需要通过信令通道与远程端通信。应用通过监听`negotiationneeded`事件，可以在需要进行信号传递的时候一直收到通知。根据表示在[NegotiationNeeded]槽内的连接 **negotiation-needed** 标记位的状态，事件被触发。

4.7.1 设置是否需要协商标记位

如果在`RTCPeerConnection`上执行的操作需要信令，则该连接将被标记为需要协商。此类操作包括添加或停止`RTCRtpTransceiver`，或添加第一个`RTCDataChannel`。 具体实现内部的变化也可能导致连接被标记为需要协商。注意，具体的更新协商标记位的程序在下方指定。

4.7.2 清除是否需要协商标记位

当`RTCSessionDescription`的应用类型为`"answer"`时，清除是否需要协商的标志，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，这意味着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。 注意，更新是否需要协商标记位的程序在下方指定。

4.7.3 更新是否需要协商标记位

以下过程在本文档的其他地方被引用。 它也可能随着实现中影响协商的内部变化而发生。如果发生此类更改，用户代理必须将更新需要协商标记位的任务加入队列。 为了 **更新是否需要协商的标记位** ，运行以下步骤：

1. 如果 *connection* 的[IsClosed]槽为`true`，终止后续步骤。
2. 如果 *connection* 的信令状态不是`stable`，终止后续步骤。 **注意：作为设置RTCSessionDescription步骤的一部分，一旦状态转移为"stable"，是否需要协商的标记位就会被更新**
3. 如果检查是否需要协商的结果为`false`，通过设 *connection* 的[NegotiationNeeded]槽为`false`将是否需要协商标记位清除，并终止后续步骤。
4. 如果 *connection* 的[NegotiationNeeded]槽已为`true`，终止后续步骤。
5. 将 *connection* 的[NegotiationNeeded]槽设为`true`。
6. 将包含以下步骤的任务加入队列：
    1. 如果 *connection* 的[IsClosed]槽为`true`，终止后续步骤。
    2. 如果 *connection* 的[NegotiationNeeded]槽为`false`，终止后续步骤。
    3. 在 *connection* 上触发名为`negotiationneeded`的事件。 **注意：将`negotiationneeded`事件入队防止了过早触发，在普通的场景中多个修改会一次发生。**

为了检查 *connection* 是否需要协商，执行以下检查：

1. 如果需要实现指定的协商，如本节刚开始中提到的，则返回`true`。
2. *description* 即 *connection.[CurrentLocalDescription]* 。
3. 如果 *connection* 创建了`RTCDataChannel`对象，且 *description* 中的m=字段没有协商获取数据，则返回`true`。
4. 对于 *connection* 中的每个收发器 *transceiver* ，执行以下检查：
    1. 如果 *transceiver* 没有`stopped`，且没有与 *description* 中的m=字段相关联，则返回`true`。
    2. 如果 *transceiver* 没有`stopped`，且已与 *description* 中的m=字段相关联，则执行一下检查：
        1. 如果 *transceiver.[Direction]* 为`sendrecv`或`sendonly`，且相关联的m=字段不包含单独的一行"a=msid"，或"a=msid"中包含MSIDs的数量，或包含MSID值本身，但MSID值与 *tranceiver.sender.[AssociatedMediaStreamIds]* 不同，则返回`true`。
        2. 如果 *description* 的类型为`offer`，且 *coonnection.[CurrentLocalDescription]* 或 *connection.[CurrentRemoteDescription]* 中关联的m=字段中的方向与 *tranceiver.[Direction]* 都不匹配，则返回`true`。
        3. 如果 *description* 的类型为`answer`，且 *description* 的m=字段中的方向与给定方向相交的 *tranceiver.[Direction]* 不匹配，（这在JSEP 5.3.1有定义），则返回`true`。
    3. 如果 *transceiver* 已`stopped`，且与m=字段相关联，但m=字段还没被 *connection.[CurrentLocalDescription]* 或 *connection.[CurrentRemoteDescription]* 拒绝，则返回`true`。

