

Broadview  
www.broadview.com.cn

[PACKT]  
PUBLISHING



Learning WebRTC

# Learning WebRTC 中文版

用WebRTC开发交互实时通信应用

[美] Dan Ristic 著

宋晓薇 王雅琼 丁坚 刘振涛 译



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>



由 扫描全能王 扫描创建



01677647

## Learning WebRTC

# Learning WebRTC 中文版

[美] Dan Ristic 著

宋晓薇 王雅琼 丁坚 刘振涛 译



电子工业出版社.

Publishing House of Electronics Industry

北京·BEIJING



由 扫描全能王 扫描创建

## 内 容 简 介

WebRTC 是一个支持网络浏览器进行实时语音对话或视频对话的软件架构。本书使用形象的案例介绍，逐步深入地阐述了 WebRTC 的幕后工作原理。通过阅读本书，读者可以快速、有效地掌握创建一个 WebRTC 应用所必需的知识，包括获取用户设备信息、创建 WebRTC 应用的客户端和服务器、连接用户并发送数据、文件共享、数据信息安全和性能优化。本书适合有一定 HTML 和 JavaScript 经验，希望了解 WebRTC 并想学习实时通信工作原理的开发者参考阅读。

Copyright © 2015 Packt Publishing. First published in the English language under the title ‘learning WebRTC’.

本书简体中文版专有版权由 Packt Publishing 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2015-6648

### 图书在版编目（CIP）数据

Learning WebRTC 中文版 / (美) 丹·里斯蒂克 (Dan Ristic) 著；宋晓薇等译. —北京：电子工业出版社，2016.7

ISBN 978-7-121-28817-3

I. ①L… II. ①丹… ②宋… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字(2016)第 111861 号

策划编辑：张春雨

责任编辑：张春雨

印 刷：北京中新伟业印刷有限公司

装 订：北京中新伟业印刷有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：10.5 字数：240.8 千字

版 次：2016 年 7 月第 1 版

印 次：2016 年 7 月第 1 次印刷

定 价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，  
联系及邮购电话：(010) 88254888, 88258888。

质量投诉请发邮件至 [zlts@phei.com.cn](mailto:zlts@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式：010-51260888-819 [faq@phei.com.cn](mailto:faq@phei.com.cn)。



由 扫描全能王 扫描创建

WebRTC 是谷歌的一个网页间实时通信的开源技术，它无须插件，就可以实现浏览器之间的交流功能，这意味着，用户无须安装任何东西，就可以在浏览器里面进行实时的声音和视频通话。当然，它不仅可以嵌入实时的音频和视频，同时还有文件共享等功能。在没有 WebRTC 之前，如果你想要在浏览器内进行通话或者共享文件，通常需要安装一个非常大的程序或者 Flash。而有了 WebRTC，用户直接单击已经开发好的网页应用，无须其他操作，即可进行视频会议。

当然，WebRTC 是一个非常年轻的技术，还有很多成长的空间，它的崛起势必会给互联网带来巨大的冲击。目前来看，Chrome 和 Firefox 浏览器已经对它有了很好的支持，而像 IE 等浏览器，还需要安装一些开源的插件来支持 WebRTC。

本书作者通过对网络视频原理的讲解，结合一些实例，非常详细地介绍了 WebRTC 在网页应用中的每一个细节。相信通过阅读本书，你将可以创建一个非常好用的 WebRTC 应用。

特别感谢寸志的引荐让我们得以参与本书的翻译，以及在整个翻译过程中提供的指导和帮助。同时也感谢这段时间团队 4 人之间的相互帮助，我们力求能正确传达作者的意思，但纰漏在所难免，存在的不足敬请广大读者批评指正。



# 关于作者

Dan Ristic

他是一位前端工程师，同时也是一名开放网络（Open Web）布道师。他致力于研究前沿技术，并以独到的创见推进 Web 领域的发展。他曾在位于美国亚利桑那州的先进技术大学（University of Advancing Technology）研习工程知识，对 Web 领域满怀热情，离开后便一直从事 Web 应用开发相关的工作。

他目前在旧金山地区生活工作，在索尼国际娱乐网络公司担任高级软件工程师一职，负责管理前端应用架构，为数百万用户提供 PlayStation 应用商店服务。业余时间里，他喜欢徒步旅行、探险、开发项目或出席一些活动。

感谢我的家人，无论我做什么事情，你们始终支持并鼓励着我。

同样感谢我的朋友们，在与你们的交流中我获益匪浅，让这本书更加充实可靠。



由 扫描全能王 扫描创建

# 关于审校者

Roy Binux 是一位软件工程师，同时也是一名开源软件开发者。在实际工作中，他致力于研究爬虫系统和信息提取技术，也会出于兴趣构建一些有趣的在线程序。他乐于尝试新技术，努力优化着从 Web 平台获取知识的过程，并借助网络的力量开发一些有用的工具。他的大部分作品已开源，你可以在 <http://github.com/binux> 上找到相关信息。

Tsahi Levent-Levi 是一位独立的 WebRTC 分析师及顾问。

他在电信、VoIP 及 3G 通信产业等领域有着 15 年的经验，在这期间他担任过工程师、经理、市场营销及首席技术官等职位。他也是一名企业家、独立分析师及顾问，协助其他公司将其电信领域技术转化为相应的商业战略。

他拥有计算机科学硕士学位和工商管理硕士学位，擅长创业及战略制定，拥有三个 3G-324M 和 VoIP 相关的专利，同时还是 IMTC（国际多媒体通信协会）多个活动小组的主席。

访问他的博客 <https://bloggeek.me>，获取有关 WebRTC 生态系统和商业机会的相关信息。

Andrii Sergienko 是一位热爱信息技术和旅行的企业家，尤其喜欢汽车旅行。他曾在多地居住过，如乌克兰、俄罗斯、白俄罗斯、蒙古国、布里亚特、西伯利亚等，每到一处都会定居多年。

自儿时起，他便对计算机程序和硬件产生了浓厚的兴趣，而在大约 20 年前，正式踏入这个领域。在过往的工作经历中，他尝试使用过包括 C、C++、Java、Assembler、Erlang、JavaScript、PHP、Riak、Shell 脚本、计算机网络、安全在内的多种语言和技术。

在他的职业生涯中，既加入过像国内互联网服务提供商这样的小公司，也加入过诸如惠普这样大型的跨国公司。他也曾创办过几家自己的公司，成功过，也失败过。



现在，他致力于 Oslikas 的成长，这家新公司也是他一手创办的，总部位于爱沙尼亚，专注于现代 IT 技术及解决方案，他们开发了一套用于创建富媒体 WebRTC 应用和服务的全栈框架。访问 <http://www.oslikas.com> 浏览更多有关 Oslikas 的信息。

## 青林审于关

我特别感谢王青林。首先感谢他的耐心和细心，帮助我将这本书从头到尾都仔细地校对了一遍，没有遗漏任何一处错误。感谢他对我在编写过程中遇到的各种问题的耐心解答，以及对书中各种技术细节的深入分析。感谢他在我完成书稿后，对书中所有内容进行仔细的校对和修改，确保了本书的质量和准确性。

我特别感谢王青林对书中所有技术细节的深入分析，以及对书中所有代码示例的细心校对。感谢他对我在编写过程中遇到的各种问题的耐心解答，以及对书中所有内容进行仔细的校对和修改，确保了本书的质量和准确性。

我特别感谢王青林对书中所有技术细节的深入分析，以及对书中所有代码示例的细心校对。感谢他对我在编写过程中遇到的各种问题的耐心解答，以及对书中所有内容进行仔细的校对和修改，确保了本书的质量和准确性。

我特别感谢王青林对书中所有技术细节的深入分析，以及对书中所有代码示例的细心校对。感谢他对我在编写过程中遇到的各种问题的耐心解答，以及对书中所有内容进行仔细的校对和修改，确保了本书的质量和准确性。

我特别感谢王青林对书中所有技术细节的深入分析，以及对书中所有代码示例的细心校对。感谢他对我在编写过程中遇到的各种问题的耐心解答，以及对书中所有内容进行仔细的校对和修改，确保了本书的质量和准确性。

我特别感谢王青林对书中所有技术细节的深入分析，以及对书中所有代码示例的细心校对。感谢他对我在编写过程中遇到的各种问题的耐心解答，以及对书中所有内容进行仔细的校对和修改，确保了本书的质量和准确性。

我特别感谢王青林对书中所有技术细节的深入分析，以及对书中所有代码示例的细心校对。感谢他对我在编写过程中遇到的各种问题的耐心解答，以及对书中所有内容进行仔细的校对和修改，确保了本书的质量和准确性。

我特别感谢王青林对书中所有技术细节的深入分析，以及对书中所有代码示例的细心校对。感谢他对我在编写过程中遇到的各种问题的耐心解答，以及对书中所有内容进行仔细的校对和修改，确保了本书的质量和准确性。



# 目录

前言 ..... XI

1 开启 WebRTC 之旅 ..... 1

    1.1 音视频通信领域的发展现状 ..... 1

    1.2 在 web 平台传输音频和视频 ..... 2

    1.3 捕捉摄像头和麦克风 ..... 3

    1.4 音频及视频的编解码 ..... 3

    1.5 传输层 ..... 4

    1.6 会话 (Session) 管理 ..... 4

    1.7 创建 web 标准 ..... 5

    1.8 浏览器支持 ..... 5

        1.8.1 Chrome、Firefox 和 Opera 的兼容性 ..... 5

        1.8.2 安卓操作系统的兼容性 ..... 6

        1.8.3 苹果操作系统兼容性 ..... 6

        1.8.4 IE 的兼容性 ..... 6

    1.9 在浏览器中使用 WebRTC ..... 6

    1.10 支持 WebRTC 的应用 ..... 7

    1.11 自测题 ..... 7

    1.12 小结 ..... 8

2 获取用户媒体 ..... 9

    2.1 访问媒体设备 ..... 9

        2.1.1 配置静态服务器 ..... 10

        2.1.2 创建我们的首个媒体流页面 ..... 11



由 扫描全能王 扫描创建

限制媒体流 .....	14
限制视频捕捉.....	15
多设备处理 .....	18
创建一个拍照室应用.....	20
修改媒体流 .....	23
自测题 .....	25
小结 .....	26
<b>3 创建简单的 WebRTC 应用 .....</b>	<b>27</b>
理解 UDP 传输协议和实时传输 .....	27
WebRTC API .....	30
RTCPeerConnection 对象.....	30
信号传递和交涉.....	31
会话描述协议 (SDP) .....	33
找到一条清晰的路线到其他用户.....	35
创建一个基本的 WebRTC 应用.....	38
创建一个 RTCPeerConnection.....	38
建立 SDP OFFER 和返回 .....	43
寻找 ICE 候选路径 .....	44
加入流和打磨.....	46
运行你的第一个 WebRTC 应用.....	47
自测题 .....	51
小结 .....	52
<b>4 创建信令服务器 .....</b>	<b>53</b>
构建信令服务器 .....	53
搭建开发环境 .....	53
获得一个连接 .....	55
测试我们的服务器.....	56
识别用户 .....	58
发起通话 .....	59
	62



呼叫应答 .....	63
处理 ICE 候选路径 .....	64
呼叫挂断 .....	65
完成信令服务器 .....	66
在实际应用中发送信令 .....	70
WebSockets 的困境 .....	70
连接其他服务 .....	71
自测题 .....	72
小结 .....	73
<b>5 把客户端连接到一起 .....</b>	<b>75</b>
客户端应用 .....	75
创建页面 .....	77
获取一个连接 .....	79
登录到应用程序 .....	81
开始一个对等连接 .....	82
发起通话 .....	85
检测通信 .....	87
挂断电话 .....	88
一个完整的 WebRTC 客户端 .....	89
改进应用程序 .....	95
自测题 .....	95
小结 .....	96
<b>6 使用 WebRTC 发送数据 .....</b>	<b>97</b>
流控制传输协议和数据传输 .....	97
RTCDATAChannel 对象 .....	99
数据通道选项 .....	101
发送数据 .....	101
加密与安全 .....	102
添加文字聊天 .....	103



---

用例 .....	107
自测题 .....	108
小结 .....	109
<b>7 文件共享 .....</b>	<b>111</b>
使用文件 API 拾取文件 .....	112
准备我们的页面 .....	114
获取对文件的引用 .....	121
文件分块 .....	122
使文件分块可读 .....	124
文件读取与发送 .....	126
在“另一端”组合文件块 .....	128
向用户展示进度 .....	130
自测题 .....	131
小结 .....	132
<b>8 高安全性与大规模优化 .....</b>	<b>133</b>
保护信令服务器 .....	133
使用编码 .....	134
使用 OAuth 提供器 .....	134
支持移动设备 .....	136
网格网络简介 .....	138
网格类型 .....	139
网格网络的缺陷 .....	143
更多用户的视频会议 .....	144
视频会议的未来 .....	146
自测题 .....	146
小结 .....	147
<b>附录 B 白测题答案 .....</b>	<b>149</b>



由 扫描全能王 扫描创建

# 前言

第一次写 HTML 代码的时候，我非常兴奋。我在键盘上输入的每一个字符都会作为指令告诉计算机该怎么处理。我很高兴我能创造出一些东西并用这些指令表达我的创造力。当我保存我的劳动成果，打开浏览器然后显示页面，这一切让我心生敬畏。当看到自己的名字以大号粗体字展现在一个叫作 GIF 的动态图片上时，心里像是有一团窜动的火焰。

web 由来已久，事实上它不仅是一个创造新事物的地方，也是展示和分享创意给他人的平台。这个表达创意的强大平台驱动着 web 以更快的速度发展。由于 web 越来越普及，我们整个生活都依赖于它。网站支持着你的邮件、娱乐、银行账号、法律文件和税收，甚至连这本书的部分撰写也是使用了 web 工具。人们想把生活搬到 web 上的需求促进了 API 的快速有力发展，例如 WebRTC。

WebRTC 是 web 平台的最实质的加法指令之一。它带来了一整套的新技术，如摄像头、流动数据，甚至整个网络的协议栈。令人吃惊的是，不仅看到有大量的工作进入 Web RTC API，而且任何应用开发者使用它都是完全免费的。

WebRTC 的目的是实现实时通信大众化。早些时候，创建一个较小的视频通信应用需要花费几个月时间，即使是做一个很小的应用程序都需要一些定制工程。但是，现在只要花费一半甚至更少的时间。同时它也把开源社区带入实时通信领域。你可以查看全世界其他的 WebRTC 例子，通过搜索一些源代码看看这些应用是怎么实现的。

正是这些创造性的表达方式和网络带来的自由促成了本书的出世。我非常高兴能有机会通过我的作品把这些东西带给大家，我希望激励其他人像我当初一样，同样由于被激励做了第一个网页。写这本书是我工作中最重要也是最困难的事之一，但我非常感激在这过程中所学习到的东西。

如果你正在寻找能创建一个新的最简单的实时通信的经验，并想分享给他人，可以读这本书。这本书像其他技术书籍一样，只是引导人们在 web 上创造一些更好的东西的途径。你不仅能学会怎样使用 WebRTC，也能够做到举一反三。这本书不仅是一个学习工具，而



由 扫描全能王 扫描创建

且也是对有激情地创造一些事情的激励。

## 这本书包含了什么

第 1 章，开启 WebRTC 之旅，囊括了 WebRTC 怎么以网页为基础，来实现音频和视频的通信。可以在你的浏览器上运行一个 WebRTC 应用。

第 2 章，获得 User Media，介绍了创建通信应用程序的第一步、网络摄像头和麦克风的获取。这一章还包括如何使用媒体和流 API 捕捉你的信息，我们也开始开发建设基础通信的例子。

第 3 章，创建简单的 WebRTC 应用，包括了对第一个 WebRTC-RTC 对等连接的介绍。同时看到 WebRTC 里面的复杂结构以及开始使用 API 后我们可以期待些什么，来奠定创建一个 WebRTC 应用程序的基础。

第 4 章，创建信令服务器，包括创建我们自己的信令服务器，来帮助客户在互联网上找到彼此的步骤。这也包括在 WebRTC 上信令是如何工作，以及如何运用到我们的示例应用程序中去的深层信息。

第 5 章，把客户端连接到一起，包括我们的信令服务器的实际使用，也包括使用 WebRTC API、媒体捕捉和在前面章节用来创建我们示例应用的信令服务器，来成功连接两个用户。

第 6 章，使用 WebRTC 发送数据，包括了 RTCDATAChannel 介绍，以及它是如何用于两个对等体之间的原始数据发送的。这一章通过为客户添加文本聊天来详细说明我们的示例。

第 7 章，文件共享，通过观察如何在两个对等体之间的共享文件，阐述了发送原始数据的概念。这将展示音频和视频共享以外的 WebRTC 的许多用途。

第 8 章，高安全性与大规模优化，包括高级的主题，比如如何提供一个大规模的 WebRTC 应用程序。我们还关注安全和其他公司使用的性能优化。

附录，自测题答案，包括每一章最后自测题的答案。



## 你需要准备什么

本书中所有的例子都是建立在 web 标准上的。由于 WebRTC 规范是相当新的，建议运行示例时使用更新过的浏览器，优先使用最新的 Firefox 和 Chrome 浏览器。

所有的服务器代码均使用 Node.js 编写。Node.js 框架大多数在 Windows、Linux 和 Mac OSX 的机器内运行。

你可以使用任何支持 JavaScript 和 HTML 代码的文本编辑器。

## 这本书是写给谁的

你应该有一些使用 HTML 和 JavaScript 构建 web 应用程序的经验。也许，你现在正在构建应用程序，或有一个利用用户间音频和视频交流力量来构建新的应用程序的想法。你可能还需要通过在用户之间转移高性能数据来实现应用程序。

你应该牢固地掌握编程概念和网络基础，但是这本书是写给新入门的网络工程师的。提到的概念很深入，并且是一点一点地深入，而不是一开始就进入高级话题。你可能完全不知道 WebRTC 或者只听说过一点，而且想学习实时通信的内部工作原理。

本书，主要讲解单线程，微博，即时语音和视频。

## 约定

本书中使用了很多格式的文本，以区分各种不同的信息。这里我们举例说明这些格式，并解释它们的含义。

文本、数据库表名、文件夹名称、文件名、文件扩展名、路径名、伪 URL、用户输入和 Twitter 中的代码，格式是这样的：“一件事需要注意的是，当我们从 theirConnection 得到一个 ICE 候选人，我们将其添加到连接，反之亦然。”

代码块的格式设置如下：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
```



由 扫描全能王 扫描创建

```

<title>Learning WebRTC - Chapter 4: Creating a
    RTCPeerConnection</title>
</head>
<body>
    <div id=""container">
        <video id=""yours"" autoplay></video>
        <video id=""theirs"" autoplay></video>
    </div>
    <script src=""main.js""></script>
</body>
</html>

```

所有命令行输入或输出写为如下：

```

> 1 + 1
2
> var hello = "world";
undefined
> "Hello" + hello;D
'Helloworld'

```

新术语和重要词汇以黑体显示。在屏幕上看到的词，例如，在菜单或对话框上，你这样表示：“现在你可以单击 Capture 按钮，捕获出现在屏幕上的一帧视频。”



警告或重要提示以这种方式表示。



技巧和窍门像这样表示。

## 下载示例代码

你可以从 <http://www.broadview.com.cn> 下载所有已购买的博文视点书籍的示例代码文件。



## 勘误表

虽然我们已经尽力谨慎地确保内容的准确性，但错误仍然存在。如果你发现了书中的错误，包括正文和代码中的错误，请告诉我们，我们会非常感激。这样，你不仅帮助了其他读者，也帮助我们改进后续的出版。如发现任何勘误，可以在博文视点网站相应图书的页面提交勘误信息。一旦你找到的错误被证实，你提交的信息就会被接受，我们的网站也会发布这些勘误信息。你可以随时浏览图书页面，查看已发布的勘误信息。

## 开启 WebRTC 之旅

音频和视频通话正风靡全球。诸如 Netflix 和 Pandora 这样的 web 应用每天都在连接上千万的用户来观看或者收听内容。然而，在如今的 web 平台上尚在形成一套成熟的消息解决方案。对于像 Facebook 这样的网站要在浏览器中加入视频通话的功能，通常会引出许多不同的解决方案。于是，WebRTC（Web 实时通信）技术应运而生了。

在本书中，我们将会探讨 WebRTC 的基础知识，包括：

- 音视频通话领域的发展现状
- WebRTC 对音频领域的影响
- WebRTC 的主要特性及使用方法

## 音视频通话领域的发展现状

人们在很久以前就利用各式各样的技术和工具进行通信，就以座机电话为例，各大电信公司建立起了大规模的音频通信网络，为全球成千上万的企业提供长途通信服务。许多个人的语音和数据传播到世界各地。

后来，视频通话也随之流行起来。基于带宽的限制，最早的视频通话只有 Skype 的视频通话采用了基带的技术，用户通过带宽较低的拨号连接进行视频通话。开发者们为了利用计算机的处理能力，通过大量的计算不断降低带宽需求，进而开发出了各种各样的相向的工程化的解决方案。例如：英特尔、高通、德州仪器、英伟达等公司的解决方案。



由 扫描全能王 扫描创建

# 开启 WebRTC 之旅

音频和视频可以在互联网上传输，诸如 Netflix 和 Pandora 这样的 web 应用每天都向成千上万的用户传输着音视频内容。然而，在如今的 web 平台上尚未形成一套成熟的实时通信解决方案，如果 Facebook 这样的网站要在浏览器中加入视频通信的功能，通常会引导用户安装一个插件来实现。于是，WebRTC（web 实时通信）技术就应运而生了。

在本章中，我们将介绍 WebRTC 的基础知识，包括：

- 音频和视频领域的发展现状
- WebRTC 对音视频领域的影响
- WebRTC 主要特性及使用方法

## 音视频通信领域的发展现状

人们在很久以前就利用各式各样的技术和工具进行通信，就以移动电话运营商为例，大型电话公司建立起大规模的音频通信网络，为全球成千上万的用户提供音频通信服务，让每个人的语音都能够传播到世界各地。

后来，视频通信也随之流行起来。基于苹果的 FaceTime、Google 的 Hangouts 还有 Skype 的视频通话这些工具背后的技术，用户可以很方便地与其他人进行视频对话。开发者们为了将用户体验优化到极致，通过大量的技术手段保障视频质量，针对每一种问题都提出了相应的工程化解决方案，例如：减少丢包、断网恢复、及时响应用户网络变化等。



由 扫描全能王 扫描创建

WebRTC 的目标是将所有这些技术都植入到浏览器中。上述解决方案大多都需要用户在 PC 和移动设备中安装相应的插件或应用程序，这些公司还会向开发者征收技术授权费，并构筑起巨大的技术壁垒以防新公司加入到这个领域中蚕食市场份额。而 WebRTC 技术一旦实现，每一位浏览器用户不再需要安装插件，开发者也不再需要缴纳昂贵的授权费，大家只需要打开特定的网站就可以立即与其他用户建立连接。

## 在 web 平台传输音频和视频

WebRTC 的成就非凡，无须借助第三方软件或插件便可在开放网络中传输高质量音频流，在过去的浏览器中一直没出现过这种免费优质的实时通信解决方案。互联网的成功很大程度上可以归因于 HTML、HTTP 还有 TCP/IP 这些高度开放且高度可用的技术，我们希望基于此来构建 WebRTC，推动互联网的进步。这便是 WebRTC 的由来。

从零开始构建一个实时通信应用，首先需要引入大量的库和框架，这些代码能够解决实际开发中将面临问题，例如：连接断开、数据丢失、NAT 穿透等。WebRTC 的优点是可以在浏览器 API 中内建上述这些库和框架，Google 也开源了许多提供高品质完整通信功能的技术。

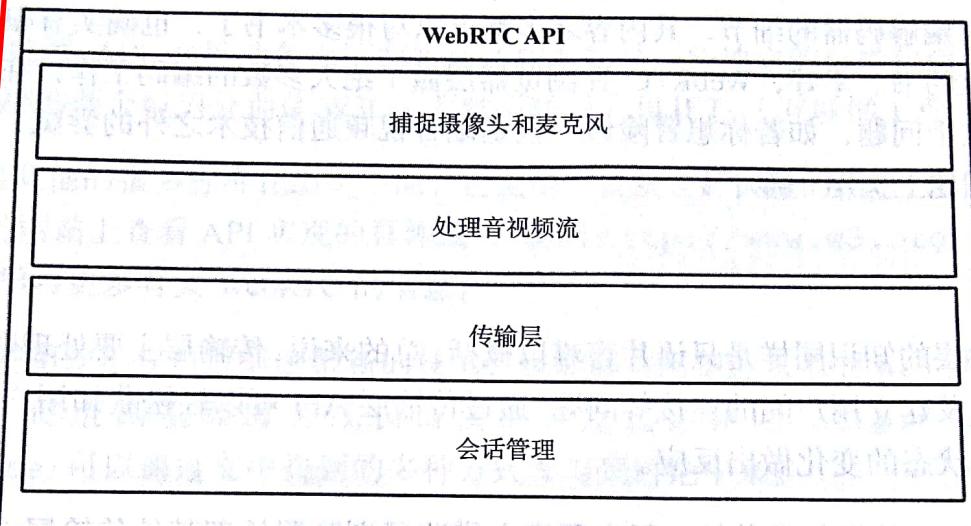


访问 <http://www.webrtc.org> 获取更多有关 WebRTC 的信息和具体实现的源码。

为了使开发者专注于实现具体的产品细节，WebRTC 在浏览器 API 中集成了大量的技术，解决了所有繁重的问题，例如：捕捉摄像头和麦克风、音视频编解码、传输层以及会话（Session）管理。



由 扫描全能王 扫描创建



## 捕捉摄像头和麦克风

建立通信平台的第一步是获取用户设备的摄像头和麦克风权限。先检测设备的可用性，然后获取用户授权并与设备建立连接，最后再从设备获取一段数据流。一切准备就绪后开始实现我们的第一个应用。

## 音频及视频的编解码

很不幸，如果我们要通过互联网发送一段音视频数据，即使优化了网络速度，数据尺寸也还是太大，根本无法处理。所以我们势必要对数据流进行编码，这个过程会将视频帧或音频声波分解成许多很小的数据块，再将它们压缩到更小的尺寸，从而可以让这些数据块在网络间更快速地传输，然后在接收端将它们解压。这项技术背后的算法通常被成为 codec（多媒体数字信号编解码器）。

如果你曾在计算机上遇到过无法播放视频文件的情况，并且深入了解过视频和音频编解码器的复杂原理，便会知晓编码音频和视频流的方法有很多种，每一种各有千秋。此外，有很多公司出于各种商业目的开发维护自己的收费编解码器。



WebRTC 内置的几种编解码器包括：H.264、Opus、iSAC，还有 VP8。当两个浏览器会话时，会综合两位使用者的情况选出最优的编解码器。众多浏览器厂商也会经常会面，讨论选取适合支撑 WebRTC 技术运行的编解码器。访问 <http://www.webrtc.org/faq> 可以了解更多有关不同 codec 的信息。

若要探讨编解码器的细节，其内容之多都可以写很多本书了，也确实有很多专门围绕这个话题写就的书。幸好，WebRTC 在浏览器层做了绝大多数的编码工作，在整本书中我们无须担心这个问题，如若你想冒险做一些基础音视频通信技术之外的尝试，很有可能会在编解码的问题上头疼一阵子了。

## 传输层

关于传输层的知识同样是只语片言难以概括，总的来说，传输层主要处理数据包丢失、数据包排序以及建立用户间的连接等问题。通过传输层 API 可以轻松获知用户网络的波动，并及时对连接状态的变化做出反应。

WebRTC 处理数据包的传输，很大程度上借鉴了浏览器处理其他传输层（例如：AJAX 和 WebSockets）的方式。浏览器提供一组易于访问且结合了事件机制的 API，能够在连接出现问题时触发相应事件。事实上，无论是在移动设备、桌面设备还是其他任何设备中，控制 WebRTC 调用的代码可能有成千上万行，这些代码可以用来处理所有的用例。

## 会话（Session）管理

WebRTC 难题的最后一部分是会话管理，虽然其原理比网络连接管理要简单得多，却仍不失为一个很重要的部分。通常来说我们称为信令（Signaling），负责在浏览器中建立并管理多个连接，我们会在第 4 章创建信令服务器中详细介绍。

数据传输也是这一系列的新特性之一。我们需要通过高质量的数据连接才能在两个客户端之间传输音视频数据，自然也可以通过这个连接来传输其他任意数据。这个功能通过 RTCDATAChannel API 暴露给 JavaScript 层，我们也将后面详加描述。

如今的 WebRTC 需要非常多的构件来协同打造超高质量的实时通信体验，Google、Mozilla、Opera，还有其他很多公司为此投入了大量的时间和精力，请最好的音视频工程师在 web 平台优化体验。WebRTC 中应用了与 VoIP（互联网协议电话）同源的技术，将在未来改变工程师构建实时通信应用的方式。



由 扫描全能王 扫描创建

# 创建 web 标准

Web 平台的优势是发展迅猛，新旧标准日新月异，浏览器可以在用户无感知的情况下自动下载升级程序并自动安装，进一步提升了快速迭代的体验。这使得 web 开发者的工作变得相对轻松了，但也意味着需要开发者时刻关注包括 WebRTC 在内的新技术。

推广浏览器 API 并推动各大厂商实现新技术的是一些独立的标准化组织，其中掌管 WebRTC 标准的两个组织分别是 W3C（万维网联盟）和 IETF（互联网工程任务组）。

W3C 与其他的很多标准化组织不同，它免费向公众公开大量的信息，让每一个人都可以在 W3C 的网站上查看 API 实现的具体细节。访问 <http://www.w3.org/TR/webrtc/> 可以参考和学习更多有关 WebRTC 的信息。

参与这些组织不仅可以紧跟最新的技术，也能够共同塑造未来的 web 世界。参与这些社区有利于促进浏览器成为成长最快的开发栈。访问 <http://www.w3.org/participate/> 可以通过文中提到的多种方式参与到讨论中来。

## 浏览器支持

尽管 WebRTC 的目标是为每一个用户服务，但这不意味着所有浏览器要同时实现相同的功能。不同浏览器可能会选择在特定的领域突进领先，如此一来在某些浏览器里可以运行的程序兴许在其他浏览器中就无法工作。接下来我们会介绍 WebRTC 在不同浏览器中的支持情况。

 你可以在很多网站上查到自己的浏览器对某项技术的支持程度，例如访问 <http://caniuse.com/rtcpeerconnection> 你就会知道自己的浏览器是否支持 WebRTC。

## Chrome、Firefox 和 Opera 的兼容性

你现在使用的浏览器很有可能就支持 WebRTC。在主流的操作系统 Windows、Mac 和 Linux 中，Chrome、Firefox 还有 Opera 这三种浏览器都支持并且默认启用了 WebRTC 功能。浏览器厂商之间，比如 Chrome 和 Firefox，也会一起合作解决互操作性的问题，目前二者间能够实现互相通信。



由 扫描全能王 扫描创建

## 安卓操作系统的兼容性

安卓系统上的 Chrome 和 Firefox 浏览器同样也支持 WebRTC。鉴于这两种浏览器都实现了桌面端和移动端程序的代码共享，所以安卓 4.0 版本（代号为 Ice Cream Sandwich）以后的浏览器都默认启用了 WebRTC，且能够与其他基于 WebRTC 的程序互相通信。

## 苹果操作系统兼容性

苹果没有针对 Safari 和 iOS 提供 WebRTC 的支持。有传言称他们正在研发相关功能，但是目前苹果官方尚未公布支持 WebRTC 的具体时间。有的人为了在 WebView 中加载 WebRTC 应用，直接在混合模式的移动应用（Hybrid App）中嵌入了 WebRTC 的代码。

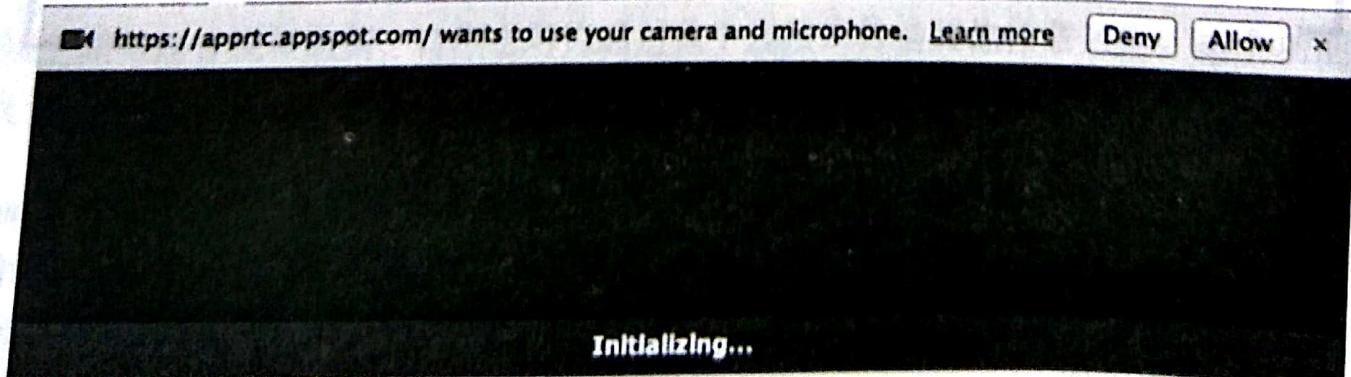
## IE 的兼容性

微软尚未宣布任何在 IE 中添加 WebRTC 支持的计划。他们也曾提出过一个可以在浏览器中支持音视频通信的可选方案，但这个方案在与 WebRTC 标准竞争后并没有得到采纳。自那时起，微软便没有过多参与 WebRTC 标准的研发。

 推荐你使用 Chrome 查看贯穿整本书的示例，切记这是一个日新月异的领域，请及时查看并更新你的浏览器。

## 在浏览器中使用 WebRTC

准备好浏览器环境后，让我们立即开始尝试 WebRTC 吧！在浏览器的地址栏中输入 <https://apprtc.appspot.com/> 访问 demo 应用。如果你使用的是 Chrome、Firefox 或 Opera 中的任意一款浏览器，你都应该看到一个下拉通知：



单击允许按钮，音频和视频就可以传输到这个 web 页面中了，你也许需要设置一下麦克风，或者选择一个可以使用的 web 摄像头。完成这一切后，你应该可以在页面中看到摄像头捕捉的影像。

页面会根据当前的会话生成一个自定义 ID，你可以在页面的 URL 中找到它，例如 <https://apprtc.appspot.com/r/359323927>。复制这个 URL，在本机或别的机器把它粘贴到其他浏览器的地址栏里，页面加载后，如若一切顺利，你会看到分别来自两个客户端的实时视频。WebRTC 是一个强大的解决方案，通过它可以很轻松地在浏览器中实现实时通信的功能。

## 支持 WebRTC 的应用

深入考究，WebRTC 的核心其实就是在两个浏览器之间建立起来的一条点对点连接，WebRTC 是这项技术在浏览器本身首次真正意义上的实现，所以任何通过点对点连接实现的程序都可以轻松地扩展到 WebRTC，例如：文件共享、文本聊天、多人游戏甚至货币流通等。现在有上百个此类应用已经在浏览器中运行起来了。

这些应用大都有一个共同点——需要在两个用户之间建立一条低延迟、高性能的连接。这要求 WebRTC 不得不使用底层协议来提供高速性能，从而加速数据在网络间的流动，实现在短时间内传输大量的数据。

WebRTC 也支持在两个用户之间建立一条安全连接，提供更高级别的隐私保护。流量在对等连接之间完全加密，而且可以直接路由到其他用户，所以在不同的连接上发送的数据包可能经过完全不同的路由，使用 WebRTC 应用的用户由此可以匿名，否则难以保证何时可以连接到一台应用服务器上。

这些只是众多 WebRTC 应用中的一部分，由于 WebRTC 构建在 JavaScript 和 web 平台的基础上，可以从已有的应用中吸取很多经验。阅读完本书后，你应该能够掌握亲自编写一个 WebRTC 应用所需的知识！

## 自测题

Q1：WebRTC 的目标是在不安装插件且不收取许可费的基础上提供易访问的实时通信



由 扫描全能王 扫描创建

功能。对或错？

Q2：下列哪个不是浏览器通过 WebRTC 提供的功能？

1. 捕捉摄像头和麦克风
2. 处理视频和音频流
3. 访问联系人列表
4. 管理会话

Q3：必须是有钱的大公司才可以加入 W3C 和 IETF。对或错？

Q4：下列哪种应用不能通过 WebRTC 实现？

1. 文件共享
2. 视频通信
3. 多人游戏
4. 以上都不是

## 小结

在本章中，我们简要介绍了 WebRTC 背后的特性和技术。你应该已经知晓 WebRTC 的目标、对 Web 平台的促进作用及可应用的场景，而且你应该已经跟随着我们在浏览器中体验了 WebRTC 的威力。

本章蕴含着大量的信息，尽管没有悉数吸收也不要担心，我们将在整本书中更详尽地讲解上述每一个主题。请尽情在网上查阅相关资料，以更好地理解 WebRTC 的方方面面。

接下来，我们将开始通过 `getUserMedia` API 来探究如何捕捉用户的摄像头和麦克风。

然后，我们会在浏览器中构建一个完整的 WebRTC 应用来处理视频和音频电话。

完成之后，我们再一起探讨如何将这个应用扩展成多用户版本，并通过添加文字聊天和文件共享等功能学习数据传输相关的知识，以及使用 WebRTC 时的最佳安全实践。



# 2

## 获取用户媒体

创建一个基于 WebRTC 的通信平台，首先需要通过用户的网络摄像头和麦克风获取实时的视频和音频流。在过去的浏览器中，我们通常用插件来实现这个功能；而现在，我们可以用 JavaScript 调用浏览器的 `getUserMedia` API 来实现。

本章将围绕以下几个话题展开：

- 如何访问媒体设备
- 如何约束媒体流
- 如何处理多种设备
- 如何修改流数据

### 访问媒体设备

在很久之前，开发者们就开始尝试将媒体设备接入浏览器中，他们曾纠结于各种解决方案，有的是基于 Flash 的，而有的基于插件，但这些方案都需要用户在浏览器中安装某些程序才能捕捉到摄像头。为此，W3C 最终决定筹备一个专门的小组来制定相关标准。在最新的浏览器中，你可以通过 JavaScript 访问 `getUserMedia` API，它又被称为 MediaStream API（译者注：目前 `getUserMedia` API 已被废除，请使用 MediaStream API）。

这组 API 有以下几个关键功能：

- 提供一个 `stream` 对象：这个对象用以表示音频或视频形式的实时媒体流。



由 扫描全能王 扫描创建

- 提供设备间切换的功能：当多个摄像头或麦克风连接到计算机上时，可以选择所需设备。
- 提供充分的安全保障：获得用户的访问许可，根据偏好设置从用户的计算机设备捕获数据流。

在我们继续进行之前，需要提前准备好相应的编程环境。首先，要有一个可以编辑 HTML 和 JavaScript 的文本编辑器，凡是购买这本书的读者很可能已经有一个趁手的编辑器了。

其次，你还需要一台服务器来托管 HTML 和 JavaScript 文件并提供伺服服务。浏览器的权限及安全限制要求：必须是通过真实服务器伺服的文件才可以连接用户的摄像头和麦克风。如果你在本地双击打开本书提供的代码，它将不会正常运行。

## 配置静态服务器

开发者们都应该先学会如何配置一台本地的 web 服务器，编程语言多种多样，用不同语言编写的服务器也数不胜数，我个人最喜爱的是 Node.js 的 node-static，这是一个出色的 web 服务器：

1. 访问 Node.js 网站 <http://nodejs.org/>，单击首页那个巨大的 INSTALL 按钮，在操作系统中安装 Node.js。

(译者注：Node.js 现分为两个分支独立发展，建议使用 v4.4.\* LTS 分支)，下载后根据指引操作可将 Node.js 安装到你的操作系统中。)

2. Node.js 的包管理器（npm）会随 Node.js 一同安装到系统中。

3. 打开终端或命令行界面并输入 `npm install -g node-static` (你很可能需要管理员权限)。

4. 选择一个你希望用来提供伺服服务的目录，置入相关的 HTML 文件。

5. 运行 `static` 指令启动一个静态 web 服务器，在浏览器中输入 `http://localhost:8080` 即可访问你的文件。

对于其他的 HTML 文件，你也可以通过类似的方式为其提供伺服服务，也可以将本提供的示例文件放到目录下访问来查看效果。



 尽管除了 node-static 外还有其他很多选择，但由于我们稍后需要使用 npm，所以非常建议你现在就了解它的相关语法。

现在就继续创建我们的第一个项目！

 在开始之前，你应该确保至少有一个摄像头以及麦克风连接到计算机上。大多数计算机都有相关的设置选项，你需要测试摄像头并确保一切可以正常工作！

## 创建我们的首个媒体流页面

我们的首个支持 WebRTC 的页面很简单：在屏幕上展示一个`<video>`元素，请求使用摄像头后在`<video>`元素里实时显示它此刻拍摄到的内容。`video` 是 HTML5 里的一个强大的特性，既可以通过它展示实时的视频流，也能用它回放很多其他的视频源。我们开始先创建一个简单的 HTML 页面，要在 `body` 标签里包含一个 `video` 元素。你可以创建一个名为 `index.html` 的文件，并且输入以下代码：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Learning WebRTC - Chapter 2: Get User Media</title>
  </head>
  <body>
    <video autoplay></video>
    <script src="main.js"></script>
  </body>
</html>
```

 切记 WebRTC 是一个彻头彻尾的 HTML5 特性，你必须使用一个支持 HTML5 标准的新浏览器。在上面的代码中，我们通过 DOCTYPE 标签将浏览器标记为兼容 HTML5 的标准模式。



由 扫描全能王 扫描创建

如果此时打开页面，你会略感失望，你将看到一个空白页面，这是因为我们没有加 main.js 文件导致的。我们来添加一个 main.js 文件，在里面贴入一段 getUserMedia 的代码：

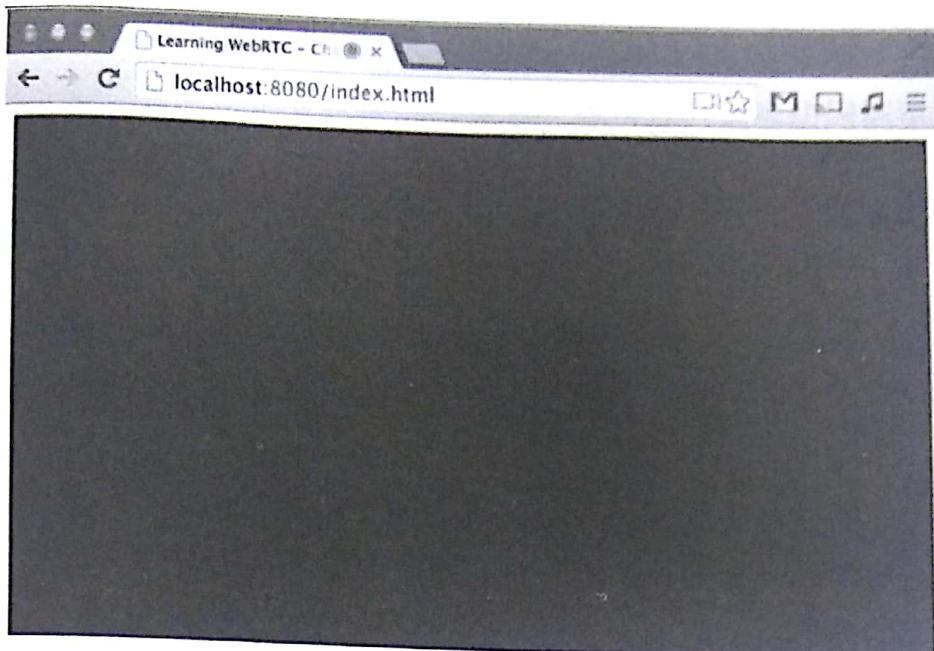
```
function hasUserMedia() {
    return !(navigator.getUserMedia || navigator.webkit GetUserMedia ||
    navigator.mozGetUserMedia || navigator.msGetUserMedia);
}
if (hasUserMedia()) {
    navigator.getUserMedia = navigator.getUserMedia || navigator.webkit GetUserMedia ||
    navigator.mozGetUserMedia || navigator.msGetUserMedia;
    navigator.getUserMedia({
        video: true,
        audio: true
    }, function (stream) {
        var video = document.querySelector('video');
        video.src = window.URL.createObjectURL(stream);
    }, function (err) {});
} else {
    alert("抱歉，你的浏览器不支持 getUserMedia.");
}
```

现在刷新页面，你应该能够看到一切都运行起来了！首先，你会看到一个授权弹框，它与之前运行 WebRTC Demo 时看到的那个类似。如果选择同意，它就会获得摄像头的权限并在页面上的<video>元素里显示它拍到的内容。

各家浏览器总喜欢领先于官方标准实现一些新特性，然后再等待这些特性成为标准。只有正确处理浏览器前缀问题才能让新的浏览器 API 正常运行。他们会创建一些与浏览器名字相似的前缀，例如 Chrome 的 Webkit，Firefox 的 Moz。这样做可以告诉浏览器这些不是标准 API，需要针对它进行特殊处理。然而不幸的是，这也导致了在多个浏览器中需要调用不同的方法来访问同一个 API。我们最终克服了这个困难，做法是：创建一个函数检测这些非标准 API 在当前浏览器中是否可用，如果可用，则将这些 API 全部赋值给一个普通的函数，然后在之后的代码中调用这个函数即可。



由 扫描全能王 扫描创建



接下来，我们要访问 `getUserMedia` 函数。这个函数接受一组参数（来确定浏览器将要做的事情）和一个回调函数，这个回调只接受一个参数：当前计算机上能够产生数据流的媒体设备。

这个对象指向一个浏览器为我们保持的媒体流。它会不断从摄像头和麦克风捕获数据，等待来自 web 应用的指令来操作这些数据。我们稍后会获取屏幕上的`<video>`元素并通过 `window.URL.createObjectURL` 函数将流加载到该元素中，由于`<video>`元素不能接受 JavaScript 作为参数，它只能通过一些字符串来获取视频流，这个函数在获取流对象后会将它转换成一个本地的 URL，这样`<video>`元素就能从这个地址获取流数据了。



请注意，`<video>`元素包含一个 `autoplay` 属性，表示视频流字节处理完成后会自动播放，如果你移除这个属性，数据流接入时不会自动播放。

现在，你已经在构建 WebRTC 应用的路上迈出了第一步！通过页面上的这些代码，你可以看到 `getUserMedia` API 的实际威力，其实，从摄像头获取 `stream` 对象并导入页面上的视频元素这个过程并不简单，仅就这一话题就可以写一整本关于 C 或 C++ 的书！



## 限制媒体流

现在我们了解了从浏览器获取流的方法，马上来学习如何通过 `getUserMedia` API 的第一个参数配置这个流。这个参数接受一个对象，通过其键值可以确定从已连接设备寻觅并处理流的方法。我们要讲的第一个选项是开关视频或音频流：

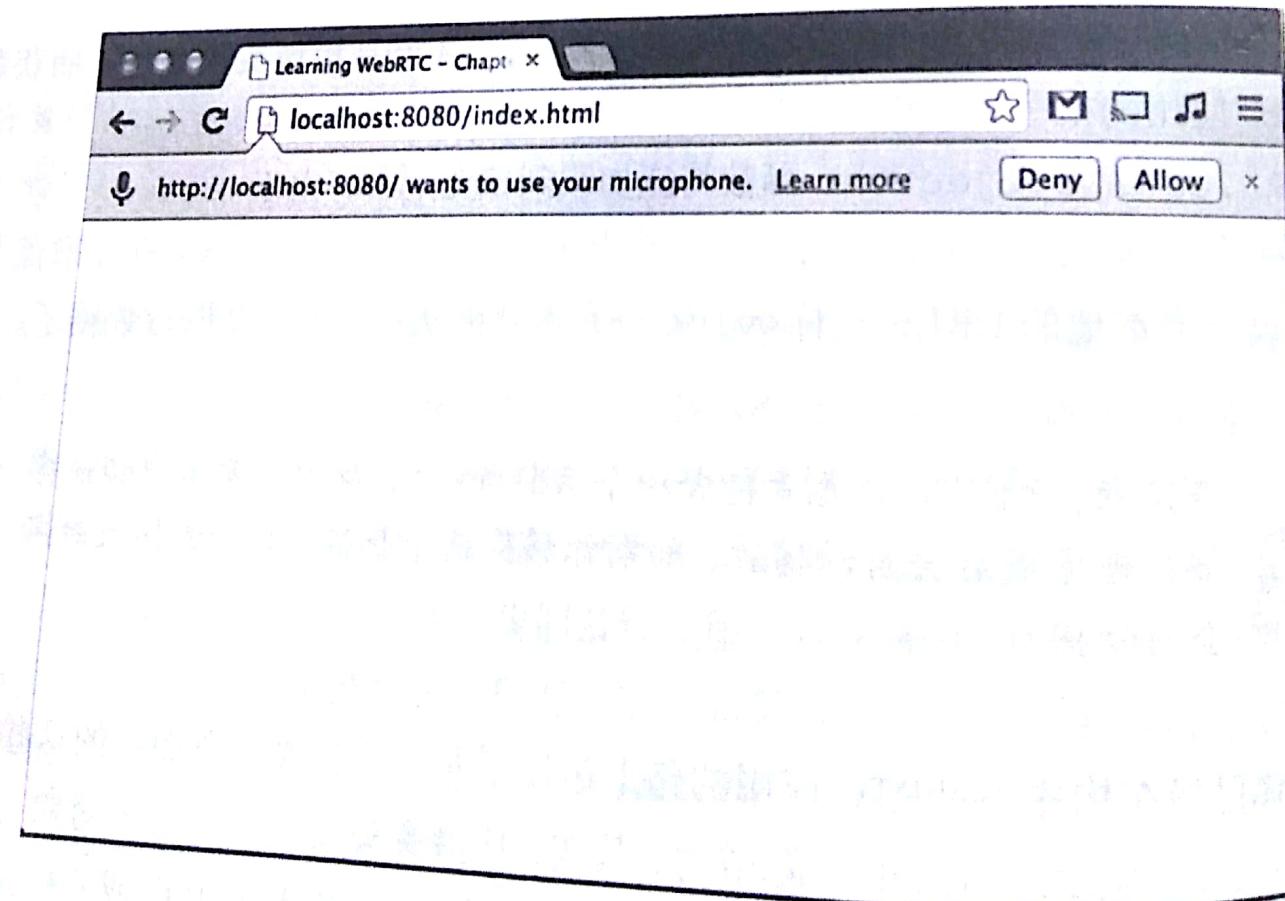
```
navigator.getUserMedia({ video: false, audio: true }, function(stream) {
  // 现在我们的数据流里不包含任何视频!
});
```

在这个示例中，当你将流添加到`<video>`元素时，不会显示来自摄像头的视频；如果对调两个配置，则只有视频没有音频。如果你不想在开发 WebRTC 应用时听一整天自己说的话，那么这是极好的！



通常来说，在开发 WebRTC 应用的时候可能会遇到音频反馈现象，当麦克风捕捉到你的声音再通过扬声器播放出来，会产生一个永无止境的回响。在开发过程中暂时关闭音频有助于解决这个问题。

示例的效果请看下面这张截图，访问 `http://localhost:8080` 的时候展示了一个下拉弹出框，这表明页面需要获取访问麦克风的权限：



由 扫描全能王 扫描创建

如果你想开发一款用来代替普通电话的应用，它理应只支持音频呼叫，这种配置方法会屏蔽视频数据。如果有人不想分享他们的视频，也可以只请求访问浏览器中的麦克风，从而也不会打开摄像头指示灯。

## 限制视频捕捉

你不仅可以通过设置 `true` 或 `false` 值来限制 `getUserMedia` API，也可以传递一个对象进行更复杂的限制。请在 <https://tools.ietf.org/html/draft-alvestrand-constraints-resolution-03> 查找限制方法的相关规范细节，例如：最低分辨率、帧速率、视频宽高比还有一些可选项，这些都可以通过配置对象传递给 `getUserMedia` API。

这可以帮助开发者应对创建 WebRTC 应用时遇到的不同场景。通过这些选项开发者可以根据用户当前场景从浏览器请求更具体的流类型。其中的一些流列出如下：

- 创建良好的用户体验，在每一位参与视频呼叫的用户间选取最小分辨率来请求。
- 保持特定风格或品牌形象，在应用中设置特定的宽和高。
- 在受限的网络连接中限制视频流的分辨率来节省电力或带宽。

举个例子，我们就假设希望将回放视频设置成 16:9 的长宽比，那么在一个像 4:3 这样更小的长宽比环境下，将无法正常导入视频。如果你在调用 `getUserMedia` 时传入以下配置，将会强制改为指定的长宽比：

```
navigator.getUserMedia({
  video: {
    mandatory: {
      minAspectRatio: 1.777,
      maxAspectRatio: 1.778
    },
    optional: [
      { maxWidth: 640 },
      { maxHeight: 480 }
    ]
  },
  audio: false
}, function (stream) {
```