4.8 连接建立接口

4.8.1 `RTCIceCandidate`接口

该接口描述了ICE候选地址，在ICE第二节中描述。除了`candidate, sdpMid, sdpMLineIndex, usernameFragment`，其余的属性都从 *candidateInitDict* 的`candidate`成员中派生，前提是它们格式完好。

```
[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** ：`RTCIceCandidate()`构造函数接收一个字典参数， *candidateInitDict* ，其内容被用来初始化新`RTCIceCandidate`对象。
  当它被调用时，运行以下步骤：
  1. 如果 *candidateInitDict* 中的`sdpMid`和`sdpMLineIndex`字典成员都为`null`，抛出一个`TypeError`错误。
  2. 设 *iceCandidate* 即新创建的`RTCIceCandidate`对象。
  3. 将 *iceCandidate* 的以下属性置为`null`：`foundation, component, priority, address, protocol, port, type, tcpType, relatedAddress, relatedPort`。
  4. 将 *iceCandidate* 的以下属性设备 *candidateInitDict* 中的对应值：`candidate, sdpMid, sdpMLineIndex, usernameFragment`。
  5. 设 *candidate* 为 *candidateInitDict* 中的`candidate`成员。如果 *candidate* 不是一个空字符串，则运行以下步骤：
     1. 使用`candidate-attribute`语法解析 *candidate* 。
     2. 如果上一步解析失败，终止步骤。
     3. 如果解析结果中的任何字段表示 *iceCandidate* 中相应属性的非法值，则中止这些步骤。
     4. 将 *iceCandidate* 中的相应属性设置为解析结果的字段值。
  6. 返回 *iceCandidate* 。

> 注意：`RTCIceCandidate`的构造函数仅对`candidateInitDict`中的字典成员进行基本的解析和类型检查。在将`RTCIceCandidate`对象传递给`addIceCandidate()`时，会完成对`candidate`，`sdpMid`，`sdpMLineIndex`，`usernameFragment`以及相应会话描述的格式完整性的详细验证。 为了保持向后兼容性，解析候选属性时发生的任何错误都将被忽略。 在这种情况下，`candidate`属性保存在`candidateInitDict`给定的原始`candidate`字符串中，但是诸如`foundation, priority`等派生的属性被设为`null`。

属性: 以下大多数属性都被定义在ICE的15.1中。

- DOMString类型的`candidate`，只读：它携带了[ICE]第15.1中定义的`candidate-attribute`。如果这个`RTCIceCandidate`代表了候选地址结束的指示，`candidate`是一个空字符串。
- DOMString类型的`sdpMid`，只读，可空：如果不为`null`，它将包含RFC5888中定义的该候选地址关联的媒体组件中的媒体流"识别标签"。
- unsigned short类型的`sdpMLineIndex`，只读，可空：如果不为`null`，它表示该候选地址关联的SDP中媒体描述的索引值（从0开始）。
- DOMString类型的`foundation`，只读，可空：允许ICE关联出现在多个`RTCIceTransport`上候选地址的唯一标识符。
- RTCIceComponent类型的`component`，只读，可空：赋予候选地址的网络组件(`rtp`或`rtcp`)。这对应于`candidate-attribute`中的`component-id`字段，解码为`RTCIceComponent`中定义的字符串表示。
- unsigned long类型的`priority`，只读，可空：赋予候选地址的优先级。
- DOMString类型的`address`，只读，可空：候选地址的地址，可以是IPv4，IPv6，或全限定域名（FQDN）。它对应于`candidate-attribute`中的`connection-address`字段。
  注意：候选者中公开的地址通过ICE收集并对`RTCIceCandidate`实例中的应用程序可见，这可以泄露有关设备和用户的更多信息（例如位置，本地网络拓扑），而不是用户在未启用WebRTC的浏览器中期望信息。
  这些地址会一直对应用公开，并可能对沟通方公开，也可能未经用户同意就公开（例如，对于在数据通道中使用的或只接收媒体的对等连接）。
  这些地址也可被用于暂时或持久的跨源状态，因此对设备的指纹表面有利。（这是一个指纹向量。）
  通过设置`RTCConfiguration`的`iceTransportPolicy`成员强制ICE代理只报告中继候选地址，应用可暂时或永久地避免将地址暴露给沟通方。
  为了限制暴露给应用本身的地址，浏览器可以向它们的用户提供不同的策略而不是共享本地地址，这定义在RTCWEB-IP-HANDLING。
- RTCIceProtocal类型的`protocal`，只读，可空：候选地址的协议(`udp`或`tcp`)。对应于`candidate-attribute`中的`transport`字段。
- unsigned short类型的`port`，只读，可空：候选地址的端口。
- RTCIceCandidateType类型的`type`，只读，可控：候选地址的类型。对应于`candidate-attribute`中的`candidate-types`字段。
- RTCIceTcpCandidateType类型的`tcpType`，只读，可空：如果`protocal`为`tcp`，则`tcpType`代表TCP候选地址的类型。否则，`tcpType`为`null`。对应于`candidate-attribute`中的`tcp-type`字段。
- DOMString类型的`relatedAddress`，只读，可空：对于从别的候选地址（中继或反射候选地址）例如派生出的候选地址，`relatedAddress`即派生源的IP地址。对于主机候选地址，`relatedAddress`为`null`。对应于`candidate-attribute`中的`real-address`字段。
- unsigned short类型的`relatedPort`，只读，可空：对于从别的候选地址（中继或反射候选地址）例如派生出的候选地址，`relatedPort`即派生源的IP端口。对于主机候选地址，`relatedPort`为`null`。对应于`candidate-attribute`中的`real-port`字段。
- DOMString类型的`usernameFragment`，只读，可空：它携带了[ICE]15.4节中定义的`ufrag`。

方法：

- *toJSON()* ：调用`RTCIceCandidate`的`toJSON()`操作将按以下步骤运行：

1. 设 *json* 即新`RTCIceCandidateInit`字典。 ，
2. 对于<"candidate", "sdpMid", "sdpMLineIndex", "usernameFragment">中的每个属性标识符 *attr*：
    1. 给定本`RTCIceCandidate`对象，设 *value* 为获取到的 *attr* 底层值。
    2. 设 `json[attr]=value`。
3. 返回 *json* 。

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

`RTCIceCandidateInit`字典成员：

- DOMString类型的`candidate`，缺省值为`""`：它携带了ICE 15.1节中定义的`candidate-attributes`。如果这表示候选地址结束指示，则`candidate`是空字符串。
- DOMString类型的`sdpMid`，可空，缺省值为`null`：如果非`null`，它将包含与此候选地址相关联的媒体组件的媒体流"识别标签"，这在RFC5888中定义。
- unsigned short类型的`sdpMLineIndex`，可空，缺省值为`null`：如果非`null`，它表示该候选地址关联的SDP中媒体描述的索引值（从0开始）。
- DOMString类型的`usernameFragment`：它携带了[ICE]15.4节中定义的`ufrag`。

4.8.1.1 candidate-attribute语法

`candidate-attribute`语法被用于解析`RTCIceCandidate()`构造函数中`candidateInitDict`变量的`candidate`成员。 `candidate-attribute`的主要语法被定义在[ICE]的15.1节。除此之外，浏览器必须支持RFC6544中定义的ICE TCP语法扩展。 浏览器也可以支持RFC中定义的其他语法扩展。

4.8.1.2 RTCIceProtocal枚举

`RTCIceProtocal`代表ICE候选地址的协议。

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

枚举值描述：

- `udp`：UDP类型候选地址，ICE中有相关描述。
- `tcp`：TCP类型候选地址，[RFC6544]中有相关描述。

4.8.1.3 RTCIceTcpCandidateType枚举

`RTCIceTcpCandidateType`代表了ICE TCP类型的枚举值，定义在[RFC6544]

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

枚举值描述：

- `active`：`active`类型的TCP候选地址通道主动建立对外的ice连接，而不接受连接请求。
- `passive`：`passive`类型的TCP候选地址通道总是接收ice连接请求，而不主动做连接。
- `so`：`so(simultaneous-open)`类型的TCP候选地址通道与对端so类型的候选地址做ice连接（优先级高与上面的两个）。

> 注意：用户代理通常只会收集`active`类型的ICE TCP候选地址。

4.8.1.4 RTCIceCandidateType枚举

`RTCIceCandidateType`代表ICE候选地址的类型，定义在[ICE]15.1节。

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

枚举值描述：

- `host`：主机候选地址，定义在ICE 4.1.1.1。
- `srflx`：服务器反射候选地址，定义在ICE 4.1.1.2。
- `prflx`：对等反射候选地址，定义在ICE 4.1.1.2。
- `relay`：中继候选地址，定义在ICE 7.1.3.2.1。

4.8.2 RTCPeerConnectionIceEvent

`RTCPeerConnection`的`icecandidate`事件使用了`RTCPeerConnectionIceEvent`接口。 当触发一个包含`RTCIceCandidate`对象的`RTCPeerConnectionIceEvent`事件时，对象必须包含`sdpMid`和`sdpMLineIndex`的值。如果`RTCIceCandidate`的类型为`srflx`或`relay`，事件的`url`属性必须被设置为候选地址获得的ICE服务器的URL地址。

> 注意：`icecandidate`有三种不同类型的表示：
>
> - 收集到了一个候选地址时，`RTCIceCandidate`事件就会触发，本端得到这个候选地址后，应该通过信号通知对端，对端通过`addIceCandidate`方法将接收到的候选地址设置到自己的`peerConnection`实例中。
> - 某一`RTCIceTransport`已结束一代候选地址的收集工作，并且提供了TRICKLE-ICE8.2节中定义的候选地址结束指示。这通过将`candidate.candidate`设为空字符串来表示。`candidate`对象应该发信号通知远程对端并向普通ICE候选地址一样传入`addIceCandidate`方法，以向远程对端提供候选地址结束指示。所有`RTCIceTransport`已结束候选地址的收集工作，且`RTCPeerConnection`的`RTCIceGatheringState`已迁移至`complete`。这通过将事件的`candidate`成员设为`null`来表示。它只

> 为了后向兼容性存在，并且本事件不需要通知远程对端。它与`"complete"`状态的`icegatheringstatechange`事件等效。

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

构造函数：

- RTCPeerConnectionIceEvent

属性：

- RTCIceCandidate类型的`candidate`，只读，可空：`candidate`属性是中造成事件的新ICE候选地址`RTCIceCandidate`对象。
  当生成的事件代表候选地址收集结束的时候，本属性被设为`null`。
  注意：即使存在多个媒体组件，也只会出发一个包含`null`候选地址的事件
- DOMString类型的`url`，只读，可空：`url`属性是在收集候选地址时被用于识别STUN/TURN服务器的STUN/TURN URL地址。如果候选地址并不是从STUN/TURN服务器收集来的，本参数为`null`。

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

`RTCPeerConnectionIceEventInit`字典成员：

- RTCIceCandidate类型的`candidate`，可空：详情请见`RTCPeerConnectionIceEvent`接口的`candidate`属性。
- DOMString类型的`url`，可空：`url`属性是在收集候选地址时被用于识别STUN/TURN服务器的STUN/TURN URL地址。

4.8.3 RTCPeerConnectionIceErrorEvent

RTCPeerConnection中的`icecandidateerror`事件使用了`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：

- DOMString类型的`hostCandidate`，只读：`hostCandidate`属性是被用来与STUN/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与服务器通信，该属性允许应用程序确定故障发生在哪一个接口上。
  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。
- DOMString类型的`url`，只读：`url`属性标识了可能发生故障的STUN/TURN服务器的URL地址。
- unsigned short类型的`errorCode`，只读：`errorCode`属性是STUN/TURN服务器返回的数字错误码，详情见STUN-PARAMETERS。
  如果没有主机候选地址可以到达服务器，`errorCode`将被设成STUN错误码范围外的701。在`RTCIceGatheringState`的"收集"阶段，每个服务器URL只会触发一次这个错误。
- USVString类型的`errorText`，只读：`errorText`属性是STUN/TURN服务器返回的错误响应文本。
  如果服务器不能到达，`errorText`将被设为一个具体实现指定的值，指示错误的细节。

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

`RTCPeerConnectionIceErrorEventInit`字典成员：

- DOMString类型的`hostCandidate`：与STUN/TURN服务器通信的本地地址和端口。
- DOMString类型的`url`：指示发生错误的STUN/TURN服务器的URL地址。
- unsigned short类型的`errorCode`，必须项：STUN/TURN服务器返回的数字错误码。
- USVString类型的`statusText`：STUN/TURN服务器返回的状态响应文本。

4.9 优先级和服务质量(QoS)模型

许多应用程序具有相同数据类型的多个媒体流，并且通常一些流程比其他流程更重要。 WebRTC使用RTCWEB-TRANSPORT和TSVWG-RTCWEB-QOS中描述的优先级和服务质量（QoS）框架为有助于在某些网络环境中提供QoS的数据包提供优先级和DSCP标记。优先级设置可用于指示各种流的相对优先级。优先级API允许JavaScript应用程序通过将`RTCRtpEncodingParameters`对象的`priority`属性设置为以下值之一来告诉浏览器特定媒体流对于应用程序来说，重要性是高，中，低还是非常低。

4.9.1 RTCPriorityType枚举

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

枚举值描述：

- `very-low`：详情见RTCWEB-TRANSPORT，第4.1和4.2节。对应于RTCWEB-DATA中定义的"below normal"。
- `low`：详情见RTCWEB-TRANSPORT，第4.1和4.2节。对应于RTCWEB-DATA中定义的"normal"。
- `medium`：详情见RTCWEB-TRANSPORT，第4.1和4.2节。对应于RTCWEB-DATA中定义的"high"。

- `high`：详情见RTCWEB-TRANSPORT，第4.1和4.2节。对应于RTCWEB-DATA中定义的"extra high"。

使用此API的应用程序应该意识到，通过降低不重要的事物的优先级而不是提高重要事物的优先级，通常可以获得更好的整体用户体验。

4.10 证书管理

`RTCPeerConnection` 实例使用的对对等连接进行鉴权的证书使用了 `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。 显式的证书管理功能是可选的。如果构造一个新 `RTCPeerConnection` 对向时，应用没有提供 `certificates` 配置选项，则用户代理必须生成新的证书集合。集合必须包括一个携带P-256曲线私钥，SHA-256哈希签名的ECDSA证书。

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

方法：

- 静态 *generateCertificate* 方法：`generateCertificate` 函数使用户代理创建并存储一个X.509证书X509V3及对应的私钥。 `RTCCertificate` 接口的表单中提供了一个信息句柄。返回的 `RTCCertificate` 可被用于控制 `RTCPeerConnection` 创建的DTLS会话中提供的证书。
  keygenAlgorithm 参数被用于控制于证书关联的私钥是如何生成的。 *keygenAlgorithm* 算法使用WebCrypto[http://w3c.github.io/webrtc-pc/#bib-WebCryptoAPI]的 `AlgorithmIdentifier` 类型。 *keygenAlgorithm* 值对于 `window.crypto.subtle.generateKey` 必须是一个合法参数，也就是说，当根据WebCrypto算法规范化过程WebCryptoAPI进行规范化时，值必须不能产生错误，其中操作名称为 `generateKey`，并且supportedAlgorithms值特定于RTCPeerConnection证书的生成。如果算法规范化的过程产生了一个错误，`generateCertificate` 调用必须以这个错误拒绝返回。
  生成的密钥中提供的签名被用于DTLS连接的鉴权。被标识的算法（由标准化 `AlgorithmIdentifier` 的 `name` 字段标识）必须是可用于产生签名的非对称算法。
  该过程产生的证书同样包含了一个签名。签名的有效性仅与兼容性相关。只有公钥和生成的证书指纹被 `RTCPeerConnection` 所用，但如果证书格式正确，则证书更有可能被接受。浏览器会选择签署证书的算法，如果需要哈希算法，浏览器应该选择SHA-256算法。
  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，则用户代理必须以一个 `NotSupportedError` 类型的 `DOMException` 拒绝对 `generateCertificate()` 的调用。
  以下值必须被用户代理所支持：`{ name: "RSASSA-PKCS1-v1_5", modulusLength: 2048, publicExponent: new Uint8Array([1, 0, 1]), hash: "SHA-256" }`, and `{ name: "ECDSA", namedCurve: "P-256" }`.。
  注意：预计用户代理将接受的值的集合较小，甚至是固定的。

4.10.1 RTCCertificateExpiration字典

`RTCCertificateExpiration` 被用于为 `generateCertificate` 生成的证书设置一个过期日期。

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

- `expires`：一个可选的`expires`属性可能被添加入传至`generateCertificate`的算法中。如果参数存在，则代表`RTCCertificate`相对于当前时间的最长有效时间。
  当以一个`object`参数调用`generateCertificate`时，用户代理会尝试将object转换为`RTCCertificateExpiration`类型。如果转换失败，会立即以一个新创建的`TypeError`拒绝并返回一个promise，然后终止后续步骤。
  用户代理生成的证书的过期时间为当前时间加上`expires`的值。返回的`RTCCertificate`中的`expires`属性被设为证书的过期时间。用户代理可选择限制`expires`属性值的大小。

4.10.2 RTCCertificate接口

`RTCCertificate`接口代表了一个用于WebRTC通信鉴权的证书。除了可见的属性，内部槽包含了生成的私有密钥子集[KeyingMaterial]的句柄，`RTCPeerConnection`与对端进行身份验证的证书[Certificate]，以及创建的对象的源[Origin]。

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

属性：

- DOMTimeStamp类型的`expires`，只读：`expires`属性指示了相对于1970-01-01T00：00：00Z的日期和时间（以毫秒为单位），在这之后的时间浏览器将认为证书失效。在这时间之后，利用此证书尝试创建`RTCPeerConnection`的操作都将失效。
  注意该值不一定在证书本身的`notAfter`参数中有所体现。

方法：

- *getSupportedAlgorithms* ：返回提供一组代表支持证书算法的序列。必须返回至少一个算法。

> 注意：例如，"RSASSA-PKCS1-v1_5"算法字典，`RsaHashedKeyGenParams`，包含了模数长度，公共指数和哈希算法的字段。实现可能支持大范围的模数长度和指数，以及有限数目的哈希算法。在这种场景下，实现为每个支持的RSA哈希算法返回一个`算法标识符`是合理的，对于`modulusLength`和`publicExponent`使用默认或推荐值（比如1024或65537）。

- *getFingerprints* ：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。

出于此API的目的，[Certificate]槽中包含未结构化的二进制数据。目前还没有提供访问[KeyingMaterial]的机制。实现必须支持应用从持久化存储中保存和取回`RTCCertificate`对象。在实现中，`RTCCertificate`可能不会直接持有私钥资料（它可能存储在某一安全模块中），私钥的引用可以保存在[KeyingMaterial]内部槽中，其中的私钥可以被存储的也可以被使用。 `RTCCertificate`对象是可序列化对象。给定 *value* 和 *serialized* ，它们的序列化步骤如下：

1. 设 *serialized.[Expires]* 值为 *value* 的`expires`属性。
2. 设 *serialized.[Certificate]* 值为 *value* 的[Certificate]槽中未结构化的二进制数据的一份拷贝。
3. 设 *serialized.[Origin]* 值为 *value* 的[Origin]槽中未结构化的二进制数据的一份拷贝。
4. 设 *serialized.[KeyingMaterial]* 值为 *value* 的[KeyingMaterial]槽中表示私钥材料序列化的结果。

给定 *value* 和 *serialized* ，它们的反序列化步骤如下：

1. 初始化 *value* 的`expires`属性为 *serialized.[Expires]* 。

2. 设 *value.[Certificate]* 槽值为 *serialized.[Certificate]* 的一份拷贝。
3. 设 *value.[Origin]* 槽值为 *serialized.[Origin]* 的一份拷贝。
4. 设 *value.[KeyingMaterial]* 槽值为 *serialized.[KeyingMaterial]* 反序列化后生成的私钥材料。

> 注意：以这种方式支持结构化克隆使得`RTCCertificate`实例持久化到存储成为可能。它还允许使用`postMessage`webmessaging这样的API将实例传递给其他源。但是，对象不能被除最初创建它的源以外的其他任何源使用。

5. RTP媒体API

RTP媒体API 使得网络应用可以在端到端对等连接上发送并接收`MediaStreamTrack`流媒体轨对象。当媒体轨被添加至`RTCPeerConnection`时会导致信令发送信号；当本信号被转发至远程对端，对应的媒体轨会在远程一侧被创建。

> 注意：`RTCPeerConnection`发送的媒体轨与另一`RTCPeerConnection`接收的媒体轨之间没有确切的1：1对应关系。比如，被发送的媒体轨的ID与被接受的媒体轨的ID不存在映射关系。同样的，即使在接收端没有创建新的媒体轨，`replaceTrack`调用也能改变`RTCRtpSender`发出的媒体轨，对应的`RTCRtpReceiver`只会持有一个媒体轨，此媒体轨可能代表了整合在一起的多个媒体数据源。`addTransceiver`和`replaceTrack`调用都可被用于多次发送同一个媒体轨，在接收端每个轨都会被单独的接收器所观察。因此，考虑将`RTCRtpSender`与另一侧`RTCRtpReceiver`之间的媒体轨建立起1：1的关系会更为准确，如果有需要的话可以使用`RTCRtpTransceiver`的`mid`值来匹配发送端和接收端。

发送媒体数据时，发送方可能需要重新调整或重新采样媒体以满足各种要求，包括SDP协商的信封。 跟从JSEP 3.6节中的规则，视频的尺寸也许会被缩小以适应SDP的约束。媒体不得为了创建未在输入源中出现的伪数据而扩大规模，除非需要满足像素计数的约束，否则不得裁剪媒体，不得更改宽高比。

> WebRTC工作组正在寻求关于需求与时间线实现的反馈，以便更复杂地处理这种情况。一些可能的设计方案已经在Github issue 1283中被讨论了。

当视频被重新调整，例如对于某一宽度或高度和`scaleResolutionDownBy`值的组合，可能出现宽度或高度不是整数的情况。这种情况下用户代理必须使用结果的整数部分。如果缩放后的宽度或高度的整数部分为零，则传输的内容由具体实现指定。 `MediaStreamTrack`的实际编码与传输过程由名为`RTCRtpSender`的对象管理。类似的，`MediaStreamTrack`的实际接收与解码过程由名为`RTCRtpReceiver`的对象管理。每个`RTCRtpSender`对象至多与一个媒体轨相关联，每个被接收的媒体轨也只能与一个`RTCRtpReceiver`关联。 每个`MediaStreamTrack`都应该被编码和传输，使其的特性（视频轨道的宽度，高度和帧率；音频轨道的体积，采样尺寸，采样率和通道数）与远程端创建的媒体轨保持一致。某些情况下也可能不适用，比如可能会在端点或网络中出现资源限制，也可能`RTCRtpSender`应用的设置指示实现表现出不同的行为。 一个`RTCPeerConnection`对象包含`RTCRtpTransceiver`的 **一个集合** ，代表了共享某些状态的发送端/接收端对。这个集合在`RTCPeerConnection`对象创建时被初始化为空集合。`RTCRtpSender`和`RTCRtpReceiver`总是由`RTCRtpTransceiver`同时创建，这样在它们的生命周期中可以保持关联。当应用通过`addTrack`方法将一个`MediaStreamTrack`附加到`RTCPeerConnection`对象上时，`RTCRtpTransceiver`会被隐式创建，应用使用`addTransceiver`方法时它会被显式创建。当一个包含新媒体描述的远程描述被应用时，`RTCRtpTransceiver`也会被创建。此外，当表示远程端点含有媒体数据要发送的远程描述被应用时，相关的`MediaStreamTrack`和`RTCRtpReceiver`会通过`track`事件被暴露给应用。

5.1 RTCPeerConnection接口扩展

RTC媒体API对以下的`RTCPeerConnection`接口作了扩展。

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

属性：

- EventHandler类型的`ontrack`：该事件句柄的事件类型为`track`。

方法：

- *getSenders*：返回一组代表RTP发送端的`RTCRtpSender`对象序列，这些对象当前正附加到`RTCPeerConnection`对象上，且属于未停止的`RTCRtpTransceiver`对象。
  当`getSenders`方法被调用，用户代理必须返回[发送端收集算法]的执行结果。
  发送端收集算法 的执行结果如下：
  1. 设 *transceivers* 为收发器收集算法的执行结果。
  2. 设 *senders* 为一个新的空序列。
  3. 对 *transceivers* 中的每个对象：
     1. 若对象的[Stopped]槽值为`false`，则将此对象的[Sender]槽加入 *senders* 。
  4. 返回 *senders* 。
- *getReceivers*：返回一组代表RTP接收端的`RTCRtpReceiver`对象序列，这些对象当前正附加到`RTCPeerConnection`对象上，且属于未停止的`RTCRtpTransceiver`对象。
  当`getReceivers`方法被调用，用户代理必须按以下步骤运行：
  1. 设 *transceivers* 为收发器收集算法的执行结果。
  2. 设 *receivers* 为一个新的空序列。
  3. 对 *transceivers* 中的每个对象：
     1. 若对象的[Stopped]槽值为`false`，则将此对象的[Receiver]槽加入 *receivers* 。
  4. 返回 *receivers* 。
- *getTransceivers*：返回一组代表RTP发送端的`RTCRtpTransceiver`对象序列，这些对象当前正附加到`RTCPeerConnection`对象上。
  `getTransceiver`方法必须返回[收发器收集算法]的执行结果。
  收发器收集算法 的执行结果如下：
  1. 设 *transceivers* 为以插入顺序排列的新`RTCRtpTransceiver`对象序列，对象来自`RTCPeerConnection`的收发器列表。
  2. 返回 *transceivers* 。
- *addTrack*：将一个被指定媒体流`MediaStream`对象包含的新媒体轨加入`RTCPeerConnection`。
  当`addTrack`方法被调用，用户代理必须按以下步骤运行：
  1. 设 *connection* 为调用方法的`RTCPeerConnection`对象。

2. 设 *track* 为作为方法第一个参数的`MediaStreamTrack`对象。
3. 设 *kind* 为 *track.kind* 。
4. 设 *streams* 为从方法剩余参数构造的`MediaStream`对象列表，如果方法被调用时只有一个参数，则为空列表。
5. 若 *connection* 的[IsClosed]槽值为`true`，抛出一个`InvalidStateError`。
6. 设 *senders* 为发送端收集算法的执行结果。如果 *senders* 中已存在一个用来发送 *track* 的发送端`RTCRtpSender`对象，则抛出一个`InvalidAccessError`。
7. 以下步骤描述了如何确定是否可以复用现有发送端。根据JSEP 5.2.2&5.3.2中的描述，这将导致未来调用的`createOffer`和`createAnswer`方法将相应的媒体描述标记为`sendrecv`或`sendonly`，并添加发送端的MSID。
   如果 *senders* 中的某个`RTCRtpSender`对象与以下所有原则相匹配，则让 *sender* 为那个对象，否则为`null`：
   - sender的媒体轨为空。
   - 与发送端相关联transceiver的类型为`RTCRtpTransceiver`，并与 *kind* 匹配。
   - 与`RTCRtpTransceiver`相关联的sender的[Stopped]槽值为`false`。
   - sender从来没有被使用过。更准确地说，与`RTCRtpTransceiver`相关联的sender的[CurrentDirection]槽值为`sendrecv`或`sendonly`。
8. 如果 *sender* 非空，运行以下步骤来使用sender：
   1. 将 *track* 赋给 *sender* 的[SenderTrack]。
   2. 将 *sender* 的[AssociatedMediaStreamIds]槽设为空集合。
   3. 对于 *streams* 中的每个 *stream* ，若[AssociatedMediaStreamIds]槽中尚未包含其`id`，则将`stream.id`加入。
   4. 设 *transceiver* 成为于 *sender* 相关联的`RTCRtpTransceiver`对象。
   5. 如果 *transceiver* 的[Direction]槽值为`recvonly`，则将此槽的值设为`sendrecv`。
   6. 如果 *transceiver* 的[Direction]槽值为`inactive`，则将此槽的值设为`sendonly`。
9. 如果 *sender* 为空，则运行以下步骤：
   1. 以 *track, kind, streams* 为参数创建RTCRtpSender，并把结果赋给 *sender* 。
   2. 以 *kind* 为参数创建RTCRtpReceiver，并把结果赋给 *receiver* 。
   3. 以 *sender, receiver* 和一个值为`sendrecv`的`RTCRtpTransceiverDirection`对象作为参数创建RTCRtpTransceiver，并把结果赋给 *transceiver* 。
   4. 将 *transceiver* 添加到 *connection* 的收发器集合中。
10. 媒体轨可能包含应用程序无法访问的内容。这可能是由于被标记了`peerIdentity`选项或任何可能追踪CORS跨源数据的操作造成的。这些媒体轨可被用于`addTrack`方法，并会为它们创建`RTCRtpSender`，但除非内容标有`peerIdentify`并且符合发送要求（详见隔离媒体流），否则不能被发送。
    所有应用无法访问的媒体流不能被发送至对等连接，将用静音（音频），黑帧（视频）或其他等效的内容替代媒体轨中的内容。
    注意，这一属性以后可能会变化。
11. 更新 *connection* 的协商标记位。
12. 返回 *sender* 。

- *removeTrack*：停止 *sender* 发送媒体数据。`RTCRtpSender`仍然存在于`getSenders`中。如JSEP 5.2.2中所述，这么做会导致未来的`createOffer`调用将对应收发器中的媒体描述标记为`recvonly`或`inactive`。
  当另一对等连接以同样的方式停止发送柜体会，媒体轨会从最初`track`事件生成的所有远程`MediaStreams`中被移除。如果`MediaStreamTrack`没有被静音，则`muted`事件会在此媒体轨上被触发。

当`removeTrack`方法被调用，用户代理必须按以下步骤运行：

1. 设`removeTrack`的参数为 *sender* 。
2. 设 *connection* 为调用此方法的`RTCPeerConnection`对象。
3. 若 *connection* 的[IsClosed]槽值为`true`，抛出一个`InvalidStateError`。
4. 若 *sender* 没有被 *connection* 创建，抛出一个`InvalidAccessError`。
5. 设 *senders* 为[发送端收集算法]的执行结果。
6. 若 *sender* 不在 *senders* 中（意味着由于将RTCSessionDescription的类型设为了"rollback"，它已被删除），则终止后续步骤。
7. 若 *sender* 的[SenderTrack]槽为空，则终止后续步骤。
8. 将 *sender* 的[SendTrack]槽设为空。
9. 设 *transceiver* 为 *sender* 对应的`RTCRtpTransceiver`对象。
10. 若 *transceiver* 的[Direction]槽值为`sendrecv`，将 *transceiver* 的[Direction]槽值设为`recvonly`。
11. 若 *transceiver* 的[Direction]槽值为`sendonly`，将 *transceiver* 的[Direction]槽值设为`inactive`。
12. 更新 *connection* 的协商标记位。

- *addTransceiver*：创建一个新的`RTCRtpTransceiver`对象并将其加入收发器集合。
  如JSEP 5.2.2中所述，以上添加收发器的动作会导致未来的`createOffer`调用将一个媒体描述添加到对应的收发器中。
  如JSEP 5.5&5.6中所述，`mid`的初始值为空。设置一个新的`RTCSessionDescription`可能会导致它变为一个非空值。
  `senEncodings`参数会被用于指定提供的联播编码的数量，以及选择性的提供它们的RID和编码参数。
  当本方法被调用，用户代理必须按以下步骤运行：

  1. 设 *init* 为第二个参数。
  2. 设 *streams* 为 *init*中的`streams`成员。
  3. 设 *sendEncodings* 为 *init* 中的`sendEncodings`成员。
  4. 设 *direction* 为 *init* 中的`direction`成员。
  5. 若第一个参数为字符串，则设 *kind* 为它并按以下步骤运行：
     1. 若 *kind* 不是一个合法的`MediaStreamTrack kind`，抛出一个`TypeError`。
     2. 设 *track* 为`null`。
  6. 若第一个参数为`MediaStreamTrack`，设 *track* 为它并设 *kind* 为 *track.kind* 。
  7. 如果 *connection* 的[IsClosed]槽值为`true`，抛出一个`InvalidStateError`。
  8. 按以下步骤验证 *sendEncodings* 的合法性：
     1. 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。如果某个RID不符合这些要求，抛出一个`TypeError`。
     2. 如果 *sendEncodings* 中的`RTCRtpEncodingParameters`字典包含除了`rid`之外的只读参数，则抛出一个`InvalidAccessError`。
     3. 验证 *sendEncodings* 中的每个`scaleResolutionDownBy`值大于等于1.0。如果某一`scaleResolutionDownBy`值不符合要求，抛出一个`RangeError`。
     4. 验证 *sendEncodings* 中的每个`maxFramerate`值大于等于0.0。如果某一`maxFramerate`值不符合要求，抛出一个`RangeError`。
     5. 设 *maxN* 为用户代理可能支持的最大同时编码数，最小值为1。这应该是一个乐观的数字，因为尚未知道编解码器的使用方法。
     6. 如果存储在 *sendEncodings* 中的`RTCRtpEncodingParameters`数量超出 *maxN* ，则从尾部开始修剪 *sendEncodings* 直到长度为 *maxN* 。

7. 如果存储在 *sendEncodings* 中的`RTCRtpEncodingParameters`数量为1，则移除其所有`rid`成员。**注意：在`sendEncodings`中提供单个默认的`RTCRtpEncodingParameters`参数，使得应用程序可以在随后未使用联播也能调用`setParameters`设置编码参数。**

9. 以 *track, kind, streams, sendEncodings* 作为参数创建一个RTCRtpSender，并设结果为 *sender* 。如果 *sendEncodings* 是集合，则接下来的`createOffer`调用将被配置用来发送JSEP 5.2.2&5.2.1中的多个RTP编码。当`setRemoteDescription`以一个对应的可以接受多个RTP编码（详见JSEP 3.7）的远程描述调用时，`RTCRtpSender`也许会发送多个RTP编码，且通过收发器的`sender.getParameters`方法取回的参数会体现协商后的编码。

10. 以 *kind* 为参数创建RTCRtpReceiver并将结果设为 *receiver* 。

11. 以 *sender, receiver, direction* 为参数创建RTCRtpTransceiver并将结果设为 *transceiver* 。

12. 将 *transceiver* 添加入 *connection* 的收发器集合。

13. 更新 *connection* 的协商标记位。

14. 返回 *transceiver* 。

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

RTCRtpTransceiverInit字典成员：

- RTCRtpTransceiverDirection类型的`direction`，缺省值为`sendrecv`：`RTCRtpTransceiver`的方向。
- sequence类型的`streams`：当远程对等连接中被触发的`track`事件与被添加的`RTCRtpReceiver`对象相对应时，这些媒体流将被放进track事件中。
- sequence类型的`sendEncodings`：一个包含发送RTP媒体编码所需参数的序列。

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

`RTCRtpTransceiverDirection`枚举值描述：

- sendrecv：`RTCRtpTransceiver`的`RTCRtpSender` *sender* 将向对端发出发送RTP的邀请，若对等端接收邀请，且`sender.getParameters().encodings[i].active`全为`true`的时候，数据将开始发送。`RTCRtpTransceiver`的`RTCRtpReceiver` *receiver* 将发出接收RTP的邀请，在远程对端接收邀请之后会开始接收数据。
- sendonly：`RTCRtpTransceiver`的`RTCRtpSender` *sender* 将向对端发出发送RTP的邀请，若对等端接收邀请，且`sender.getParameters().encodings[i].active`全为`true`的时候，数据将开始发送。`RTCRtpTransceiver`的`RTCRtpReceiver` *receiver* 不会发出接收RTP的邀请，也不会接收数据。
- recvonly：`RTCRtpTransceiver`的`RTCRtpSender` *sender* 不会发出发送RTP的邀请，也不会发送数据。`RTCRtpTransceiver`的`RTCRtpReceiver` *receiver* 将发出接收RTP的邀请，在远程对端接收邀请之后会开始接收数据。

- Inactive：`RTCRtpTransceiver`的`RTCRtpSender` *sender* 不会发出发送RTP的邀请，也不会发送数据。`RTCRtpTransceiver`的`RTCRtpReceiver` *receiver* 不会发出接收RTP的邀请，也不会接收数据。

5.1.1 处理远程媒体流轨

应用可以通过调用`RTCRtpTranceiver.stop()`停用两个方向，并拒绝即将传入的媒体描述，或将收发器的方向设为`sendonly`来拒绝即将到达的那一侧。 给定`RTCRtpTranceiver` *transceiver* 和 *trackEventInits* ，为了**处理远程媒体轨的加入**， 用户代理必须按以下步骤运行：

1. 设 *receiver* 为 *transceiver* 的[Receiver]槽。
2. 设 *track* 为 *receiver* 的[ReceiverTrack]槽。
3. 设 *streams* 为 *receiver* 的[AssociatedRemoteMediaStreams]槽。
4. 将 *receiver, track, streams, transceiver* 作为成员变量，创建一个`RTCTrackEventInit`字典，并把它加入 *trackEventInits* 。

给定`RTCRtpTranceiver` *transceiver* 和 *muteTracks* ，为了**处理远程媒体轨的移除**，用户代理必须按以下步骤运行：

1. 设 *receiver* 为 *transceiver* 的[Receiver]槽。
2. 设 *track* 为 *receiver* 的[ReceiverTrack]槽。
3. 若 *track.muted* 为`false`，则将 *track* 加入 *muteTracks* 。

给定`RTCRtpReceiver` *receiver, msids, addList, removeList* ，为了**设置关联的远程媒体流**，用户代理必须按以下步骤运行：

1. 设 *connection* 为与 *receiver* 相关联的`RTCPeerConnection`对象。
2. 对 *msids* 中的每个MSID，若之前没有以该`id`在本 *connection* 上创建过`MediaStream`对象，则以该`id`创建一个新的`MediaStream`对象。
3. 设 *streams* 为在本 *connection* 上以 *msids* 中的各个`id`创建的`MediaStream`对象列表。
4. 设 *track* 为 *receiver* 的[ReceiverTrack]。
5. 对于 *receiver* 的[AssociatedRemoteMediaStreams]中的每个流，若它尚未存在于 *streams* 中，则将它连同 *track* 作为一对，加入 *removeList* 。
6. 对于 *streams* 中的每个流，若它尚未存在于 *receiver* 的[AssociatedRemoteMediaStreams]中，则将它连同 *track* 作为一对，加入 *addList* 。
7. 将 *receiver* 的[AssociatedRemoteMediaStreams]槽中内容赋给 *streams* 。

5.2 `RTCRtpSender`接口

`RTCRtpSender`接口允许应用控制一个`MediaStreamTrack`如何被编码并被发送至远程对端。当一个`RTCRtpSender`对象的`setParameters`方法被调用时，编码会以合适的方式被改变。 为了用现有的MediaStreamTrack对象 *track* ，字符串 *kind* ，MediaStream对象列表 *streams* 以及可选的`RTCRtpEncodingParameters`对象列表 *sendEncodings* **创建一个RTCRtpSender对象**，运行以下步骤：

1. 设 *sender* 为一个新`RTCRtpSender`对象。
2. *sender* 内部创建[SenderTrack]槽，并初始化为 *track* 。
3. *sender* 内部创建[SenderTransport]槽，并初始化为`null`。
4. *sender* 内部创建[Dtmf]槽，并初始化为`null`。
5. 若 *kind* 值为`audio`，则创建一个`RTCDTMFSender`对象 *dtmf* ，并将其[Dtmf]槽设为 *dtmf* 。
6. *sender* 内部创建[SenderRtcpTransport]槽，并初始化为`null`。
7. *sender* 内部创建[AssociatedMediaStreamIds]槽，代表与 *sender* 相关联的MediaStream对象的`id`列表。如JSEP 5.2.1所述，当在SDP中代表 *sender* 时，[AssociatedMediaStreamIds]槽会被使用。
8. 设 *sender* 的[AssociatedMediaStreamIds]槽为一个空集合。

9. 对于 *streams* 中的每个 *stream* ，若 *stream.id* 不存在于[AssociatedMediaStreamIds]槽，则将它添加进去。
10. *sender* 内部创建[SendEncoding]槽，代表 `RTCRtpEncodingParameters` 字典列表。
11. 若将 *sendEncodings* 作为本算法的输入，且它非空，则将[SendEncodings]槽设为 *sendEncodings* 。否则，将槽的内容设为只包含一个 `RTCRtpEncodingParameters` 对象的列表，其 `active` 成员为 `true` 。
12. *sender* 内部创建[LastReturnedParameters]槽，它将被用于匹配 `getParameters` 和 `setParameters` 事务。
13. 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，可空： `track` 属性是与 `RTCRtpSender` 对象关联的媒体轨。如果 `track` 已结束或它的输出被关闭（例如，媒体轨被关闭或静音），`RTCRtpSender` 必须发送静音（音频），黑帧（视频）或其他不携带信息的等效内容。在视频的场景下，`RTCRtpSender` 应该每秒发送一个黑帧。如果 `track` 为空，则 `RTCRtpSender` 不发送。请求此属性时，该属性必须返回[SenderTrack]槽的值。
- RTCDtlsTransport类型的 `transport` ，只读，可空： `transport` 属性是从 `track` 以RTP分组形式发出的媒体数据的传输通道。在构造 `RTCDtlsTransport` 对象之前， `transport` 属性为空。当使用捆绑，多个 `RTCRtpSenders` 对象将共享同一个 `transport` 并在此传输通道上发送RTP/RTCP。请求此属性时，该属性必须返回[SenderTransport]槽的值。
- RTCDtlsTransport类型的 `rtcpTransport` ，只读，可空： `rtcpTransport` 属性是RTCP发送与接收的传输通道。在构造 `RTCDtlsTransport` 对象之前， `rtcpTransport` 属性为空。当使用RTCP多路复用（或使用捆绑，捆绑用到了RTCP多路复用）， `rtcpTransport` 将为 `null` ，RTP和RTCP数据都会在 `transport` 指定的传输通道上流动。请求此属性时，该属性必须返回[SenderRtcpTransport]槽的值。

方法：

- *getCapabilities*，静态： `getCapabilities()` 方法返回最乐观的系统功能视图，用于发送给定类型的媒体数据。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 `audio` 和 `video` 的两种类型。如果系统没有与 `kind` 参数值表示的类型相对应的功能，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，浏览器可以考虑暂缓操作，例如仅报告功能的公共子集。（这是一个指纹向量。）
- *setParameters*： `setParameters` 方法更新媒体轨 `track` 的编码与传输方式。
  当 `setParameters` 方法被调用，用户代理必须按以下步骤运行：
  1. 设 *parameters* 为方法的第一个参数。
  2. 设 *sender* 为调用此方法的 `RTCRtpSender` 对象。
  3. 设 *transceiver* 为与 *sender* 关联的 `RTCRtpTransceiver` 对象（sender即transceiver中的[Sender]槽）。

4. 如果 *transceiver* 的[Stopped]槽值为`true`，以一个新建的`InvalidStateError`拒绝此promise。
5. 如果 *sender* 内部的[LastReturnedParameters]槽为空，以一个新建的`InvalidStateError`拒绝此promise.
6. 按以下步骤验证 *parameter* 合法性：
    1. 设 *encodings* 为`parameter.encodings`。
    2. 设 *codec* 为`parameters.codecs`。
    3. 设 *N* 为*sender*内部的[SendEncodings]槽中存储的`RTCRtpEncodingParameters`对象数量。
    4. 如果以下任一条件被满足，则以一个新建的`InvalidModificationError`拒绝promise并返回：
        1. `encodings.length`与N不相等。
        2. *encodings* 被重新排序。
        3. *parameter* 中的任何参数都被标记为**只读参数**（例如RID），且某一参数的值与 *sender* 的[LastReturnedParameters]槽中对应参数的值不相同。注意，这同样适用于 *transactionId* 。
    5. 确认 *encodings* 中的每个`scaleResolutionDownBy`值都大于等于1.0。如果任一`scaleResolutionDownBy`不满足本需求，则以一个新建的`RangeError`拒绝promise并返回。
    6. 确认 *encodings* 中的每个`maxFramerate`值都大于等于0.0。如果任一`maxFramerate`不满足本需求，则则以一个新建的`RangeError`拒绝promise并返回。
    7. 对于0到编码数量范围中的每个值i，0到编解码器数量范围中的每个值j，检查`encodings[i].codecPayloadType`（如果已设置）是否与`codec[j].payloadType`相对应。如果没有对应关系，或者`codec[j] .mimeType`的MIME子类型部分等于`"red"`, `"cn"`, `"telephone-event"`, `"rtx"`或前向纠错编解码器（`"ulpfec"`或`"flexfec"`），则以一个新建的`InvalidAccessrror`拒绝promise。
7. 创建一个新的promise对象 *p* 。
8. 并行地用 *parameters* 配置媒体栈，以发送 *sender* 的[SenderTrack]中的数据。
    1. 如果用 *parameters* 成功配置了媒体栈，则将包含以下步骤的任务加入操作队列：
        1. 将 *sender* 内部的[LastReturnedParameters]槽置空。
        2. 将 *sender* 内部的[SendEncodings]槽设为`parameters.encodings`。
        3. 用`undefined`解析 *p* 。
    2. 如果配置过程出现了错误，则将包含以下步骤的任务加入操作队列：
        1. 如果是因为硬件资源不可用而发生错误，则以一个新建的`RTCError`拒绝 *p* ，并将其`errorDetail`设为"hardware-encoder-not-available"，然后终止步骤。
        2. 如果是因为硬件编码器不支持 *parameters* 而发生错误，则以一个新建的`RTCError`拒绝 *p* ，并将其`errorDetail`设为"hardware-encoder-error"，然后终止步骤。
        3. 对于其他的错误，以一个新建的`OperationError`拒绝 *p* 。
9. 返回 *p* 。 如果应用程序通过`codecPayloadType`选择编解码器，并且此编解码器从后续的邀请/应答协商中被删除了，那么在下一次调用`getParameters`方法时将清空`codecPayloadType`，并且实现将回退到其默认编解码器选择策略，直到选择了新的编解码器。
`setParameters`不会导致SDP重新协商，只能用于更改媒体栈在由邀请/应答协商的信封内发送或接收的内容。 `RTCRtpSendParameters`字典中的属性旨在不启用本特性，因此`cname`等属性是只读的，无法被更改。其他的，如比特率使用`maxBitrate`等限制进行控制，用户代理需要确保它不超过`maxBitrate`指定的最大比特率，同时确保它满足别处指定的比特率限制，例如SDP。

- *getParameters*：`getParameters()`方法返回`RCTRtpSender`对象当前持有的控制媒体轨`track`的编码与传输方式的参数。
  当`getParameters`被调用，`RTCRtpSendParameters`字典按以下流程构建：
  - `transactionId`被设为一个新的唯一标识符，用于将本次`getParameters`调用与将来可能发生的`setParameters`调用进行匹配。
  - `encodings`被设为内部[SendEncodings]槽的值。
  - `headerExtensions`序列根据已协商的发送头部扩展名进行填充。
  - `codec`序列基于已协商用于发送的编解码器以及用户代理当前能够发送的编解码器进行填充。在协商完成之前，`codec`序列为空。
  - `rtcp.cnmae`被设为与其关联的`RTCPeerConnection`对象的CNAME。如果已经达成了裁剪发送RTCP的协商，则`rtcp.reducedSize`为`true`，否则为`false`。
  - `degradationPreference`被设为传入`setParameters`的最新值，如果`setParameters`从未被调用，则为默认值"balanced"。

  返回的`RTCRtpSendeParameters`字典必须被保存在`RTCRtpSender`对象的[LastReturnedParameters]槽中。`getParameters`可以以下的方式，和`setParameters`配合使用来改变参数：