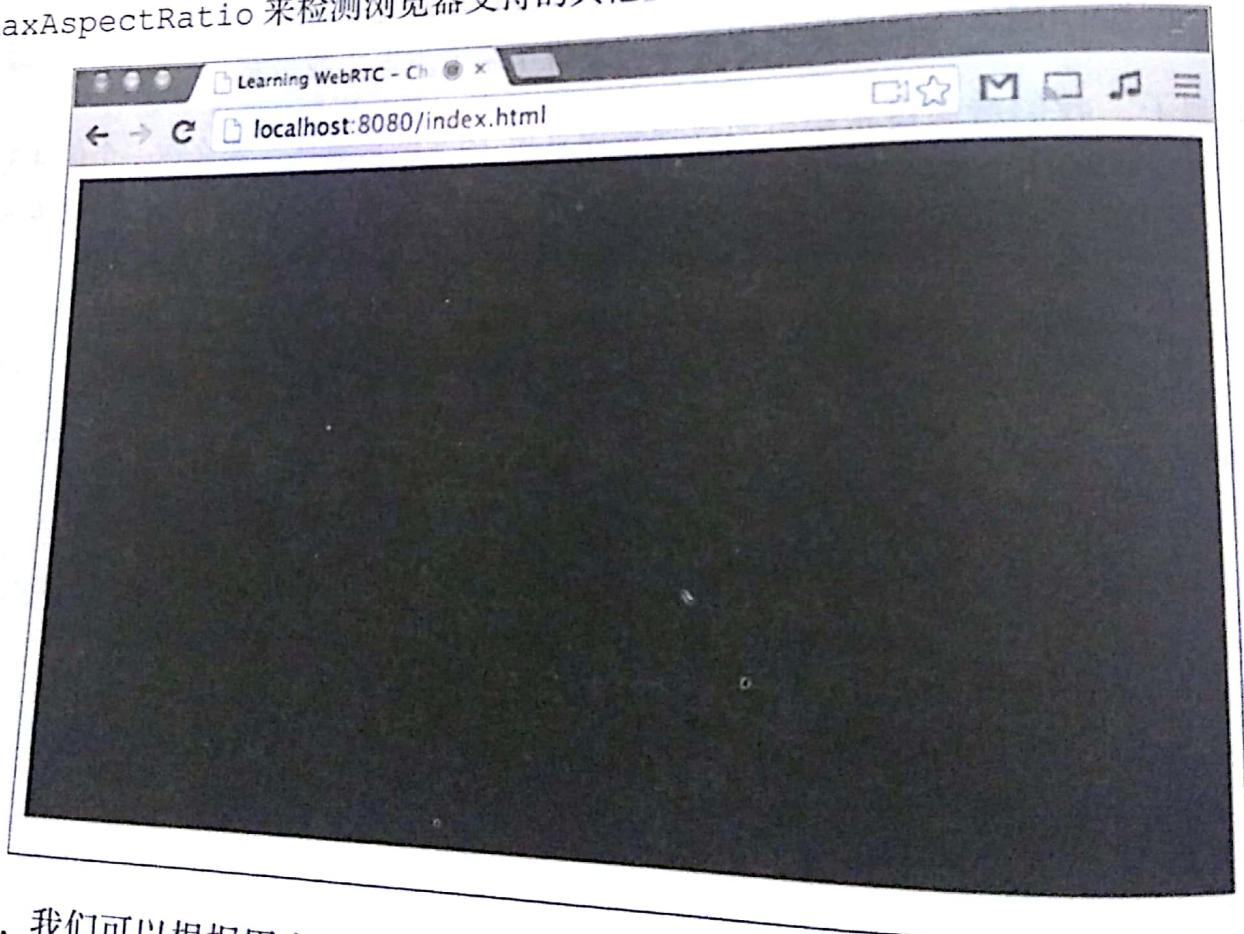
```

var video = document.querySelector('video');
video.src = window.URL.createObjectURL(stream);
}, function (error) {
  console.log("Raised an error when capturing:", error);
});

```

刷新浏览器后并授权页面捕捉摄像头，现在显示的视频比以前更宽。在配置对象的第一部分，我们将长宽比限制为 16:9 或 1.777。在 optional 这一部分，我们将分辨率限制为  $640 \times 480$ 。如果可以，浏览器将根据 optional 代码块的设置尽量尝试平衡这些需求。你所看到的视频分辨率很可能是  $640 \times 360$ ，这是在当前限制下大多数摄像头普遍支持的解决方案。

你应该也注意到我们在调用 `getUserMedia` 时传入了第二个函数，当捕获媒体流时遇到任何问题都会调用这个出错回调函数。在之前的示例中，如果你的摄像头不支持 16:9 分辨率，就会触发这个函数。请时刻注意浏览器中的开发者控制台，看看当错误发生时是否会调用这个出错回调。如果当前项目成功运行，你也可以试着更改 `minAspectRatio` 或 `maxAspectRatio` 来检测浏览器支持的其他参数：



现在，我们可以根据用户的使用环境适配不同情形，提供最好的视频流体验，每个用户的浏览器环境不尽相同，因此这些配置非常有用。如果有很多用户使用你的 WebRTC



用，就必须为每个独立的使用环境提供独立的解决方案。支持移动端设备是数个最大的痛点之一，这些设备的运行资源有限，其屏幕空间也捉襟见肘。如果要节约电量、处理器和带宽，在手机上可以按照  $480 \times 320$  或更小的分辨率来捕获视频。通过对比浏览器的 user agent 字符串与常见的移动 web 浏览器，可以检测用户是否在使用移动设备。将 getUserMedia 调用改成以下这段代码可以实现上述功能：

```

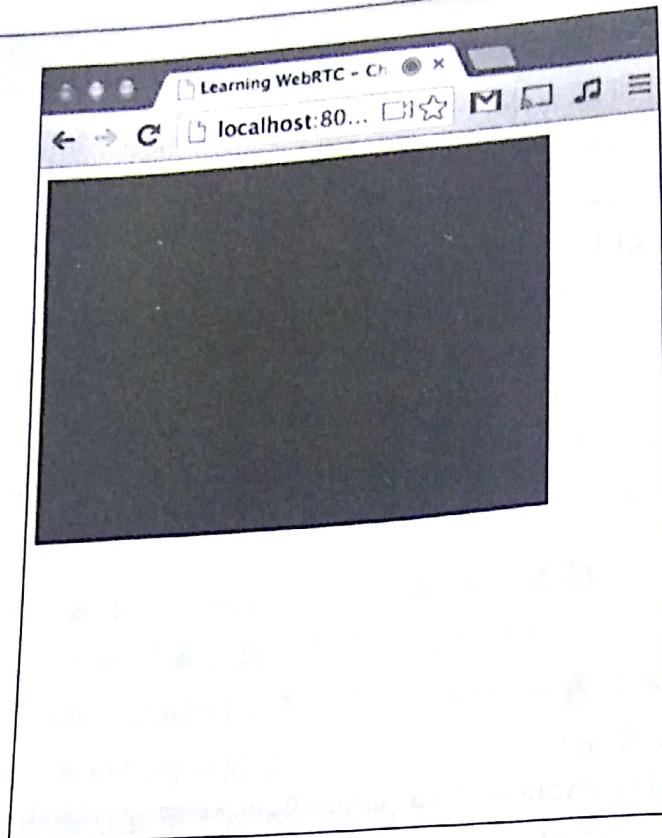
var constraints = {
  video: {
    mandatory: {
      minWidth: 640,
      minHeight: 480
    }
  },
  audio: true
};

if (/Android|webOS|iPhone|iPad|iPod|BlackBerry|IEMobile|Opera
Mini/i.test(navigator.userAgent)) {
  // 用户正在使用移动设备，请降低我们的最小分辨率
  constraints = {
    video: {
      mandatory: {
        minWidth: 480,
        minHeight: 320,
        maxWidth: 1024,
        maxHeight: 768
      }
    },
    audio: true
  };
}

navigator.getUserMedia(constraints, function (stream) {
  var video = document.querySelector('video');
  video.src = window.URL.createObjectURL(stream);
}, function (error) {
  console.log("Raised an error when capturing:", error);
});

```





你必须要重视限制配置的重要性，通过简单的调整就可以有效提升 WebRTC 应用的性能。当通读本章后，你应当考虑应用未来的使用环境及每种环境下最好的解决方案，在第 8 章高安全性与大规模优化中我们将深入探究如何优化 WebRTC 的性能。

## 多设备处理

在某些情况下，用户可能在他们的设备上接驳多台摄像头或麦克风。现在的智能手机基本都有前置和后置两个摄像头，此时你需要查遍所有可用的摄像头和麦克风，然后选择适当的设备来满足用户的需求。好在浏览器暴露出了一个名为 MediaSourceTrack 的 API，可以很方便地管理多个设备。

 在编写本书时，只有最新版本的 Chrome 支持 MediaSourceTrack API，由于很多类似的 API 仍在被创造的过程中，因此不是所有浏览器都支持这些特性。

我们可以通过 MediaSourceTrack 请求设备列表并从中选择我们所需的设备：



由 扫描全能王 扫描创建

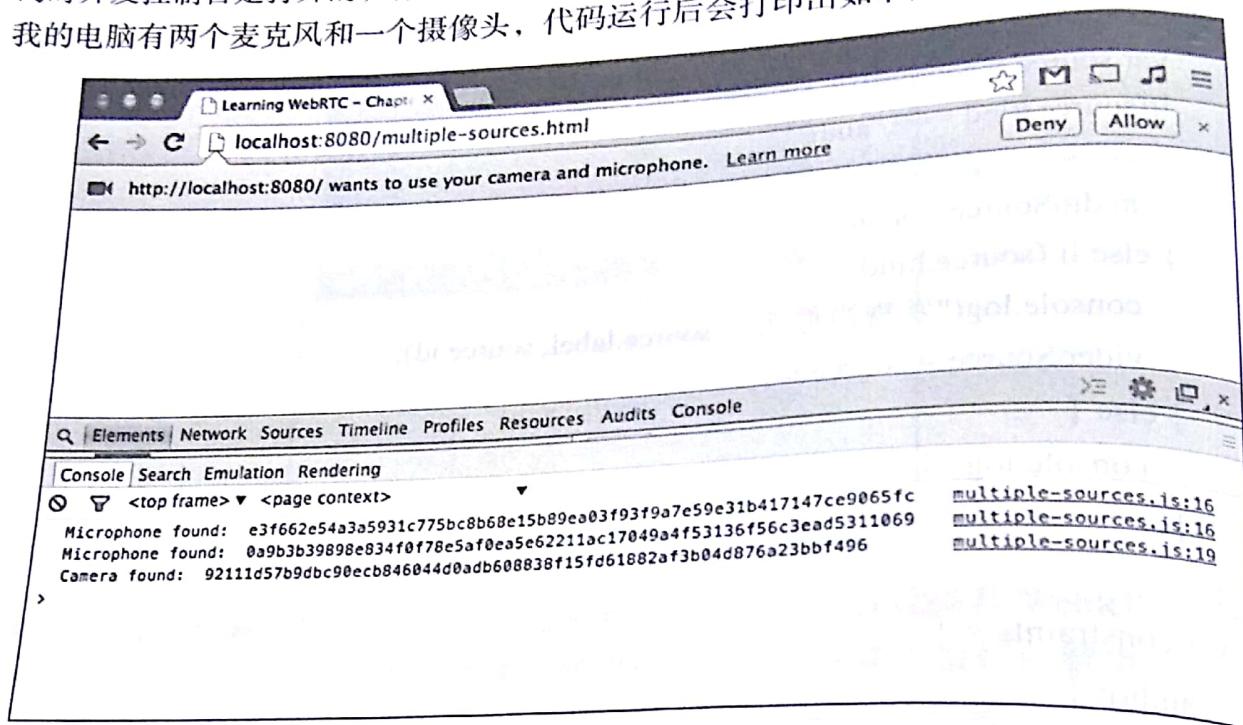
```

MediaStreamTrack.getSources(function(sources) {
    var audioSource = null;
    var videoSource = null;
    for (var i = 0; i < sources.length; ++i) {
        var source = sources[i];
        if(source.kind === "audio") {
            console.log("发现麦克风:", source.label, source.id);
            audioSource = source.id;
        } else if (source.kind === "video") {
            console.log("发现摄像头:", source.label, source.id);
            videoSource = source.id;
        } else {
            console.log("发现未知资源:", source);
        }
    }
    var constraints = {
        audio: {
            optional: [{sourceId: audioSource}]
        },
        video: {
            optional: [{sourceId: videoSource}]
        }
    };
    navigator.getUserMedia(constraints, function (stream) {
        var video = document.querySelector('video');
        video.src = window.URL.createObjectURL(stream);
    }, function (error) {
        console.log("Raised an error when capturing:", error);
    });
});

```



在上面这段代码中，我们调用 `MediaSourceTrack` 对象的 `getSources` 方法返回一个连接到用户机器的设备列表，遍历后可以选一个与你的应用更契合的设备。如果运行代码时开发控制台是打开的，你会发现当前连接到计算机上的设备都被打印出来了。例如，我的电脑有两个麦克风和一个摄像头，代码运行后会打印出如下图所示的内容：



`source` 对象中也包含其他的一些信息，比如有助于设备选择的朝向信息。经过日积累，浏览器或可提供更多的信息，例如：分辨率、帧率（FPS）以及更多设备的信息。记得时常查看 `getUserMedia` 和 `MediaStreamTrack` 这两个 API 的更新信息，你可以了解到各个浏览器中新增的特性。

## 创建一个拍照室应用

Web 平台最棒的地方在于，它的所有子集可以很好地协同运转，通过 `Canvas` 就可以轻松地创建一个复杂的拍照室应用。这个应用能够让你在屏幕上看到自己，也可以随时捕捉自己的照片，就像一个真的拍照室一样。`Canvas` 是一系列在屏幕上绘制线条、图形和图片的 API，因其可以制作游戏并且实现其他交互应用，从而在 web 平台上流行起来。

在这个项目中，我们将使用 `Canvas API` 绘制其中一帧视频到屏幕上。从 `<video>` 元素里获取当前的流，将其转换为一张图片，再把图片绘制到 `<canvas>` 元素中。我们通过



一个简单的 HTML 文件来启动项目：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>Learning WebRTC - Chapter 2: Get User Media</title>
    <style>
        video, canvas {
            border: 1px solid gray;
            width: 480px;
            height: 320px;
        }
    </style>
</head>
<body>
    <video autoplay></video>
    <canvas></canvas>
    <button id="capture">Capture</button>
    <script src="photobooth.js"></script>
</body>
</html>
```

我们需要在页面上添加 canvas 标签, 然后加载 photobooth.js 文件, 我们的 JavaScript 文件里全是逻辑功能代码:

```
function hasUserMedia() {
    return !(navigator.getUserMedia || navigator.webkit GetUserMedia ||
navigator.mozGetUserMedia || navigator.msGetUserMedia);
}

if (hasUserMedia()) {
    navigator.getUserMedia = navigator.getUserMedia ||
navigator.webkit GetUserMedia || navigator.mozGetUserMedia ||
navigator.msGetUserMedia;
    var video = document.querySelector('video'),
        canvas = document.querySelector('canvas'),
        streaming = false;
```



```
navigator.getUserMedia({
  video: true,
  audio: false
}, function (stream) {
  video.src = window.URL.createObjectURL(stream);
  streaming = true;
}, function (error) {
  console.log("Raised an error when capturing:", error);
});

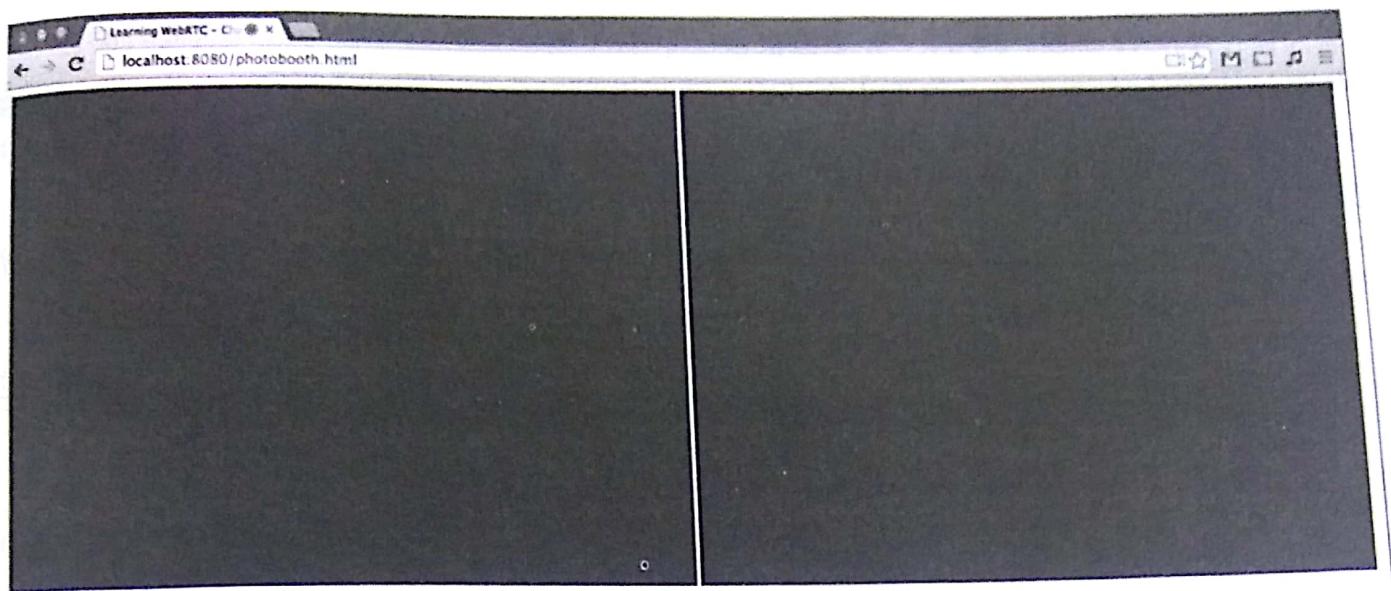
document.querySelector('#capture').addEventListener('click', function
(event) {
  if (streaming) {
    canvas.width = video.clientWidth;
    canvas.height = video.clientHeight;
    var context = canvas.getContext('2d');
    context.drawImage(video, 0, 0);
  }
});
} else {
  alert("对不起，你的浏览器不支持getUserMedia.");
}
```

```

现在，如果单击 Capture 按钮便可捕捉视频并将其中的一帧绘制到 canvas 上。此时的 <canvas> 元素里显示的是一个静止的帧。你也可以一遍又一遍地保存截图来替换 canvas 上的图片。



由 扫描全能王 扫描创建



## 修改媒体流

我们可以继续改进这个项目。如今的大多数图片分享应用总会准备很多滤镜，你可以给图片使用滤镜让它们看起来更酷。在 web 平台上通过 CSS 滤镜就可以提供不同的效果。我们先编写几个 CSS 类，再将这些滤镜应用到<canvas>元素：

```
<style>
  .grayscale {
    -webkit-filter: grayscale(1);
    -moz-filter: grayscale(1);
    -ms-filter: grayscale(1);
    -o-filter: grayscale(1);
    filter: grayscale(1);
  }

  .sepia {
    -webkit-filter: sepia(1);
    -moz-filter: sepia(1);
    -ms-filter: sepia(1);
    -o-filter: sepia(1);
    filter: sepia(1);
  }
</style>
```



```

    .invert {
      -webkit-filter: invert(1);
      -moz-filter: invert(1);
      -ms-filter: invert(1);
      -o-filter: invert(1);
      filter: invert(1);
    }
  </style>

```

然后，我们再添加一些 JavaScript，当用户单击的时候改变滤镜：

```

var filters = ['', 'grayscale', 'sepia', 'invert'],
  currentFilter = 0;

document.querySelector('video').addEventListener('click', function (event) {
  if (streaming) {
    canvas.width = video.clientWidth;
    canvas.height = video.clientHeight;
    var context = canvas.getContext('2d');
    context.drawImage(video, 0, 0);
    currentFilter++;
    if (currentFilter > filters.length - 1) currentFilter = 0;
    canvas.className = filters[currentFilter];
  }
});

```

当你加载了这个页面，无论何时你从摄像头拍了一个新的快照，它总会被应用新的滤镜。CSS 滤镜的威力强大，可以动态修改 Canvas 输出的内容，浏览器会为你处理一切，比如应用滤镜和展示新图片。

一旦可以通过这种途径把流导入 canvas，将有无限的可能。canvas 是一个低阶且强有力绘图工具，支持很多功能，例如：绘制线条、图形和文字等。举个例子，下面这段代码可以给你的图片添加一些文字：

```

context.fillStyle = "white";
context.fillText("Hello World!", 10, 10);

```



当捕捉到图片时，你应该可以看到图片左上角角落里有一行字：Hello World！请大胆地使用 Canvas API 修改文字、尺寸或其他内容。在此之上更进一步的是 WebGL，这项技术支持在浏览器中渲染 3D 物体，集 JavaScript 之大成且效果惊人。你可以把一个流视频源当作 WebGL 中的纹理贴在 3D 空间里的对象上！网络上有成千上万个相关的示例，我建议你多看看，了解浏览器强大的功能。

## 自测题

Q1. 在浏览器中，打开来自文件的页面和从 web 服务器请求的页面均可以访问摄像头和麦克风。对或错？

Q2. 下列哪一个是不正确的浏览器前缀？

1. webkit GetUserMedia
2. moz GetUserMedia
3. blink GetUserMedia
4. ms GetUserMedia

Q3. getUserMedia API 的第三个参数接受一个函数，当从摄像头或麦克风获取数据流时如果有错误发生则调用这个函数。对或错？

Q4. 下列哪一个不是限制视频流的好处？

1. 给视频流加密
2. 节省运算处理所消耗的电力
3. 提供一个好的用户体验
4. 节省带宽

Q5. getUserMedia API 可以与 Canvas API 和 CSS 滤镜结合，给应用添加更多的功能。对或错？



## 小结

到目前为止，你应该掌握了如何通过不同的方式捕捉麦克风和摄像头的数据流。我们同样也介绍了限制数据流以及从多个设备中做出选择的方法。在最后一个项目中，我们将所有的功能组织起来，结合滤镜和图像捕捉的功能打造了一款独立的拍照室应用。

在本章中，我们介绍了如何访问媒体设备、限制媒体流、处理多设备、修改流数据等内容。

MediaStream 规范正在不断更新中，目前计划为这个 API 添加更多新特性，以此来构建更有趣的 web 应用。当你开发 WebRTC 应用时，请时刻关注最新的规范，关注各大浏览器厂商的最新动态。

本章的内容尽管看起来不多，但在接下来的章节中将发挥重大作用。后续我们将了解如何修改输入的流来确保 WebRTC 应用正常运行。

在接下来的章节中，我们将使用在本章中学习的知识通过 WebRTC 技术给另一位用户发送我们的数据流。



由 扫描全能王 扫描创建

# 3

## 创建简单的 WebRTC 应用

开发任何 WebRTC 应用的首个步骤就是创建 `RTCPeerConnection`。成功创建一个 `RTCPeerConnection` 的前提就是需要理解浏览器创建对等连接的内部工作原理。在这一章中，我们首先会对 WebRTC 的内在原理做一些知识储备，然后利用这些知识来创建一个简单的 WebRTC 视频聊天应用。

在这一章中，我们会涵盖以下内容：

- 理解 UDP 传输协议和实时传输
- 在本地与其他用户发送信令和交涉
- 在 Web 上找到其他用户和 NAT 穿透
- 创建 `RTCPeerConnection`

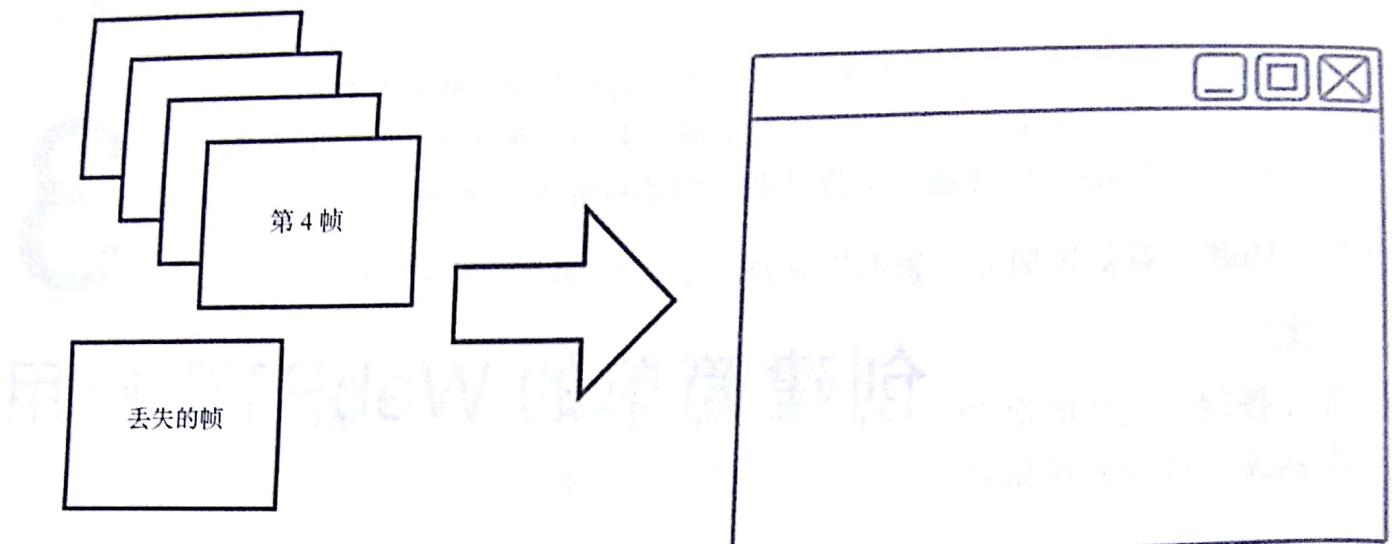
### 理解 UDP 传输协议和实时传输

数据的实时传输要求双方间有快速的连接速度。一个典型的网络连接需要将音频和视频都放到同一帧中，并以每秒 40 到 60 帧的速度发送给另一个用户，以此来获得较好的效果。在这种要求下，为了保持连接速度，音频和视频应用可允许部分数据帧的丢失。这意味着传送最新的数据帧比确保每一帧都不丢失来得更为重要。

现今的视频播放应用也采用了类似的做法。由于人类大脑的特殊性，视频游戏和流媒体播放器可以忍受些许数据帧的丢失。在看视频或游戏时我们的大脑会试着去填补这些空白。若每秒播放 30 帧，其中第 28 帧丢失，很多时候用户根本就不会察觉到。这就让我们的视频应用有了一套不同的需求：



由 扫描全能王 扫描创建



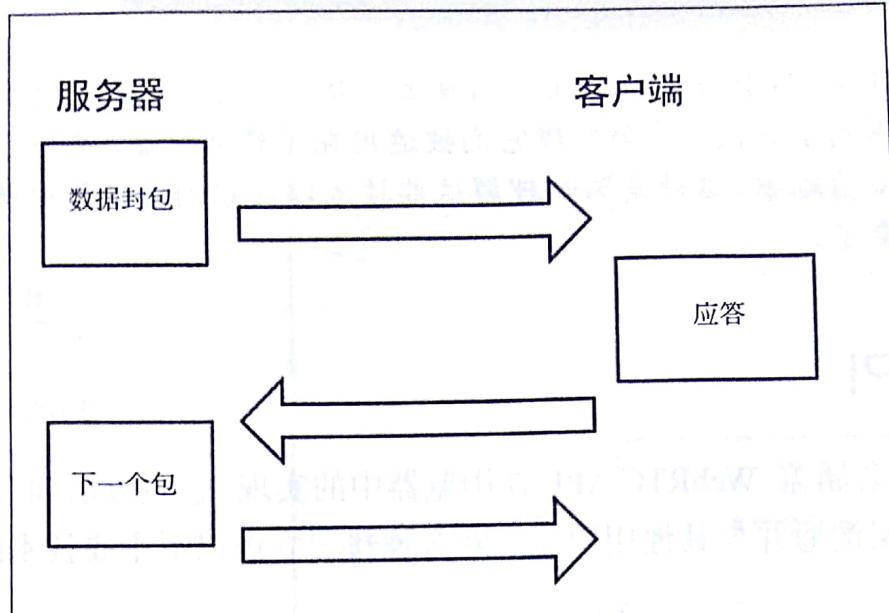
这就是为什么我们选择用户数据报协议( User Datagram Protocol, UDP )来作为 WebRTC 应用的传输协议。UDP 给了我们能力, 确切地说是它的不管控的能力, 来创建高性能应用。现今大部分的网络应用都建立在传输控制协议 ( Transmisson Control Protocol, TCP )之上。这是因为它保证了如下几点:

- 任何送出的数据都有送达的确认。
- 任何未到达接收端的数据会被重传并停止发送更多的数据。
- 数据是唯一的, 接收端不会有重复的数据。

正是这些特性让 TCP 成为现今网络应用的最佳选择。如果你传送一个 HTML 页面, 就必须确保所有的数据按照正确的顺序到达接收端。遗憾的是这项技术并不适用于所有情况。例如多人游戏中的流数据。视频游戏里的大部分数据在一秒甚至不到一秒的时间内就已经不是最新的了。这意味着用户只会关心最后几秒发生了什么。如果每一块数据都需要保证到达接收端, 当数据丢失时就会造成巨大的瓶颈:



由 扫描全能王 扫描创建



TCP 的这些限制使得 WebRTC 开发者们选择了 UDP 作为传输协议。WebRTC 的音频、视频和数据在浏览器间的传输不需最可靠但需要最快。这意味着允许丢帧，也就是说对这类应用来讲 UDP 是一个更好的选择。



这并不意味着 WebRTC 从不使用 TCP 协议来传输。之后我们将学习 Traversal Using Relays around NAT (TURN) 服务器以及它们是如何在高安全网络间使用 TCP 来协助传输 WebRTC 数据的。

UDP 不保证可靠性交付的特性实现了这个场景。为了对传送数据进行较少的确认，它被构建成不那么可靠的传输层。以下是它不保证的事情：

- 它不保证数据发送或接收的先后顺序。
- 它不保证每一个数据包都能够传送到接收端；一些数据可能在半路丢失。
- 它不跟踪每一个数据包的状态，即使接收端有数据丢失也会继续传输。

这样 WebRTC 就能以尽可能最快的方式来发送音频和视频。这也揭示了为何 WebRTC 是一个如此复杂的话题。并不是所有的网络都允许使用 UDP。为了防止恶意连接，拥有企业级防火墙的大型网络会完全阻止使用 UDP。这些连接所通过的路径与大部分网页下载的路径不同。为了让广大用户都能正常地使用 UDP，已有了许多解决方案。这些问题只是 WebRTC 的冰山一角。接下来的部分，我们会讲述其他让浏览器能开启 WebRTC 的技术。



由 扫描全能王 扫描创建

 UDP 和 TCP 并不仅仅用在网页上，今天你看到的众多互联网传输都用到了它们。你会发现它们被运用在了移动设备、电视、汽车和更多的地方。这就是为何理解这些技术以及它们的工作原理是多么重要了。

## WebRTC API

接下来的部分会涵盖 WebRTC API 在浏览器中的实现。这些方法和对象使得开发者们可以与 WebRTC 层沟通并与其他用户建立对等连接。它由以下主要技术组成：

- RTCPeerConnection 对象
- 信号传递和交涉
- 会话描述协议（SDP）
- 交互式连接建立（ICE）

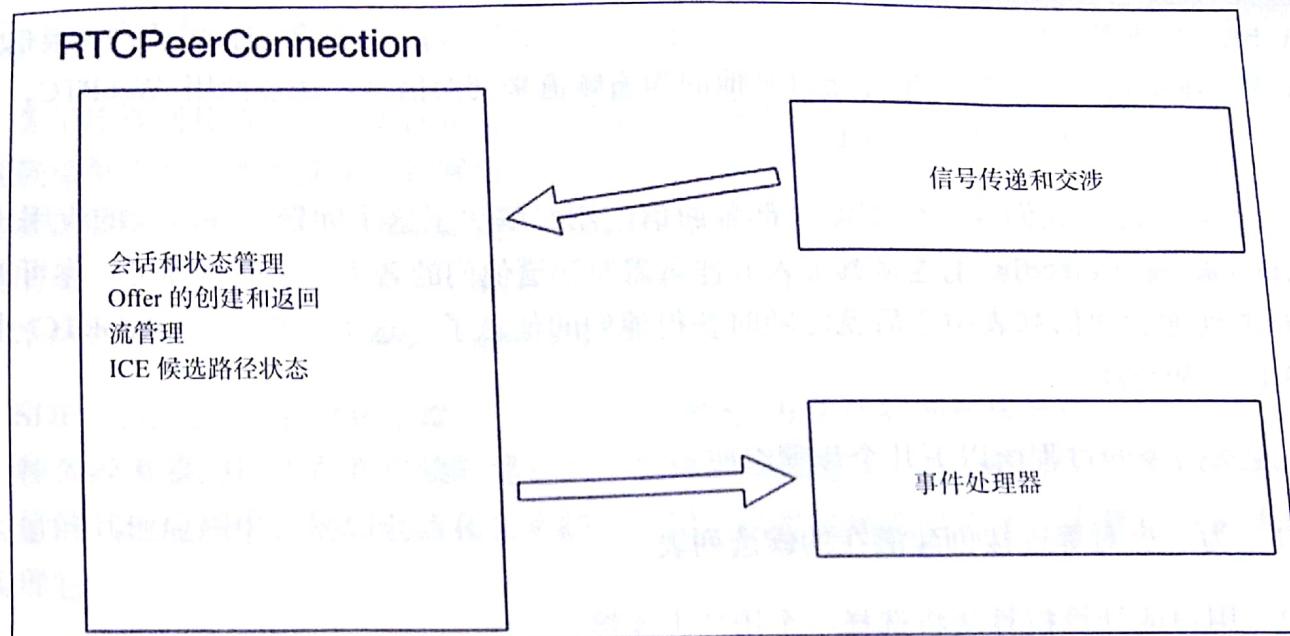
## RTCPeerConnection 对象

RTCPeerConnection 对象是 WebRTC API 的主入口。我们通过它初始化一个连接、连接他人以及传送流媒体信息。它负责与另一用户建立 UDP 连接。请记住这个名词，它将时常出现在本书剩下的内容中。

RTCPeerConnection 对象的功能是维护浏览器内会话和对等连接的状态。它也负责对等连接的建立。它将所有的这些封装起来并暴露出一系列的事件，它们会在连接过程的关键点被触发。这些事件使你可以访问配置项，同时了解对等连接的内部正在发生什么：



由 扫描全能王 扫描创建



`RTCPeerConnection` 对象是浏览器内一个简单的对象，可通过 new 来实例化：

```

var myConnection = new RTCPeerConnection(configuration);
myConnection.onaddstream = function(stream) {
  // 在此处使用 stream 参数
};
  
```

这个连接接收一个 `configure` 对象，稍后会在本章中说到这个对象。在这个例子里，我们为 `onaddstream` 事件添加了一个处理器。当终端用户在对等连接中添加视频或音频流时该事件将会被触发。稍后谈及这个处理器。

## 信号传递和交涉

一般来说，我们需要知道对方浏览器的网络地址才能连接到它。通常地址由 IP 地址和端口号组成，这就像街道地址一样，它可以让别人找到你。你的电脑或移动设备的 IP 地址可以允许其他的网络设备来直接传送数据给你的设备；`RTCPeerConnection` 建立在这之上。当这些设备知道如何在互联网上找到其他设备后，它们需要知道如何与其他设备进行沟通。这意味着它们需要交换数据，这些数据包括各设备支持哪种协议以及视频和音频的解码器及其他信息。

所以为了连接到其他用户，你需要了解它们。一种方式是在你的电脑上建一张列表来存储那些你能连接的用户。为了与其他用户沟通，你仅需要交换联络信息，其他都交由



WebRTC 处理。然而这种方式也有缺点，你必须手动地与每一个你想连接的用户交换信息。你需要维护这些用户的名单并通过其他的沟通频道来交换信息。通过使用 WebRTC，我们可以让这个过程变得更自动化。

幸运的是，我们今天使用的大部分通信应用已解决了这个问题。在流行的应用比如 Facebook 或 LinkedIn 上连接其他人，你只需要知道他们的名字然后搜索即可。你可以把他们加到你的通信列表中之后就能随时获得他们的信息了。这个过程就是 WebRTC 中的发送信令和交涉。

发送信令的过程由以下几个步骤组成：

1. 为一个对等连接创建潜在的候选列表。
2. 用户或计算机算法将选择一个用户去连接。
3. 信令层将通知那个用户有人想要连接他 / 她，用户可以选择接受或拒绝。
4. 当连接的请求被接受时，第一个用户会被通知。
5. 若接受，第一个用户将初始化 RTCPeerConnection。
6. 双方将通过信令通道交换各自电脑的硬件和软件信息。
7. 双方将通过信令通道交换各自电脑的位置信息。
8. 用户之间的连接将成功或失败。

这只是一个 WebRTC 发送信令的例子。实际上，WebRTC 规范并不包含两个用户该如何交换信息的标准。这是因为连接用户的标准一直在扩充。现在已经有许多的规范，发送信令和交涉过程中还将有更多的规范被创建。WebRTC 规范的作者们认为试着去统一这个规范将会阻碍它的发展。

在本书中，我们将自己实现发送信令和交涉的过程。我们将写一个可以在两个浏览器间传送信息的简单服务器。尽管它是一个简单的可能会有安全瑕疵的服务器，但你将对它如何在 WebRTC 中工作有深刻的理解。与此同时，在众多公司提供的大量的信令选择中尽情探索吧。现在有上百种发送信令和交涉的解决方案，且每天都有更多的方案涌现。它们中的一些与现在的电话或聊天工具集成，比如 XMPP 或 SIP，有些则使用崭新的发送信令方式。



## 会话描述协议 (SDP)

为了连接到其他用户，你首先需要对他们有一些了解。你需要了解对方所支持的音频和视频编解码器、他们使用何种网络以及他们的电脑可以处理多少数据。双方也需要能方便地传送数据。因为我们并未规定这些数据该如何传送，它也应当具有在各种传输协议中传送的能力。这意味着我们需要一张有用户所有信息的字符串形式的名片来传送给其他用户。幸运的是 SDP 为我们提供了这些功能。

SDP 的好处在于它存在了很久——它在 20 世纪 90 年代后期初次起草。这意味着 SDP 是一种久经考验的用于在客户端间建立基于媒体的连接方式。在 WebRTC 之前它就被用在大量的其他应用中，例如电话和文字聊天应用。这也意味着有许多非常棒的资源来使用和实现它。

SDP 是由浏览器提供的基于字符串的二进制数据对象。这种字符串的形式是一系列的键值对，由换行符分隔：

<key>=<value>\n

key 是一个单字符，用来表明值的类型。value 是由机器可读的配置项组成的有结构的一组文本。不同的键值对由换行符分隔。

SDP 涵盖一个指定用户的描述、时间配置和对媒体的限制。SDP 是在与其他用户建立连接的过程中由 RTCPeerConnection 对象给出的。在开始与 RTCPeerConnection 打交道时，你可以轻易地把它在 JavaScript 的控制面板里打印出来。你可以清楚地看到 SDP 里有什么，它们可能像下面这样：

```
v=0
o=- 1167826560034916900 2 IN IP4 127.0.0.1
s=- 
t=0 0
a=group:BUNDLE audio video
a=msid-semantic: WMS K44HTOZVjyAyAlvUVD3pOLu8i0LdytHiWRp1
m=audio 1 RTP/SAVPF 111 103 104 0 8 106 105 13 126
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:V15FBUBecw/U3EzQ
a=ice-pwd:OtsNG6FzUH8uhNEh0g9/hprb
```



由 扫描全能王 扫描创建

```

a=ice-options:google-ice
a=fingerprint:sha-256
FB:56:7D:B6:E0:C7:E7:39:FE:47:5A:12:6C:B4:4E:0E:2D:18:CE:AE:33:92:
A9:60:3F:14:E4:D9:AA:0D:BE:0D
a=setup:actpass
a=mid:audio
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=sendrecv
a=rtpmap:mux
a=crypto:1 AES_CM_128_HMAC_SHA1_80
inline:zE+3pkUbJyFG4UmmvPxG/OFC4+QE24X8Zf3iOSCf
a=rtpmap:111 opus/48000/2
a=fmtp:111 minptime=10
a=rtpmap:103 ISAC/16000
a=rtpmap:104 ISAC/32000
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:106 CN/32000
a=rtpmap:105 CN/16000
a=rtpmap:13 CN/8000
a=rtpmap:126 telephone-event/8000
a=maxptime:60
a:ssrc:4274470304 cname:+j4Ma6UfMsCcQCWK
a:ssrc:4274470304 msid:K44HTOZVjyAyAlvUVD3pOLu8i0LdytHiWRp1
a1751f6b-98de-469b-b6c0-81f46e19009d
a:ssrc:4274470304 mslabel:K44HTOZVjyAyAlvUVD3pOLu8i0LdytHiWRp1
a:ssrc:4274470304 label:a1751f6b-98de-469b-b6c0-81f46e19009d
m=video 1 RTP/SAVPF 100 116 117
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:Vl5FBUBecw/U3EzQ
a=ice-pwd:OtsNG6FzUH8uhNEhOg9/hprb
a=ice-options:google-ice
a=fingerprint:sha-256 FB:56:7D:B6:E0:C7:E7:39:FE:47:5A:12:6C:B4:4E:0E:
2D:18:CE:AE:33:92:
A9:60:3F:14:E4:D9:AA:0D:BE:0D
a=setup:actpass

```



```

a=mid:video
a=extmap:2 urn:ietf:params:rtp-hdrext:toffset
a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-
time
a=sendrecv
a=rtcp-mux
a=crypto:1 AES_CM_128_HMAC_SHA1_80
inline:zE+3pkUbJyFG4UmmvPxG/OFC4+QE24X8Zf3iOSCf
a=rtpmap:100 VP8/90000
a=rtcp-fb:100 ccm fir
a=rtcp-fb:100 nack
a=rtcp-fb:100 nack pli
a=rtcp-fb:100 goog-remb
a=rtpmap:116 red/90000
a=rtpmap:117 ulpfec/90000
a:ssrc:3285139021 cname:+j4Ma6UfMsCcQCWK
a:ssrc:3285139021 msid:K44HTOZVjyAyAlvUVD3pOLu8i0LdytHiWRp1
bd02b355-b8af-4b68-b82d-7b9cd03461cf
a:ssrc:3285139021 mslabel:K44HTOZVjyAyAlvUVD3pOLu8i0LdytHiWRp1
a:ssrc:3285139021 label:bd02b355-b8af-4b68-b82d-7b9cd03461cf

```

上面这段代码是在会话初始化过程中从我自己的电脑中截取的。如你所见，这些生成的代码看上去复杂难懂。它先用 IP 地址来识别连接，之后创建请求的基本信息，比如是在请求音频还是视频或者两者都是。随后会创建一些音频信息，包括诸如编码类型和 ice 配置。它同样会创建视频信息。我们的目的并不是去理解每一行代码的意义，而是去加深理解 SDP 有何用途。在本书中你不需要直接使用它，但未来某天可能会需要。

总之，SDP 类似你电脑的名片，其他用户可以通过它来试着联系到你。SDP 与发送指令和交涉结合，构成对等连接的前半部分。在之后的部分，我们将介绍当双方用户都知道如何找到对方之后发生了什么。

## 找到一条清晰的路线到其他用户

网络安全是现今大部分网络的一大重点。你所使用的任何网络可能都有好几层的访问控制，告诉你数据在哪以及如何发送它们。这意味着你需要在自己和对方的网络里找到一条清晰的路线来连接到对方。为了达到这个目的，WebRTC 用到了多种技术：



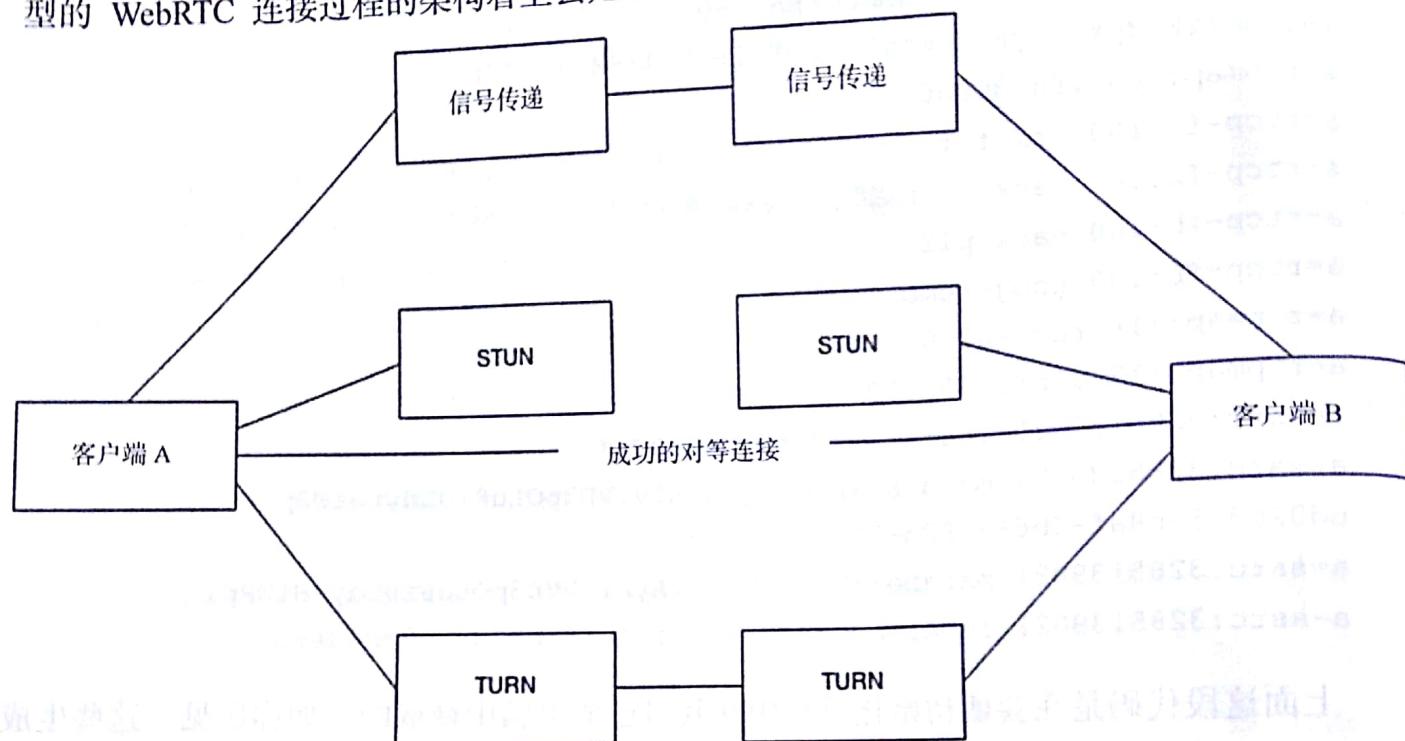
由 扫描全能王 扫描创建

- NAT 会话穿透工具 (STUN)

- 使用中继技术穿透 NAT (TURN)

- 交互式连接建立 (ICE)

这个过程需要许多服务器和连接的建立。为了理解它们如何工作，首先要知道一个典型的 WebRTC 连接过程的架构看上去是怎样的：



首先要找到你的 IP 地址。连接到互联网的几乎所有设备都有一个 IP 地址来标明它们在网络上的地址。通过它你才能让数据包到达正确的地址。当你的网络位于路由器之后时，问题就来了。路由器会隐藏你电脑的 IP 地址并用其他的地址来替代，这是为了增加安全性以及为了允许多台电脑使用同一个网络地址。通常，你有多个 IP 地址、你的电脑、网络路由器以及公网。

## NAT 会话穿透工具 (STUN)

STUN 是双方建立稳定连接的第一步。它有助于在互联网上识别对方，同时在创建对等连接时常被其他协议使用。首先发一个请求给服务器，以开启 STUN 协议。之后服务器识别发出请求的客户端的 IP 地址，并将其返回给客户端。客户端之后可通过返回的 IP 地址来识别自己。



由 扫描全能王 扫描创建

使用 STUN 协议需要有一个支持 STUN 协议的服务器。目前，在 Firefox 和 Chrome 中，浏览器商直接提供了支持此协议的默认服务器。这极利于快速上手和测试。



尽管你可能会对无服务器会话赞赏有加，但建立高质量的 WebRTC 应用实际上需要多个服务器。你需要提供一套 STUN 和 TURN 服务器来让你的客户使用。现在已有许多非常棒的服务对此提供，请务必搜寻更多有用的信息。

## 使用中继技术穿透 NAT (TURN)

在一些情况下，受防火墙限制，可能不允许任何基于 STUN 的访问。这种情况可能存在于企业级 NAT 中，它利用端口随机化来让数以千计的设备连接，你无法使用通用的方法找到用户。在这种情况下，我们需要使用不同的方法来连接另一个用户。这个标准叫作 TURN。

它的工作原理是代表客户端在对等连接的双方之间增加一个转播。客户端之后从 TURN 服务器得到信息，这有些类似于向服务器请求从热门视频网站加载视频。这就要求 TURN 服务器去下载、处理并重定向每一个用户发送过来的数据包。这是为什么 TURN 被认为是用来建立 WebRTC 连接的最后一种手段，因为建立高质量的 TURN 服务代价很大。

关于使用 STUN 还是 TURN 有许多不同的数据资料，但它们都得出了相同的一个结论——在大部分情况下，不使用 TURN 不会有任何问题。在大多数网络设置下 WebRTC 使用 STUN 也工作得好好的。建立你自己的 WebRTC 服务时，查询一下这些资料总是好的，你可以自己决定使用 TURN 是否值得。



你可能注意到没有一个例子含有 TURN 服务器的配置参数。本书假定你的网络可以兼容 STUN。如果在连接的时候碰到了问题，那么你可能需要找一公用的利用率低的 TURN 服务器，用于后面的例子。

## 交互式连接建立

我们已经讲了 STUN 和 TURN，把它们结合在一起的一个标准叫 ICE。这个过程利用



STUN 和 TURN 为对等连接提供正确的路由。 寻找一系列双方可用的地址并按顺序测试每一个地址，直到找到一个双方都可用的组合。

ICE 过程开始并不知道每一个用户的网络配置。它通过一些步骤来一步步发现双方的网络是如何建立的。这个过程将会使用不同的技术。其目的是发现双方网络足够多的信息，以此来建立一个成功的连接。

每一个 ICE 候选路径都是通过 STUN 和 TURN 来找到的。通过查询 STUN 服务器来找到外部 IP 地址，并附上 STUN 服务器的地址，当连接失败时可用作备份。当浏览器来找到外部 IP 地址，并附上 STUN 服务器的地址，当连接失败时可用作备份。当足够找到一个新的候选路径时，通知客户端程序将使用信令信道来发送 ICE 候选路径。当足够多的地址被发现且通过验证，并且连接建立后，这个过程就结束了。

## 创建一个基本的 WebRTC 应用

现在对 WebRTC 的各部分如何使用都有了较深的理解，开始建立我们的第一个 WebRTC 应用。在这章的末尾，将有一个可以工作的 WebRTC 网页，能看到它的实际应用。我们将把所有讲过的内容集成到一个容易开发的例子里。这将包括：

- 创建一个 RTCPeerConnection
- 创建 SDP offer 和回应
- 为双方找到 ICE 候选路径
- 创建一个成功的 WebRTC 连接

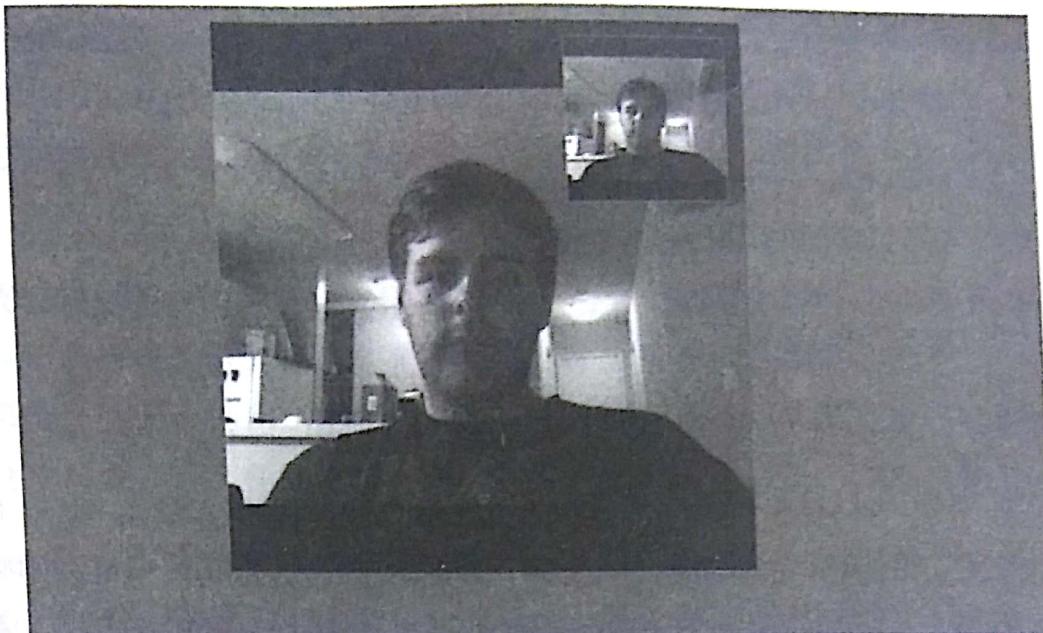
### 创建一个 RTCPeerConnection

让人遗憾的是，我们将创建的应用并不是一个功能完整的应用，除非正好你喜欢照镜子欣赏自己。这章的主要目标是连接一个浏览器窗口，从用户的摄像头加载视频数据。最终目的是在页面上获得两个视频流，一个直接来自摄像头，另一个来自浏览器在本地创建的 WebRTC 连接。