```
    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);
    }
}
```

  调用`setParamters`之后，后续的`getParameters`调用叫返回修改后的参数集合。

- *replaceTrack*：不经过协商，尝试用另一个已有的媒体轨（或一个空轨）替代当前`RTCRtpSender`中的媒体轨`track`。
  当`replaceTrack`方法被调用，用户代理必须按以下步骤运行：
  1. 设 *sender* 为调用此`replaceTrack`方法的`RTCRtpSender`对象。
  2. 设 *transceiver* 为与 *sender* 关联的`RTCRtpTransceiver`对象。
  3. 设 *connection* 为与 *sender* 关联的`RTCPeerConnection`对象。
  4. 设 *withTrack* 为此方法的参数。
  5. 如果`withTrack`非空且`withTrack.kind`与 *transceiver* 的收发器类型不同，则以一个新创建的`TypeError`拒绝promise并返回。
  6. 将包含以下步骤的任务加入 *connection* 的操作队列，并返回执行结果：
     1. 如果 *transceiver* 的[Stopped]的槽为`true`，则以一个新创建的`InvalidStateError`拒绝promise并返回。
     2. 创建一个新promise对象 *p* 。
     3. 如果 *transceiver* 的[CurrentDirection]槽值为`"sendrecv"`或`"sendonly"`，则设 *sending* 为`true`，否则为`false`。

4. 并行运行以下步骤：
    1. 若 *sending* 为`true`且 *withTrack* 为`null`，发送端停止发送。
    2. 若 *sending* 为`true`且 *withTrack* 不为`null`，确定发送端是否可以立即发送 *withTrack* 而不违反发送端已经协商的信封，如果不能，则以一个新创建的`InvalidModificationError`拒绝 promise，并终止后续步骤。
    3. 若 *sending* 为`true`且 *withTrack* 不为`null`，则发送方无缝切换到发送 *withTrack* ，而不是发送现有的媒体轨。
    4. 将包含以下步骤的任务加入队列：
        1. 如果 *transceiver* 的[Stopped]槽值为`true`，终止后续步骤。
        2. 设 *sender* 的`track`属性为 *withTrack* 。
        3. 以`undefined`解析 *p* 。
    5. 返回 *p* 。 **注意：**更改尺寸和/或帧速率也许不需要协商。需要协商的场景包含以下：
        1) 如RFC6236中描述的，将分辨率更改为协商好的imageattr范围之外的值。
        2) 将帧速率更改为导致编解码器超出阻塞速率的值。
        3) 视频轨与原始格式和预编码格式不同。
        4) 具有不同通道数的音轨。
        5) 同样编码的源（通常是硬件编码器）可能无法提供支持协商的编解码器; 同样，软件源可能不会实现支持编码源协商的编解码器。

- *setStreams*：设置与该发送端媒体轨相关联的`MediaStreams`。
  当`setStreams`方法被调用，用户代理必须按以下步骤运行：
    1. 设 *sender* 为调用此方法的`RTCRtpSender`对象。
    2. 设 *connection* 为此方法调用发生的`RTCPeerConnection`对象。
    3. 如果 *connection* 的[IsClosed]槽为`true`，则抛出一个`InvalidStateError`。
    4. 设 *streams* 为从方法参数中构建的`MediaStream`对象列表，如果没有参数，那么它为一个空列表。
    5. 对于 *streams* 中的每个 *stream* ，如果 *stream.id* 尚未存在于[AssociatedMediaStreamIds]，则将它加入。
    6. 更新 *connection* 的协商标记位。
- *getStats*：仅收集该发送端的状态信息，并异步报告结果。
  当`getStats()`方法被调用，用户代理必须按以下步骤运行：
    1. 设 *selector* 为调用此方法的`RTCRtpSender`对象。
    2. 设 *p* 为新创建的promise，并行地运行以下步骤：
        1. 根据状态选择算法收集选择器指示的状态数据。
        2. 将包含收集到状态信息的`RTCStatsReport`对象用来解析 *p* 。
    3. 返回 *p* 。

5.2.1 `RTCRtpParameters`字典

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

`RTCRtpParameters`字典成员

- sequence类型的`headerExtensions`，必需项：一个包含RTP头部拓展参数的序列。该参数为只读参数。

- RTCRtcpParameters类型的`rtcp`，必需项：RTCP使用的参数。该参数为只读参数。
- sequence类型的`codecs`，必需项：一个包含`RTCRtpSender`可选的媒体编解码器的序列，这些编解码器同样也是RTX，RED，FEC机制的条目。对于启用RTX重传的每个媒体编解码器，在编解码器数组`codec[]`中会有一个带有`mimeType`属性的条目，该条目指示了通过"audio/rtx"重传还是通过"video/rtx"重传，该条目还具有`sdpFmtpLine`属性（提供"apt"参数和"rtx-time"参数）。该参数是只读参数。

5.2.2 `RTCRtpSendParameters`字典

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

`RTCRtpSendParameters`字典成员：

- DOMString类型的`transcationId`，必需项：最新应用的参数的唯一标识符。确保只能基于先前的`getParameters`调用`setParameters`，并且没有干预更改。该参数为只读参数。
- sequence类型的`encodings`，必需项：一个包含RTP媒体编码参数的序列。
- RTCDegradationPreference类型的`degradationPreference`，缺省值为`"balanced"`：当带宽被限制，且`RtpSender`需要在降低分辨率和降低帧率之间做出选择，`degradationPreference`表示哪项是首选。
- RTCPriorityType类型的`priority`，缺省值为`"low"`：表示编码的优先级。它在RTCWEB-TRANSPORT第四节被定义。

5.2.3 `RTCRtpReceiveParameters`字典

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

`RTCRtpReceiveParameters`字典成员：

- sequence类型的`encodings`，必需项：一个包含传入的RTP媒体编码信息的序列。

> 风险等级2：支持`RTCRtpReceiveParameters`的`encodings`属性被标记为是有风险的特性，因为实现者对实现方式没有明确的承诺。

5.2.4 `RTCRtpCodingParameters`字典

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

`RTCRtpCodingParameters`字典成员：

- DOMString类型的`rid`：如果被设置了，该RTP编码会随着RID头部拓展一起被发送（JSEP 5.2.1）。RID不能被`setParameters`调用修改。它只能被设置，或者在发送端被`addTransceiver`调用修改。该参数为只读参数。

5.2.5 `RTCRtpDecodingParameters`字典

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

5.2.6 `RTCRtpEncodingParameters`字典

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

`RTCRtpEncodingParameters`字典成员：

- octet类型的`codecPayloadType`：被用于选择一个编解码器发送。必须从`RTCRtpParameters`的`codec`成员中引用有效内容类型。如果未被设置，实现将根据其默认策略选择编解码器。
- RTCDtxStatus类型的`dtx`：只有当发送端的类型`kind`为`"audio"`时才使用此成员。它表示是否使用不连续传输。将其设置为`disabled`会关闭不连续传输。将其设置为`enabled`会在协商时（通过编解码器指定的参数或通过协商CN编解码器）打开不连续传输; 如果没有协商（例如将`voiceActivityDetection`设为`false`），则无论`dtx`的值是什么，都会关闭不连续操作，即使检测到静音也会发送媒体数据。
- boolean类型的`active`，缺省值为`true`：表示正在活跃地发送此编码。 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。
- RTCPriorityType类型的`priority`，缺省值为`"low"`：表示当前编码的优先级。这在RTCWEB-TRANSPORT第四节中规定。
- unsigned long类型的`ptime`：如果该成员存在，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。如JSEP 5.10中所述，该值必须优先于远程描述中的任何`"ptime"`属性。注意，如RFC4566第6节中所述，用户代理必须仍然遵守所有`"maxptime"`属性施加的约束。
- unsigned long类型的`maxBitrate`：如果该成员存在，则表示可用于发送此编码的最大比特率。编码还可能受到除了这里指定的最大值之外的其他限制（例如每次传输/每次会话中的`maxFramerate`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，后者不计算IP/TCP/UDP等其他传输层协议所需的最大带宽。
- double类型的`maxFramerate`：如果该成员存在，则表示可用于发送此编码的最大帧率，即每秒的帧数。
- double类型的`scaleResolutionDownBy`：该成员仅在发送端的类型`kind`为`"video"`时才会存在。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，则视频不受影响。该值必须大于或等于1.0。默认情况下，发送端不会应用任何缩放（即`scaleResolutionDownBy`为1.0）。

5.2.7 `RTCDtxStatus`枚举

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

`RTCDtxStatus`枚举值描述：

- disabled：关闭不连续传输。
- enabled：如果协商发生，则启动连续传输。

5.2.8 `RTCDegradationPreference`枚举

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

`RTCDegradationPreference`枚举值描述：

- maintain-framerate：降低分辨率以保持帧率。
- maintain-resolution：降低帧率以保持分辨率。
- balanced：平衡地降低帧率和分辨率。

5.2.9 `RTCRtcpParameters`字典

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

`RTCRtcpParameters`字典成员：

- DOMString类型的`cname`：RTCP（如SDES消息）使用的规范名称（CNAME）。该参数为只读参数。
- boolean类型的`reducedSize`：若为`true`，表示配置后的RTCP裁剪尺寸（RFC5506），若为`false`，表示RFC3550中指定的复合RTCP。

5.2.10 `RTCRtpHeaderExtensionParameters`字典

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

`RTCRtpHeaderExtensionParameters`字典成员：

- DOMString类型的`uri`，必需项：如RFC5285中定义，表示RTP头部扩展的URI。该参数为只读参数。
- unsigned short类型的`id`，必需项：放置在RTP分组中标识头部扩展的值。该参数为只读参数。
- boolean类型的`encrypted`：表示头部扩展是否被加密。该参数为只读参数。

> 注意：`RTCRtpHeaderExtensionParameters`字典使应用程序能够确定是否配置了头部扩展，使得可以在`RTCRtpSender`或`RTCRtpReceiver`中使用。对于一个`RTCRtpTransceiver`收发器，应用程序不必解析SDP就可以确定头部扩展中的"direction"参数（在RFC5285中定义），如下所示：1) sendonly：头部扩展只包含在`transceiver.sender.getParameters().headerExtensions`。
> 2) recvonly：头部扩展只包含在`transceiver.receiver.getParameters().headerExtensions`。

> 3) sendrecv：头部扩展同时包含在`transceiver.sender.getParameters().headerExtensions`和`transceiver.receiver.getParameters().headerExtensions`。
> 4) inactive：头部扩展均不包含在`transceiver.sender.getParameters().headerExtensions`和`transceiver.receiver.getParameters().headerExtensions`。

5.2.11 `RTCRtpCodecParameters`字典

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

`RTCRtpCodecParameters`字典成员：

- octet类型的`payloadType`：被用于标识该编解码器的RTP有效内容类型。该参数为只读参数。
- DOMString类型的`mimeType`：编解码器的MIME媒体类型/子类型。IANA-RTP-2中罗列了有效的媒体类型/子类型。该参数为只读参数。
- unsigned long类型的`clockRate`：以赫兹表示的编解码器时钟速率。该参数为只读参数。
- unsigned short类型的`channels`：当该成员存在时，表示通道的数量（mono=1, stereo=2）。该参数为只读参数。
- DOMString类型的`sdpFmtpLine`：如果存在，如JSEP 5.8所定义，表示SDP中对应于编解码器的"a = fmtp"行中的"format specific parameters"字段。对于`RTCRtpSender`，这些参数来自远程描述，对于`RTCRtpReceiver`，它们来自本地描述。该参数为只读参数。

5.2.12 `RTCRtpCapabilities`字典

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

`RTCRtpCapabilities`字典成员：

- sequence类型的`codecs`，必需项：支持的媒体编解码器以及RTX，RED和FEC机制的条目。`sdpFmtpLine`不存在时，在编解码器数组`codecs []`中只有一个条目用于通过RTX进行重传。
- sequence类型的`headerExtensions`，必需项：支持的RTP头部扩展。

5.2.13 `RTCRtpCodecCapability`字典

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

`RTCRtpCodecCapability`字典成员： `RTCRtpCodecCapability`字典提供了关于编解码器能力的信息。只有在生成的SDP邀请中可以利用不同有效载荷类型的能力组合会被提供。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。
2. 两个不同时钟速率的CN编解码器。

- DOMString类型的`mimeType`，必需项：编解码器的MIME媒体类型/子类型。IANA-RTP-2中罗列了有效的媒体类型/子类型。
- unsigned long类型的`clockRate`，必需项：以赫兹表示的编解码器时钟速率。
- unsigned short类型的`channels`：如果存在，则表示通道最大数（mono=1, stereo=2）。
- DOMString类型的`sdpFmtpLine`：表示SDP中对应于编解码器的"a = fmtp"行中的"format specific parameters"字段。

5.2.14 `RTCRtpHeaderExtensionCapability`字典

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

`RTCRtpHeaderExtensionCapability`字典成员：

- DOMString类型的`uri`：如RFC5285中定义，表示RTP头部扩展的URI。

5.3 `RTCRtpReceiver`接口

`RTCRtpReceiver`接口允许应用程序监视流媒体轨`MediaStreamTrack`的接收情况。 给定一个表示类别的字符串参数 *kind* ，按以下步骤创建一个 **RTCRtpReceiver** ：

1. 设 *receiver* 为新的`RTCRtpReceiver`对象。
2. 设 *track* 为新的`MediaStreamTrack`对象（详见GETUSERMEDIA）。 *track* 的源是一个由 *receiver* 提供的 **远程源** 。注意 *track.id** 由用户代理生成，且与远程端的媒体轨ID没有任何映射关系。
3. 将 *track.kind* 初始化为 *kind* 。
4. 将 *track.label* 初始化为 *kind* 与字符串`"remote "`连接后的结果。
5. 将 *track.readyState* 初始化为`live`。
6. 将 *track.muted* 初始化为`true`。查阅MediaStreamTrack这一节，获得`muted`属性如何反应`MediaStremTrack`接受状态的相关信息。
7. 将 *receiver* 的[ReceiverTrack]槽初始化为 *track* 。
8. 将 *receiver* 的[ReceiverTransport]槽初始化为`null`。
9. 将 *receiver* 的[ReceiverRtcpTransport]槽初始化为`null`。
10. *receiver* 的[AssociatedRemoteMediaStreams]槽代表与`MediaStreamTrack`对象相关联的`MediaStream`对象的列表，将它初始化空列表。
11. 返回 *receiver* 。

```
[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：

- MediaStreamTrack类型的`track`，只读：`track`属性代表与`RTCRtpReceiver`对象 *receiver* 相关联的媒体轨。
  注意`track.stop()`是最后一步，但克隆后的媒体轨不受影响。`receiver.track.stop()`并不会隐式停止 *receiver* ，因此接收情况的报告还会继续发送。请求读值时，该属性必须返回[ReceiverTrack]槽的值。
- RTCDtlsTransport类型的`transport`，只读，可空：`transport`属性是以RTP分组的形式接收 *reciver.track* 媒体数据的传输通道。在构造`RTCDtlsTransport`对象之前，`transport`属性为`null`。 使用捆绑时，多个`RTCRtpReceiver`对象将共享同一个传输，并通过同一传输接收RTP和RTCP。
  请求读值时，属性必须返回[[ReceiverTransport]]槽的值。
- RTCDtlsTranport类型的`rtcpTransport`，只读，可空：`rtcpTransport`属性是发送和接收RTCP的传输通道。在构造`RTCDtlsTransport`对象之前，`rtcpTransport`属性为`null`。当使用RTCP多路复用（或捆绑强制执行RTCP多路复用）时，`rtcpTransport`将为`null`，并且RTP和RTCP的流量都将通过传输流动。
  请求读值，属性必须返回[ReceiverRtcpTransport]槽的值。

方法：

- *getCapabilities* ，静态：`getCapabilities()`方法返回的系统功能视图，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`"audio"`和`"videp"`的 *kind* 值。如果系统没有与 *kind* 值相对应的功能，则`getCapabilities`返回`null`。
  这些功能通常在设备上持续提供跨源信息，从而增加了应用程序的指纹表面。 在隐私敏感的上下文中，浏览器可以考虑暂缓操作，例如仅报告功能的公共子集。（这是一个指纹矢量。）

- *getParameters* ：`getParameters()`方法返回`RTCRtpReceiver`对象的当前参数，来指示`track`如何解码。
  当`getParameters`被调用，`RTCRtpReceiveParameters`字典会按以下步骤构造：

  1. 根据当前远程描述中存在的RID填充`encodings`。
  2. 根据接收端当前准备接收的头部扩展来填充`headerExtensions`序列。
  3. 根据接收端当前准备接收的编解码器来填充`codecs`序列。
     注意： 本地和远程描述都可能影响此编解码器列表。例如，如果提供了三个编解码器，接收端准备好接收每一个编解码器，并通过`getParameters`把它们返回。但是如果远程端点只对其中两个作了应答，则`getParameters`将不再返回缺失的编解码器，因为接收端不再处于准备接收它的状态了。
  4. 如果接收端正在准备接收裁剪后的RTCP分组，则将`rtcp.reducedSize`设为`true`，否则为`false`。`rtcp.cname`被遗漏。

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

- *getSynchronizationSources* ：返回此`RTCRtpReceiver`在过去10秒内收到的每个唯一SSRC标识符的`RTCRtpSynchronizationSources`对象。
- *getStats* ：仅收集此接收端的状态信息并异步地报告结果。
  当`getStats()`方法被调用，用户代理必须按以下步骤运行：
  1. *selector* 即调用此方法的`RTCRtpReceiver`对象。
  2. 设 *p* 为新的promise对象，并行地执行以下步骤：
     1. 根据状态信息选择算法收集 *selector* 指示地状态信息。
     2. 用包含收集到的状态信息的`RTCStatsReport`对象解析 *p* 。
  3. 返回 *p* 。

`RTCRtpContributingSource`和`RTCRtpSynchronizationSource`字典分别包含给定贡献源（CSRC）或同步源（SSRC）的相关信息，包括数据源最近一次贡献数据包的时间。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，用户代理必须将任务加入队列，任务会根据数据包的内容更新`RTCRtpContributingSource`和`RTCRtpSynchronizationSource`字典的相关信息。与SSRC标识符对应的`RTCRtpSynchronizationSource`字典相关的信息每次都会被更新，并且如果RTP分组包含CSRC标识符，还会更新与CSRC标识符对应的`RTCRtpContributingSource`字典相关的信息。

> 注意：如一致性一节中所述，只要最终结果是等效的，可以用任何方式实现该算法。因此，实现不需要为每个数据包都创建处理任务并加入操作队列。只要最终结果是在单个事件循环中执行得到的，特定`RTCRtpReceiver`返回的所有`RTCRtpSynchronizationSource`和`RTCRtpContributingSource`字典都包含来自RTP流中的单点信息。

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

`RTCRtpContributingSource`成员：

- DOMHighResTimeStamp类型的`timestamp`，必需项：DOMHighResTimeStamp类型HIGHRES-TIME的时间戳表示从该源发出的RTP分组中到达的媒体最近的播出时间。时间戳被定义为播放时调用`performance.timeOrigin + performance.now()`得到的值。
- unsigned long类型的`source`，必需项：贡献源或同步源的CSRC/SSRC标识符。
- double类型的`audioLevle`：仅适用于音频接收端。 这是一个0...1（线性）之间的值，其中1.0表示0dBov，0表示静音，0.5表示声压等级从0dBov变化到约6dBSPL。
  对于CSRC，如果头部扩展存在，则该值必须转化为RFC6465中定义的等级值，否则该成员不存在。
  对于SSRC，如果头部扩展存在，则该值必须转化为RFC6464中定义的等级值，否则用户代理必须从音频数据中计算出该值（对于音频接收端，该值一定存在）。
  两个RFC都将级别定义为从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音频电平。因此，0代表系统可能编码的最大声信号，127代表静音。
  要将这些值转换为0...1的线性范围，值127将被转换为0，并使用以下公式转换所有其他值：`10^(-rfc_level/20)`。

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

`RTCRtpSynchronizationSource`字典成员：

- boolean类型的`voiceActivityFlag`：仅适用于音频接收端。表示该源播放的最后一个RTP数据包是否包含语音活动（true or false）。 如[RFC6464]第4节所述，如果头部扩展不存在，或者对端通过将"vad"扩展属性设置为"off"来表示它没有使用V位，则`voiceActivityFlag`将不存在。

5.4 `RTCRtpTransceiver`接口

`RTCRtpTransceiver`接口表示共享公共`mid`的`RTCRtpSender`和`RTCRtpReceiver`的组合。如JSEP 3.4.1中定义的，如果`RTCRtpTransceiver`的`mid`属性非空，则称其与媒体描述**相关联**，否则不相关。概念上说，一个被关联的收发器代表上一次会话描述中应用的收发器。
`RTCRtpReceiver`的 **收发器类型** 在与之关联的`RTCRtpReceiver`对象持有的`MediaStream`对象中定义。
利用一个现有的`RTCRtpReceiver`对象 *receiver* ，`RTCRtpSender`对象 *sender* ，`RTCRtpTransceiverDirection`值 *direction* ，按以下步骤创建一个`RTCRtpTransceiver`对象：

1. 设 *transceiver* 为一个新`RTCRtpTransceiver`对象。
2. 将 *transceiver* 的[Sender]槽初始化为 *sender* 。
3. 将 *transceiver* 的[Receiver]槽初始化为 *receiver* 。
4. 将 *transceiver* 的[Stopped]槽初始化为`false`。
5. 将 *transceiver* 的[Direction]槽初始化为 *direction* 。
6. 将 *transceiver* 的[Receptive]槽初始化为`fasle`。
7. 将 *transceiver* 的[CurrentDirection]槽初始化为`null`。
8. 将 *transceiver* 的[FiredDirection]槽初始化为`null`。
9. 返回 *transceiver* 。

> 注意：创建一个收发器并不会创建底层的`RTCDtlsTransport`和`RTCIceTransport`对象。这一过程只会作为"设置一个`RTCSessionDescription`"的子步骤发生。

```
[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：

- DOMString类型的`mid`，只读，可空：如JSEP 5.2.1&5.3.1所述，`mid`属性是协商好的存在于本地和远程描述中的`mid`值。在协商完成之前，`mid`值为空。在回滚发生之后，该值可能从非空变为空。

- RTCRtpSender类型的`sender`，只读：`sender`属性将发送的RTP媒体中与mid=`mid`对应的`RTCRtpSender`对象公开。当请求读值时，该属性必须返回[Sender]槽的值。
- RTCRtpReceiver类型的`receiver`，只读：`receiever`属性表示与mid=`mid`RTP媒体相对应的`RTCRtpReceiver`对象。当请求读值时，该属性必须返回[Receiver]槽的值。
- boolean类型的`stopped`，只读：`stopped`属性表示此收发器的发送端将不再发送，接收端将不再接收。如果已经调用了`stop`或者设置本地或远程描述导致`RTCRtpTransceiver`被停止，则它将变为`true`。当请求读值时，该属性必须返回[Stopped]槽的值。
- RTCRtpTransceiverDirection类型的`direction`：如JSEP 4.2.4中定义的，`direction`属性表示`createOffer`和`createAnswer`调用时收发器的首选方向。方向的更新不会立即生效，相反的，将来调用`createOffer`和`createAnswer`时会将相应的媒体描述标记为`sendrecv`，`sendonly`，`recvonly`或`inactive`，这定义在JSEP 5.2.2&5.3.2。
  当请求读值时，该属性必须返回[Direction]槽的值。
  当请求写值时，必须按以下步骤运行：
  1. 设 *transceiver* 为调用此方法的`RTCRtpTransceiver`对象。
  2. 设 *connection* 为与 *transceiver* 关联的`RTCPeerConnection`对象。
  3. 若 *connection* 的[IsClosed]槽为`true`，抛出一个`InvalidStateError`错误。
  4. 若 *transceiver* 的[Stopped]槽位`true`，抛出一个`InvalidStateError`错误。
  5. 设 *newDirection* 为写函数的参数。
  6. 若 *newDirection* 与 *transceiver* 的[Direction]槽相同，则终止步骤。
  7. 将 *transceiver* 的[Direction]槽值设为 *newDirection* 。
  8. 更新 *connection* 的协商标记位。
- RTCRtpTransceiverDirection类型的`currentDirection`，只读，可空：如JSEP 4.2.5定义，`currentDirection`代表当前协商好的收发器的方向。`currentDirection`的值与`RTCRtpEncodingParameters.active`的值无关，因为两者无因果关系。如果此收发器从未存在于邀请/应答的数据交换过程中，或者收发器已停止，则该值为空。请求读值时，该属性必须返回[CurrentDirection]槽的值。

方法：

- *stop* ：不可逆地停止`RTCRtpTransceiver`。此收发器的发送端将不再发送，接收端将不再接收。调用`stop()`会更新与`RTCRtpTransceiver`关联的`RTCPeerConnection`的需要协商标记位。
  如JSEP 4.2.1所述，停止收发器将导致将来`createOffer`或`createAnswer`的调用在相应收发器的媒体描述中生成零端口。
  注意： 如果该方法在应用远程描述与创建应答之间被调用，且该收发器已与打上了"邀请"标签的媒体描述相关联了，则将导致捆绑组中所有其他收发器都停止。为了避免这样的情况发生，我们可以在信令状态为`"stable"`时，在执行后续的邀请/应答数据交换时停止收发器。
  当`stop`方法被调用，用户代理必须按以下步骤运行：
  1. 设 *transceiver* 为调用此方法的`RTCRtpTransceiver`对象。
  2. 设 *connection* 为与 *transceiver* 关联的`RTCPeerConnection`对象。
  3. 如果 *connection* 的[IsClosed]槽为`true`，则抛出一个`InvalidStateError`。
  4. 如果 *transceiver* 的[Stopped]槽为`true`，则终止以下步骤。
  5. 根据下述步骤停止 *transceiver* 。
  6. 更新 *connection* 的协商标记位。 **停止RTCRtpTransceiver**算法如下：
  7. 设 *sender* 为 *transceiver* 的[Sender]槽。
  8. 设 *receiver* 为 *transceiver* 的[Receiver]槽。
  9. 停止 *sender* 发送媒体数据。
  10. 根据RFC3550， *sender* 向每个RTP流发送一个RTCP BYE信号。

11. 停止 *receiver* 接收媒体数据。
12. 执行结束 *receiver* 的[ReceiverTrack]槽的步骤。
13. 将 *transceiver* 的[Stopped]槽设为 `true`。
14. 将 *transceiver* 的[Receptive]槽设为 `false`。
15. 将 *transceiver* 的[CurrentDirection]槽设为 `null`。

- *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 `codecs` 参数中指定的顺序使用指定的编解码器，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过程。它还允许应用程序使远程对端偏好列表中首先出现的编解码器。
  对于所有包含此 `RTCRtpTransceiver` 的 `createOffer` 和 `createAnswer` 的调用，编解码器首选项仍然有效，直到再次调用此方法。将 `codecs` 设置为空序列将使编解码器首选项重置为所有默认值。
  传递给 `setCodecPreferences` 的 `codecs` 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，无法修改 `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` 错误。
  注意： 根据SDP的建议，`createAnswer` 的调用应只包含编解码器首选项和邀请中出现的编解码器的公共子集。例如，如果编解码器首选项为"C, B, A"，但邀请中只提供了"A, B"，则应答中只能包含"B, A"编解码器。但是JSEP 5.3.1允许添加不在邀请中出现的编解码器，因此具体实现可以表现得不一样。

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，但仍可以控制发送流的数量和这些流的特征。利用 `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。利用 `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，当利用能够发送[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好的编码。

> 注意：在选择性转发单元（SFU）在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，但是因为被重新排序，来自多个RTP流的分组可能在接收端混合。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来说是可选的。

5.4.1.1 编码参数样例

使用编码参数实现联播场景的样例：

```
EXAMPLE 4
// Example of 3-layer spatial simulcast with all but the lowest resolution layer disabled
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}
];
```

5.4.2 "暂停"功能

`direction`和`replaceTrack`两个属性使得开发者可以实现"暂停"场景。 将音乐发送给对端并停止呈现接收的音频：

```
EXAMPLE 5
async function playMusicOnHold() {
  try {
    // Assume we have an audio transceiver and a music track named musicTrack
    await audio.sender.replaceTrack(musicTrack);
    // Mute received audio
    audio.receiver.track.enabled = false;
    // Set the direction to send-only (requires negotiation)
    audio.direction = 'sendonly';
  } catch (err) {
    console.error(err);
  }
}
```

响应远程对端的"sendonly"邀请：

```
EXAMPLE 6
async function handleSendonlyOffer() {
  try {
    // Apply the sendonly offer first,
    // to ensure the receiver is ready for ICE candidates.
    await pc.setRemoteDescription(sendonlyOffer);
    // Stop sending audio
    await audio.sender.replaceTrack(null);
    // Align our direction to avoid further negotiation
    audio.direction = 'recvonly';
    // Call createAnswer and send a recvonly answer
    await doAnswer();
  } catch (err) {
    // handle signaling error
  }
}
```

停止发送音乐并发送从麦克风捕获的音频，以及呈现接收到的音频：

```
EXAMPLE 7
async function stopOnHoldMusic() {
  // Assume we have an audio transceiver and a microphone track named micTrack
  await audio.sender.replaceTrack(micTrack);
  // Unmute received audio
  audio.receiver.track.enabled = true;
  // Set the direction to sendrecv (requires negotiation)
  audio.direction = 'sendrecv';
}
```

响应被远程对端的取消暂停操作：

```
EXAMPLE 8
async function onOffHold() {
  try {
    // Apply the sendrecv offer first, to ensure receiver is ready for ICE candidates.
    await pc.setRemoteDescription(sendrecvOffer);
    // Start sending audio
    await audio.sender.replaceTrack(micTrack);
    // Set the direction sendrecv (just in time for the answer)
    audio.direction = 'sendrecv';
    // Call createAnswer and send a sendrecv answer
    await doAnswer();
  } catch (err) {
    // handle signaling error
  }
}
```

5.5 `RTCDtlsTransport`接口

`RTCDtlsTransport`接口允许应用程序访问有关`RTCRtpSender`和`RTCRtpReceiver`对象发送和接收RTP和RTCP数据包的数据报传输层安全性（DTLS）传输的信息，以及数据通道发送和接收的其他数据，如SCTP数据包。特别的，DTLS为底层传输增加了安全性，`RTCDtlsTransport`接口允许访问有关底层传输和添加的安全性信息。`RTCDtlsTransport`对象调用`setLocalDescription()`和`setRemoteDescription()`构造。每个`RTCDtlsTransport`对象表示特定`RTCRtpTransceiver`的RTP或RTCP组件的DTLS传输层，或者一组`RTCRtpTransceivers`（如果这样的组已通过捆绑BUNDLE协商）。

> 注意：现有`RTCRtpTransceiver`的新DTLS关联将由现有`RTCDtlsTransport`对象表示，其状态将相应更新，而不是由新对象表示。

`RTCDtlsTransport`对象有一个被初始化为`new`的内部槽[DtlsTransportState]。 当底层DTLS传输需要更新相应`RTCDtlsTransport`对象的状态时，用户代理必须将包含以下步骤的任务加入队列：

1. 设 *transport* 为接收状态更新的`RTCDtlsTransport`对象。
2. 设 *newState* 为新状态。
3. 将 *transport* 的[DtlsTransportState]槽设为 *newState* 。
4. 在 *transport* 上触发名为`statechange`的事件。

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

属性：

- RTCIceTransport类型的`iceTransport`，只读：`iceTransport`属性是用于发送和接收数据包的底层传输。多个活跃的`RTCDtlsTransport`对象之间可能不共享底层传输。
- RTCDtlsTransportState类型的`state`，只读：请求读值时`state`属性必须返回[DtlsTransport]槽的值。
- EventHandler类型的`onstatechange`：该事件处理器的事件类型为`statechange`。
- EventHandler类型的`onerror`：该事件处理器的事件类型为`error`。

方法：

- *getRemoteCertificates* ：返回远程端使用的证书链，每个证书以二进制可辨别编码规则（DER）X690编码。`getRemoteCertificates()`将在选择远程证书之前返回一个空列表，该列表将在`RTCDtlsTransportState`转换为`"connected"`时完成。

RTCDtlsTransportState枚举

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

枚举值描述：

- new：DTLS还未启动协商。
- connecting：DTLS正处于协商一个安全连接并验证远程指纹的过程中。
- connected：DTLS已完成安全连接的协商和远程指纹的验证过程。
- closed：由于收到close_notify告警或调用`close()`，传输已被故意关闭。
- failed：由于错误（例如收到错误警报或无法验证远程指纹），传输失败。

5.5.1 `RTCDtlsFingerprint`字典

`RTCDtlsFingerprint`字典包含哈希算法及RFC4572中的证书指纹。

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

`RTCDtlsFingerprint`字典成员：

- DOMString类型的`algorithm`："哈希函数文本名称"注册表IANA-HASH-FUNCTION中定义的哈希函数算法之一。
- DOMString类型的`value`：使用RFC4572第5节中"指纹"语法表示的小写十六进制字符串中的证书指纹值。

5.6 `RTCIceTransport`接口

`RTCIceTransport`接口允许应用程序访问有关发送和接收数据包的ICE传输的信息。特别地，ICE管理涉及应用可能想要访问的状态的点对点连接。`RTCIceTransport`对象通过调用`setLocalDescription()`和`setRemoteDescription()`被构造。底层ICE状态由ICE代理管理; 因此如下所述，当ICE代理向用户代理发起命令时，`RTCIceTransport`的状态改变。每个`RTCIceTransport`对象表示特定`RTCRtpTransceiver`的RTP或RTCP组件的ICE传输层，或者一组`RTCRtpTransceivers`（如果这样的组已通过BUNDLE协商）。

> 注意：现有`RTCRtpTransceiver`的ICE重启将由现有的`RTCIceTransport`对象表示，其状态将相应更新，而不是由新对象表示。

当ICE代理指示它开始为`RTCIceTransport`收集一代候选地址时，用户代理必须将包含以下步骤的任务加入操作队列：

1. 设 *connection* 为与ICE代理相关联的`RTCPeerConnection`对象。
2. 如果 *connection* 的[IsClosed]槽为`true`，则终止后续步骤。
3. 设 *transport* 为启动收集候选地址时的`RTCIceTransport`对象。
4. 将 *transport* 的[IceGatherState]槽设为`gathering`。
5. 在 *transport* 之上触发名为`gatheringstatechange`的时间。
6. 更新 *connection* 的ICE收集状态。

当ICE代理指示已为`RTCIceTransport`完成一代候选地址的收集工作时，用户代理必须将包含以下步骤的任务加入操作队列：

1. 设 *connection* 为与ICE代理关联的`RTCPeerConnection`对象。
2. 如果 *connection* 的[IsClosed]槽为`true`，则终止后续步骤。
3. 设 *transport* 为结束候选地址收集工作的`RTCIceTransport`对象。
4. 创建一个新`RTCIceCandidate`对象，命名为 *newCandidate* ，其`sdpMid`和`sdpMLineIndex`值设为与之关联的`RTCIceTransport`的对应值，`usernameFragment`设为收集完一代候选地址的用户名片段，`candidate`设为一个空字符串，其他所有可空的成员设为`null`。
5. 利用`candidate`属性设为 *newCandidate* 的`RTCPeerConnectionIceEvent`接口触发名为`icecandidate`的事件。
6. 如果另一代候选地址正在被收集，则终止步骤。 **注意：** 如果在ICE代理仍在收集上一代候选地址时启动ICE重启，则可能会发生这种情况。
7. 将 *transport* 的[IceGatherState]槽设为`complete`。
8. 在 *transport* 上触发名为`gatheringstatechange`的事件。
9. 更新 *connection* 的ICE收集状态。

当ICE代理指示新的ICE候选地址可用于`RTCIceTransport`时，从ICE候选池中取一个或重新开始收集一个候选地址，用户代理将包含以下步骤的任务加入操作列队：

1. 设 *connection* 为与ICE代理关联的`RTCPeerConnection`对象。
2. 如果 *connection* 的[IsClosed]槽为`true`，则终止后续步骤。
3. 设 *transport* 为被提供可用候选地址的`RTCIceTransport`。
4. 如果 *connection.[PendingLocalDescription]* 非空，且代表收集 *candidate* 的ICE代，则将 *candidate* 添加入 *connection.[PendingLocalDescription].sdp* 。

5. 若 *connection.[CurrentLocalDescription]* 非空，且代表收集 *candidate* 的ICE代，则将 *candidate* 添加入 *connection.[CurrentLocalDescription].sdp* 。
6. 创建一个代表候选地址的 `RTCIceCandidate` 对象，命名为 *newCandidate* 。
7. 将 *newCandidate* 加入 *transport* 的本地候选地址集合。
8. 利用 `candidate` 属性设为 *newCandidate* 的 `RTCPeerConnectionIceEvent` 接口触发名为 `icecandidate` 的事件。

当ICE代理指示 `RTCIceTransport` 的 `RTCIceTransportState` 已被更改时，用户代理将包含以下步骤的任务加入操作列队：

1. 设 *connection* 为与ICE代理关联的 `RTCPeerConnection` 对象。
2. 如果 *connection* 的[IsClosed]槽为 `true` ，则终止后续步骤。
3. 设 *transport* 为状态改变的 `RTCIceTransport` 对象。
4. 设 *newState* 为表示改变后状态的 `RTCIceTransportState` 对象。
5. 将 *transport* 的[IceTransportState]槽设为 *newState* 。
6. 在 *transport* 上触发名为 `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代理指示 `RTCIceTransport` 的所选候选对已更改时，用户代理将包含以下步骤的任务加入操作列队：

1. 设 *connection* 为与ICE代理关联的 `RTCPeerConnection` 对象。
2. 如果 *connection* 的[IsClosed]槽为 `true` ，则终止后续步骤。
3. 设 *transport* 为所选候选对已更改的 `RTCIceTransport` 对象。
4. 若有候选对被选中，则设 *newCandidate* 为代表所选候选对的新创建 `RTCIceCandidatePair` 对象，否贼 *newCandidate* 为空。
5. 将 *transport* 的[SelectedCandidatePair]设为 *newCandidate* 。
6. 在 *transport* 上触发名为 `selectedcandidatepairchange` 的事件。

一个 `RTCIceTransport` 对象持有以下槽：

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

```
[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：

- DOMString类型的`role`，只读：`role`属性必须返回传输的ICE角色。
- RTCIceComponent类型的`component`，只读：`component`属性必须返回传输的ICE组件。当启动RTCP多路复用，单个传输RTP/RTCP的`RTCIceTransport`，其`component`会被设为"RTP"。
- RTCIceTransportState类型的`state`，只读：请求读值时，`state`属性必须返回[IceTransportState]的值。
- RTCIceGathererState类型的`gatheringState`，只读：请求读值时，`gatheringState`属性必须返回[IceGathererState]的值。
- EventHandler类型的`onstatechange`：该`statechange`事件类型的事件处理器，一旦`RTCIceTransport`状态改变就会被触发。
- EventHandler类型的`ongatheringstatechange`：该`gatheringstatechange`事件类型的事件处理器，一旦`RTCIceTransport`的收集状态改变就会被触发。
- EventHandler类型的`onselectedcandidatepairchange`：该`gatheringstatechange`事件类型的事件处理器，一旦`RTCIceTransport`选中的候选对改变就会被触发。

方法：

- *getLocalCandidates* ：返回一个序列，描述为此`RTCIceTransport`收集并在`onicecandidate`中发送的本地ICE候选地址。
- *getRemoteCandidates* ：返回发送数据包的选定候选对。此方法必须返回[SelectedCandidatePair]槽的值。
- *getLocalParameters* ：返回此`RTCIceTransport`通过`setLocalDescription`方法接收的本地ICE参数，如果尚未接收，则返回`null`。
- *getRemoteParameters* ：返回此`RTCIceTransport`通过`setRemoteDescription`方法接收的远程ICE参数，如果尚未接收，则返回`null`。

5.6.1 `RTCIceParameters`字典

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

`RTCIceParameters`字典成员：

- DOMString类型的`usernameFragment`：ICE 7.1.2.3中定义的ICE用户名片段。
- DOMString类型的`password`：ICE 7.1.2.3中定义的ICE密码。