尽管这个应用没什么大用，但它能帮助我们让代码更直观可读。之后将学习如何用服务器做远程连接。因要连接的就是本地浏览器，我们并不需要操心网络稳定性或搭建服务器。在完成这个项目后，你的应用看上去应该类似这样：



由 扫描全能王 扫描创建



撇开我帅气的脸蛋不谈，你可以看到这是一个相当简单的例子。我们将像第 2 章里的第一个例子那样，用一些类似的步骤来开始。你需要创建另一个 HTML 页面并用本地 Web 服务器加载它。可以参考第 2 章中获得媒体设备一节的创建静态服务器部分，复习一下如何搭建开发环境。

首先我们将创建一些函数来处理跨浏览器支持的问题。这些函数可以告诉我们当前的浏览器是否支持需要的特性。这也会将 API 标准化，以确保无论在哪种浏览器下我们总是可以调用同一个函数。



大部分的浏览器通过使用前缀来调用正在开发中的方法。确保你使用的是最新版本的浏览器，并决定应该在代码中使用哪个前缀。现在也有 JavaScript 库来帮助处理这些前缀。

我们通过创建一个引用 JavaScript 源文件的新网页来开始。我们的 HTML 需要含有两个 `video` 元素，分别给不同的客户端。

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Learning WebRTC - Chapter 4: Creating a
      RTCPeerConnection</title>
```



由 扫描全能王 扫描创建

```

</head>
<body>
  <div id=""container"">
    <video id=""yours"" autoplay></video>
    <video id=""theirs"" autoplay></video>
  </div>
  <script src=""main.js""></script>
</body>
</html>

```

如果你经常创建 HTML5 网页，应该对 `html` 和 `head` 标签很熟悉。这是 HTML5 兼容页面的标准格式。创建页面可以使用许多不同的模板，但这是我认为可以用来实现功能而又最简单的模板了。只要页面上有 `video` 元素，我们应用程序的工作方式并不会发生巨变，如若你想对这个文件做一些更改，请随意。

请注意这两个 `video` 元素被标记为 `yours` 和 `theirs`。这将是我们的两个视频源，它们将模拟连接对方。`yours` 代表本地初始化连接的用户。`theirs` 代表通过 WebRTC 连接到的远程用户，尽管他们的实际距离并不远。

最后引入 `script` 函数。请始终记得在 HTML 页面的最后加上它。这确保了 `body` 中的元素已经可以被使用了，而且此时页面已经完全加载好，只等 JavaScript 去与其交互。

接下来，我们将创建 JavaScript 源码。创建一个叫 `main.js` 的文件，并粘贴如下代码：

```

function hasUserMedia() {
  navigator.getUserMedia = navigator.getUserMedia || 
  navigator.webkit GetUserMedia || navigator.mozGetUserMedia || 
  navigator.msGetUserMedia;
  return !!navigator.getUserMedia;
}

function hasRTCPeerConnection() {
  window.RTCPeerConnection = window.RTCPeerConnection || 
  window.webkitRTCPeerConnection || window.mozRTCPeerConnection;
  return !!window.RTCPeerConnection;
}

```

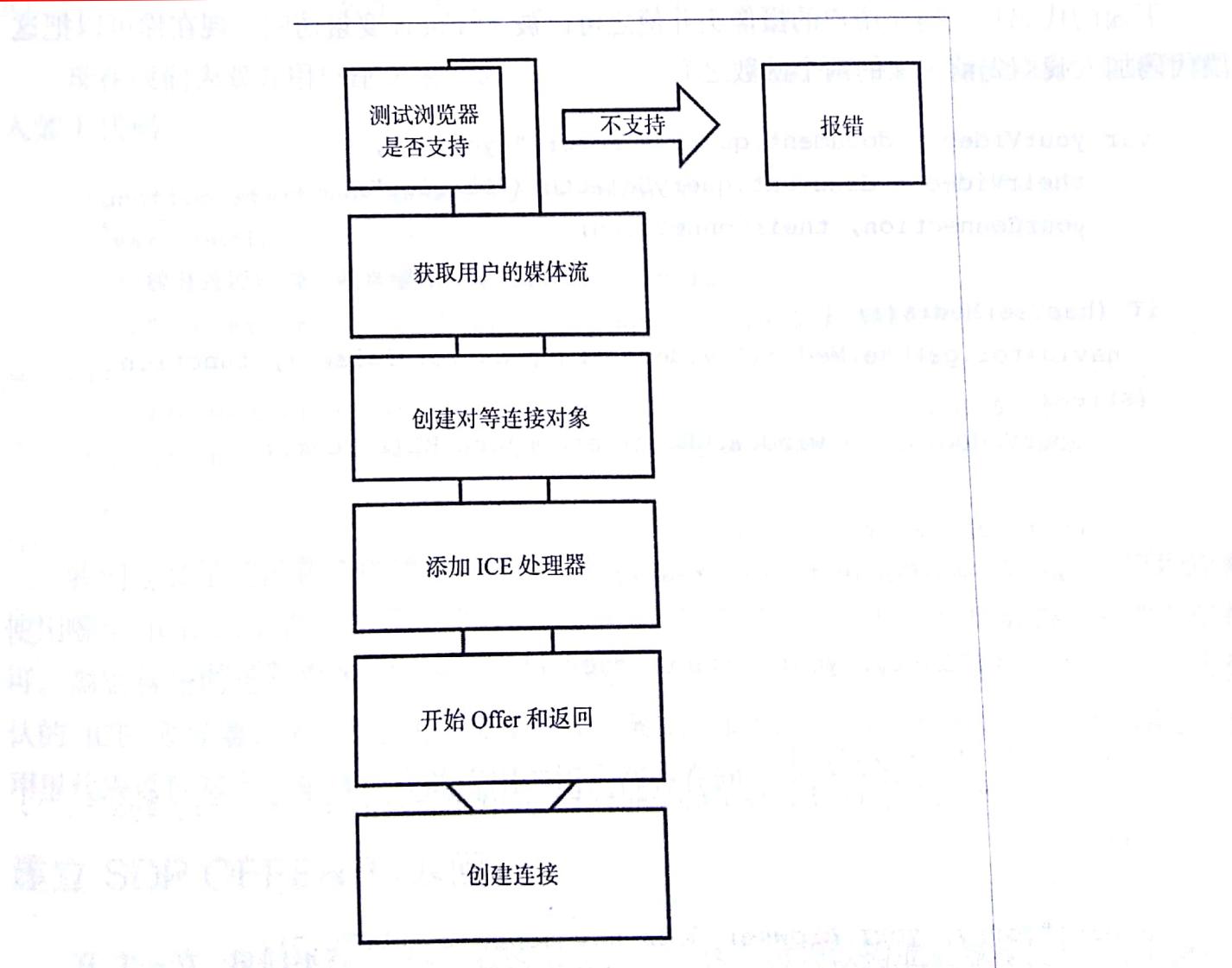


由 扫描全能王 扫描创建

第一个函数处理了 getUserMedia API，你应该很熟悉此函数。第二个函数用相同的方法处理 RTCPeerConnection 对象，以确保我们能在浏览器里使用它。先试着将浏览器内置的 WebRTC 函数赋给全局函数，最后通过返回变量的值来确定浏览器内是否存在该功能。

现在我们知道用户的浏览器支持哪些 API 了，让我们继续前进并开始使用它们。后续的步骤你应该也相当熟悉。我们将重复一些在第 2 章中修改媒体流部分的一些功能来获得用户的视频流。在开始使用 WebRTC 前，应先从用户那里获得本地视频流。这确保用户已准备好建立对等连接，在创建对等连接前我们无须再等待用户同意视频共享。

用 WebRTC 建立的大部分应用都将经过一系列的状态变化。WebRTC 最难的部分是分步骤地执行顺序。如果顺序出错，应用会很快崩溃。这些状态是阻塞的，这意味着我们必须完成前面的一个步骤，才能进入下一个状态。以下是应用如何工作的流程图：



由 扫描全能王 扫描创建

首先需要从用户那里得到媒体流。这确保流已经准备就绪且用户同意共享他们的摄像头和麦克风。

之后，建立对等连接。此过程由断开状态开始。现在我们可以设置 WebRTC 连接将使用的 ICE 服务器。此时，浏览器处于空闲状态并等待连接过程的开始。

当一方用户创建一个 `offer` 时魔法就开始了。它让浏览器即刻开始工作，为与另一方用户建立对等连接而做准备。我们之前提及过，`offer` 和返回都是发送信令过程中的部分。

与此同时，浏览器也准备寻找合适的端口和 IP 的组合来让对方可以连接。寻找将在一段时间内持续，直到连接成功或失败。一旦这一步成功，WebRTC 连接过程就结束了，双方可以开始共享信息。

下面的代码片段捕获用户的摄像头并使之可以被 `stream` 变量访问。现在你可以把这段代码加入我们先前定义的两个函数之后。

```

var yourVideo = document.querySelector("#yours"),
    theirVideo = document.querySelector("#theirs"),
    yourConnection, theirConnection;

if (hasUserMedia()) {
  navigator.getUserMedia({ video: true, audio: false }, function
    (stream) {
      yourVideo.src = window.URL.createObjectURL(stream);

      if (hasRTCPeerConnection()) {
        startPeerConnection(stream);
      } else {
        alert("Sorry, your browser does not support WebRTC.");
      }
    }, function (error) {
      alert("Sorry, we failed to capture your camera, please try again.");
    });
} else {
  alert("Sorry, your browser does not support WebRTC.");
}

```



我们先从文档中筛选出 video 元素并创建一些随后会用到的变量。此时假定浏览器支持 querySelector API。之后检查用户是否可以获取 getUserMedia API。如果不行，程序在此处终止并提醒用户浏览器不支持 WebRTC。

如果成功，我们将试图从用户处获得摄像头。由于用户必须同意共享他们的摄像头，因此这是一个异步的操作。如果这一步成功，我们将设置本地的视频流为用户的流，这样他们知道此步骤成功了。如果失败了，我们将通知用户出现了错误并停止这个过程。

最后我们检查用户的浏览器是否可以获取 RTCPeerConnection API。如果浏览器支持，我们调用函数开始连接过程（将在下一节定义此函数）。如果不支持，将在此步停止并再一次提示用户。

下一步是实现上一节调用的 startPeerConnection 函数。这个函数会创建 RTCPeerConnection 对象，建立 SDP offer 和返回，为双方寻找 ICE 候选路径。

现在我们为双方用户建立 RTCPeerConnection 对象。在你的 JavaScript 文件里加入如下代码：

```
function startPeerConnection(stream) {
  var configuration = {
    //放开这段注释代码来增加自定义的 iceServers
    // "iceServers": [{ "url": "stun:127.0.0.1:9876" }]
  };
  yourConnection = new webkitRTCPeerConnection(configuration);
  theirConnection = new webkitRTCPeerConnection(configuration);
}
```

我们定义了此函数来创建连接对象。在 configuration 对象里，你可以为应用设置使用哪个 ICE 服务器。如果要使用自定义的 ICE 服务器，放开注释的代码并改变其值即可。浏览器在创建对等连接时会自动使用我们定义的配置项。由于浏览器一般会有一套默认的 ICE 服务器，这些配置并不是必需的。随后，我们建立两个对等连接对象，用来在应用里代表通信双方。请谨记在此应用中用户都将在同一个浏览器中。

## 建立 SDP OFFER 和返回

在这一节，我们执行 offer 和返回 answer 这个过程以构成对等连接。以下代码在双



方间建立 offer 并返回 answer 流。

```

function startPeerConnection(stream) {
  var configuration = {
    //放开这段注释代码来增加定制的 iceServers
    // "iceServers": [{ "url": "stun:127.0.0.1:9876" }]
  };
  yourConnection = new webkitRTCPeerConnection(configuration);
  theirConnection = new webkitRTCPeerConnection(configuration);

  //开始offer
  yourConnection.createOffer(function (offer) {
    yourConnection.setLocalDescription(offer);
    theirConnection.setRemoteDescription(offer);

    theirConnection.createAnswer(function (offer) {
      theirConnection.setLocalDescription(offer);
      yourConnection.setRemoteDescription(offer);
    });
  });
}

```

在解释了一整章后，你可能已经发现这段代码看上去相当简单。这是由于通信双方都在同一个浏览器窗口中。采用这种方式，我们能确保当用户收到 offer 时不用执行多次异步操作，使用这种方式实现 offer/answer 机制简单易懂。你能明确知道成功创建一个对等连接所需要的步骤以及顺序。如果你使用浏览器内置的调试工具，你可以执行一遍这些步骤并在每一步检查 RTCPeerConnection 对象来了解到底发生了什么。

下一章，我们将就此话题展开深度讨论。通常，你连接的用户与你并不在同一个浏览器中，这意味着你将需要一个服务器，用于在浏览器窗口中连接双方。考虑到这些步骤不但需要以正确的顺序执行，而且也需要跨多个浏览器窗口执行，这个过程将变得更加复杂。此过程将发生许多的同步操作，有时可能会导致环境的不稳定。

## 寻找 ICE 候选路径

建立对等连接的最后一部分是在双方间传递 ICE 候选路径，以使他们可以互相连接。



由 扫描全能王 扫描创建

现在将 startPeerConnection 更改如下：

```

function startPeerConnection(stream) {
    var configuration = {
        //放开这段注释代码来增加定制的 iceServers
        // "iceServers": [{ "url": "stun:127.0.0.1:9876" }]
    };
    yourConnection = new webkitRTCPeerConnection(configuration);
    theirConnection = new webkitRTCPeerConnection(configuration);

    //创建 ICE 处理
    yourConnection.onicecandidate = function (event) {
        if (event.candidate) {
            theirConnection.addIceCandidate(new RTCIceCandidate(event.candidate));
        }
    };
    theirConnection.onicecandidate = function (event) {
        if (event.candidate) {
            yourConnection.addIceCandidate(new RTCIceCandidate(event.candidate));
        }
    };

    //开始 offer
    yourConnection.createOffer(function (offer) {
        yourConnection.setLocalDescription(offer);
        theirConnection.setRemoteDescription(offer);

        theirConnection.createAnswer(function (offer) {
            theirConnection.setLocalDescription(offer);
            yourConnection.setRemoteDescription(offer);
        });
    });
};

}


```



你可能已经注意到了，这一部分的代码完全是事件驱动的。这是因为寻找 ICE 候选路径总是异步的。浏览器会不停地搜寻，直至找到尽可能多的能创建良好且稳定的对等连接的候选路径。

在后续的几章，我们将构建发送功能，它将通过信令通道来发送 ICE 候选路径。有一点要注意的是，当我们从 `theirConnection` 中获取 ICE 候选路径时，需要将路径加入到 `yourConnection` 中，反之亦然。当另一方跟我们不在同一网络时，这些数据会横跨互联网。

## 加入流和打磨

使用 WebRTC 在对等连接中加入流非常容易。API 会负责流的建立和发送。当另一方在对等连接中加入流时，会发送提醒，告诉第一个用户有变更。浏览器通过调用 `onaddstream` 来通知用户，流已被加入。

```
// 监听流的创建
yourConnection.addStream(stream);
theirConnection.onaddstream = function (e) {
    theirVideo.src = window.URL.createObjectURL(e.stream);
};
```

通过为流对象的地址创建 URL，我们可以将该流加入本地的视频中。它通过在浏览器中创建一个值来标识流，使 `video` 元素与其交互。这个值是视频流的唯一 ID，用来通知 `video` 元素从本地流播放视频数据。

最后，我们将为应用增加一些样式。目前最流行的视频通信软件的样式是那些我们在应用中看到的，比如 Skype。这些样式已被许多基于 WebRTC 的 demo 复制。通常，你呼叫的那个人的画面出现在应用中的前方，而你自己的画面出现在一个大窗口内的小窗口中。由于我们是在创建一个网页，这些样式都可以由一些简单的 CSS 来实现，如下：

```
<style>
    body {
        background-color: #3D6DF2;
        margin-top: 15px;
    }
    video {
```



由 扫描全能王 扫描创建

```

background: black;
border: 1px solid gray;
}

#container {
position: relative;
display: block;
margin: 0 auto;
width: 500px;
height: 500px;
}

#yours {
width: 150px;
height: 150px;
position: absolute;
top: 15px;
right: 15px;
}

#theirs {
width: 500px;
height: 500px;
}

</style>

```

只要把这些加到你的 HTML 页面中，你就会有一个布局优美的 WebRTC 应用。如果觉得我们的应用看上去很蠢，可以再加入一些样式。后续几章，将在此基础上开发，用 CSS 来让 demo 变得好看些永远令人激动。

## 运行你的第一个 WebRTC 应用

现在运行网页来测试。当你运行这个页面时它应当会询问是否可以让浏览器共享你的摄像头。当你同意后，将开始 WebRTC 的连接过程。浏览器会立即运行我们之前讨论的那些步骤，建立起一个连接。你应当可以看到两个你的影像，一个来自你的摄像头，另一



个是通过 WebRTC 加载的流。

我们有这个例子的所有代码，可供你参考。下面是 index.html 文件中的代码：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Learning WebRTC - Chapter 4: Creating a
    RTCPeerConnection</title>
```

```
<style>
  body {
    background-color: #3D6DF2;
    margin-top: 15px;
  }
  video {
    background: black;
    border: 1px solid gray;
  }
```

```
#container {
  position: relative;
  display: block;
  margin: 0 auto;
  width: 500px;
  height: 500px;
}
```

```
#yours {
  width: 150px;
  height: 150px;
  position: absolute;
  top: 15px;
  right: 15px;
```



由 扫描全能王 扫描创建

```

    }
}

#theirs {
    width: 500px;
    height: 500px;
}

```

</style>

</head>

<body>

<div id=""container"">

<video id=""yours"" autoplay></video>

<video id=""theirs"" autoplay></video>

</div>

<script src=""main.js""></script>

</body>

</html>

以下是 main.js 的代码：

```

function hasUserMedia() {
    navigator.getUserMedia = navigator.getUserMedia ||
    navigator.webkit GetUserMedia || navigator.mozGetUserMedia ||
    navigator.msGetUserMedia;
    return !!navigator.getUserMedia;
}

function hasRTCPeerConnection() {
    window.RTCPeerConnection = window.RTCPeerConnection ||
    window.webkitRTCPeerConnection || window.mozRTCPeerConnection;
    return !!window.RTCPeerConnection;
}

var yourVideo = document.querySelector("#yours"),
    theirVideo = document.querySelector("#theirs"),
    yourConnection, theirConnection;

```

if (hasUserMedia()) {



```

navigator.getUserMedia({ video: true, audio: false }, function
  (stream) {
    yourVideo.src = window.URL.createObjectURL(stream);

    if (hasRTCPeerConnection()) {
      startPeerConnection(stream);
    } else {
      alert("Sorry, your browser does not support WebRTC.");
    }
  }, function (error) {
    console.log(error);
  });
} else {
  alert("Sorry, your browser does not support WebRTC.");
}

function startPeerConnection(stream) {
  var configuration = {
    //放开这段代码来增加定制的 iceServers
    // "iceServers": [{ "url": "stun:127.0.0.1:9876" }]
  };
  yourConnection = new webkitRTCPeerConnection(configuration);
  theirConnection = new webkitRTCPeerConnection(configuration);

  //创建 ICE 处理
  yourConnection.onicecandidate = function (event) {
    if (event.candidate) {
      theirConnection.addIceCandidate(new
        RTCIceCandidate(event.candidate));
    }
  };

  theirConnection.onicecandidate = function (event) {
    if (event.candidate) {
      yourConnection.addIceCandidate(new
        RTCIceCandidate(event.candidate));
    }
  };
}

```



```

};

//开始 offer
yourConnection.createOffer(function (offer) {
    yourConnection.setLocalDescription(offer);
    theirConnection.setRemoteDescription(offer);

    theirConnection.createAnswer(function (offer) {
        theirConnection.setLocalDescription(offer);
        yourConnection.setRemoteDescription(offer);
    });
});
}

```

## 自测题

Q1. UDP 比较适合 WebRTC 对等连接,因为它传送数据包时的不保证送达。对或错?

Q2. 发送信令和交涉是 WebRTC 标准中的一部分,并且它们完全由浏览器负责实现。对或错?

Q3. 会话描述协议 (SDP) 是:

1. WebRTC 的配置文件
2. 发现使用何种视频编码方式的方法
3. 你电脑的名片
4. 没有人懂的令人困惑的技术文档

Q4. 交互式连接建立 (ICE) 在建立网络时有助于发现双方间清晰的路线。对或错?

Q5. 关于 TURN 哪一项是不正确的?

1. 相对于普通的连接来说它需要更多的带宽和处理能力
2. 在尝试了各种连接方法后, TURN 应该是最后的手段
3. TURN 服务器将会处理双方间传递的每一个数据包
4. TURN 方法是由浏览器提供的



## 小结

恭喜你到达这一步。如果你成功地学习完这一章，你已能够创建大型 WebRTC 应用了。这一章的目标不仅仅是创建一个 WebRTC 应用，还需要理解这个过程中每一个步骤中发生了什么。

说了这么多，你应当已知道 WebRTC 是一种复杂的技术。我们讲解了许多 WebRTC 内在工作原理。尽管我们并不需要知道浏览器是如何实现 WebRTC 的所有细节的，但理解这些重要的部分如何一起工作有助于你理解接下来的例子。

在这一章中，我们讲述了浏览器创建对等的内部工作原理；讲述了所需的技术，包括 UDP、SDP 和 ICE。你应当对两个浏览器如何在互联网中找到对方并与之沟通有浅显的认知。

复习到现在为止所讲授的资料将有助于完全理解例子中 WebRTC 是如何工作的。了解每一步骤以及步骤的顺序都十分重要。在之后的章节中，我们将会介绍更多复杂的功能，这将有助于你调试 WebRTC 应用中出现的问题。

本书剩下的部分将在这个例子的基础上继续，会比现在的更加复杂。我们将增加一些功能，让不同环境下的多个用户可以通过多个浏览器连接。每一章都将讲述 WebRTC 过程的一个部分，并讲解得更深入，包括一些常见的缺点，并将处理一些极端情况，比如网络的安全性和稳定性。

下一章，我们将开始建立信令服务器来连接远程用户。本书的剩余部分都将使用这个信令服务器。我们也能用它来建立第一个真正的呼叫应用，类似 Google Hangouts。



由 扫描全能王 扫描创建

# 4

## 创建信令服务器

创建一个完整的 WebRTC 应用，有些时候你需要抛开客户端的开发，转而进行服务器端的开发。大部分 WebRTC 应用并不仅仅用于视频和音频交流，通常需要用到功能来让应用变得有趣。在这一章里，将使用 JavaScript 和 Node.js 来深入服务器端编程。我们将创建一个有基本功能的信令服务器，在剩余的章节里将使用这个服务器。

在这一章里，我们将涵盖以下主题：

- 搭建 Node.js 的开发环境
- 使用 WebSockets 连接客户端
- 识别用户
- 发起和应答 WebRTC 通话
- 处理 ICE 候选路径的传送
- 挂断通话

在这一章中，我们将关注点完全放在应用的服务器部分。下一章，我们将创建这个例子的客户端部分。这个实例服务器只有最基础的功能，但对于创建 WebRTC 对等连接来说绰绰有余。

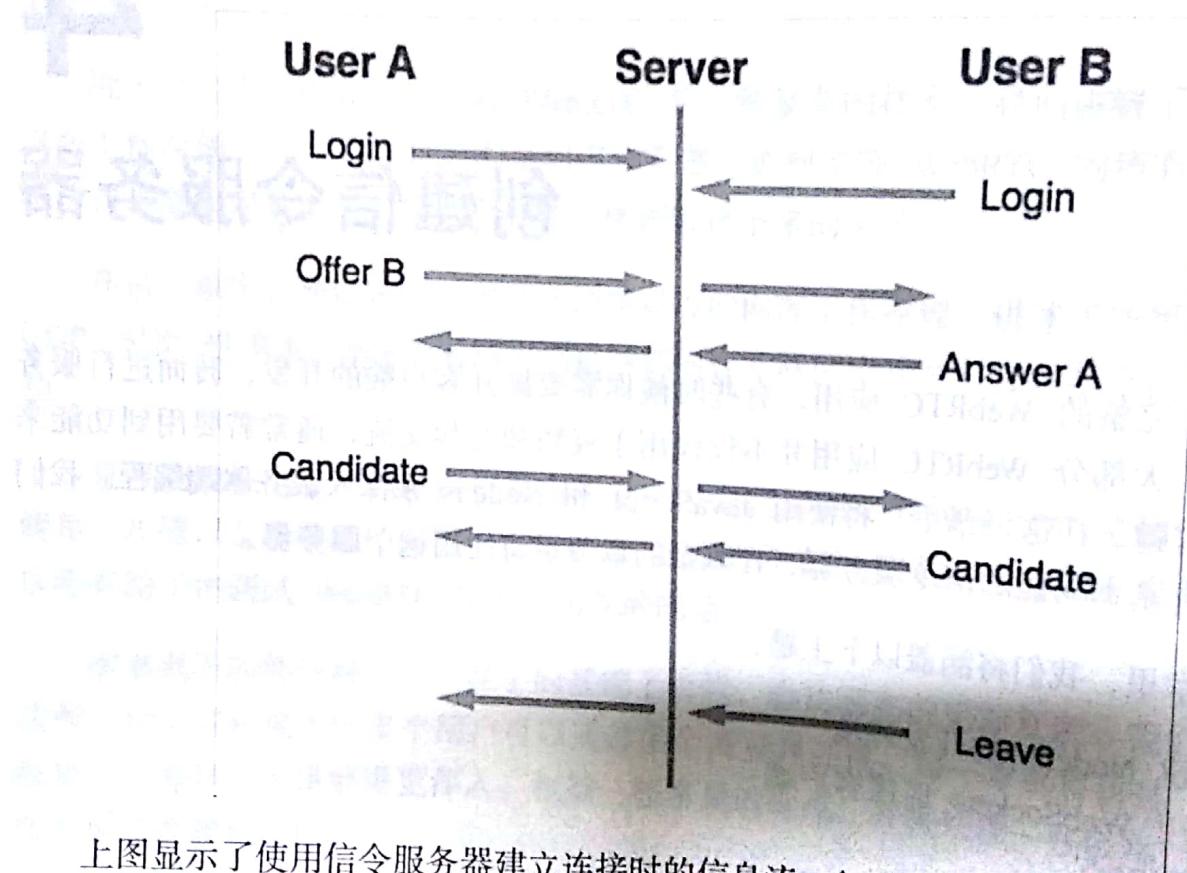
### 构建信令服务器

即将创建的服务器能帮助我们将不在同一个电脑中的两个用户连接起来。此服务器的目的是通过网络传输替代原先的信令机制。该服务器的实现简洁明了，只支持 WebRTC 连接最基础的功能。



由 扫描全能王 扫描创建

我们的实现将对多个用户做出回应。这将通过一个简单的双向通信系统实现。它允  
一方用户呼叫另一个用户从而在双方间建立 WebRTC 连接。一旦用户呼叫了另一方，服  
务器将会在双方间传递请求，应答和 ICE 候选路径。这将让用户成功地建立起一个 WebRT  
C 连接。



上图显示了使用信令服务器建立连接时的信息流。每一方都将从登录服务器开始。登录仅仅只是向服务器端发送一个字符串形式的用户标识，并确保此标识码没有被使用。一旦双方都登录到服务器，他们便可呼叫另一方。通过使用对方的标识码发送请求即可。一方用户应答，最后，双方会发送候选地址直到他们能成功建立连接为止。任何时刻，用户都可通过发送离开信息来终止连接。这个实现是简单的，它主要用来作为互相发送信息的通道。



请记住这只是信令服务器的一个例子罢了，由于发送信令的实现并没有任何规则，你可以使用你喜欢的任意协议、技术或模式！



由 扫描全能王 扫描创建

# 搭建开发环境

我们将使用 Node.js 来搭建服务器。如果你没有 Node.js 的编程经验，请不用担心！这项技术使用 JavaScript 引擎来工作。这意味着我们将用 JavaScript 来编程，没有新的语言要学。现在让我们通过以下步骤来搭建 Node.js 环境：

1. 运行 Node.js 服务器的第一步是安装 Node.js。

 关于如何安装，你可以访问 <http://nodejs.org> 来获得更多信息。每一个操作系统下都有多个版本，选择最适合你的那个。在写这本书的时候，Node.js 最新的版本是 v0.12.4。

2. 现在打开你的终端并使用 node 命令来运行 Node.js 虚拟机。Node.js 基于 Google Chrome 的 V8 JavaScript 引擎。这意味着它的工作方式十分接近于浏览器编译 JavaScript 的方式。通过输入一些命令来了解它如何工作：

```
>1+ 1  
2  
> var hello = "world";  
undefined  
> "Hello" + hello;  
'Helloworld'
```

3. 现在可以开始创建服务器程序了。幸运的是，Node.js 可以像终端命令行那样运行 JavaScript 文件。用以下的内容创建一个 index.js 文件并使用 node index.js 命令来运行：

```
console.log("Hello from node!");
```

当你运行 node index.js 命令时，你可以在 Node.js 终端看到下面的输出：

```
Hello from node!
```

之后我们将不再提及 Node.js 的概念。我们实现的信令服务器并不是最高级的，因为深入讲解服务器编程需要一整本书。随着我们的深入，请多花些时间学习 Node.js 或者将其转换为你最喜欢的编程语言！



由 扫描全能王 扫描创建

## 获得一个连接

建立 WebRTC 连接所需的步骤必须都是实时的。这意味着客户端不能使用 WebRTC 对等连接在双方间实时传输信息。这就需要使用另一个 HTML5 的强大功能——WebSockets。

WebSockets 就像它听上去的那样——是在两个终端间——浏览器和网络服务器间的一个双向 socket 连接。你可以利用 socket 以字符串和二进制码方式双向发送信息。浏览器和网络服务器都需要实现此功能方可再不使用 AJAX 请求的情况下实现双方间的沟通。

WebSocket 协议起始于 2010 年，它是一个严格定义的标准，现在大部分的浏览器都已实现了此协议。它在跨浏览器方面支持性很好，有许多服务器技术框架已开始使用它们。甚至已有完全依赖 WebSocket 技术的框架，比如 Meteor JavaScript framework。

WebSocket 协议和 WebRTC 协议最大的不同是在 TCP 堆栈的使用上。WebSockets 被设计成是一个客户端到服务器端使用 TCP 传输协议来建立可靠连接的协议。这意味着它有着许多 WebRTC 没有的瓶颈，我们已经在第 3 章理解 UDP 传输协议和实时传输部分讲解过。这也是它为什么作为一个信令传输协议非常好的原因。由于它的可靠性，信令很少会丢失，这让我们可以获得更稳定的连接。它被集成在了浏览器里，因此使用 Node.js 来搭建非常方便，这让信令服务器的实现更容易理解。

在项目中使用 WebSockets，必须先为 Node.js 安装一个支持 WebSockets 的库。我们将使用 npm 上的 ws 项目。安装这个库，请去服务器的目录下执行以下命令：

npm install ws

你将看到如下输出：



由 扫描全能王 扫描创建

```

dan:webrtc-book/ (master) $ npm install ws
npm http GET https://registry.npmjs.org/ws
npm http 304 https://registry.npmjs.org/ws
npm http GET https://registry.npmjs.org/tinycolor
npm http GET https://registry.npmjs.org/options
npm http GET https://registry.npmjs.org/commander
npm http GET https://registry.npmjs.org/nan
npm http 304 https://registry.npmjs.org/tinycolor
npm http 304 https://registry.npmjs.org/options
npm http 200 https://registry.npmjs.org/commander
npm http 200 https://registry.npmjs.org/nan

> ws@0.4.31 install /Users/dan/workspace/webrtc-book/node_modules/ws
> (node-gyp rebuild 2> builderror.log) || (exit 0)

CXX(target) Release/obj.target/bufferutil/src/bufferutil.o
SOLINK_MODULE(target) Release/bufferutil.node
SOLINK_MODULE(target) Release/bufferutil.node: Finished
CXX(target) Release/obj.target/validation/src/validation.o
SOLINK_MODULE(target) Release/validation.node
SOLINK_MODULE(target) Release/validation.node: Finished
ws@0.4.31 node_modules/ws
├─ tinycolor@0.0.1
├─ options@0.0.5
├─ commander@0.6.1
└─ nan@0.3.2

```

 npm 是 Node.js 的包管理工具。它存放一些开源框架，任何人都可以下载并使用它们。想要了解更多，请去 <https://www.npmjs.org/> 查看更多内容。

现在已经安装了 WebSocket 库，可以在服务器中使用它了，在 index.js 文件中插入以下代码：

```

var WebSocketServer = require("ws").Server,
wss = new WebSocketServer({ port: 8888 });

wss.on("connection", function (connection) {
  console.log("User connected");

  connection.on("message", function (message) {
    console.log("Got message:", message);
  });
});

```



```

    });
    connection.send("Hello World");
});

```

第一行代码获取我们在之前命令中安装的 WebSocket 库。然后创建 WebSocket 服务器，告诉它连接监听哪个端口。如有需要，你可以更改端口号。

下一步，监听服务器端的 connection 事件。当用户与服务器建立 WebSocket 连接时这段代码将被调用。它会给你一个连接对象，这个对象有连接方的所有信息。

之后监听用户发送的所有消息。现在，我们仅将这些消息在控制台打印出来。

最后，向客户端发送一个 Hello World 的回应。这将在服务器与客户端建立 WebSocket 连接后即刻发送。

请注意当有任何用户连接到服务器时，connection 事件就会被触发。这意味着同时可以有多个用户连接同一个服务器且他们都将各自触发 connection 事件。这种基于异步的代码在 Node.js 里很常见，是其优势之一。

现在通过运行 node index.js 来启动服务器。服务器应已启动并且等待着处理 WebSocket 连接。它将持续运作直到你停止此过程。

## 测试我们的服务器

为了测试代码能正常工作，我们可以使用 ws 库里的 wscat 命令。npm 的一个好处是你能仅在你的应用里安装库，也可以全局安装然后使用命令行工具。通过运行 npm install -g ws 就可全局安装，全局安装时你需要使用管理员权限来运行这行命令。

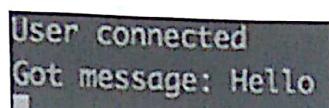
该库提供了一个新的命令 wscat。这个工具让我们能够直接在命令行连接 WebSocket 服务器来测试它们。在终端窗口中运行服务器，然后再新开一个窗口运行 wscat -c ws://localhost:8888 命令。你会发现 ws:// 是 WebSocket 协议的自定义指令而非 HTTP。你的输出应该与此类似：



由 扫描全能王 扫描创建

```
dan:webrtc-book/ (master) $ wscat -c ws://localhost:8888
connected (press CTRL+C to quit)
< Hello World
>
```

你的服务器应该也会在控制台记录这个连接。



如果这两种情况没有发生，那么照着列表检查一遍代码，并阅读 ws 库、Node.js 和 npm 的文档。这些工具在不同的环境下可能会不同，某些情况下需要一些额外的工作。如果所有都能正常工作，那么请自我鼓励一下，你只用 12 行代码就写了一个 WebSocket 服务器哦。

## 识别用户

在一个典型的网络应用中，服务器需要一种方法来识别连接的用户。今天大部分的应用都遵循唯一规则，让每一个用户有一个字符串形式的标识，即用户名。在我们发送信令的应用中，也将遵循此规则。我们的实现不会像现实使用的方法那般复杂，甚至都不需要用户输入密码。对每一个连接我们仅需一个 ID 来标识，这样我们就能知道往哪发送消息。

我们先从修改 connection 处理器开始，它应当与此相似：

```
connection.on("message", function (message) {
  var data;
  try {
    data = JSON.parse(message);
  } catch (e) {
    console.log("Error parsing JSON");
    data = {};
  }
});
```

这将更改 WebSocket 的实现让它只接受 JSON 格式的消息。由于 WebSocket 连接只允许字符和二进制数据，我们需要一种方法来传送结构化的数据。JSON 允许我们定义结



构化的数据并将之序列化为字符串，从而可以通过 WebSocket 连接传送。这也是 JavaScript 中最简单的序列化形式。

接下来，需要找个方法来存储所有连接的用户。由于我们的服务器只有最简单的功能，我们将使用哈希图来存储数据，它在 JavaScript 中也叫对象。我们可以将文件顶部改成这样：

```
var WebSocketServer = require("ws").Server,
    wss = new WebSocketServer({ port: 8888 }),
    users = {};
```

当用户发送了 login 类型的信息才能登录。为了支持此功能，我们将为客户端发送的每一个信息增加 type 字段。这样服务器就能知道该如何处理收到的数据了。首先，我们定义用户试图登录时需要执行的动作：

```
connection.on("message", function (message) {
    var data;

    try {
        data = JSON.parse(message);
    } catch (e) {
        console.log("Error parsing JSON");
        data = {};
    }

    switch (data.type) {
        case "login":
            console.log("User logged in as", data.name);
            if (users[data.name]) {
                sendTo(connection, {
                    type: "login",
                    success: false
                });
            } else {
                users[data.name] = connection;
                connection.name = data.name;
                sendTo(connection, {
```



```

        type: "login",
        success: true
    });
}

break;
default:
    sendTo(connection, {
        type: "error",
        message: "Unrecognized command: " + data.type
    });
}

break;
}
});

```

使用 switch 语句就消息的类型进行分别处理。如用户发送 login 类型的消息，首先需要确定有没有其他用户已经使用此 ID 登录到服务器了。如果有，通知用户登录失败，需要使用一个新名字。如果没有人使用这个 ID，将此 ID 作为用户对象的属性名来存放连接对象。如果没有识别出消息的类型，我们会向客户端发送消息告知在处理请求时发生了错误。

我也添加了一个叫 sendTo 的帮助函数来负责向连接发送消息。它可以加在文件的任何地方：

```

function sendTo(conn, message) {
    conn.send(JSON.stringify(message));
}

```

此函数确保所有的消息都以 JSON 格式编码。这也有助于减少代码数量。将消息的发送放在一个地方是最佳实践，以防向客户端发送消息时有其他变更。

最后当连接断开时，我们需要提供一个方法来收尾。幸运的是，当断开发生时我们的库提供了一个事件。我们可以监听此事件，并通过以下方法删除用户。

```

connection.on("close", function () {
    if (connection.name) {
        delete users[connection.name];
    }
});

```



```

    }
});

```

这段应当被当作消息处理器加在 connection 事件里。

现在可以使用 login 命令来测试我们的服务器了。像之前那样，使用客户端测试 login 命令。请记住，为了被服务器接受，现在发送的消息已被编码成 JSON 格式。

当连接成功后，我们可以向服务器发送以下消息：

```
{ "type": "login", "name": "Foo" }
```

你收到的输出应与此类似：

```

dan@webrtc-book/ (master) $ wscat -c ws://localhost:8888
connected (press CTRL+C to quit)
> { "type": "login", "name": "Foo" }
< {"type":"login","success":true}
>

```

## 发起通话

从现在起，将不会有比 login 处理器更复杂的代码了。我们将创建一系列处理器正确地为每一个步骤传递消息。offer 处理器是登录后的首批通话之一，它意味着有用户想要呼叫另一方。

将呼叫的初始化过程与 WebRTC 的 offer 步骤分开是一个好主意。在这个例子中，我们把这两步放在一起，让我们的 API 更简单。但在大部分的设置中，这两个步骤是分开的。它常见于一些应用中，比如 Skype，另一方用户必须接受呼叫才能建立连接。

现在将 offer 处理器加到代码中：

```

case "offer":
  console.log("Sending offer to", data.name);
  var conn = users[data.name];
  if (conn != null) {
    connection.otherName = data.name;
  }
}

```



```

    sendTo(conn, {
      type: "offer",
      offer: data.offer,
      name: connection.name
    });
  }
}

break;
}

```

首先获取试图呼叫的用户的 `connection` 对象。由于我们已将 `connection` 对象存放在用户对象的 `ID` 属性中，这将十分简单。之后我们检查另一用户是否存在与服务器上，若存在则向他们发送 `offer` 的细节。我们也为用户的 `connection` 对象添加一个 `otherName` 属性，便于之后能方便地获取。你可能也注意到了，以上的代码并非 WebRTC 专用。它可能用于任何双方间的呼叫技术。我们将在这章后半部分细说。

你可能也发现缺少了错误处理。这可能是 WebRTC 最冗长乏味的部分。在这个过程中，由于呼入可能在任何地方失败，我们的连接也可能在任何地方失败。失败的原因也多种多样，比如网络的可用性、防火墙问题和其他更多的问题。在本书中，我们将让用户以他们喜欢的方式来分别处理各种错误情况。

## 呼应应答

应答就跟发送 `offer` 一样容易。我们按照相似的模式，让客户端做大部分工作。我们的服务器仅将消息作为 `answer` 传递给另一方。把以下代码加在 `offer` 处理器后：

```

case "answer":
  console.log("Sending answer to", data.name);
  var conn = users[data.name];
  if (conn != null) {
    connection.otherName = data.name;
    sendTo(conn, {
      type: "answer",
      answer: data.answer
    });
  }
}

```



```
break;
```

你可以发现这些代码跟之前的十分相似。请注意，我们也依赖来自另一方的 answer。服务器在许多方面都不足够，但应付到下一章节还是绰绰有余的。

对 WebRTC 的 offer 和 answer 机制来说这是一个好的开始。你可以发现它遵循了 RTCPeerConnection 的 createOffer 和 createAnswer 功能。这是我们的服务器连接处理远程客户端的开始。

我们能使用之前使用过的 WebSocket 客户端来测试我们的实现。同时连接两个客户端让我们可以在这两者之间传递 offer 和回应。这将给你更多的洞察力来了解它是怎样工作的。通过在终端窗口同时运行两个客户端可以看到结果，截图如下：

The screenshot shows two terminal windows side-by-side. Both windows have the command 'wschat -c http://localhost:8888' at the top. The left window shows the initial connection and a message from UserA: 'connected (press CTRL+C to quit) > { "type": "login", "name": "UserA" }'. The right window shows UserB's response: '[2:03:18] | connected (press CTRL+C to quit) | > { "type": "login", "name": "UserB" } | < { "type": "login", "success": true } | < { "type": "offer", "offer": "Hello", "name": "UserA" } | > { "type": "answer", "name": "UserA", "answer": "Hello to you too!" } | >'.

在这个例子里，我的 offer 和 answer 都是简单的字符串消息。如果你还记得第 3 章创建简单的 WebRTC 应用的 WebRTC API 部分，我们详细讲述了会话描述协议( SDP )。当创建一个 WebRTC 呼叫时，这就是 offer 和 answer 字符串将实际被填入的内容。如果你不记得 SDP 是什么，请翻到第 3 章重温。

## 处理 ICE 候选路径

WebRTC 信令的最后一部分是在用户间处理 ICE 候选路径。这里，我们使用之前的技术在用户间传递消息。此类消息的不同在于每一个用户可能都需要发送多次且在双方用户间会以任何顺序发送。还好，我们的服务器能以一种简单的方式解决这个问题。在文件里加上此 candidate 处理器：

```
case "candidate":
    console.log("Sending candidate to", data.name);
    var conn = users[data.name];
```



```

    if (conn != null) {
        sendTo(conn, {
            type: "candidate",
            candidate: data.candidate
        });
    }
}

break;

```

由于通信已经建立，我们不需要在这个函数里添加另一方用户的名字。现在请用 WebSocket 客户端测试一下。它类似于 offer 和 answer 函数，在双方间传递消息。

## 呼叫挂断

我们最后的一点内容并不涉及 WebRTC 规范，但拥有总是好的——挂断。这可以让我们从另一方断开，这样他们就可以呼叫其他用户了。这也会通知我们的服务器断开用户引用。你可以加入 leave 处理器，如下所示：

```

case "leave":
    console.log("Disconnecting user from", data.name);
    var conn = users[data.name];
    conn.otherName = null;

    if (conn != null) {
        sendTo(conn, {
            type: "leave"
        });
    }
}

break;

```

这段代码也会通知另一个用户 leave 事件的触发，这样他们可以相应地断开对等连接。当用户从信令服务器掉线时我们也需要做相应的处理。这意味着我们不再提供服务，需要结束通话。可以将之前使用的 close 处理器更改为如下：

```
connection.on("close", function () {
```



```

        if (connection.name) {
            delete users[connection.name];

            if (connection.otherName) {
                console.log("Disconnecting user from",
                    connection.otherName);
                var conn = users[connection.otherName];
                conn.otherName = null;

                if (conn != null) {
                    sendTo(conn, {
                        type: "leave"
                    });
                }
            }
        );
    }
}

```

如果用户碰巧意外地与服务器断开连接，上述代码将断开与用户的连接。如果我们还处于发送 offer、answer 或 candidate 状态但对方用户已关闭浏览器窗口，这将非常有帮助。在这个场景下，WebRTC API 将不再发送事件，我们需要另一种方法来知道用户已经离开了。让信令服务器处理这种场景能让我们的应用更可靠和稳定。

## 完成信令服务器

以下是信令服务器的完整代码。这包括登录和处理所有响应类型。我在最后也增加了一个监听处理器，当服务器已准备好接受 WebSocket 连接时，它将通知用户：

```

var WebSocketServer = require("ws").Server,
wss = new WebSocketServer({ port: 8888 }),
users = {};

wss.on("connection", function (connection) {
    connection.on("message", function (message) {
        var data;

```



```

try {
    data = JSON.parse(message);
} catch (e) {
    console.log("Error parsing JSON");
    data = {};
}
switch (data.type) {
    case "login":
        console.log("User logged in as", data.name);
        if (users[data.name]) {
            sendTo(connection, {
                type: "login",
                success: false
            });
        } else {
            users[data.name] = connection;
            connection.name = data.name;
            sendTo(connection, {
                type: "login",
                success: true
            });
        }
        break;
    case "offer":
        console.log("Sending offer to", data.name);
        var conn = users[data.name];
        if (conn != null) {
            connection.otherName = data.name;
            sendTo(conn, {
                type: "offer",
                offer: data.offer,
                name: connection.name
            });
        }
        break;
    case "answer":

```



```

        console.log("Sending answer to", data.name);
        var conn = users[data.name];

        if (conn != null) {
            connection.otherName = data.name;
            sendTo(conn, {
                type: "answer",
                answer: data.answer
            });
        }
        break;
    case "candidate":
        console.log("Sending candidate to", data.name);
        var conn = users[data.name];

        if (conn != null) {
            sendTo(conn, {
                type: "candidate",
                candidate: data.candidate
            });
        }
        break;
    case "leave":
        console.log("Disconnecting user from", data.name);
        var conn = users[data.name];
        conn.otherName = null;
        if (conn != null) {
            sendTo(conn, {
                type: "leave"
            });
        }
        break;
    default:
        sendTo(connection, {

```



```

        type: "error",
        message: "Unrecognized command: " + data.type
    });
}

break;
}
});

connection.on('close', function () {
    if (connection.name) {
        delete users[connection.name];
    }

    if (connection.otherName) {
        console.log("Disconnecting user from",
            connection.otherName);
        var conn = users[connection.otherName];
        conn.otherName = null;
        if (conn != null) {
            sendTo(conn, {
                type: "leave"
            });
        }
    }
});
});

function sendTo(conn, message) {
    conn.send(JSON.stringify(message));
}

wss.on('listening', function () {
    console.log("Server started...");
});

```





你可能注意到了我们在例子中使用的是不安全的 WebSocket 服务。实际应用中，WebSocket 协议支持 SSL，这有些类似 HTTP 支持 HTTPS。你可以在连接到服务器的时候通过简单地使用 `wss://` 来开启这个功能。

请随意使用 WebSocket 客户端来测试服务器应用。你甚至可以试着用三四个或更多的用户连接服务器，看它是如何处理多个连接的。你可能会发现有许多我们的服务器不能处理的场景。发现这些场景是好事，可以改进服务器来处理它们。

## 在实际应用中发送信令

我们花了许多时间来建立一个基本的信令服务器，用于连接两个 WebRTC 用户。此时，你可能会好奇现实世界的产品是如何搭建信令服务器的。由于信令是一个十分抽象的概念，在 WebRTC 规范里也没有被定义，答案是怎样创建都行。因为它的“怎样都行”，信令是一个十分复杂和难解的问题。WebRTC 的创建者提供了许多的资源，但没有一个能确切详细地指出信令该如何实现最好。这里有许多的问题需要解决，每一个场景中并不是所有的都是一样的。有些开发者可能需要一个较高扩展性的解决方案来让全球数百万用户连接。另一些开发者可能需要一个可以与 Facebook 集成的方案，还有一些可能需要与 Twitter 集成。这是一个极有挑战性的话题，需要许多的时间和研究来找到最佳方案。这里，我们将详细阐述连接信令服务器时一些最为常见的缺陷和它们的解决方案。

## WebSockets 的困境

WebSockets 的好处是它为浏览器带来了双向通信。许多人认为 WebSockets 是他们所有问题的解决方案，直接连接服务器让 socket 连接更快。这样说吧，WebSockets 还有一些问题需要解决。

其中一个便是网络防火墙问题。在理想情况下，WebSockets 是可靠的，但不像 HTTP 副本那样，WebSockets 在代理设置的情况下很容易变得不稳定。虚拟专用网络（VPN）的额外开销或复杂的防火墙系统可导致连接的成功率急剧下降。这意味着你将回到使用其他的技术，比如 HTTP 流，来完成同样的工作。



这将引起 WebRTC 的紊乱。网络通道的延时能导致消息处理混乱，连接 WebRTC 时效果很差。请记住当创建一个 WebRTC 连接时，顺序是十分重要的，如果没有按照正确的步骤会导致连接失败。

除此之外，WebSockets 是一个伟大的技术。这个故事的中心是当创建一个真实的产品时，使用 WebSockets 做信令服务器技术会有停顿。许多公司高效地使用它们，但因网络状况 WebSockets 不能工作时也提供了备选方案。

## 连接其他服务

WebRTC 最令人激动的特点之一是它不但可作为一个单独的解决方案，也可与其他技术一起使用。在 WebRTC 出现之前就已经有许多对等连接应用了，由于它的出现，人们做了许多努力来让 WebRTC 向后兼容。这包括我们现今使用的一些常见的即时通信系统和手机所使用的工作。

### XMPP

XMPP 是一个即时通信协议，追溯到 20 世纪 90 年代，它叫 Jabber。这个协议的目的在于定义一个常用的方法来实现即时通信、用户在线和通讯录。它是一个开源标准，所有人都可以使用并将其合成到他们的应用中。有许多的大型即时通信平台在某一时刻将 XMPP 集成到了他们的服务中，包括 Google Talk、Facebook Chat 和 AOL Instant Messenger。

正是这大量的历史数据让 XMPP 是一个易用的平台。由于许多视频和音频通信平台在某些时候需要上线状态和通讯录信息，它给了任何典型的 WebRTC 应用无穷的力量。除此以外，它很安全，拥有很多的文档，实现起来特别灵活。有许多针对 JavaScript 和 WebRTC 的端口，也有许多公司致力于提供基于 WebRTC 的 XMPP 服务。如果你能让 XMPP 工作，这会比我们在这章创建的简单的信令服务器要好很多。

### 会话初始协议

会话初始协议（SIP）是另一个 20 世纪 90 年代使用的协议。它是一个致力于移动网络和电话系统的信令协议。它是一个被大部分移动网络和网络设备提供者使用的定义严格的支持广泛的协议。



SIP 和 WebRTC 集成的目的是为不支持 WebRTC 的基于 SIP 的电话设备提供通信支持。如果我们与一个可以翻译我们信息的服务器连接，与移动电话或其他通信设备的连接也将十分容易。如果它使用 SDP，它也会支持现今电话所拥有的许多功能。

SIP 本身就是另一个巨大的话题。你可以在网上找到数不清的有关 SIP 和 WebRTC 集成的资源。与 XMPP 截然相反，就困难和复杂而言这是另一个极端。基于电话的通信是完全不同的一个话题，它有自己的技术和标准。本书不会阐述这个集成，但你可以自己寻找一些信息。