5.6.2 `RTCIceCandidatePair`字典

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

`RTCIceParameters`字典成员：

- RTCIceCandidate类型的`local`：本地ICE候选地址。
- RTCIceCandidate类型的`remote`：远程ICE候选地址。

5.6.3 `RTCIceGathererState`枚举

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

`RTCIceGathererState`枚举值描述

- new：`RTCIceTransport`刚被创建，尚未启动候选地址收集。
- gathering：`RTCIceTransport`正在候选地址收集的过程中。
- complete：`RTCIceTransport`已完成收集，并已发送此传输的候选地址终止指示。在ICE重启之前，它不会再次收集候选地址。

5.6.4 `RTCIceTransportState`枚举

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

`RTCIceTransportState`枚举值描述：

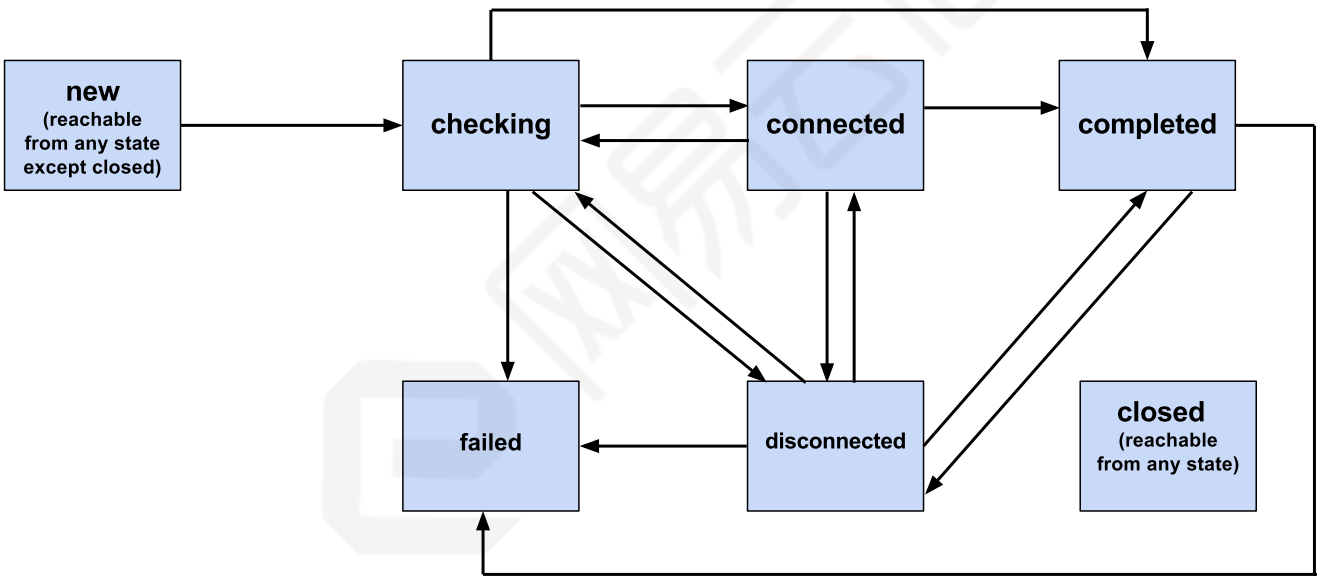
- new：`RTCIceTransport`正在收集候选地址和 /或正在等待远程候选地址被提供，且还未启动检查。
- checking：`RTCIceTransport`已经收到至少一个远程候选地址，正在检查候选对，且尚未找到可用连接或许可检查RFC7675在先前所有成功的候选对都失败了。除了检查，它还可能仍在收集。
- connected：`RTCIceTransport`找到了一个可用的连接，但仍在检查其他候选对，以查看是否有更好的连接。它可能仍然在收集和/或等待其他远程候选人。如果许可检查[RFC7675]在正在使用的连接上失败，并且没有其他成功的候选对可用，则状态转换为`"checking"`（还有待检查的候选对）或`"disconnected"`（没有候选对要检查，但对端仍在收集和/或等待其他远程候选地址）。
- completed：`RTCIceTransport`已完成收集，收到一条表明没有更多远程候选地址的指示，已完成所有候选对的检查并找到一个连接。如果许可检查[RFC7675]随后在所有成功的候选对上都失败，则状态转换为`"failed"`。
- disconnected：ICE代理已确定此`RTCIceTransport`当前已丢失连接。这是一种瞬时状态，可能会在片状网络上间歇性地触发（并在没有其他动作的情况下自行解决）。确定此状态的方式取决于具体实现。以下是例子：
  - 丢失了正在使用的连接的网络接口。
  - 重复无法收到STUN请求的响应。 另外的，`RTCIceTransport`已完成所有现有候选对的检查但未找到连接（或者许可检查[RFC7675]之前成功，但现已失败），但它仍在收集和/或等待其他远程候选地址。
- failed：`RTCIceTransport`已完成收集，收到一个表明没有更多的远程候选地址的指示，已完成检查所有候选对，并且所有候选对的连接检查都失败或不被许可。这是一个最终状态。
- closed：`RTCIceTransport`已关闭且不再响应STUN请求。

ICE重启导致候选地址收集和连接检查重新启动，如果在`"completed"`状态下开始则会转移到`"connected"`状态。如果在瞬时`"disconnected"`状态下启动，则会导致状态转移到`"checking"`，从而很快忘记先前已丢失的连接。

`"failed"`和`"completed"`状态要求一个没有其他远程候选地址的指示。可以用一个候选值调用`addIceCandidate`方法来指示，该候选值的`candidate`属性设为空字符串或`canTrickleIceCandidates`属性设为`false`。

以下是一些状态转移的例子：

- （`RTCIceTransport`作为`setLocalDescription`和`setRemoteDescription`的调用结果被创建）：`new`。
- （`new`，远程候选地址被接收）：`checking`。
- （`checking`，找到可用连接）：`connected`。
- （`checking`，检查失败，但仍在收集过程中）：`disconnected`。
- （`checking`，放弃）：`failed`。
- （`disconnected`，新的本地候选地址）：`checking`。
- （`connected`，结束所有检查）：`completed`。
- （`completed`，失去连接）：`disconnected`。
- （`disconnected`或`failed`，ICE开始重启）：`checking`。
- （`completed`，ICE开始重启）：`connected`。
- `RTCPeerConnection.close(): closed`



5.6.5 `RTCIceRole`枚举

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

`RTCIceRole`枚举值描述：

- controlling：[ICE]第三节定义的控制代理。
- controlled：[ICE]第三节定义的被控制代理。

5.6.6 `RTCIceComponent`枚举

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

`RTCIceComponent` 枚举值描述：

- rtp：如[ICE 4.1.1.1]中定义，ICE传输用于RTP（或RTCP复用）。与RTP（例如数据信道）复用的协议共享其组件ID。 这表示在 `candidate-attribute` 中编码的 `component-id` 值 `1`。
- rtcp：如[ICE 4.1.1.1]中定义，ICE传输用于RTCP。 这表示在 `candidate-attribute` 中编码的 `component-id` 值 `2`。

5.7 `RTCTrackEvent`

使用 `RTCTrackEvent` 接口的 `track` 事件。

```
[Constructor(DOMString type, RTCTrackEventInit eventInitDict),
  Exposed=Window]
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

属性：

- RTCRtpReceiver类型的 `receiver`，只读： `receiver` 属性代表与事件关联的 `RTCRtpReceiver` 对象。
- MediaStreamTrack类型的 `track`，只读： `track` 属性代表与以 `receiver` 标识的 `RTCRtpReceiver` 对象关联的 `MediaStreamTrack` 对象。
- FrozenArray类型的 `streams`，只读： `streams` 属性返回一个 `MediaStream` 对象数组，代表此事件的 `track` 所属的MediaStream。
- RTCRtpTransceiver类型的 `transceiver`，只读： `transceiver` 属性代表与事件关联的 `RTCRtpTransceiver`。

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

`RTCTrackEventInit` 字典成员：

- RTCRtpReceiver类型的 `receiver`，必需项： `receiver` 属性代表与事件关联的 `RTCRtpReceiver` 对象。

- MediaStreamTrack类型的`track`，必需项：：`track`属性代表与以`receiver`标识的`RTCRtpReceiver`对象关联的`MediaStreamTrack`对象。
- sequence类型的`streams`，缺省值为`[]`：`streams`属性返回一个`MediaStream`对象数组，代表此事件的`track`所属的MediaStream。
- RTCRtpTransceiver类型的`transceiver`，必需项：`transceiver`属性代表与事件关联的`RTCRtpTransceiver`。

6. 点对点数据API

点对点数据API允许Web应用程序以点对点的方式发送和接收通用的应用程序数据。用于发送和接收数据的API模拟WebSocketsWEBSOCKETS-API的行为。

6.1 RTCPeerConnection接口扩展

点对点数据API扩展了以下描述的`RTCPeerConnection`接口。

```
partial interface RTCPeerConnection {
  readonly attribute RTCSctpTransport? sctp;
  RTCDataChannel createDataChannel(USVString label,
  optional RTCDataChannelInit dataChannelDict);
  attribute EventHandler ondatachannel;
};
```

属性：

- RTCSctpTransport类型的`sctp`，只读，可空：SCTP数据的发送和接收的SCTP传输通道。如果尚未协商SCTP，则该值为`null`。该属性必须返回存储在[SctpTransport]内部槽中的`RTCSctpTransport`对象。
- EventHandler类型的`ondatachannel`：本事件处理器的事件类型的是`datachannel`。

方法：

- *createDataChannel* ：根据给定标签创建一个新`RTCDataChannel`对象。`RTCDataChannelInit`字典可用于配置底层通道的属性，例如数据可靠性。
  当`createDataChannel`被调用，用户代理必须按以下步骤运行：
  1. 设 *connection* 为调用此方法的`RTCPeerConnection`对象。
  2. 若 *connection* 的[IsClosed]槽为`true`，则抛出一个`InvalidStateError`错误。
  3. 创建一个`RTCDataChannel`， *channel* 。
  4. 将 *channel* 的[DataChannelLabel]初始化为第一个参数的值。
  5. 如果[DataChannelLabel]长度大于65535字节，抛出一个`TypeError`。
  6. 设 *options* 为第二个参数。
  7. 若 *options* 的`maxPacketLifeTime`成员存在的话，将 *channel* 的[MaxPacketLifeTime]槽初始化为它，否则为`null`。
  8. 若 *options* 的`maxRetransmits`成员存在的话，将 *channel* 的[MaxRetransmits]槽初始化为它，否则为`null`。
  9. 将 *channel* 的[Ordered]槽初始化为 *options* 的`ordered`成员。
  10. 将 *channel* 的[DataChannelProtocol]槽初始化为 *options* 的`protocol`成员。
  11. 如果[DataChannelProtocol]长度大于65535字节，抛出一个`TypeError`。
  12. 将 *channel* 的[Negotiated]槽初始化为 *options* 的`negotiated`成员。

13. 若 *options* 的`id`成员存在且其[Negotiated]槽值为`true`的话，将 *channel* 的[DataChannelId]槽初始化为它，否则为`null`。 **注意：** 这意味着如果数据通道在频内协商完成，则忽略`id`成员; 这是故意所为。如RTCWEB-DATA-PROTOCOL中所述，频内协商的数据通道应根据DTLS角色选择ID。
14. 若[Negotiated]槽为`true`且[DataChannelId]为`null`，抛出一个`TypeError`。
15. 将 *channel* 的[DataChannelPriority]槽初始化为 *options* 的`priority`成员。
16. 若[MaxPacketLifeTime]和[MaxRetransmits]属性都被设置了（非空），则抛出`TypeError`。
17. 如果[MaxPacketLifeTime]或[MaxRetransmits]设置已被设为指示不可靠模式，且该值超过用户代理支持的最大值，则必须将值设置为用户代理支持的最大值。
18. 如果[DataChannelId]等于65535（大于最大允许的ID 65534但仍在`unsigned short`精度范围内），则抛出`TypeError`。
19. 如果[DataChannelId]槽为`null`（由于没有ID传递到`createDataChannel`，或者[Negotiated]为`false`），并且已经协商了SCTP传输的DTLS角色，则根据RTCWEB-DATA-PROTOCOL，以用户代理生成的值初始化[DataChannelId]，并跳到下一步。如果无法生成可用ID，或者现有`RTCDataChannel`正在使用[DataChannelId]槽的值，则抛出`OperationError`异常。 **注意：** 若此步骤完成后[DataChannelId]槽为`null`，则它会在设置`RTCSessionDescription`的过程中，确定DTLS角色后被填充。
20. 设 *transport* 为 *connection* 的[SctpTransport]槽。
    若[DataChannel]槽非空， *transport* 处于`connected`状态，且[DataChannelId]大于等于 *transport* 的[MaxChannels]，则抛出一个`OperationError`。
21. 若 *channel* 为 *connection* 中创建的第一个`RTCDataChannel`对象，则更新 *connection* 的协商标记位。
22. 返回 *channel* 然后并行地执行下面步骤。
23. 创建与 *channel* 关联的底层数据传输，并根据 *channel* 中的相关属性对它进行配置。

6.1.1 `RTCSctpTransport`接口

`RTCSctpTransport`接口允许应用程序访问与特定SCTP相关联的SCTP数据通道中的信息。

6.1.1.1 创建实例

用一个可选的初始状态 *initialState* **创建一个RTCSctpTransport** 实例的步骤如下：

1. 设 *transport* 为一个新的`RTCSctpTransport`对象。
2. 如果提供了 *initialState* ，则将 *transport* 的[SctpTransportState]槽初始化为它，否则为`"neww"`。
3. 为 *transport* 创建[MaxMessageSize]槽，并运行更新消息大小的最大值标记的步骤来初始化它。
4. 将 *transport* 的[MaxChannels]槽初始化为`null`。
5. 返回 *transport* 。

6.1.1.2 更新消息大小的最大值

运行以下步骤更新`RTCSctpTransport`的 **消息大小的最大值** 。

1. 设 *transport* 为被更新的`RTCSctpTransport`对象。
2. 如SCTP-SDP第六节所述，设 *remoteMaxMessageSize* 为从远程描述中读取的SDP属性`"max-message-size"`的值，若该属性丢失，则设为65536。
3. 设 *canSendSize* 为本客户端可以发送的字节数（例如，本地发送缓冲区的大小），若实现可以处理任意大小的消息则设为0。
4. 若 *remoteMaxMessageSize* 和 *canSendSize* 均为0，则将[MaxMessageSize]设为一个正无穷值。
5. 否则，若 *remoteMaxMessageSize* 和 *canSendSize* 其中一个为0，则将[MaxMessageSize]设为两者中的较大值。
6. 否则将[MaxMessageSize]设为 *remoteMaxMessageSize* 和 *canSendSize* 中的较小值。

6.1.1.3 连接过程

一旦一个SCTP传输被连接 ，意味着已经与一个`RTCSctpTransport`建立了SCTP关联，运行以下步骤：

1. 设 *transport* 为上下文中的`RTCSctpTransport`对象。
2. 设 *connection* 为与 *transport* 关联的`RTCPeerConnection`对象。
3. 将[MaxChannels]设为协商完成的传入和传出SCTP流的最小值。
4. 在 *transport* 上触发名为`statechange`的事件。
5. 对 *connection* 中的每个`RTCDataChannel`对象：
    1. 设 *channel* 为`RTCDataChannel`对象。
    2. 如果 *channel* 的[DataChannelId]槽值大于等于 *transport* 的[MaxChannels]槽值，则将其视为一个错误并关闭 *channel* ，否则宣称 *channel* 已经开启。

```
[Exposed=Window]
interface RTCSctpTransport {
  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：发送和接收数据通道中的所有SCTP数据包的传输。
- RTCSctpTransportState类型的`state`，只读：SCTP传输的当前状态。请求读值时，此属性必须返回[SctpTransportState]槽的值。
- unrestricted double类型的`maxMessageSize`，只读：可被传入`RTCDataChannel.send()`方法的最大数据代销。请求读值时，此属性必须返回[MaxMessageSize]槽的值。
- unsigned short类型的`maxChannels`，只读，可空：可同时使用的最大`RTCDataChannel`数量。请求读值时，属性必须返回[MaxChannels]槽的值。 **注意：** 在SCTP传输转移为`connected`状态之前，此属性的值一直为`null`。
- EventHandler类型的`onstatechange`：该事件处理器的事件类型为`statechange`。

6.1.2 `RTCSctpTransportState`枚举

`RTCSctpTransportState`枚举代表SCTP传输的状态。

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

`RTCSctpTransportState`枚举描述：

- connecting：`RTCSctpTransport`正在协商建立关联的过程中。这是`RTCSctpTransport`创建时其[SctpTransportState]槽的初始状态。

- connected：当建立关联的协商完成时，一个将[SctpTransportState]槽更新为`"connected"`的任务将被加入操作队列。
- closed：当收到SHUTDOWN或ABORT块或者有意关闭SCTP关联时（例如关闭对等连接或应用一个拒绝数据或更改SCTP端口的远程描述），一个将[SctpTransportState]槽更新为`"closed"`的任务将被加入操作队列。

6.2 `RTCDataChannel`

`RTCDataChannel`接口表示两个对端之间的双向数据信道。`RTCDataChannel`由`RTCPeerConnection`对象中的工厂方法创建。浏览器间发送的消息在RTCWEB-DATA和RTCWEB-DATA-PROTOCOL中有相关描述。

有两种方法可以用`RTCDataChannel`建立连接。第一种方法是在其中一个对端创建一个`RTCDataChannel`，并且`RTCDataChannelInit`字典的成员`negotiated`未被设置或已设为默认值`false`。这将在频内公布新通道，并在对端上触发带有相应`RTCDataChannel`对象的`RTCDataChannelEvent`。第二种方法是让应用程序协商一个`RTCDataChannel`。为此，创建一个`RTCDataChannel`对象，将`RTCDataChannelInit`字典的成员`negotiated`设为`true`，并通过频外信号（例如通过Web服务器）向另一方发出信号，另一方应该创建一个对应的`RTCDataChannel`，其`RTCDataChannelInit`字典的 成员`negotiated`设为`true`且和通道有相同的`id`。这将连接两个单独创建的`RTCDataChannel`对象。第二种方法使得创建具有非对称属性的通道成为可能，并通过指定匹配`id`以声明方式创建通道。

每个`RTCDataChannel`有一个与之关联的的 **底层数据传输** 来向对端传输实际的数据。在SCTP数据通道使用一个`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（例如顺序送达设置和可靠性模式）由对端在创建通道时配置。通道的属性在创建后便不再更改。端与端之间的实际有线协议由`WebRTC DataChannel`协议规范RTCWEB-DATA指定。

可将`RTCDataChannel`配置在不同的可靠性模式下操作。一个可靠的通道通过重传确保向对端送达数据。不可靠通道限制了重传次数（`maxRetransmits`）或允许传输（包括重传）的时间（`maxPacketLifeTime`）。这些属性不能同时使用，尝试这样做会导致错误。对这些属性没有限制的话即为可靠通道。

使用`createDataChannel`创建或通过`RTCDataChannelEvent`调度的`RTCDataChannel`刚开始必须处于`"connecting"`状态。当`RTCDataChannel`对象的底层数据传输就绪时，用户代理必须宣称`RTCDataChannel`已开启。

运行以下步骤 **创建一个`RTCDataChannel`**：

1. 设 *channel* 为新创建的`RTCDataChannel`对象。
2. 将 *channel* 的[ReadyState]槽初始化为`"connecting"`。
3. 将 *channel* 的[BufferedAmount]槽初始化为`0`。
4. 为 *channel* 创建`[DataChannelLabel]`，`[Ordered]`，`[MaxPacketLifeTime]`，`[MaxRetransmits]`，`[DataChannelProtocol]`，`[Negotiated]`，`[DataChannelId]`，`[DataChannelPriority]`槽。
5. 返回 *channel*。

当用户代理 **宣称`RTCDataChannel`已开启** 时，用户代理必须将包含以下步骤的任务加入操作队列：

1. 若与之关联的`RTCPeerConnection`的[IsClosed]槽为`true`，则终止后续步骤。
2. 设 *channel* 为将要宣称为开启的`RTCDataChannel`对象。
3. 如果 *channel* 的[ReadyState]槽为`closing`或`closed`，则终止后续步骤。
4. 将 *channel* 的[ReadyState]槽设为`open`。
5. 在 *channel* 上触发名为`open`的事件。

当一个底层数据传输将被宣布开启时（对端创建了一个未设置`negotiated`属性或被设为`false`的通道），未启动创建过程的对端用户代理必须将包含以下步骤的任务加入操作队列：

1. 若与之关联的`RTCPeerConnection`的[IsClosed]槽为`true`，则终止后续步骤。
2. 创建一个`RTCDataChannel`，命名为 *channel* 。
3. 设 *configuration* 为从对端接收的信息包，作为建立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 *channel* 的`[DataChannelLabel]`, `[Ordered]`, `[MaxPacketLifeTime]`, `[MaxRetransmits]`, `[DataChannelProtocol]`, `[DataChannelId]`槽初始化为 *configuration* 中的对应值。
5. 将 *channel* 的[Negotiated]槽设为`false`。
6. 基于 *configuration* 中的整数优先级值初始化 *channel* 的[DataChannelPriority]槽，具体的映射关系如下：

***configuration* 优先级值**	**`RTCPriorityType`值**
0到128	very-low
129到256	low
257到512	medium
513及更大	high

7. 将*channel* 的[ReadyState]设为`open`（但未触发`open`事件）。 **注意：** 这允许在触发`open`事件之前开始在`datachannel`事件处理器内发送消息。
8. 利用`RTCDataChannelEvent`接口触发名为`datachannel`的事件，接口使用的`RTCPeerConnection`对象的`channel`属性被设为 *channel* 。
9. 宣布数据通道已开启。

通过运行**关闭程序**，可以以非突发的方式解除`RTCDataChannel`对象的底层数据传输。当这种情况发生时，用户代理必须将包含以下步骤的任务加入操作队列：

1. 设 *channel* 为将要关闭传输的`RTCDataChannel`对象。
2. 除非该过程是由 *channel* 的`close`方法启动的，否则将 *channel* 的[ReadyState]槽设为`closing`。
3. 并行地运行以下步骤：
    1. 结束当前 *channel* 中所有等待中的消息的发送工作。
    2. 遵循为 *channel* 的底层传输定义的关闭过程：
        1. 若传输基于SCTP，则遵循RTCWEB-DATA 6.7节中的做法。
    3. 按照相关步骤完成 *channel* 的数据传输关闭。

当一个`RTCDataChannel`的底层数据传输 **已被关闭** ，用户代理必须将包含以下步骤的任务加入操作队列：

1. 设 *channel* 为传输已关闭的`RTCDataChannel`对象。
2. 将 *channel* 的[ReadyState]设为`closed`。
3. 若关闭传输时 **出错** ，利用`RTCErrorEvent`接口在 *channel* 上触发名为`error`的事件，其`errorDetail`属性被设为"sctp-failure"。
4. 在 *channel* 上触发名为`close`的事件。

在某些情况下，用户代理可能 **无法创建** `RTCDataChannel`的底层数据传输。例如，数据通道的`id`可能超出SCTP握手中RTCWEB-DATA协商好的范围。当用户代理确定无法创建`RTCDataChannel`的底层数据传输时，用户代理必须将包含以下步骤的任务加入操作队列：

1. 设 *channel* 为用户代理无法创建底层数据传输的`RTCDataChannel`对象。
2. 将 *channel* 的[ReadyState]设为`closed`。
3. 利用`RTCErrorEvent`接口在 *channel* 上触发名为`error`的事件，其`errorDetail`属性被设为"data-channel-failure"。
4. 在 *channel* 上触发名为`close`的事件。

当通过`type`类型和`rawData`数据的底层数据传输 **接收** 到`RTCDataChannel`消息时，用户代理必须将包含以下步骤的任务加入操作队列：

1. 设 *channel* 为用户代理已收到消息的`RTCDataChannel`对象。
2. 若 *channel* 的[ReadyState]槽值不是`open`，则终止后续步骤并忽略 *rawData* 。
3. 根据 *type* 与 *channel* 的`binaryType`的匹配结果执行以下子步骤：
    - 若 *type* 表示 *rawData* 为`string`类型：设 *data* 为 *rawData* 经UTF-8格式解码后的DOMString。
    - 若 *type* 表示 *rawData* 为二进制类型且`binaryType`为`"blob"`：设 *data* 为包含 *rawData* 作为原始数据源的新`Blob`对象。
    - 若 *type* 表示 *rawData* 为二进制类型且`binaryType`为`"arrayBuffer"`：设 *data* 为包含 *rawData* 作为原始数据源的新`ArrayBuffer`对象。
4. 利用`MessageEvent`接口触发名为`message`的事件，其`origin`属性初始化为创建了与 *channel* 关联的`RTCPeerConnection`的文档源，并且`data`属性初始化为 *channel* 上的 *data* 。

```
[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`属性表示被用于区分不同`RTCDataChannel`对象的标签。可以用脚本为同一标签创建多个`RTCDataChannel`对象。请求读值时，该属性必须返回[DataChannelLabel]槽的值。
- boolean类型的`ordered`，只读：如果`RTCDataChannel`是有序的，则`ordered`属性返回`true`，如果允许按其他顺序到达，则返回`false`。请求读值时，属性必须返回[Ordered]槽的值。
- unsigned short类型的`maxPacketLifeTime`，只读，可空：`maxPacketLifeTime`属性返回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）。请求读值时，属性必须返回[MaxPacketLifeTime]槽的值。
- USVString类型的`protocol`，只读：`protocol`属性返回`RTCDataChannel`中使用的子协议的名字。请求读值时，属性必须返回[DataChannelProtocol]槽的值。
- boolean类型的`negotiated`，只读：若本`RTCDataChannel`已经由应用程序协商完毕，则`negotiated`属性返回`true`，否则返回`false`。请求读值时，属性必须返回[Negotiated]槽的值。
- unsigned short类型的`id`，只读，可空：`id`属性返回此`RTCDataChannel`的ID。该值初始时为`null`，如果在创建通道时未提供ID，且尚未协商SCTP传输的DTLS角色，则返回`null`。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。ID设置为非空值后就不会被更改。请求读值时，属性必须返回[DataChannelId]槽的值。
- RTCPriorityType类型的`priority`，只读：`priority`属性返回此`RTCDataChannel`的优先级。优先级在通道创建时由用户代理指定。请求读值时，属性必须返回[DataChannelPriority]槽的值。
- RTCDataChannelState类型的`readyState`，只读：`readyState`属性代表`RTCDataChannel`对象的状态。请求读值时，属性必须返回[ReadyState]槽的值。
- unsigned long类型的`bufferedAmount`，只读：请求读值时，`bufferedAmount`属性必须返回[BufferedAmount]槽的值。该属性公开通过`send()`方法加入发送队列的应用程序数据（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，但为了防止数据竞争，在当前任务返回事件循环之前，不得减小返回值。该值不包括协议产生的帧开销，或由操作系统/网络硬件实现的缓冲。只要[ReadyState]槽处于`open`状态，[BufferedAmount]槽的值只会随着每次调用`send()`方法而增加; 但通道关闭，槽的值也不会重置为零。当底层数据传输从其发送队列发送数据时，用户代理必须将一个任务加入操作队列，该任务将[BufferedAmount]中的值减去发送的字节数。
- unsigned long类型的`bufferedAmountLowThreshold`：`bufferedAmountLowThreshold`属性设置`bufferedAmount`的最低阈值。当`bufferedAmount`的值减小至小于等于此阈值时，将触发`bufferedamountlow`事件。每个新`RTCDataChannel`都将`bufferedAmountLowThreshold`初始化为零，但应用程序可能随时更改其值。
- EventHandler类型的`onopen`：该事件处理器的事件类型为`open`。
- EventHandler类型的`onbufferedamountlow`：该事件处理器的事件类型为`bufferedamountlow`。
- EventHandler类型的`onerror`：该事件处理器的事件类型是`RTCErrorEvent`。其`errorDetail`包含"sctp-failure"，`sctpCauseCode`包含SCTP Cause Code值，`message`包含SCTP Cause-Specific-Information，也可能包含其他文本。
- EventHandler类型的`onclose`：该事件处理器的事件类型为`close`。
- EventHandler类型的`onmessage`：该事件处理器的事件类型为`message`。
- DOMString类型的`binaryType`：请求读值时，`binaryType`属性必须返回最新设置的值。请求写值时，如果新的值是`"blob"`字符串或`"arraybuffer"`字符串，则将IDL属性设为这个新值，否则抛出一个`SyntaxError`。当一个`RTCDataChannel`对象被创建，其`binaryType`属性必须被初始化为`"blob"`字符串。此属性控制脚本读取二进制数据的方式。详见WEBSOCKETS-API。

方法：

- *close* ：关闭`RTCDataChannel`。不论`RTCDataChannel`是被本端创建还是对端创建，这个方法都可以被调用。当`close`方法被调用时，用户代理必须运行以下步骤：

    1. 设 *channel* 为即将关闭的`RTCDataChannel`对象。

2. 若 *channel* 的[ReadyState]值为`closing`或`closed`，则终止后续步骤。
3. 将 *channel* 的[ReadyState]值设为`closing`。
4. 若关闭程序还未启动，则将它启动。

- *send* ：以`string`对象作为参数，运行send()算法中指定步骤。
- *send* ：以`Blob`对象作为参数，运行send()算法中指定步骤。
- *send* ：以`ArrayBuffer`对象作为参数，运行send()算法中指定步骤。
- *send* ：以`ArrayBufferView`对象作为参数，运行send()算法中指定步骤。

```
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";
};
```

`RTCDataChannelInit`字典成员：

- boolean类型的`ordered`，缺省值为`true`：若被设为`false`，则允许数据无序到达。默认值为`true`，保证数据一定按需到达。
- unsigned short类型的`maxPacketLifeTime`：限制通道在未得到确认的情况下传输或重传数据的时间（以毫秒为单位）。如果该值超过用户代理支持的最大值，则可以限制使用该值。
- unsigned short类型的`maxRetransmit`：在数据未送达对端的情况下限制通道数据重传的次数。如果该值超过用户代理支持的最大值，则可以限制使用该值。
- USVString类型的`protocol`，缺省值为`""`：该通道使用的子协议名。
- boolean类型的`negotiated`，缺省值为`false`：缺省值`false`指示用户代理在频内公布此通道并指示其他端分派相应的`RTCDataChannel`对象。如果被设为`true`，则由应用程序协商通道并在对端创建具有相同`id`的`RTCDataChannel`对象。 **注意：** 如果被设为`true`，则应用程序必须注意，在对端创建了一个数据通道来接收它之前不要发送消息。在没有关联数据通道的SCTP流上接收消息是未定义行为，消息可能会以静默的方式丢弃。只要两个端在第一个邀请/应答交换完成之前创建其数据通道，这样的情况就不可能发生。
- unsigned short类型的`id`：覆盖此通道默认选择的ID。
- RTCPriorityType类型的`priority`，缺省值为`low`：此通道的优先级。

`send()`方法以重载的方式处理不同的数据参数类型。当此方法的任意版本被调用时，用户代理必须运行以下步骤：

1. 设 *channel* 为将要发送数据的`RTCDataChannel`对象。
2. 若 *channel* 的[ReadyState]槽值不是`open`，抛出一个`InvalidStateError`。
3. 根据方法的参数运行以下对应的子步骤：
    - `string`对象：设 *data* 为一个字节缓冲区，代表方法参数以UTF-8解码后的结果。
    - `Blob`对象：设 *data* 为由`Blob`对象表示的源数据。
    - `ArrayBuffer`对象：设 *data* 为存储在缓冲区中由`ArrayBuffer`对象表示的数据。

- `ArrayBufferView`对象：设 *data* 为存储在缓冲区部分中的数据，数据由被`ArrayBufferView`对象引用的`ArrayBuffer`对象表示。 **注意：** 除了本方法已被重载的数据参数类型，其他所有数据参数类型都会造成`TypeError`。这同样包括`null`和`undefined`类型。

4. 若 *data* 的字节数超出与 *channel* 相关联的`RTCSctpTransport`对象中的`maxMessageSize`值，则抛出一个`TypeError`。
5. 将 *data* 加入 *channel* 的底层数据传输队列。如果因为没有足够多可用的缓存空间导致 *data* 的入队操作失败，则抛出一个`OperationError`。 **注意：** 实际的数据发送是并行的。如果发送数据时导致了一个SCTP级别的错误，则会通过`onerror`事件异步地将错误通知应用。
6. 将[BufferedAmount]槽中的值加上 *data* 的字节数。

```
enum RTCDataChannelState {
    "connecting",
    "open",
    "closing",
    "closed"
};
```

`RTCDataChannelState`枚举值描述：

- connecting：用户代理正尝试建立底层数据传输。无论`RTCDataChannel`是由`createDataChannel`方法创建的，还是作为`RTCDataChannelEvent`的一部分指派的，它的初始状态都是`connecting`。
- open：底层数据传输已建立且可能发生通信。
- closing：关闭底层数据传输的关闭程序已经启动。
- closed：底层数据传输已被关闭或不能建立。

6.3 `RTCDataChannelEvent`

`datachannel`事件使用`RTCDataChannelEvent`接口。

```
[Constructor(DOMString type, RTCDataChannelEventInit eventInitDict),
 Exposed=Window]
interface RTCDataChannelEvent : Event {
    readonly attribute RTCDataChannel channel;
};
```

构造函数：

`RTCDataChannelEvent`

属性：

- RTCDataChannel类型的`channel`，只读：`channel`属性代表与事件相关联的`RTCDataChannel`对象。

```
dictionary RTCDataChannelEventInit : EventInit {
    required RTCDataChannel channel;
};
```

`RTCDataChannelEventInit`字典成员：

- RTCDataChannel类型的`channel`，必需项：由事件宣布的`RTCDataChannel`对象。

6.4 垃圾回收

如果满足以下情况，则`RTCDataChannel`对象一定不能被回收：

- [ReadyState]槽为`connecting`且为`open`事件，`message`事件，`error`事件，或`close`事件注册了至少一个事件监听器。
- [ReadyState]槽为`open`且`message`事件，`error`事件，或`close`事件注册了至少一个事件监听器。
- [ReadyState]槽为`closing`且为`error`事件，或`close`事件注册了至少一个事件监听器。
- 底层数据传输已被创建且数据已被加入发送队列。

7. 点对点DTMF

本节介绍`RTCRtpSender`上的一个接口，用于在`RTCPeerConnection`上发送DTMF（按键）值。关于如何将DTMF发送到其他对等端，详见RTCWEB-AUDIO。

7.1 RTCRtpSender接口扩展

点对点DTMF的`RTCRtpSender`接口API扩展如下所示。

```
partial interface RTCRtpSender {
  readonly attribute RTCDTMFSender? dtmf;
};
```

属性：

- RTCDTMFSender类型的`dtmf`，只读，可空：请求读值时，`dtmf`属性返回[Dtmf]内部槽的值，其值表示可用于发送DTMF的`RTCDTMFSender`，如果未被设置，则返回`null`。 当`RTCRtpSender`的[SenderTrack]的类型为`"audio"`时，[Dtmf]内部槽会被设置。

7.3 `RTCDTMFSender`

用户代理运行以下步骤 **创建一个`RTCDTMFSender`** ：

1. 设 *dtmf* 为新创建的`RTCDTMFSender`对象。
2. 为 *dtmf* 创建[Duration]槽。
3. 为 *dtmf* 创建[InterToneGap]槽。
4. 为 *dtmf* 创建[ToneBuffer]槽。

```
[Exposed=Window]
interface RTCDTMFSender : EventTarget {
  void insertDTMF(DOMString tones,
  optional unsigned long duration = 100,
  optional unsigned long interToneGap = 70);
  attribute EventHandler ontonechange;
  readonly attribute boolean canInsertDTMF;
  readonly attribute DOMString toneBuffer;
};
```

属性：

- EventHandler类型的`ontonechange`：该事件处理器的事件类型为`tonechange`。
- boolean类型的`canInsertDTMF`，只读：指示`RTCDTMFSender`类型的 *dtmfSender* 能否发送DTMF。请求读值时，用户代理必须返回确认DTMF能否被发送的运行结果。
- DOMString类型的`toneBuffer`，只读：`toneBuffer`属性必须返回剩余要播放的音调列表。此列表的语法，内容和解释，详见insertDTMF。

方法：

- *insertDTMF* ：`RTCDTMFSender`对象的`insertDTMF`方法被用于DTMF音调。
  `tone`参数被视为一系列字符。相关的DTMF音调由字符0-9，A-D，#和*生成。字符a-d必须在输入时被标准化为大写字符，等同于A-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','，它指示在处理`tone`参数中的下一个字符之前的延迟2秒。所有其他字符（仅其他字符）必须被视为 **无法识别** 。
  `duration`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`参数指示音调之间的间隔（ms）。用户代理将其限制为30毫秒到6000毫秒的某个值。默认值为70毫秒。
  浏览器可以增加`duration`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，但不能超过单个RTP音频数据包的持续时间。
  当`insertDTMF()`方法被调用，用户代理必须按以下步骤运行：

  1. 设 *sender* 为用来发送DTMF的`RTCRtpSender`。
  2. 设 *transceiver* 为与 *sender* 相关联的`RTCRtpTransceiver`对象。
  3. 若 *transceiver* 的[Stopped]槽为`true`，则抛出一个`InvalidStateError`。
  4. 若 *transceiver* 的[CurrentDirection]槽为`recvonly`或`inactive`，则抛出一个`InvalidStateError`。
  5. 设 *dtmf* 为与 *sender* 相关联的`RTCDTMFSender`。
  6. 若确认DTMF能否被发送的运行结果为`false`，则抛出一个`InvalidStateError`。
  7. 设 *tones* 为方法的第一个参数。
  8. 若 *tones* 包含不能识别的字符，则抛出一个`InvalidCharacterError`。
  9. 设[ToneBuffer]槽为 *tones* 。
  10. 设 *dtmf* 的[Duration]槽为`duration`参数的值。
  11. 设 *dtmf* 的[InterToneGap]槽为`interToneGap`参数的值。
  12. 若`duration`参数的值少于40ms，则将 *dtmf* 的[Duration]槽设为40ms。
  13. 若`duration`参数的值大于6000ms，则将 *dtmf* 的[Duration]槽设为6000ms。
  14. 若`interToneGap`参数的值少于30ms，则将 *dtmf* 的[InterToneGap]槽设为30ms。
  15. 若`interToneGap`参数的值大于6000ms，则将 *dtmf* 的[InterToneGap]槽设为6000ms。
  16. 若[ToneBuffer]槽为空字符串，则终止后续步骤。
  17. 如果计划运行 *Playout* 任务，则终止这些步骤; 否则将包含以下步骤的任务（播出任务）加入操作队列：
      1. 若 *transceiver* 的[Stopped]槽为`true`，则终止这些步骤。
      2. 若 *transceiver* 的[CurrentDirection]槽为`recvonly`或`inactive`，则终止这些步骤。
      3. 若[ToneBuffer]槽包含空字符串，则利用`RTCDTMFToneChangeEvent`接口触发名为`tonechange`的事件，其`RTCDTMFSender`对象的`tone`属性被设为空字符串，最后终止这些步骤。

4. 从[ToneBuffer]中移除第一个字符，并设那个字符为 *tone* 。
5. 如果 *tone* 为","，则在与之关联的RTP媒体流上延迟发送音调2000ms，并将一个被标记为 *Playout* 的任务加入操作队列，2000ms后执行。
6. 如果 *tone* 不是","，则在[Duration]ms后使用合适的编解码器开始在与之关联的RTP媒体流上播放 *tone* ，然后将一个被标记为 *Playout* 的任务加入操作队列，[InterToneGap]ms后执行。
7. 利用`RTCDTMFToneChangeEvent`接口触发名为`tonechange`的事件，其`RTCDTMFSender`对象的`tone`属性被设为空字符串。

因为`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]槽中）和附加在一起的新音调的字符串调用`insertDTMF`。使用空`tone`参数调用`insertDTMF`可用于取消在当前播放音调之后等待播放的所有音调。

7.3 canInsertDTMF算法

为了确定`RTCDTMFSender`实例 *dtmfSender* **能否发送DTMF** ，用户代理必须将包含以下步骤的任务加入操作队列：

1. 设 *sender* 为与 *dtmfSender* 相关联的`RTCRtpSender`对象。
2. 设 *transceiver* 为与 *sender* 相关联的`RTCRtpTransceiver`对象。
3. 设 *connection* 为与 *sender* 相关联的`RTCPeerConnection`对象。
4. 若 *connection* 的[RTCPeerConnectionState]不是`"connected"`，则返回`false`。
5. 若 *sender* 的[SenderTrack]为`null`，则返回`false`。
6. 若 *transceiver* 的[CurrentDirection]槽不是`"sendrecv"`或`"sendonly"`，则返回`false`。
7. 若 *sender* 的`[SendEncodings][0].active`为`false`，则返回`false`。
8. 如果该 *sender* 没有为mimetype `"audio/telephone-event"`协商好编解码器，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),
  Exposed=Window]
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（详见[insertDTMF]）。 如果该值为空字符串，则表示[ToneBuffer]槽为空字符串，并且前一个音调已完成播放。

```
dictionary RTCDTMFToneChangeEventInit : EventInit {
  required DOMString tone;
};
```

`RTCDTMFToneChangeEventInit`字典成员：

- DOMString类型的`tone`：`tone`属性包含刚刚开始播放的音调（包括","）的字符（详见[insertDTMF]）。 如果该值为空字符串，则表示[ToneBuffer]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 介绍

基本统计模型是浏览器以统计对象的形式维护的一组受监控对象的统计信息。 **选择器** 可以引用一组相关的对象。例如，选择器可以是`MediaStreamTrack`。要使媒体轨成为有效的选择器，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，该算法将用到选择器的发送端或接收端。 `stats`对象中返回的统计信息被设计为可以根据`RTCStats`字典的`id`成员将重复查询链接起来。因此，Web应用程序可以通过在该时段的开始和结束时刻请求测量，来对给定时间段内的应用状态进行完整的统计。 除少数异常情况外，受监控对象一旦创建，就会在与其关联的`RTCPeerConnection`期间一直存在。这样可以确保`getStats()`返回的结果中的统计信息在关闭对端连接之后仍然可用。 只有少数受监控对象的生命周期较短。对于这些对象，它们的生命周期在算法将它们删除时结束。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，其包含将要同时被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中的对象描述中由关于何时删除这些被监视对象的阐述。

8.2 `RTCPeerConnection`接口扩展

统计API扩展的`RTCPeerConnection`接口如下所示：

```
partial interface RTCPeerConnection {
  Promise<RTCStatsReport> getStats(optional MediaStreamTrack? selector = null);
  attribute EventHandler onstatsended;
};
```

属性：

- EventHandler类型的`onstatsend`：本事件处理器的事件类型为`statsend`。
  为了 **删除** 与`RTCPeerConnection`对象 *connection* 相关联的一些被监控对象的 **统计信息** ，用户代理必须并行地运行以下步骤：
  1. 只收集将被删除的被监控对象的统计信息。这些统计信息必须代表被删除时的最终值。这些被监控对象的统计信息一定不能在后续的`getStats()`调用中出现。
  2. 将包含以下信息的任务加入操作队列：
     1. 设 *report* 为一个新的`RTCStatsReport`对象。
     2. 对于每个被监控对象，利用为该受监视对象收集的统计信息创建一个新的相关统计信息对象，并将其添加至 *report* 中。
     3. 利用`RTCStatsEvent`接口触发名为`statsended`的事件，其`report`属性被设为 *report* 。

方法：

- *getStats* ：为给定的选择器收集信息，并异步地报告结果。
  当`getStats()`方法被调用，用户代理必须按以下步骤运行：
  1. 设 *selectorArg* 为方法的第一个参数。
  2. 设 *connection* 为调用此方法的`RTCPeerConnection`对象。

3. 若 *selectorArg* 为`null`，则设 *selector* 为`null`。
4. 如果 *selectorArg* 是`MediaStreamTrack`类型，则设 *selector* 为 *connection* 上`track`成员与 *selectorArg* 匹配的`RTCRtpSender`或`RTCRtpReceiver`对象。如果不存在这样的发送端或接收端，或者由多个发送端或接收端符合此条件，则用新创建的`InvalidAccessError`拒绝promise并返回。
5. 设 *p* 为一个新的promise。
6. 并行地运行以下步骤：
    1. 根据统计信息选择算法收集由 *selector* 表示的统计信息。
    2. 利用上一步得到的包含收集到的统计信息的`RTCStatsReport`对象解析 *p* 。
7. 返回 *p* 。

8.3 `RTCStatsReport`对象

`getStats()`方法以`RTCStatsReport`对象的形式提供成功的运行结果。`RTCStatsReport`对象是标识被检查对象（`RTCStats`实例中的`id`属性）的字符串和对应`RTCStats`派生词典之间的映射。`RTCStatsReport`可以由几个`RTCStats`派生的字典组成，每个字典为底层对象报告其统计信息，底层对象的实现与选择器相关。通过对所有统计数据中的某种类型求和，可以求得选择器的总量; 例如，如果`RTCRtpSender`使用多个SSRC通过网络传输媒体轨，则`RTCStatsReport`可以为每个SSRC持有一个对应的`RTCStats`派生字典（可以通过"ssrc"统计属性的值来区分）。