## 自测题

Q1: 信令服务器的目的在于连接两个在不同网络的用户，这样他们可以建立对等连接。对或错？

Q2: WebSockets 使用哪种技术在客户端和服务期间建立双向连接？

1. UDP
2. TCP
3. ICE
4. STUN

Q3: 使用 JSON 用于向服务器端传送消息给了我们以下哪个好处？

1. 易于传输的基于字符串的数据包
2. 在消息内定义结构复杂数据
3. 广泛支持的编码和解码方法
4. 以上所有

Q4: 在信令中，操作的顺序是 offer, answer，然后互相发送候选直到连接成功。对或错？



由 扫描全能王 扫描创建

## 小结

在这章里，我们阐述了信令过程的每一个步骤。我们经过了创建 Node.js 应用，识别用户，在用户间发送 offer/answer。我们也详细讲述了断开连接、退出连接和在用户间发送候选。

你现在应该对信令服务器如何工作有了扎实的理解。我们创建的服务器是简单直观的，仅供学习使用。我们可以通篇讲解那些可以加入服务器的新功能特性，比如验证、通讯录等。如果你有兴趣，请随意在我们的实现里添加任何你想添加的功能。

我们也略微阐述了一些现实世界的信令应用。就信令的信息而言，这些只是冰山一角。WebRTC 信令已有大量的场景和实现了。我的建议是尽可能早地收集你的需求并坚持以最简单的方案来达到你所有的需求。这有可能是任何事，从一个简单的 WebSocket 服务器到最复杂的 SIP 实现。

在下一章中，我们将把我们的服务器与一个真实的 WebRTC 客户端集成。这使得我们能在不同地址的用户之间建立 WebRTC 连接。对那些能将全世界的人们连接起来的成熟应用来说，这只是一个开端。



由 扫描全能王 扫描创建

# 5

## 把客户端连接到一起

现在信令服务器已经开发好，是时候利用它来创建一个应用了。在本章中，我们将创建一个客户端应用，实现让两个用户通过 WebRTC 来进行实时通信。在本章的最后，我们将会做出一个设计精良且可正常工作的示例，它提供大部分 WebRTC 应用所拥有的功能。

在本章中，我们将涵盖以下主题：

- 从客户端获取到服务器的连接
- 识别各个连接端的用户
- 两个远程用户发起通话
- 结束通话

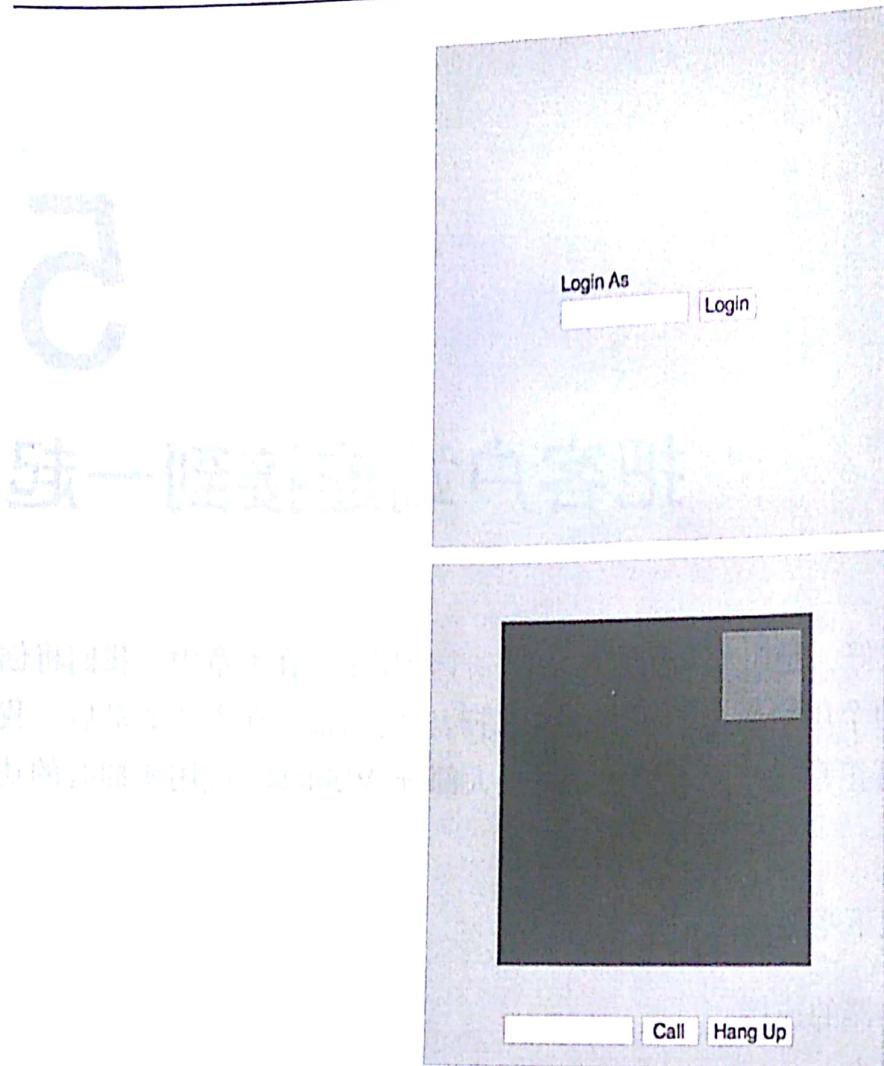
如果你还未完成第 4 章，现在正是时候回去学习。本章的学习将以在第 4 章搭建的服务器为基础，因此你必须知道如何在本地搭建和运行服务器。

### 客户端应用

客户端应用的目标是让不同地方的用户互相连接并进行通信。这类程序被视为 WebRTC 程序的 hello world 版，网上或 WebRTC 的技术会议中有大量的例子。你很可能之前已使用过类似的应用。



由 扫描全能王 扫描创建



我们将创建的应用包含两个页面：一个页面用来输入用户名，另一个页面用来呼叫其他用户。记住，页面本身很简单，我们主要关注如何实现 WebRTC 的功能。在开发应用前，我们先来看一下原始的线框模型。

这个应用并没有什么复杂的地方。这两个页面就是些 `div` 标签，我们将使用 JavaScript 来切换它们。而且大部分的输入将使用简单的事件处理器来处理。如果你已经具备 HTML5 和 JavaScript 编程基础，本章的代码对你来说会非常简单。

接下来将关注客户端与信令服务器的集成。我们将在两个不同的页面传递 `RTCPeerConnection` 对象（详见第 3 章 WebRTC API 一节）。一种测试方式是在浏览器的两个标签页中分别打开这个程序页面，并让它们相互进行通话。



由 扫描全能王 扫描创建

## 创建页面

首先，我们需要创建一个简单的 HTML 页面。以下是模板文件，复制下面的代码到你的 index.html 文件中：

```
... html
<html lang="en">
  <head>
    <meta charset="utf-8" />

    <title>Learning WebRTC - Chapter 5: Connecting Clients Together</title>

  <style>
    body {
      background-color: #3D6DF2;
      margin-top: 15px;
      font-family: sans-serif;
      color: white;
    }

    video {
      background: black;
      border: 1px solid gray;
    }

    .page {
      position: relative;
      display: block;
      margin: 0 auto;
      width: 500px;
      height: 500px;
    }

    #yours {
      width: 150px;
      height: 150px;
    }
  </style>
</head>
<body>
  <video id="yours" ...></video>
</body>
</html>
```



```
position: absolute;
top: 15px;
right: 15px;
}

#theirs {
width: 500px;
height: 500px;
} </style>
</head>
<body>
<div id="login-page" class="page">
<h2>Login As</h2>
<input type="text" id="username" />
<button id="login">Login</button>
</div>

<div id="call-page" class="page">
<video id="yours" autoplay></video>
<video id="theirs" autoplay></video>
<input type="text" id="their-username" />
<button id="call">Call</button>
<button id="hang-up">Hang Up</button>
</div>

<script src="client.js"></script>
</body>
</html>
```

到目前为止，这部分代码看上去比较容易。我们使用 `div` 标签来标识两个页面，并通过 `display` 属性来控制它们的隐藏显示。此外，我们创建了几个按钮和输入框来让用户进行输入。最后，使用两个 `video` 元素来分别显示你和远程用户的视频流。你也可以添加一些 CSS 来让这些页面看上去与众不同一些。





别忘了使用静态 web 服务器来浏览这些页面。如果直接在浏览器打开这个页面，由于安全限制，大部分浏览器将不允许你创建 WebRTC 连接。

## 获取一个连接

第一件要做的就是与信令服务器创建连接。我们在第 4 章中创建的信令服务器完全基于 WebSocket 协议。使用这个技术的好处是，我们不需要依赖其他的类库去连接服务器。它仅使用现在大部分最新的浏览器都支持的 WebSocket 的内置功能。我们可以直接创建一个 WebSocket 对象，即刻就能连接到服务器。



使用 WebSockets 的另一个好处是，它的标准也是非常完善的。现在的大部分浏览器，使用时不需要去检测和添加前缀，它会自动帮你实现这些功能。然而，你还是需要检查浏览器的最新文档。

我们可以从创建 HTML 页面中需要使用到的 `client.js` 文件开始。你可以添加以下用于连接的代码：

```
var name,
    connectedUser;

var connection = new WebSocket('ws://localhost:8888');

connection.onopen = function () {
    console.log("Connected");
};

// 通过回调函数处理所有的消息
connection.onmessage = function (message) {
    console.log("Got message", message.data);
    var data = JSON.parse(message.data);
}
```



```

    switch(data.type) {
        case "login":
            onLogin(data.success);
            break;
        case "offer":
            onOffer(data.offer, data.name);
            break;
        case "answer":
            onAnswer(data.answer);
            break;
        case "candidate":
            onCandidate(data.candidate);
            break;
        case "leave":
            onLeave();
            break;
        default:
            break;
    }
};

connection.onerror = function (err) {
    console.log("Got error", err);
};

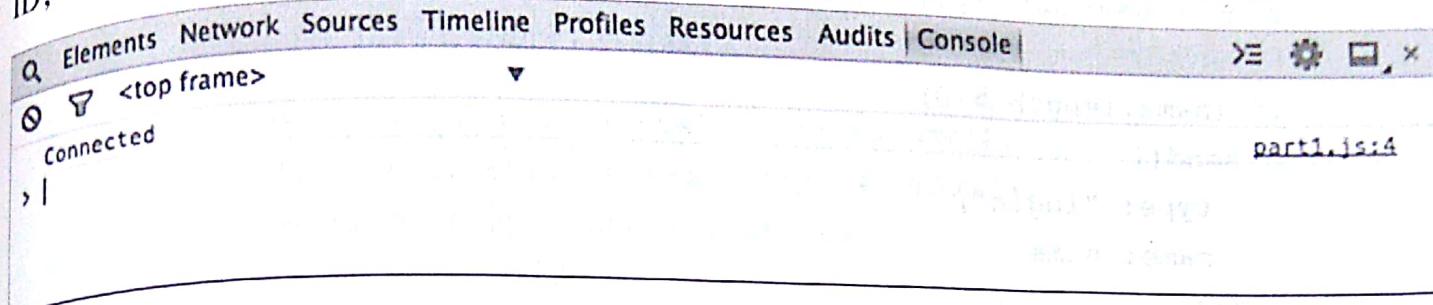
// Alias 以JSON格式发送消息
function send(message) {
    if (connectedUser) {
        message.name = connectedUser;
    }
    connection.send(JSON.stringify(message));
}

```

我们首要做的是建立与服务器的连接。通过传递服务器地址，再加上在 ws:// 协议作为前缀来实现。然后，设置一系列的事件处理器。最需要关注的是 onmessage 方法，通过它来获取所有基于 WebRTC 的消息。switch 方法会根据消息类型来调用不同的函数，



这些函数将在本章的剩余部分补充。最后，创建一个简单的 send 方法，它将自动在消息中添加另一用户的 ID，并对消息编码。我们还设置了一些变量来保存用户名和另一用户的 ID，以便后面使用。如果你现在打开这个文件，应该会看到一条简单的连接消息：



WebSocket API 是创建实时应用的基础。正如在本章中所看到的，它使我们能够实时在浏览器和服务器之间来回发送消息。我们不仅可以用它来传输信令数据，同样也可以传输一些其他的消息。像多人在线游戏、股市行情等许多不同类型的网站都使用了 WebSocket 技术。

## 登录到应用程序

使用唯一的用户名登录是与服务器的首次交互行为。它用来识别我们的身份，同时给别的用户一个唯一的标识来呼叫我们。要这样做，我们只需要发送一个用户名给服务器，它会告诉我们这个用户名是否已经被使用了。在这个应用中，我们允许用户选择任何他喜欢的用户名。

要实现这个，需要在我们的应用脚本文件中增加一点功能。你可以添加下面的代码到 JavaScript 中：

```
var loginPage = document.querySelector('#login-page'),
    usernameInput = document.querySelector('#username'),
    loginButton = document.querySelector('#login'),
    callPage = document.querySelector('#call-page'),
    theirUsernameInput = document.querySelector('#their
username'),
    callButton = document.querySelector('#call'),
    hangUpButton = document.querySelector('#hang-up');

callPage.style.display = "none";
```



```

    // 单击按钮登录
    loginButton.addEventListener("click", function (event) {
        name = usernameInput.value;

        if (name.length > 0) {
            send({
                type: "login",
                name: name
            });
        }
    });

    // 准备好通话的通道
    startConnection();
}

};


```

首先，选择一些页面上的元素，以便和用户进行交互，并通过各种方式反馈给用户。然后让 callPage 区域隐藏，这样就只有登录区域呈现给用户。接着，给 Login 按钮添加一个事件，当用户单击时，我们会发送一个消息，通知服务器用户需要登录。最后，我们实现了在前面的消息回调中调用的 onLogin 方法。如果登录成功，这个应用将显示 callPage 区域，并且进行一些必需的设置，来创建一个 WebRTC 连接。

## 开始一个对等连接

startConnection 方法是任何 WebRTC 连接的第一步。由于整个过程不依赖任何其他用户的连接，在用户试图和其他用户进行通话之前，可以提前设置这一步。所含步骤如下：



由 扫描全能王 扫描创建

1. 从相机中获取视频流
2. 验证用户的浏览器是否支持 WebRTC
3. 创建 RTCPeerConnection 对象

下面的 JavaScript 代码对此进行了实现：

```

var yourVideo = document.querySelector('#yours'),
    theirVideo = document.querySelector('#theirs'),
    yourConnection, connectedUser, stream;

function startConnection() {
    if (hasUserMedia()) {
        navigator.getUserMedia({ video: true, audio: false },
            function(myStream) {
                stream = myStream;
                yourVideo.src = window.URL.createObjectURL(stream);

                if (hasRTCPeerConnection()) {
                    setupPeerConnection(stream);
                } else {
                    alert("Sorry, your browser does not support WebRTC.");
                }
            }, function (error) {
                console.log(error);
            });
    } else {
        alert("Sorry, your browser does not support WebRTC.");
    }
}

function setupPeerConnection(stream) {
    var configuration = {
        "iceServers": [{ "url": "stun:stun.1.google.com:19302" }]
    };
    yourConnection = new RTCPeerConnection(configuration);
    // 设置流的监听
}

```



```

        yourConnection.addStream(stream);
        yourConnection.onaddstream = function (e) {
            theirVideo.src = window.URL.createObjectURL(e.stream);
        };

        // 设置 ice 处理事件
        yourConnection.onicecandidate = function (event) {
            if (event.candidate) {
                send({
                    type: "candidate",
                    candidate: event.candidate
                });
            }
        };
    }

    function hasUserMedia() {
        navigator.getUserMedia = navigator.getUserMedia ||
        navigator.webkit GetUserMedia || navigator.mozGetUserMedia ||
        navigator.msGetUserMedia;
        return !!navigator.getUserMedia;
    }

    function hasRTCPeerConnection() {
        window.RTCPeerConnection = window.RTCPeerConnection ||
        window.webkitRTCPeerConnection || window.mozRTCPeerConnection;
        window.RTCSessionDescription = window.RTCSessionDescription ||
        window.webkitRTCSessionDescription || window.mozRTCSessionDescription;
        window.mozRTCSessionDescription;
        window.RTCIceCandidate = window.RTCIceCandidate ||
        window.webkitRTCIceCandidate || window.mozRTCIceCandidate;
        return !!window.RTCPeerConnection;
    }
}

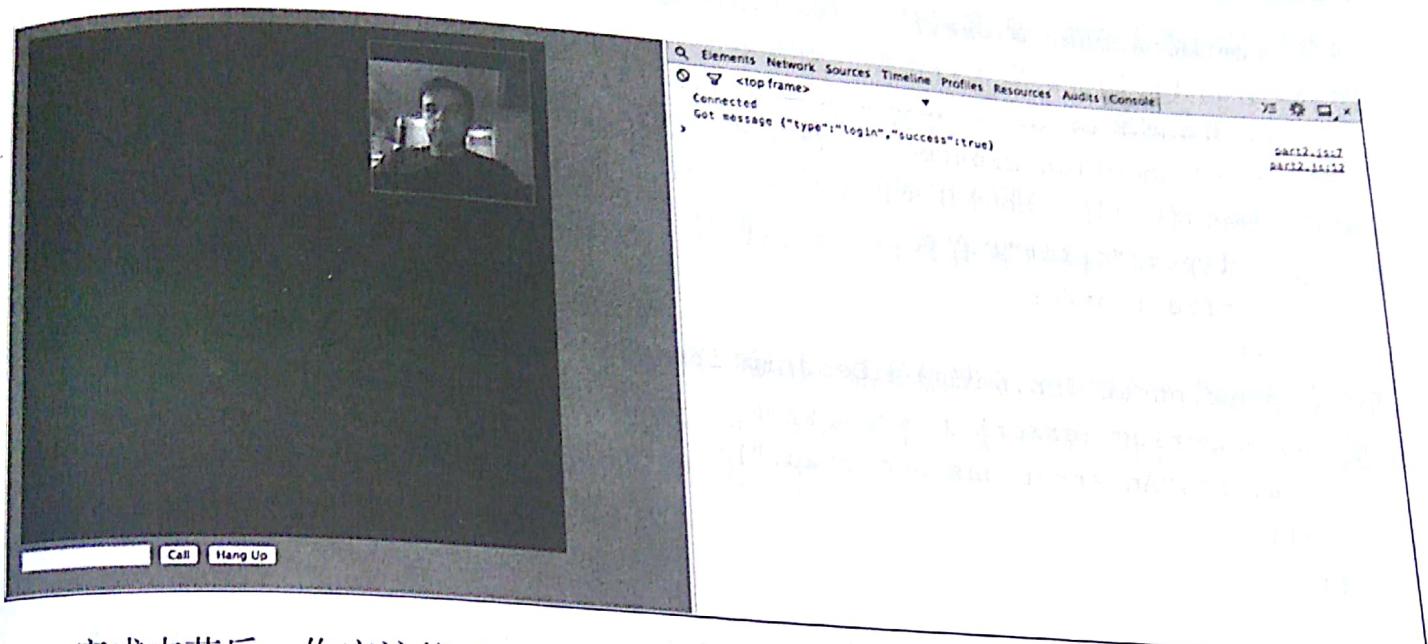
```

这段代码现在看起来应该感觉相当熟悉了，大部分都是从第 3 章创建简单的 WebRTC 应用中展示的例子复制过来的。一如既往，我们检查正确的浏览器前缀和设置相应错误处理的代码。如果运行此代码，现在应该允许你登录，并要求获取在你的设备上使用相机视频流功能的权限。同时，正如你所记得的，audio 参数设置为 false，用来避免在同一设



由 扫描全能王 扫描创建

备上测试连接时回应声音太大。



完成本节后，你应该能看到与上面的截图类似的界面。如果出现了问题，你可以去回顾一下前面的章节，确保你的服务器设置正确。同时，确保你的文件托管在本地服务器中，以便 `getUserMedia` 方法能够正常运行。

## 发起通话

现在，所有的事情都已经准备妥当，我们可以和远程用户发起通话。首先发送 `offer` 给另一个用户来开始整个过程，一旦用户得到这个 `offer`，他将创建一个响应并开始交换 ICE 候选，直到成功连接到服务器。这个过程和第 3 章创建简单的 WebRTC 应用提到的完全一样，除了现在使用的是远程服务器。为了实现此功能，添加以下代码到我们的脚本中：

```
callButton.addEventListener("click", function () {
  var theirUsername = theirUsernameInput.value;

  if (theirUsername.length > 0) {
    startPeerConnection(theirUsername);
  }
});

function startPeerConnection(user) {
```



```

connectedUser = user;

// 开始创建 offer
yourConnection.createOffer(function (offer) {
    send({
        type: "offer",
        offer: offer
    });
    yourConnection.setLocalDescription(offer);
}, function (error) {
    alert("An error has occurred.");
});

};

function onOffer(offer, name) {
    connectedUser = name;
    yourConnection.setRemoteDescription(new
        RTCSessionDescription(offer));

    yourConnection.createAnswer(function (answer) {
        yourConnection.setLocalDescription(answer);
        send({
            type: "answer",
            answer: answer
        });
    }, function (error) {
        alert("An error has occurred");
    });
};

function onAnswer(answer) {
    yourConnection.setRemoteDescription(new RTCSessionDescription(answer));
};

function onCandidate(candidate) {
    yourConnection.addIceCandidate(new RTCIceCandidate(candidate));
};

```



首先给 Call 按钮添加单击事件，用来启动这个过程。然后实现一些服务器的消息处理器所需要的方法。这些方法将异步运行，直到双方成功地进行了连接。因为大部分工作已经在服务器和 WebSocket 层做了，使得这部分的实现更简单。

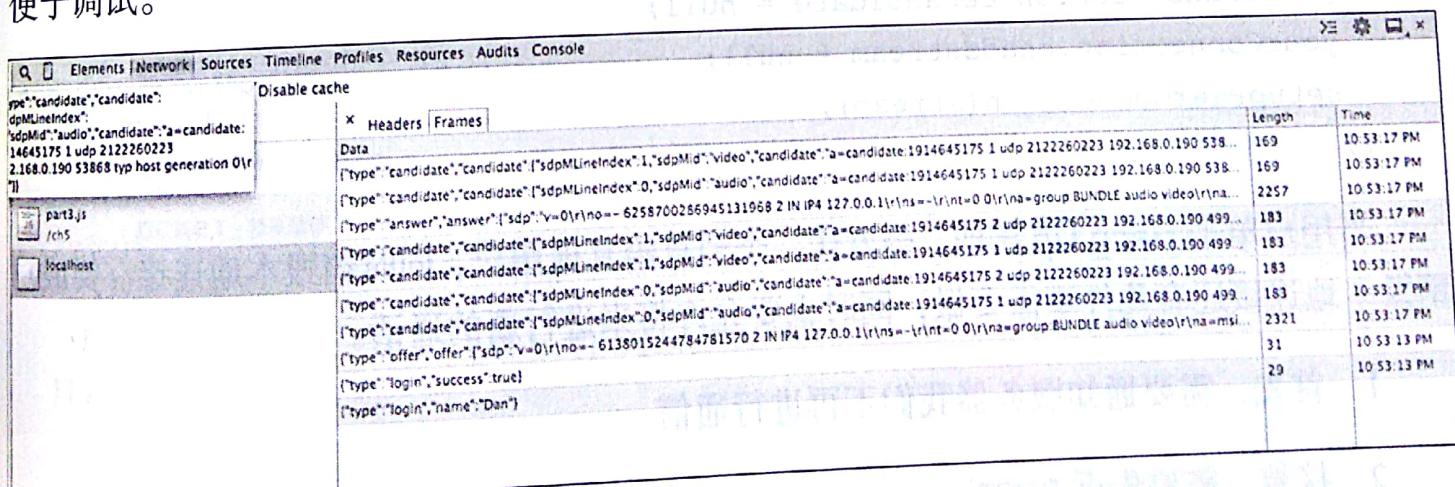
运行这段代码后，你应该在两个浏览器标签页中分别使用不同的用户名登录到服务器。之后使用通话功能来呼叫另一个标签页中的用户，这将在客户端之间成功创建一个 WebRTC 连接。

恭喜你已经创建了一个完整的 WebRTC 应用。这是创造令人惊叹的对等网络应用的重要一步。通常，要实现如此强大的功能，需要翻阅多本书籍并使用不同的框架，但使用这种强大的技术我们通过短短几章的学习就实现了。

检测通信

调试实时应用是非常困难的。由于许多件事都在同一时刻发生，要完整描述某一时刻发生了什么很难。拥有 WebSocket 协议的现代浏览器就能帮上忙。现在的大部分浏览器不仅可以查看连接到服务器的 WebSocket 连接，也可以查看每一个发送的包。

在我的例子中，我使用 Chrome 浏览器来检查网络通信。通过导航中的 View→Developer→Developer Tools，我们会看到一系列调试 web 应用程序的工具。单击 Network 选项卡，会显示所有当前网页的网络通信。如果你没有看到任何网络通信，请在 Developer Tools 打开的状态下刷新网页。从这个列表中容易找到 localhost 连接，选中它，可以看到已经使用 WebSocket 连接发送的帧。它以人们可读的格式来显示每个发送的包，便于调试。



你可以在这个图中看到每一个步聚，上面的截图中显示了 login、offer、answer，以及每一个发送的 ICE 候选。这样，我可以检查消息中的错误，比如错误的数据。在调试



在 web 应用程序时，尽可能地使用浏览器内置工具。

在电脑上还有其他许多方式来获得这些信息。在服务器和客户端使用控制台输出也是一种不错的获取部分信息的方式。你也可以使用网络代理和抓包工具拦截浏览器发送的数据包。这种设置起来比较麻烦，但可以得到服务器和客户端发送的数据的更多信息。我将把它留给读者作为练习，来找出更多的方法调试 web 应用程序。

## 挂断电话

我们将实现的最后一个功能是挂断进行中的通话。这将通知其他用户我们试图关闭通话并停止发送信息。需要添加几行代码在 JavaScript 中：

```
hangUpButton.addEventListener("click", function () {
  send({
    type: "leave"
  });
  onLeave();
});

function onLeave() {
  connectedUser = null;
  theirVideo.src = null;
  yourConnection.close();
  yourConnection.onicecandidate = null;
  yourConnection.onaddstream = null;
  setupPeerConnection(stream);
}
```

当用户单击 Hang Up 按钮，会发送一个消息给其他用户，同时销毁本地连接。要成功销毁本地连接还有几件事要做，同时也要允许将来进行新的通话。

1. 首先，需要通知服务器我们不再进行通信。
2. 接着，需要告诉 RTCPeerConnection 进行关闭，这将停止发送数据流给其他用户。



由 扫描全能王 扫描创建

3. 最后，再次设置连接，把连接实例设置为打开状态，以便我们接受新的通话。

## 一个完整的 WebRTC 客户端

下面是完整的客户端程序中所使用的 JavaScript 代码。这些代码包括用户 UI，连接到信令服务器，并开始与另一个用户 WebRTC 连接：

```

var connection = new WebSocket('ws://localhost:8888'),
    name = "";

var loginPage = document.querySelector('#login-page'),
    usernameInput = document.querySelector('#username'),
    loginButton = document.querySelector('#login'),
    callPage = document.querySelector('#call-page'),
    theirUsernameInput = document.querySelector('#their-
username'),
    callButton = document.querySelector('#call'),
    hangUpButton = document.querySelector('#hang-up');

callPage.style.display = "none";

// 单击按钮登录
loginButton.addEventListener("click", function (event) {
    name = usernameInput.value;

    if (name.length > 0) {
        send({
            type: "login",
            name: name
        });
    }
});

connection.onopen = function () {
    console.log("Connected");
};