```
[Exposed=Window]
interface RTCStatsReport {
  readonly maplike<DOMString, object>;
};
```

此接口有"entries", "forEach", "get", "has", "keys", "values", @@iterator方法和一个`readonly maplike`的"size"获取器。 使用这些方法来检索此统计报告由`RTCStats`组成的各种字典。所有被支持的属性名称WEBIDL-1的集合被定义为此统计报告生成的所有`RTCStats`派生词典的id集合。

8.4 `RTCStats`字典

`RTCStats`字典表示通过检查特定受监视对象而构造的`stats`对象。`RTCStats`字典是一种基本类型，它指定一组默认属性，例如`timestamp`和`type`。通过扩展`RTCStats`字典添加特定的统计信息。 注意，虽然统计信息名称已被标准化，但任何给定的实现都可能使用实验值或对Web应用程序透明的值。因此，应用程序必须准备好处理未知的统计数据。 统计数据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，以便"average packet size"可以被计算为"bytes/packets" - 如果时间间隔不同，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;
};
```

`RTCStats`字典成员：

- DOMHighResTimeStamp类型的`timestamp`：DOMHighResTimeStamp类型HIGHRES-TIME的`timestamp`与本对象关联。本时间戳代表相对于UNIX纪元（1970年1月1日，UTC）的时间。对于来自远程数据源（例如来自

接收的RTCP分组）的统计数据，时间戳表示信息到达本端的时间。如果适用的话，可以在`RTCStats`的派生字典中的附加字段中找到远程时间戳。

- RTCStatsType类型的`type`：本对象的类型。`type`属性必须被初始化为本`RTCStats`字典代表的最具体类型的名字。
- DOMString类型的`id`：与本对象关联的唯一标识符，用于生成此`RTCStats`对象。如果从两个不同的`RTCStatsReport`对象中提取出的两个`RTCStats`对象是通过检查相同的底层对象生成的，则它们必须具有相同的`id`。用户代理可以自由选择`id`的格式，只要它符合上述要求即可。

`RTCStatsType`的合法值集合，以及它们表示的`RTCStats`的派生字典，都被记录在WEBRTC-STATS。

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`对象相应子类的`stats`对象，给出受监视对象生命周期结束时的相关统计信息值。

```
dictionary RTCStatsEventInit : EventInit {
  required RTCStatsReport report;
};
```

`RTCStatsEventInit`字典成员：

- RTCStatsReport类型的`report`，必需项：包含`RTCStats`对象，提供生命周期已结束对象的统计信息。

8.6 统计信息选择算法

统计信息选择算法 如下所示：

1. 设 *result* 为一个空`RTCStatsReport`对象。
2. 若 *selector* 为`null`，则为整个 *connection* 收集统计信息，将它们添加入 *result* 并将其返回，然后终止后续步骤。
3. 若 *selector* 为`RTCRtpSender`，则收集统计信息并将以下对象加入 *result* 。
    - 所有代表被 *selector* 发送的RTP流的`RTCOutboundRTPStreamStats`对象。
    - 所有被`RTCOutboundRTPStreamStats`对象直接或间接引用的统计对象。
4. 若 *selector* 为`RTCRtpReceiver`，则收集统计信息并将以下对象加入 *result* 。
    - 所有代表被 *selector* 接收的RTP流的`RTCOutboundRTPStreamStats`对象。
    - 所有被`RTCOutboundRTPStreamStats`对象直接或间接引用的统计对象。

5. 返回 *result* 。

8.7 强制实施统计数据

WEBRTC-STATS中罗列的统计数据应该能覆盖大范围的使用场景。但并非所有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，以及这些类型对该对象来说有效时所列出的以下属性：

- `RTCRTPStreamStats` 及其 `ssrc, kind, transportId, codecId, nackCount` 属性。
- `RTCReceivedRTPStreamStats` 及其继承字典类型属性中的所有必需项，除此之外还有 `packetsReceived, packetsLost, jitter, packetsDiscarded` 属性。
- `RTCInboundRTPStreamStats` 及其继承字典类型属性中的所有必需项，除此之外还有 `bytesReceived, trackId, receiverId, remoteId, framesDecoded` 属性。
- `RTCRemoteInboundRTPStreamStats` 及其继承字典类型属性中的所有必需项，除此之外还有 `localId, roundTripTime` 属性。
- `RTCSentRTPStreamStats` 及其继承字典类型属性中的所有必需项，除此之外还有 `packetsSent, bytesSent` 属性。
- `RTCOutboundRTPStreamStats` 及其继承字典类型属性中的所有必需项，除此之外还有 `trackId, senderId, remoteId, framesEncoded` 属性。
- `RTCRemoteOutboundRTPStreamStats` 及其继承字典类型属性中的所有必需项，除此之外还有 `localId, remoteTimestamp` 属性。
- `RTCPeerConnectionStats` 及其 `dataChannelsOpened, dataChannelsClosed` 属性。
- `RTCDataChannelStats` 及其 `label, protocol, datachannelId, state, messagesSent, bytesSent, messagesReceived, bytesReceived` 属性。
- `RTCMediaStreamStats` 及其 `streamIdentifer, trackIds` 属性。
- `RTCMediaStreamTrackStats` 及其 `detached` 属性。
- `RTCMediaHandlerStats` 及其 `trackIdentifier, remoteSource, ended` 属性。
- `RTCAudioHandlerStats` 及其 `audioLevel` 属性。
- `RTCVideoHandlerStats` 及其 `frameWidth, frameHeight, framesPerSecond` 属性。
- `RTCVideoSenderStats` 及其 `framesSent` 属性。
- `RTCVideoReceiverStats` 及其 `framesReceived, framesDecoded, framesDropped, framesCorrupted` 属性。
- `RTCCodecStats` 及其 `payloadType, codec, clockRate, channels, sdpFmtpLine` 属性。
- `RTCTransportStats` 及其 `bytesSent, bytesReceived, rtcpTransportStatsId, selectedCandidatePairId, localCertificateId, remoteCertificateId` 属性。
- `RTCIceCandidatePairStats` 及其 `transportId, localCandidateId, remoteCandidateId, state, priority, nominated, bytesSent, bytesReceived, totalRoundTripTime, currentRoundTripTime` 属性。
- `RTCIceCandidateStats` 及其 `address, port, protocol, candidateType, url` 属性。
- `RTCCertificateStats` 及其 `fingerprint, fingerprintAlgorithm, base64Certificate, issuerCertificateId` 属性。

实现可以支持生成WEBRTC-STATS中定义的任何其他统计信息，也可以生成尚未记录文档中的统计信息。

8.8 GetStats例子

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况。可能使用以下示例代码：

```
EXAMPLE 9
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. 用于网络的Medis Stream API扩展

9.1 介绍

GETUSERMEDIA规范中定义的`MediaStreamTrack`接口通常代表一路音频流或者视频流数据。一个或多个`MediaStreamTrack`可被`MediaStream`所收集（严格来说，[GETUSERMEDIA]中定义的`MediaStream`可以包含零或多个`MediaStreamTrack`对象）。 可以扩展`MediaStreamTrack`以表示来自或被发送到远程对端（例如，不仅仅是本地相机）的媒体流。本节将介绍在`MediaStreamTrack`对象上启用此功能所需的扩展。 RTCWEB-RTP，RTCWEB-AUDIO和RTCWEB-TRANSPORT描述了如何将媒体传输到对端。 发送给另一个对端的`MediaStreamTrack`将作为一个且仅一个`MediaStreamTrack`在接收端显示。对端被定义为支持该规范的用户代理。此外，发送端一侧的应用程序可以指示`MediaStreamTrack`所属的`MediaStream`对象。对应的`MediaStream`对象将在接收端一侧被创建（如果尚未存在）并相应地填充。 正如本文档前面提到的，应用程序可以使用`RTCRtpSender`和`RTCRtpReceiver`对象来对`MediaStreamTrack`的传输和接收进行更细粒度的控制。 通道是`MediaStream`规范中的最小单元。通道旨在被编码在一起以便传输，例如，RTP有效载荷类型。需要被编解码器共同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媒体轨中的所有通道。 `MediaStreamTrack`输入和输出的概念也同样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）将把远程对端接收的数据作为输入。类似地，来自本源的

MediaStreamTrack（例如通过GETUSERMEDIA的摄像机）将具有输出，该输出表示传输到远程对端的内容，前提是该对象与RTCPeerConnection对象一起使用。 GETUSERMEDIA中提到的复制MediaStream和MediaStreamTrack对象的概念也适用于此处。例如，可以在视频会议场景中使用此功能，以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，对使用"视频静音"功能的用户作出响应）。 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。

> 注意：在本文档中，我们仅指定以下与RTCPeerConnection一起使用的相关对象的各个方面。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 MediaStream

9.2.1 id

MediaStream中指定的id属性返回该流的唯一标识id，因此流可以被远程对端的RTCPeerConnection API识别。当MediaStream被创建为代表从远程对等端获取的流时，id属性根据远程数据源提供的信息初始化。

> 注意：MediaStream对象的id对于流的数据源来说是唯一的，但这并不意味着不能以流的副本结束整个流程。例如，本地生成的流媒体轨可以使用RTCPeerConnection从一个用户代理发送到远程对等端，然后以相同的方式发送回原用户代理，在这种情况下，原用户代理将具有多个相同id的流（本地生成的id和远程peer发送的id）。

9.3 MediaStreamTrack

在非本地媒体源的场景下（RTP源，每个MediaStreamTrack都与一个RTCRtpReceiver相关联），MediaStreamTrack对象一直是MediaStream对象的强引用。 每当RTCRtpReceiver在对应MediaStreamTrack的静音RTP源上接收数据，并且RTCRtpTraceceiver对象的[Receptive]槽是RTCRtpReceiver成员，且值为true时，它必须将设置相应MediaStreamTrack的静音状态为false的任务加入操作队列。 当RTCRtpReceiver已接收的RTP源媒体流中的某一SSRC由于收到BYE信号或因超时而被删除时，它必须将一个把相应MediaStreamTrack的静音状态设置为true的任务加入操作队列并等待执行。注意，setRemoteDescription还可以将媒体轨的静音状态设置为true。 **添加媒体轨，移除媒体轨和设置媒体轨静音状态**的操作在GETUSERMEDIA中指定。 当RTCRtpReceiver接收端生成的MediaStreamTrack轨已经结束GETUSERMEDIA时（例如通过调用receiver.track.stop），用户代理可以选择释放为传入流预先分配的资源，例如关闭解码器接收器。

9.3.1 MediaTrackSupportedConstraints, MediaTrackCapabilities, MediaTrackConstraints及MediaTrackSettings

[GETUSERMEDIA]概述了MediaTrackSupportedConstraints，MediaTrackCapabilites，MediaTrackConstraints和MediaTrackSettings的基本内容。但是，由RTCPeerConnection提供的MediaStreamTrack对象中的MediaTrackSettings的成员变量，将通过setRemoteDescription应用的远程RTCSessionDescription描述和实际RTP数据提供的数据进行填充。这意味着某些成员（例如facingMode，echoCancellation，latency，deviceId和groupId）将始终缺失。

10. 例子与调用流程

10.1 简易的点对点示例

当两个对等端决定要建立彼此的连接时，它们都将经历这些步骤。STUN/TURN服务器配置描述了可用于获取公共IP地址或设置NAT遍历的服务器。在通信最开始，它们还必须使用相同的频外机制互相发送信令通道内的数据。

```
EXAMPLE 10
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 进阶的点对点示例-热身

当两个对等端决定彼此建立连接并希望ICE，DTLS和媒体连接"热身"以便它们准备好立即发送和接收媒体数据时，它们都会经历这些步骤。

```
EXAMPLE 11
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 点对点传输示例-媒体数据先于信号

应答方可能希望在发送应答的同时并行地发送媒体数据，邀请方可能希望在应答到达之前渲染媒体。

```
EXAMPLE 12
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 联播示例

客户端可能希望向服务端发动多个RTP编码（联播）。

```
EXAMPLE 13
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

此示例展示如何创建`RTCDataChannel`对象并执行将通道连接到其他对等端所需的邀请/应答交换。`RTCDataChannel`用于简单聊天应用程序的上下文中，并且当通道准备就绪，接收到消息以及通道关闭时，监听器都会连接到监视器。

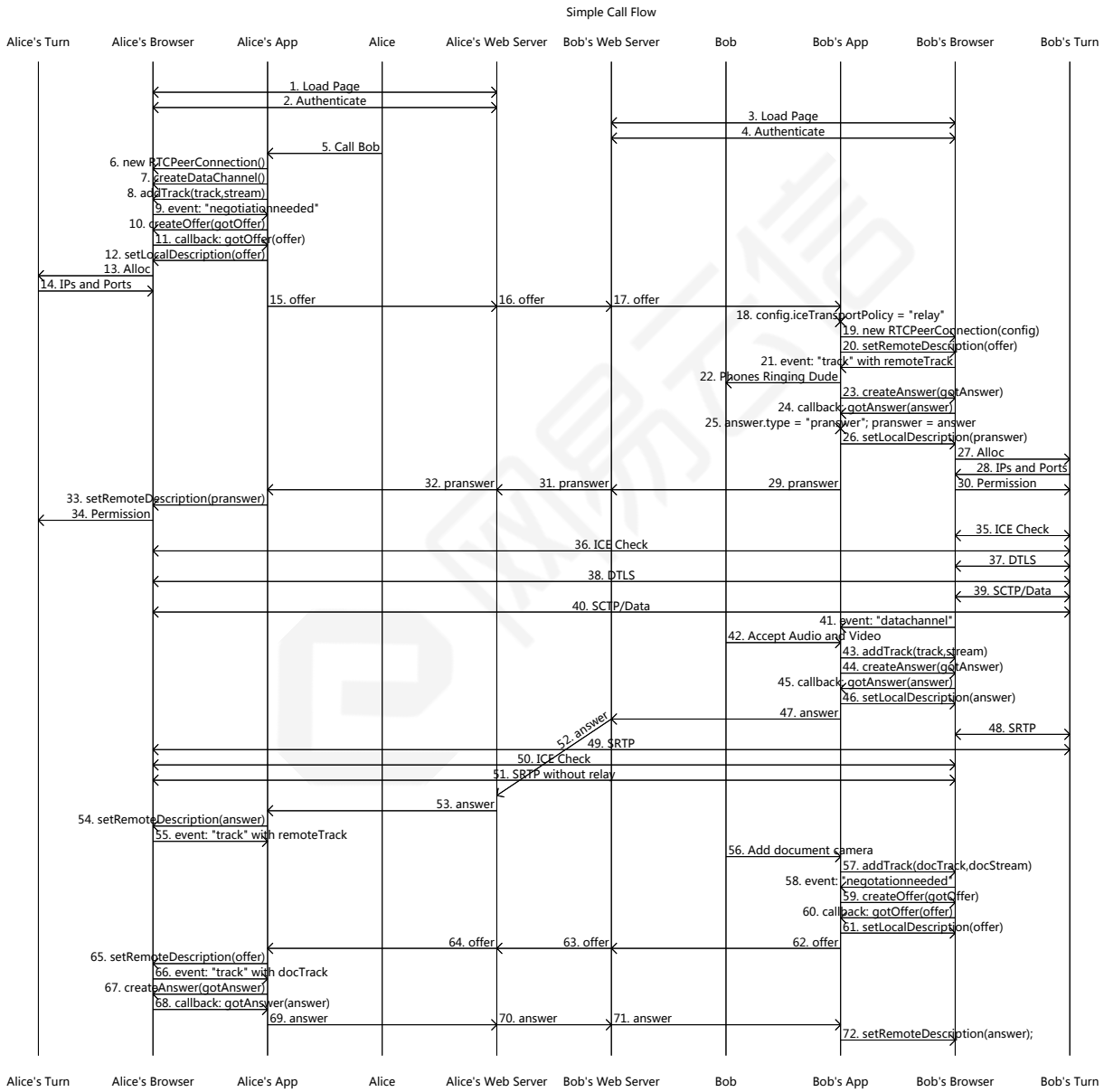
```
EXAMPLE 14
const signaling = new SignalingChannel(); // handles JSON.stringify/parse
const configuration = {iceServers: [{urls: 'stuns:stun.example.org'}]};
let pc;
let channel;
```

```
// 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 浏览器间的调用流程

这展示了两个浏览器之间一个可能发生的呼叫流程示例。它并不展示访问本地媒体或每个回调被触发的过程，而是尝试将其减少为仅显示关键事件和消息。



10.7 DTMF示例

示例假设发送端是一个`RTCRtpSender`。 每个音调每隔500ms发送一次DTMF信号"1234"。

```
EXAMPLE 15
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"之后终止。

```
EXAMPLE 16
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"，并在播放音调时利用`lightKey(key)`点亮活动键（假设`lightKey("")`会将所有键熄灭）。

```
EXAMPLE 17
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');
}
```

追加到音调缓冲区始终是安全的。此示例在所有音调播放开始之前以及播放期间追加。

```
EXAMPLE 18
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');
}
```

发送一个一秒的"1"音调并紧跟着一个两秒的"2"音调。

```
EXAMPLE 19
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

本节及其小节本节扩展了ECMASCRIPT-6.0中定义的 `Error` 子类列表，它们遵循该规范第19.5.6节中的 `NativeError` 模式。作假设如下：

- [Something]和 %something% 的语法使用遵循ECMASCRIPT-6.0。
- ECMAScript标准内置对象（[ECMASCRIPT-6.0]，第17节）的规则在本节中有效。
- 新的内部对象 `%RTCError%` 和 `%RTCErrorPrototype%` 已被包含在（[ECMASCRIPT-6.0]，表7）和所有引用部分中，例如（[ECMASCRIPT-6.0]，第8.2.2节），因此它们可用且行为正确。

11.1 ECMAScript 6术语

本节中使用的以下术语在[ECMASCRIPT-6.0]中定义。

Term/Notation	Section in [ECMASCRIPT-6.0]
Type(X)	6
intrinsic object	6.1.7.4
[[ErrorData]]	19.5.1
internal slot	6.1.7.2
NewTarget	various uses, but no definition
active function object	8.3
OrdinaryCreateFromConstructor()	9.1.14
ReturnIfAbrupt()	6.2.2.4
Assert	5.2
String	4.3.17-19, depending on context
PropertyDescriptor	6.2.4
[[Value]]	6.1.7.1
[[Writable]]	6.1.7.1
[[Enumerable]]	6.1.7.1
[[Configurable]]	6.1.7.1
DefinePropertyOrThrow()	7.3.7
abrupt completion	6.2.2
ToString()	7.1.12
[[Prototype]]	9.1
%Error%	19.5.1
Error	19.5
%ErrorPrototype%	19.5.3
Object.prototype.toString	19.1.3.6

11.2 RTCError对象

11.2.1 RTCError构造函数

RTCError构造函数是`%RTCError%`内部对象。当`RTCError`作为函数而不是构造函数被调用时，它会创建并初始化一个新的`RTCError`对象。将对象作为函数调用等通于调用具有相同参数的构造函数。因此，函数调用`RTCError()...)`等效于具有相同参数的对象创建表达式`new RTCError(...)`。 `RTCError`构造函数被设计为可继承。它可以被用作类定义中`extends`子句的值。计划继承指定`RTCError`行为的子类构造函数必须包含对`RTCError`

构造函数的`super`父类调用，以使用[ErrorData]内部槽创建并初始化子类实例。

11.2.1.1 `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"
};
```

枚举值描述：

- data-channel-failure：数据通道已失败。
- dtls-failure：DTLS协商失败或连接因为某个严重错误被终止了。`message`包含了与错误性质有关的信息。如果收到严重的DTLS警报，则`receivedAlert`属性将被设为收到的DTLS警报值。如果发送了致命的DTLS警报，则`sentAlert`属性将被设为发送的DTLS警报值。
- fingerprint-failure：`RTCDtlsTransport`的远程证书与SDP中提供的所有指纹都不匹配。如果远程对端无法将本地证书与提供的指纹匹配，不会生成此错误。相反，可能会从远程对等方接收到`"bad_certificate"`（42）DTLS警报，从而导致`"dtls-failure"`。
- idp-bad-script-failure：从身份提供程序加载的脚本不是有效的JavaScript代码，或没有实现正确的接口。
- idp-execution-failure：身份提供程序抛出一个异常或返回了一个被拒绝的promise。
- idp-load-failure：加载IDP URI失败。`httpRequestStatusCode`属性被设为响应中的HTTP状态码。
- idp-need-login：身份提供程序需要用户登陆。`idpLoginUrl`属性被设为用于登录的URL。
- idp-timeout：IDP定时器已过期。
- idp-tls-failure：用于IDP HTTPS连接的TLS证书不可信。
- idp-token-expired：IDP令牌已过期。
- idp-token-invalid：IDP令牌非法。
- sctp-failure：SCTP协商失败或连接因为某个严重错误被终止了。`sctpCauseCode`属性被设为SCTP错误码。
- sdp-syntax-error：SDP语法非法。`sdpLineNumber`属性被设为SDP中检测出语法错误的行号。
- hardware-encoder-not-available：请求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。

11.2.1.2 RTCError(errorDetail, message)

当`RTCError`函数被 *errorDetail* 和 *message* 参数调用时，以下步骤会被采取：

1. 若NewTarget **未定义** 时，设 *newTarget* 为活跃函数对象，否则为NewTarget。
2. 设 *O* 为`OrdinaryCreateFromConstructor(newTarget, "%RTCErrorPrototype%", «[ErrorData]»)`。

3. 调用ReturnIfAbrupt(O)。
4. 若 *errorDetail* 非 **未定义** ，则：
    1. 设 *errorDetail* 为 `PropertyDescriptor{[[Value]]: errorDetail, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}.`
    2. 设 *cStatus* 为 `DefinePropertyOrThrow(O, "errorDetail", errorDetailDesc).`
    3. 断言： *eStatus* 表示正常完成，过程没有被打断。
5. 若 *message* 非 **未定义** ，则：
    1. 设 *msg* 为 `ToString(message).`
    2. 设 *msgDesc* 为 `PropertyDescriptor{[[Value]]: msg, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}.`
    3. 设 *mStatus* 为 `DefinePropertyOrThrow(O, "message", msgDesc).`
    4. 断言：*mStatus* 表示正常完成，过程没有被打断。
6. 返回 *O* 。

11.2.2 RTCError构造函数的属性

`RTCError` 构造函数的[Prototype]槽的值是内部对象 `%Error%` 。 除了 `length` 属性（其值为 `1` ），`RTCError` 构造函数还有以下属性：

11.2.2.1 RTCError.prototype

`RTCError.prototype` 的初始值是 `RTCError` 原型对象。此属性具有属性 `{[[Writable]]：false，[[Enumerable]]：false，[[Configurable]]：false}` 。

11.2.3 RTCError原型对象属性

`RTCError` 原型对象是普通对象。它不是 `Error` 实例，也没有[ErrorData]内部槽。 `RTCError` 原型对象的[Prototype]内部槽的值是内部对象 `%ErrorPrototype%` 。

11.2.3.1 RTCError.prototype.constructor

`RTCError` 构造函数的原型的 `constructor` 属性的初始值是内部对象 `%RTCError%` 。

11.2.3.2 RTCError.prototype.errorDetail

`RTCError` 构造函数的原型的 `errorDetail` 属性的初始值是空字符串。

11.2.3.3 RTCError.prototype.sdpLineNumber

`RTCError` 构造函数的原型的 `sdpLineNumber` 属性的初始值是0。

11.2.3.4 RTCError.prototype.httpRequestStatusCode

`RTCError` 构造函数的原型的 `httpRequestStatusCode` 属性的初始值是0。

11.2.3.5 RTCError.prototype.sctpCauseCode

`RTCError` 构造函数的原型的 `sctpCauseCode` 属性的初始值是0。

11.2.3.6 RTCError.prototype.receivedAlert

一个无符号整数，表示收到的DTLS警报值。 `RTCError` 构造函数的原型的 `receivedAlert` 属性的初始值为 `null` 。

11.2.3.7 RTCError.prototype.sentAlert

一个无符号整数，表示发送的DTLS警报值。`RTCError`构造函数的原型的`sentAlert`属性的初始值为`null`。

11.2.3.8 RTCError.prototype.message

`RTCError`构造函数的原型的`message`属性的初始值是空字符串。

11.2.3.9 RTCError.prototype.name

`RTCError`构造函数的原型的`name`属性的初始值是`"RTCError"`。

11.2.4 RTCError实例的属性

`RTCError`实例是从`RTCError`原型对象继承属性并具有[ErrorData]内部槽的普通对象，槽值 **未定义** 。[ErrorData]的唯一指定用途是通过`Object.prototype.toString`（ECMASCRIPT-6.0，第19.1.3.6节）识别`Error`的实例或其各种子类。

`RTCErrorEvent`接口为各种将`RTCError`作为事件抛出的场景而定义：

```
[Exposed=Window,
 Constructor(DOMString type, RTCErrorEventInit eventInitDict)]
interface RTCErrorEvent : Event {
    readonly attribute RTCError? error;
};
```

构造函数：

- RTCErrorEvent：构造一个新`RTCErrorEvent`。

属性：

- RTCError类型的`error`，只读，可空：描述触发事件的错误的描述（如果存在的话）。

```
dictionary RTCErrorEventInit : EventInit {
    RTCError? error = null;
};
```

`RTCErrorEventInit`字典成员：

- RTCError类型的`error`，可空，缺省值为`null`：与事件相关联的错误的描述（如果存在的话）。

12. 事件摘要

以下事件在`RTCDataChannel`对象上触发。

Event name	Interface	Fired when
open	Event	当`RTCDataChannel`对象的底层数据传输已建立（或重新建立）
message	MessageEventwebmessaging	消息已被成功接收
bufferedamountlow	Event	`RTCDataChannel`对象的`bufferedamount`从高于`bufferedAmountLowThreashold`减少到小于等于`bufferedAmountLowThreashold`
error	RTCErrorEvent	数据通道中发生了一个错误
close	Event	`RTCDataChannel`的底层数据传输已被关闭

以下事件在`RTCPeerConnection`对象上触发。

Event name	Interface	Fired when
track	RTCTrackEvent	已为特定的`RTCRtpReceiver`协商了新的传入媒体，并且该接收端的`track`已添加到所有与之关联的远程`MediaStream`中。
negotiationneeded	Event	浏览器希望通知应用程序需要完成会话协商（即`createOffer`调用后跟`setLocalDescription`）。
signalingstatechange	Event	信令状态已改变状态。状态的改变原因是调用了`setLocalDescription`或`setRemoteDescription`。
iceconnectionstatechange	Event	`RTCPeerConnection`的ICE连接状态已改变。
icegatheringstatechange	Event	`RTCPeerConnection`的ICE收集状态已改变。
icecandidate	RTCPeerConnectionIceEvent	新的`RTCIceCandidate`已对脚本可见。
connectionstatechange	Event	`RTCPeerConnection`的`connectionState`属性已改变。
icecandidateerror	RTCPeerConnectionIceErrorEvent	在收集ICE候选项阶段发生错误
datachannel	RTCDataChannelEvent	为了响应对端创建通道的请求，新的`RTCDataChannel`被调度到脚本中。
isolationchange	Event	当`MediaStreamTrack`上的`isolated`属性改变时，新的`Event`被调度到脚本中。
stateended	RTCStatsEvent	为了响应一个或多个受监控对象被同时删除，新的`Event`被调度到脚本中。

以下事件在`RTCDTMFSender`对象上触发：

Event name	Interface	Fired when
tonechange	RTCDTMFToneChangeEvent	`RTCDTMFSender`对象刚刚开始播放音调（返回`tone`属性）或刚刚结束播放`toneBuffer`中的音调（返回空的`tone`属性）。

以下事件在`RTCIceTransport`对象上触发：

Event name	Interface	Fired when
statechange	Event	`RTCIceTransport` 状态改变。
gatheringstatechange	Event	`RTCIceTransport` 收集状态改变。
selectedcandidatepairchange	Event	`RTCIceTransport` 选中的候选项对改变。

以下事件在 `RTCDtlsTransport` 对象上触发：

Event name	Interface	Fired when
statechange	Event	`RTCDtlsTransport` 状态改变。
error	RTCErrorEvent	在 `RTCDtlsTransport` 上发生错误（"dtls-error"或"fingerprint-failure"）。

以下事件在 `RTCSctpTransport` 对象上触发：

Event name	Interface	Fired when
statechange	Event	`RTCSctpTransport` 状态改变。

13. 隐私与安全考量

本节并非规范;它没有指定新的行为，而是总结了规范其他部分已有的内容。WebRTC中使用的一般API和协议集的整体安全性考量在RTCWEB-SECURITY-ARCH中有描述。

13.1 对同源策略的影响

本文档拓展了能够在浏览器和其他设备（包括其他浏览器）之间建立实时的直接通信Web平台。 这意味着数据和媒体可以在运行在不同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非浏览器的应用程序之间共享，这是Web模型中常见屏障的扩展，用于在具有不同来源的实体之间发送数据。 WebRTC规范不提供用户提示或Chrome指示符进行通信;它假设一旦允许网页访问媒体数据，就可以自由地与其他实体共享该媒体。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视角的WebRTC数据通道对等交换，类似于服务器介导的交换（例如，通过WebSockets）可以在没有用户参与的情况下发生。 `peerIdentity` 机制从充当身份提供者的第三方服务器加载并执行JavaScript代码。该代码在单独的JavaScript域中执行，不会影响相同原策略提供的防护。

13.2 泄露IP地址

即使没有WebRTC，提供Web应用程序的Web服务器也知道应用程序将要到达的公网IP地址。设置通信会向Web应用程序公开有关浏览器网络上下文的其他信息，甚至可能包括浏览器用于WebRTC通信的一组（可能是私有的）IP地址。其中一些信息必须传递给相应方才能建立通信会话。 泄露IP地址可能会泄漏位置和连接方式，这是很敏感的。根据网络环境，它还可以增加指纹表面并创建持久且跨源的用户无法轻易清除的状态。 连接将始终显示建议用于与对应端通信的IP地址。应用程序可以通过一些方法限制这种情况，比如使用 `RTCIceTransportPolicy` 字典的设置选择避开一些特定地址，以及使用中继连接（例如TURN服务器）代替通信参与者之间的直接连接。通常我们假设TURN服务器的IP地址不是敏感信息。 这些选择可以由应用程序做出，例如基于用户是否已经表示同意开始与另一方进行媒体连接。 暂缓向应用程序暴露IP地址本身需要限制可用的IP地址，这将影响端之间的最直接路径上进行通信的能力。

浏览器应该根据用户的安全需求，提供适当的控制来决定哪些IP地址可供应用程序使用。本地策略控制公开哪些地址（详见RTCWEB-IP-HANDLING）。

13.3 对本地网络的影响

由于浏览器是在可信网络环境（防火墙内）中执行的活动平台，因此限制浏览器对本地网络上其他元素可以造成的损害非常重要，保护数据免受不受信任的参与者拦截，操纵和修改非常重要。 缓解措施包括：

- 用户代理将始终请求对端用户代理的许可使用ICE进行通信。这可以确保用户代理只能发送给您具有共享凭据的合作伙伴。
- 用户代理将始终请求持续的许可，以继续使用ICE发送持续的同意。这使得接收端可以撤回接收数据的同意请求。
- 用户代理将始终对数据进行加密，并且每次会话都有独立的密钥（DTLS-SRTP）。
- 用户代理将始终使用拥塞控制。这保证WebRTC不会造成网络拥塞。

这些措施在相关的IETF文件中有详细说明。

13.4 通信保密性

通信内容实际上不能向具备观测网络能力的攻击方隐藏，因此必须将其视为公开信息。 现有的 `peerIdentity` 机制为Javascript提供了请求相同javascript无法访问媒体的选项，但只能发送给某些其他实体。

13.5 WebRTC公开的持久性信息

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地址外，WebRTC API还通过 `RTCRtpSender.getCapabilities` 和 `RTCRtpReceiver.getCapabilities` 方法公开有关底层媒体系统的信息，包括系统能够生成和使用的编解码器的详细且有序的信息。该信息的子集可在会话协商期间生成，公开和传输的SDP会话描述中表示。在大多数情况下，该信息在不同时间和不同源上都是持久的，并且增加了给定设备的指纹表面。 如果设置了默认ICE服务器，则它们可由 `RTCPeerConnection` 实例上的 `getDefaultIceServers` 方法公开，并提供持久的跨时间和跨源信息，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化的证书（例如，在IndexedDB中）。这些证书不在原数据库之间共享，但在原始数据库清除持久存储时会被清除。

A. 致谢

编辑们感谢工作组主席和团队联系人Harald Alvestrand，StefanHåkansson，Erik Lagerway和DominiqueHazaël-Massieux的支持。许多人提供了本规范中的大量文本，包括Martin Thomson，Harald Alvestrand，Justin Uberti，Eric Rescorla，Peter Thatcher，Jan-Ivar Bruaroey和Peter Saint-Andre。Dan Burnett感谢Voxeo和Aspect在本规范制定过程中给予的大力支持。 RTCRtpSender和RTCRtpReceiver对象最初在W3C ORTC CG中定义，先已被此规范采用。

B. 参考文献

B.1 规范性参考文献

[BUNDLE] Negotiating Media Multiplexing Using the Session Description Protocol (SDP). C. Holmberg; H. Alvestrand; C. Jennings. IETF. 31 August 2017. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-mmusic-sdp-bundle-negotiation

[DOM] DOM Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://dom.spec.whatwg.org/

[ECMASCRIPT-6.0] ECMA-262 6th Edition, The ECMAScript 2015 Language Specification. Allen Wirfs-Brock. Ecma International. June 2015. Standard. URL: http://www.ecma-international.org/ecma-262/6.0/index.html

[fetch] Fetch Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://fetch.spec.whatwg.org/

[FILEAPI] File API. Marijn Kruisselbrink; Arun Ranganathan. W3C. 6 November 2018. W3C Working Draft. URL: https://www.w3.org/TR/FileAPI/

[FIPS-180-4] FIPS PUB 180-4 Secure Hash Standard. U.S. Department of Commerce/National Institute of Standards and Technology. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[GETUSERMEDIA] Media Capture and Streams. Daniel Burnett; Adam Bergkvist; Cullen Jennings; Anant Narayanan; Bernard Aboba. W3C. 3 October 2017. W3C Candidate Recommendation. URL: https://www.w3.org/TR/mediacapture-streams/

[HIGHRES-TIME] High Resolution Time Level 2. Ilya Grigorik; James Simonsen; Jatinder Mann. W3C. 1 March 2018. W3C Candidate Recommendation. URL: https://www.w3.org/TR/hr-time-2/

[HTML] HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[HTML51] HTML 5.1 2nd Edition. Steve Faulkner; Arron Eicholz; Travis Leithead; Alex Danilo. W3C. 3 October 2017. W3C Recommendation. URL: https://www.w3.org/TR/html51/

[IANA-HASH-FUNCTION] Hash Function Textual Names. IANA. URL: https://www.iana.org/assignments/hash-function-text-names/hash-function-text-names.xml

[ICE] Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. J. Rosenberg. IETF. April 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc5245

[INFRA] Infra Standard. Anne van Kesteren; Domenic Denicola. WHATWG. Living Standard. URL: https://infra.spec.whatwg.org/

[JSEP] Javascript Session Establishment Protocol. Justin Uberti; Cullen Jennings; Eric Rescorla. IETF. 10 October 2017. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-jsep/

[RFC2119] Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC3550] RTP: A Transport Protocol for Real-Time Applications. H. Schulzrinne; S. Casner; R. Frederick; V. Jacobson. IETF. July 2003. Internet Standard. URL: https://tools.ietf.org/html/rfc3550

[RFC3986] Uniform Resource Identifier (URI): Generic Syntax. T. Berners-Lee; R. Fielding; L. Masinter. IETF. January 2005. Internet Standard. URL: https://tools.ietf.org/html/rfc3986

[RFC4566] SDP: Session Description Protocol. M. Handley; V. Jacobson; C. Perkins. IETF. July 2006. Proposed Standard. URL: https://tools.ietf.org/html/rfc4566

[RFC4572] Connection-Oriented Media Transport over the Transport Layer Security (TLS) Protocol in the Session Description Protocol (SDP). J. Lennox. IETF. July 2006. Proposed Standard. URL: https://tools.ietf.org/html/rfc4572

[RFC5389] Session Traversal Utilities for NAT (STUN). J. Rosenberg; R. Mahy; P. Matthews; D. Wing. IETF. October 2008. Proposed Standard. URL: https://tools.ietf.org/html/rfc5389

[RFC5761] Multiplexing RTP Data and Control Packets on a Single Port. C. Perkins; M. Westerlund. IETF. April 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc5761

[RFC5888] The Session Description Protocol (SDP) Grouping Framework. G. Camarillo; H. Schulzrinne. IETF. June 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc5888

[RFC6236] Negotiation of Generic Image Attributes in the Session Description Protocol (SDP). I. Johansson; K. Jung. IETF. May 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6236

[RFC6464] A Real-time Transport Protocol (RTP) Header Extension for Client-to-Mixer Audio Level Indication. J. Lennox, Ed.; E. Ivov; E. Marocco. IETF. December 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6464

[RFC6465] A Real-time Transport Protocol (RTP) Header Extension for Mixer-to-Client Audio Level Indication. E. Ivov, Ed.; E. Marocco, Ed.; J. Lennox. IETF. December 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6465

[RFC6544] TCP Candidates with Interactive Connectivity Establishment (ICE). J. Rosenberg; A. Keranen; B. B. Lowekamp; A. B. Roach. IETF. March 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6544

[RFC6749] The OAuth 2.0 Authorization Framework. D. Hardt, Ed.. IETF. October 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6749

[RFC7064] URI Scheme for the Session Traversal Utilities for NAT (STUN) Protocol. S. Nandakumar; G. Salgueiro; P. Jones; M. Petit-Huguenin. IETF. November 2013. Proposed Standard. URL: https://tools.ietf.org/html/rfc7064

[RFC7065] Traversal Using Relays around NAT (TURN) Uniform Resource Identifiers. M. Petit-Huguenin; S. Nandakumar; G. Salgueiro; P. Jones. IETF. November 2013. Proposed Standard. URL: https://tools.ietf.org/html/rfc7065

[RFC7515] JSON Web Signature (JWS). M. Jones; J. Bradley; N. Sakimura. IETF. May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7515

[RFC7635] Session Traversal Utilities for NAT (STUN) Extension for Third-Party Authorization. T. Reddy; P. Patil; R. Ravindranath; J. Uberti. IETF. August 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7635

[RFC7656] A Taxonomy of Semantics and Mechanisms for Real-Time Transport Protocol (RTP) Sources. J. Lennox; K. Gross; S. Nandakumar; G. Salgueiro; B. Burman, Ed.. IETF. November 2015. Informational. URL: https://tools.ietf.org/html/rfc7656

[RFC7675] Session Traversal Utilities for NAT (STUN) Usage for Consent Freshness. M. Perumal; D. Wing; R. Ravindranath; T. Reddy; M. Thomson. IETF. October 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7675

[RTCWEB-AUDIO] WebRTC Audio Codec and Processing Requirements. JM. Valin; C. Bran. IETF. May 2016. Proposed Standard. URL: https://tools.ietf.org/html/rfc7874

[RTCWEB-DATA] RTCWeb Data Channels. R. Jesup; S. Loreto; M. Tuexen. IETF. 14 October 2015. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-data-channel

[RTCWEB-DATA-PROTOCOL] RTCWeb Data Channel Protocol. R. Jesup; S. Loreto; M. Tuexen. IETF. 14 October 2015. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-data-protocol

[RTCWEB-RTP] Web Real-Time Communication (WebRTC): Media Transport and Use of RTP. C. Perkins; M. Westerlund; J. Ott. IETF. 17 March 2016. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage

[RTCWEB-TRANSPORT] Transports for RTCWEB. H. Alvestrand. IETF. 31 October 2016. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-transports

[SCTP-SDP] Session Description Protocol (SDP) Offer/Answer Procedures For Stream Control Transmission Protocol (SCTP) over Datagram Transport Layer Security (DTLS) Transport. C. Holmberg; R. Shpount; S. Loreto; G. Camarillo. IETF. 20 March 2017. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-mmusic-sctp-sdp

[SDP] An Offer/Answer Model with Session Description Protocol (SDP). J. Rosenberg; H. Schulzrinne. IETF. June 2002. Proposed Standard. URL: https://tools.ietf.org/html/rfc3264

[STUN-BIS] Session Traversal Utilities for NAT (STUN). M. Petit-Huguenin; G. Salgueiro; J. Rosenberg; D. Wing; R. Mahy; P. Matthews. IETF. 16 February 2017. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-tram-stunbis

[TRICKLE-ICE] Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol. E. Ivov; E. Rescorla; J. Uberti. IETF. 20 July 2015. Internet Draft (work in progress). URL: http://datatracker.ietf.org/doc/draft-ietf-mmusic-trickle-ice

[TSVWG-RTCWEB-QOS] DSCP Packet Markings for WebRTC QoS. S. Dhesikan; C. Jennings; D. Druta; P. Jones; J. Polk. IETF. 22 August 2016. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-tsvwg-rtcweb-qos

[WebCryptoAPI] Web Cryptography API. Mark Watson. W3C. 26 January 2017. W3C Recommendation. URL: https://www.w3.org/TR/WebCryptoAPI/

[WEBIDL] Web IDL. Cameron McCormack; Boris Zbarsky; Tobie Langel. W3C. 15 December 2016. W3C Editor's Draft. URL: https://heycam.github.io/webidl/

[WEBIDL-1] WebIDL Level 1. Cameron McCormack. W3C. 15 December 2016. W3C Recommendation. URL: https://www.w3.org/TR/2016/REC-WebIDL-1-20161215/

[webmessaging] HTML5 Web Messaging. Ian Hickson. W3C. 19 May 2015. W3C Recommendation. URL: https://www.w3.org/TR/webmessaging/

[WEBRTC-IDENTITY] Identity for WebRTC 1.0. Adam Bergkvist; Daniel Burnett; Cullen Jennings; Anant Narayanan; Bernard Aboba; Taylor Brandstetter. W3C. W3C Candidate Recommendation. URL: https://w3c.github.io/webrtc-identity/identity.html

[WEBRTC-STATS] Identifiers for WebRTC's Statistics API. Harald Alvestrand; Varun Singh. W3C. 3 July 2018. W3C Candidate Recommendation. URL: https://www.w3.org/TR/webrtc-stats/

[X509V3] ITU-T Recommendation X.509 version 3 (1997). "Information Technology - Open Systems Interconnection - The Directory Authentication Framework" ISO/IEC 9594-8:1997. ITU.

[X690] Recommendation X.690 — Information Technology — ASN.1 Encoding Rules — Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). ITU. URL: https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf

B.2 非规范性参考文献

[API-DESIGN-PRINCIPLES] API Design Principles. Domenic Denicola.29 December 2015. URL: https://w3ctag.github.io/design-principles/

[IANA-RTP-2] RTP Payload Format media types. IANA. URL: https://www.iana.org/assignments/rtp-parameters/rtp-parameters.xhtml#rtp-parameters-2

[INDEXEDDB] Indexed Database API. Nikunj Mehta; Jonas Sicking; Eliot Graff; Andrei Popescu; Jeremy Orlow; Joshua Bell. W3C. 8 January 2015. W3C Recommendation. URL: https://www.w3.org/TR/IndexedDB/

[OAUTH-POP-KEY-DISTRIBUTION] OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution. J. Bradley; P. Hunt; M. Jones; H. Tschofenig. IETF. 5 March 2015. Internet Draft (work in progress). URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-pop-key-distribution/

[RFC3890] A Transport Independent Bandwidth Modifier for the Session Description Protocol (SDP). M. Westerlund. IETF. September 2004. Proposed Standard. URL: https://tools.ietf.org/html/rfc3890

[RFC5285] A General Mechanism for RTP Header Extensions. D. Singer; H. Desineni. IETF. July 2008. Proposed Standard. URL: https://tools.ietf.org/html/rfc5285

[RFC5506] Support for Reduced-Size Real-Time Transport Control Protocol (RTCP): Opportunities and Consequences. I. Johansson; M. Westerlund. IETF. April 2009. Proposed Standard. URL: https://tools.ietf.org/html/rfc5506

[RTCWEB-IP-HANDLING] WebRTC IP Address Handling Recommendations. Guo-wei Shieh; Justin Uberti. IETF. 20 March 2016. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-ip-handling

[RTCWEB-OVERVIEW] Overview: Real Time Protocols for Brower-based Applications. H. Alvestrand. IETF. 14 February 2014. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-overview

[RTCWEB-SECURITY] Security Considerations for WebRTC. Eric Rescorla. IETF. 22 January 2014. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-security

[RTCWEB-SECURITY-ARCH] WebRTC Security Architecture. Eric Rescorla. IETF. 10 December 2016. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch

[STUN-PARAMETERS] STUN Error Codes. IETF. IANA. April 2011. IANA Parameter Assignment. URL: https://www.iana.org/assignments/stun-parameters/stun-parameters.xhtml#stun-parameters-6

[WEBSOCKETS-API] The WebSocket API. Ian Hickson. W3C. 20 September 2012. W3C Candidate Recommendation. URL: https://www.w3.org/TR/websockets/

[XMLHttpRequest] XMLHttpRequest Level 1. Anne van Kesteren; Julian Aubourg; Jungkee Song; Hallvord Steen et al. W3C. 6 October 2016. W3C Note. URL: https://www.w3.org/TR/XMLHttpRequest/

网易云信以稳定的技术和全方位的服务赢得客户信赖

60W+

60W+**开发者接入**

7.5亿+

SDK**覆盖用户超过7.5亿**

196个

覆盖全球196个国家

网易云信

网易云信是集网易19年IM以及音视频技术打造的PaaS服务产品，稳定易用且功能全面，致力于提供全球领先的技术能力和场景化解决方案。
开发者通过集成客户端SDK和云端OPEN API，即可快速实现IM、音视频通话、直播、点播、互动白板、短信等功能。
点此立即体验DEMO。
如需了解更多信息，欢迎访问： https://yunxin.163.com/
联系我们：4009-000-123

网易云信Netease YunXin
助您轻松实现IM、音视频能力！

扫码关注公众号
获取更多技术干货