```



```

// 通过回调函数处理所有的消息
connection.onmessage = function (message) {
    console.log("Got message", message.data);

    var data = JSON.parse(message.data);

    switch(data.type) {
        case "login":
            onLogin(data.success);
            break;
        case "offer":
            onOffer(data.offer, data.name);
            break;
        case "answer":
            onAnswer(data.answer);
            break;
        case "candidate":
            onCandidate(data.candidate);
            break;
        case "leave":
            onLeave();
            break;
        default:
            break;
    }
};

connection.onerror = function (err) {
    console.log("Got error", err);
};

// Alias 以JSON格式发送消息
function send(message) {
    if (connectedUser) {
        message.name = connectedUser;
    }
}

```



```

connection.send(JSON.stringify(message));
};

function onLogin(success) {
  if (success === false) {
    alert("Login unsuccessful, please try a different name.");
  } else {
    LoginPage.style.display = "none";
    callPage.style.display = "block";
  }
}

// 准备好通话的通道
startConnection();
}

};

callButton.addEventListener("click", function() {
  var theirUsername = theirUsernameInput.value;
  if (theirUsername.length > 0) {
    startPeerConnection(theirUsername);
  }
});

hangUpButton.addEventListener("click", function() {
  send({
    type: "leave"
  });
  onLeave();
});

});

function onOffer(offer, name) {
  connectedUser = name;
  yourConnection.setRemoteDescription(new
    RTCSessionDescription(offer));
}

```



```

    yourConnection.createAnswer(function (answer) {
        yourConnection.setLocalDescription(answer);
        send({
            type: "answer",
            answer: answer
        });
    }, function (error) {
        alert("An error has occurred");
    });
}

function onAnswer(answer) {
    yourConnection.setRemoteDescription(new
RTCSessionDescription(answer));
}

function onCandidate(candidate) {
    yourConnection.addIceCandidate(new RTCIceCandidate(candidate));
}

function onLeave() {
    connectedUser = null;
    theirVideo.src = null;
    yourConnection.close();
    yourConnection.onicecandidate = null;
    yourConnection.onaddstream = null;
    setupPeerConnection(stream);
}

function hasUserMedia() {
    navigator.getUserMedia = navigator.getUserMedia ||

navigator.webkit GetUserMedia || navigator.mozGetUserMedia ||

navigator.msGetUserMedia;

    return !!navigator.getUserMedia;
}

function hasRTCPeerConnection() {
}

```



```

window.RTCPeerConnection = window.RTCPeerConnection || window.mozRTCPeerConnection;
window.webkitRTCPeerConnection || window.mozRTCPeerConnection;
window.RTCSessionDescription = window.RTCSessionDescription || window.mozRTCSessionDescription;
window.webkitRTCSessionDescription || window.mozRTCSessionDescription;
window.RTCIceCandidate = window.RTCIceCandidate || window.mozRTCIceCandidate;
window.webkitRTCIceCandidate || window.mozRTCIceCandidate;
return !!window.RTCPeerConnection;
}

var yourVideo = document.querySelector('#yours'),
theirVideo = document.querySelector('#theirs'),
yourConnection, connectedUser, stream;

function startConnection() {
if (hasUserMedia()) {
navigator.getUserMedia({ video: true, audio: false }, function
(myStream) {
stream = myStream;
yourVideo.src = window.URL.createObjectURL(stream);
if (hasRTCPeerConnection()) {
setupPeerConnection(stream);
} else {
alert("Sorry, your browser does not support WebRTC.");
}
}, function (error) {
console.log(error);
});
} else {
alert("Sorry, your browser does not support WebRTC.");
}
}

function setupPeerConnection(stream) {
var configuration = {
"iceServers": [{ "url": "stun:stun.1.google.com:19302" }]
}

```



```

};

yourConnection = new RTCPeerConnection(configuration);
yourConnection.onicecandidate = function (e) {
    theirVideo.src = window.URL.createObjectURL(e.stream);
};

// 设置流的监听
yourConnection.addStream(stream);
yourConnection.onaddstream = function (e) {
    theirVideo.src = window.URL.createObjectURL(e.stream);
};

// 设置 ice 处理事件
yourConnection.onicecandidate = function (event) {
    if (event.candidate) {
        send({
            type: "candidate",
            candidate: event.candidate
        });
    }
};

function startPeerConnection(user) {
    connectedUser = user;

    // 开始创建 offer
    yourConnection.createOffer(function (offer) {
        send({
            type: "offer",
            offer: offer
        });

        yourConnection.setLocalDescription(offer),
        function (error) {
            alert("An error has occurred.");
        });
    });
}

```

如果你在运行客户端时遇到问题，花一点时间检查以确保你正确地复制了代码。另外，



检查浏览器的具体实现。浏览器之间有细微的差别，留意在控制台中出现的任何错误。

## 改进应用程序

通过本章课程的建设，现在是时候来让程序变得更加完美了。它提供了点对点通信应用所需的基本功能。从这里开始，我们添加一些常用的 web 应用程序特性来增强用户体验。

登录体验是一个开始改进的点。像 Facebook、Google 这些公共平台已经提供非常完善的服务来进行用户认证。它们整合的 API 不仅简洁易懂，也确保每个用户的唯一性。同时，它们还能提供好友列表。因此，他/她第一次使用应用时，就可以和好友列表中的用户进行通话。

此外，应用程序需要万无一失，确保最好的体验。用户的输入应该在客户端和服务端都进行检查。另外，有些情况会导致 WebRTC 连接失败，如技术不支持、防火墙阻拦、没有足够的带宽进行视频通话。大量的工作已经用来使普通的电话通信平台稳定以避免断线，同样的工作也需要进入到 WebRTC 平台中，使它变得稳定。

## 自测题

Q1. 在大部分浏览器中创建一个 WebSocket 连接需要安装多个框架才能正常工作。对或错？

Q2. 要成功运行本章创建的例子，有哪些技术或者技术点需要用户浏览器支持？

1. WebRTC
2. WebSockets
3. 媒体捕捉和视频流
4. 以上全部

Q3. 该应用允许用户在一个视频通话中连两个以上的用户。对或错？

Q4. 为让我们的应用更加稳定，在尝试通话时减少错误，需要添加：

1. 更多 CSS 样式



2. Facebook 联合登录
3. 每一步进行错误检查和验证
4. 炫酷的动画

## 小结

在完成本章后，你或许应该退后一步，思索一番，庆祝目前所做的。通过这一章，我们已经讲完了整本书的前半部分并完成了一个完备的 WebRTC 应用。对于非常复杂的点对点连接应用，通过短短的 5 个章节就成功创建了是非常了不起的。你现在可以放下聊天工具，使用自己新手创建的工具和全世界的人进行聊天。

现在你应该已经对 WebRTC 应用程序总体架构有了了解。我们不仅覆盖到了客户端的实现，而且还支持信令服务器。甚至还结合了其他 HTML5 技术，如 WebSocket，来帮助我们进行远程的点对点连接。

如果你有任何问题，现在是时候放下这本书休息一会。这个应用程序是一个起点，你可以进行自己的 WebRTC 应用程序的原型设计并加入一些创新的特性。读到这里，到网络上去搜索一些其他的 WebRTC 应用程序并研究它们的开发方法也是一个不错的主意。对 WebRTC 应用程序内部工作原理有了大致了解后，你应该可以从网络中一些其他开源代码例子中学到很多。

在接下来的章节，我们将继续扩大，接触在网络上创建点对点应用的许多高级主题。音频和视频通话是你现在能做的 WebRTC 应用的临界点，我们将继续探索该技术的许多其他功能，内容也会覆盖到如何创建一个更加健壮的应用来连接多个用户，以及移动电话和安全话题对于 WebRTC 应用的关系。



由 扫描全能王 扫描创建

# 使用 WebRTC 发送数据

到目前为止，我们的关注点都在 WebRTC 的音频和视频功能上。然而，有一个主题，还没有开始讨论——任意数据的传输。WebRTC 擅长进行数据传输，不仅仅是音频和视频流，还包括我们希望的任意数据类型。

本章我们将学习 WebRTC 数据通道协议以及它在通信应用的使用。纵观这章，我们将讨论以下主题：

- 如何理解数据通道适应 WebRTC 难题
- 如何在对等连接中创建一个数据通道对象
- 加密和安全问题
- 数据通道的潜在用例

## 流控制传输协议和数据传输

总的来说，在对等连接中传输数据是一项艰巨的任务。在用户之间发送数据，目前通常的做法是使用严格的 TCP 连接。即使用如 AJAX 和 WebSockets 技术将数据发送到服务端，并在另一端进行接收。这种缓慢而笨重的方式对于高性能的应用是一个瓶颈。为了在两个用户间传送数据，开发者需要购买服务器网络，这通常非常昂贵，因为即使购买一个小型的分布式网络服务器也需要每月花费上千美元。

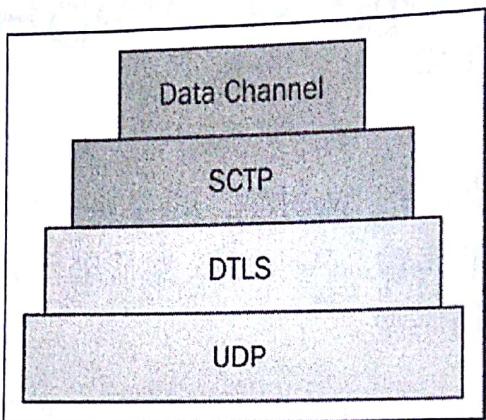
在我们的 WebRTC 模型中，已经实现了用户之间高速、低延时的连接。通过这个连接，我们可以快速地发送音频和视频数据。目前使用的协议是专为视频和音频流的帧设计的。这就是为什么 WebRTC 在刚建立的对等连接中使用流控制传输协议（SCTP）作为数



由 扫描全能王 扫描创建

据传输的方式。

SCTP 同样也是另一种术语，位于独立的 WebRTC 堆栈中。这就是为什么 WebRTC 是一个如此有趣的技术。它给 JavaScript 开发者提供了许多新的方法进行数据传输。SCTP 位于数据传输层安全协议（DTLS）之上，每个 WebRTC 连接都实现了它并且为数据传输提供出口绑定到数据通道上。所有这些都位于 UDP 之上，它为 WebRTC 数据提供基本的传输方法，就如第 3 章创建简单的 WebRTC 应用中所讨论的。目前为止，我们的堆栈看上去是这样的：



这看上去有点复杂，但这都是为了强大的 SCTP。WebRTC 设计者认识到，每个应用都希望有自己独特的方式来使用强大的数据通道。有些希望像 TCP 协议那样进行可靠的数据传输，另一些需要像 UDP 协议那样高效。因为这些原因促使他们选择了 SCTP，因为 SCTP 能同时满足这些需求。以下是 SCTP 的特点：

- 传输层的安全性，基于 DTLS 层。
- 传输层可以运行在可靠的或不可靠的模式中。
- 传输层可以担保或者无担保数据包顺序。
- 数据是面向消息进行传播的，允许消息分解传输，在接收端重组。
- 传输层支持流量和阻塞协议。

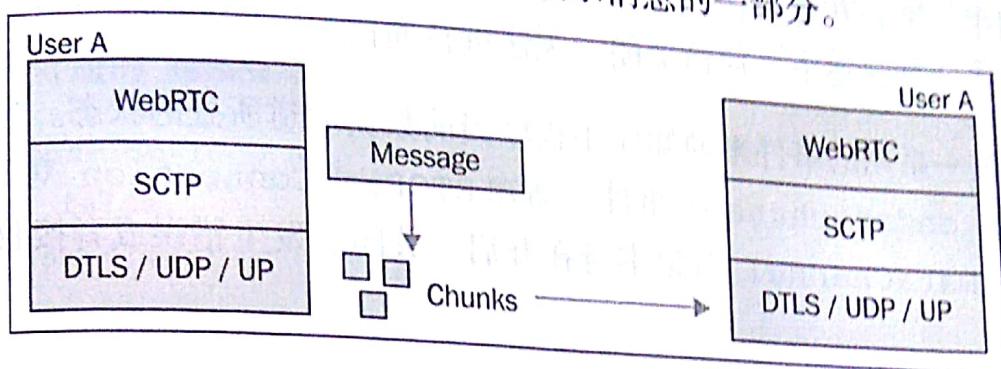
 其中某些特点听上去类似于 UDP 和 TCP 协议。这是因为 SCTP 的产生是由于这两种协议的限制。它设计出来用来解决 TCP 的一些问题，同时利用了 UDP 的传输能力。

SCTP 规范定义了使用多个端点，把消息分解成多个块进行发送数据，以下是一些相关解释：



由 扫描全能王 扫描创建

- 端点：在两个 IP 位置之间定义任意数据的连接。
- 消息：任意从应用发送到 SCTP 层的数据。
- 块：正准备通过电缆传输的数据包，表示消息的一部分。



整个协议总体跨越 16 个部分，通过这些开始让你了解 WebRTC 的工作模式。本节你需要记住的是，在浏览器中，数据通道采用完全不同的渠道，而不像其他基于数据的传输层所使用的通道。它是可配置的并且高效地控制数据传输。WebRTC 提供给每个 JavaScript 应用开发者一种非常强大的技术。

## RTCDataChannel 对象

现在我们已经清楚相关的基本技术，可以开始学习实际应用中 RTCDataChannel 对象 API 是如何工作的。一个好消息是这些 API 的学习相对于 SCTP 的基础工作原理复杂度要低。创建一个通道这个主要功能已经在 RTCDataChannel 对象中实现了：

```
var peerConnection = new RTCPeerConnection();
// 建立你的对等连接使用信号

var dataChannel = peerConnection.createDataChannel("myLabel",
  dataChannelOptions);
```

这就是你需要开始的一切。WebRTC 会处理好所有的事情包括浏览器内部层。一旦信令完成，连接建立，这所有的都将发生。RTCPeerConnection 对象关闭前，你能在这个过程的任何时候来创建一个数据通道，如果它关闭了，你还尝试创建一个新的通道，它将抛出一个异常。

数据通道会存在于以下几种状态中：



由 扫描全能王 扫描创建

- 连接中：这是默认状态，当数据通道等待一个连接。
- 开启：这种状态下，连接已经被建立，可以进行通信。
- 关闭中：通道正在被销毁。
- 关闭：这种状态下，通道关闭，无法进行通信。

浏览器通过一系列的事件来通知应用程序当前数据通道所处的状态。当有其他人试图创建一个通道，`ondatachannel` 事件会通知 `RTCPeerConnection` 对象，已经有通道被创建了。`RTCDataChannel` 对象本身在开启、关闭、发生错误或者接收到消息时会触发对应的事件。

```
dataChannel.onerror = function (error) {
  console.log("Data channel error:", error);
};

dataChannel.onmessage = function (event) {
  console.log("Data channel message:", event.data);
};

dataChannel.onopen = function () {
  console.log("Data channel opened, ready to send messages!");
  dataChannel.send("Hello World!");
};

dataChannel.onclose = function () {
  console.log("Data channel has been closed.");
};
```

这些事件的代码看上去简单明了。当创建一个数据通道后，你必须等 `onopen` 事件触发后才能发送消息。在通道打开前发送消息，会抛出一个异常，这个异常指出通道还没有准备好发送消息。

这些看上去和我们前面使用的 WebSocket 协议很类似，是故意这么做的。因为 WebSocket 协议标准有着很好的定义，并且容易理解。数据通道也使用这样的方式，容易被开发者使用。当然，也别弄糊涂了，WebSockets 发送数据给远程时，使用完全不同的路



由 扫描全能王 扫描创建

你应该已经注意到例子中引入了 `dataChannelOptions` 对象。SCTP 在发送数据给别一端时，可以进行不同配置，这些配置选项已经提供给开发人员。传入的配置项是可选的，并且是一个普通的 JavaScript 对象：

```
var dataChannelOptions = {  
    reliable: false,  
    maxRetransmitTime: 3000  
};
```

这些配置项可以使应用在 UDP 或者 TCP 的优势之间进行变化。一些选项使得通道更加稳定可靠，另外一些使得应用运行更加快。这些选项包括：

- `reliable`: 设置消息传递是否进行担保。
- `ordered`: 用来设置消息的接受是否需要按照发送时的顺序。
- `maxRetransmitTime`: 设置消息发送失败时，多久重新发送。
- `maxRetransmits`: 设置消息发送失败时，最重发次数。
- `protocol`: 设置强制使用其他子协议，但当用户代理不支持该协议时会报错。
- `negotiated`: 此选项用来设置开发人员是否有责任在两边创建数据通道，还是浏览器来自动完成这个步骤。稍后将替换该选项。
- `id`: 这个用来设置通道的唯一标识，可以在多通道时进行区分。

这些配置项很多，不过大部分只在高级应用中才会使用。主要使用的配置项是 `reliable` 和 `ordered`，当设置为 `true` 时数据通道表现更像 TCP，设置为 `false` 时表现更像 UDP。

`negotiated` 参数为解决两边用户同步创建数据通道而设置。用来处理 `ondatachannel` 事件触发在 `RTCPeerConnection` 对象上。它的默认值是 `false`，表示浏览器会自动触发一件事件在通道的另一边，告诉他/她这个新通道。如果设置成 `true`，开发者需要自己在通道两边创建相同 ID 的数据通道。

## 发送数据

数据通道的 `send` 方法像 WebSockets 上的 `send` 方法一样进行了重载，这样允许在传输层上发送不同的 JavaScript 数据类型。使用不同的数据类型有助于提升应用的性能。这



由 扫描全能王 扫描创建

是由于大部分基于字符串编码的数据过于庞大，需要发送更多的数据包，目前，数据通道支持如下类型：

- `String`: JavaScript 基本的字符串。
- `Blob`: 一种文件格式的原始数据。
- `ArrayBuffer`: 确定数组长度的数据类型。
- `ArrayBufferView`: 基础的数组视图。

将需要的变量传入到 `send` 方法中，浏览器会做剩下的工作。你可以在接收消息的一边通过测试来确定数据类型。

```
dataChannel.onmessage = function (event) {
  var data = event.data;

  if (data instanceof Blob) {
    // 处理 Blob
  } else if (data instanceof ArrayBuffer) {
    // 处理 ArrayBuffer
  } else if (data instanceof ArrayBufferView) {
    // 处理 ArrayBufferView
  } else {
    // 处理 string
  }
};
```

## 加密与安全

使消息能安全地进行传输引起 WebRTC 协议的设计者高度重视。背后的原因是许多大公司由于 WebRTC 应用没有正常的安全保障而不考虑使用它。为了能增加被采用的概率，在设计 API 接口必须要考虑安全性能被保障。

你应该注意到，WebRTC 运行时，对于所有协议的实现，都会强制执行加密功能。这意味着浏览器间的每一个对等连接，都自动处于高的安全级别中。所使用的加密技术都应该满足对等应用的以下几个要求：

- 信息在传输过程中被窃取，无法进行读取。



由 扫描全能王 扫描创建

- 第三方不能够伪造消息，使消息看上去像是连接者发送的。
- 消息在传输给另一方时不能进行编辑。
- 加密算法的快速性，以支持客户端之间的最大带宽。

为了满足这些要求，我们选择了 DTLS 技术。作为一个 web 开发者，你可能知道后面三个字母 TLS。TLS 协议可以追溯到 1999 年，并且到目前为止，是被最广泛使用的网络加密协议。它的流行是因为直接嵌入到应用和传输层，使得很少甚至不需要改变应用本身的逻辑而使应用变得安全。使用 TLS 的唯一缺点就是它必基于 TCP 的应用，但我们的 WebRTC 协议并不是在其基础上工作的。

所以这里引进了 DTLS，它源于希望能有一个像 TLS 一样简单易用的协议，但拥有 UDP 传输层的功能。它吸取了 TLS 协议许多相同的概念，并增加了对 UDP 的支持。

TCP 和 UDP 最大的区别是前者能够保证消息到达另一边。这也是 TLS 和 DTLS 的主要区别。DTLS 有能力处理消息丢失和接收消息时顺序不正确的能力，它能够在 UDP 协议基础上使用。这使 DTLS 满足像 WebRTC 这类应用程序的加密要求，它是非常不错的选择。

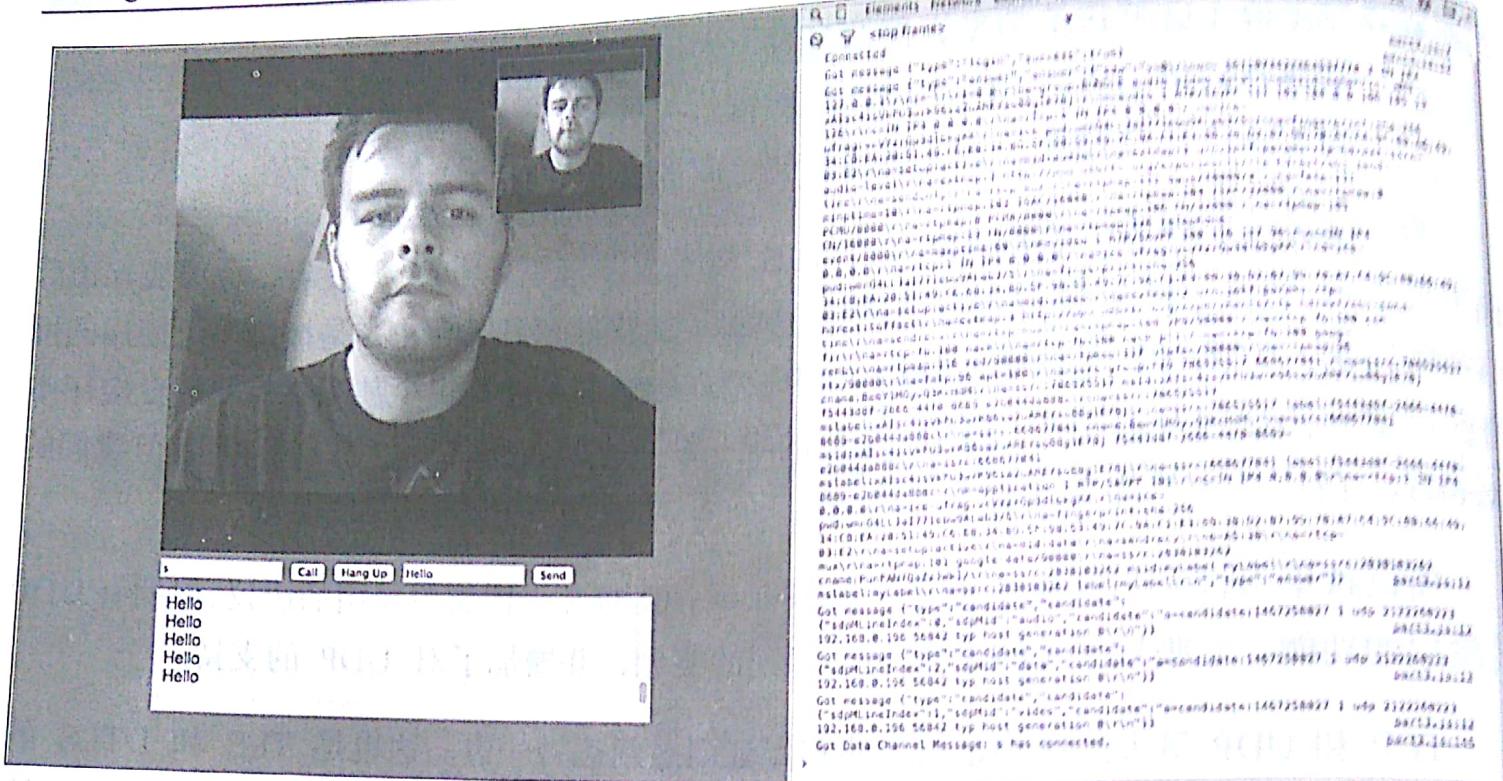
到这里最大的收获是，你需要知道在数据通道和 WebRTC 应用中，数据是安全的。一般情况下，已经采取措施来确保数据在点对点连接中是不会被修改的。这是一般的点对点应用的基本工作要求。关于 DTLS 的规范可以在 <https://tools.ietf.org/html/rfc4347> 看到，通过这个，你可以确定 DTLS 提供的安全是否符合你的应用。

## 添加文字聊天

现在我们将使用本章所学到的，为我们的通信应用程序添加数据通道支持。因为数据通道可以用于任意数据，我们为我们的应用程序添加另外一个特性，增加文字聊天功能。用户可以有一个文本框输入消息，同时能显示当前通话的所有消息。完成这个功能后，整个客户端界面看上去如下所示：



由 扫描全能王 扫描创建



首先，对应用程序的呼叫页面做一些调整，添加三个新的元素——文本框、按钮、div。文本框和按钮将允许用户输入文本，div 用来保留所有用户的消息。

```
<div id="call-page" class="page">
  <video id="yours" autoplay></video>
  <video id="theirs" autoplay></video>
  <input type="text" id="their-username" />
  <button id="call">Call</button>
  <button id="hang-up">Hang Up</button>

  <input type="text" id="message"></input>
  <button id="send">Send</button>
  <div id="received"></div>
</div>
```

接下来，给页面添加 JavaScript 和相关的数据通道功能：

```
function openDataChannel() {
  var dataChannelOptions = {
    reliable: true
  };
```



由 扫描全能王 扫描创建

```

dataChannel = yourConnection.createDataChannel("myLabel",
dataChannelOptions);

dataChannel.onerror = function (error) {
  console.log("Data Channel Error:", error);
};

dataChannel.onmessage = function (event) {
  console.log("Got Data Channel Message:", event.data);

  received.innerHTML += "recv: " + event.data + "<br />";
  received.scrollTop = received.scrollHeight;
};

dataChannel.onopen = function () {
  dataChannel.send(name + " has connected.");
};

dataChannel.onclose = function () {
  console.log("The Data Channel is Closed");
};
}

```

这个脚本根据我们的连接设置数据通道，当 `yourConnection` 变量实例化并准备好被使用后，将回调它。你可以在 `yourConnection` 变量创建以后调用这个函数。这段代码将为我们的数据通道添加一系列的监听事件。

- `onerror`: 此监听器将检测任何连接问题。
- `onmessage`: 此监听器将接收其他用户发送的消息。
- `onopen`: 此监听器用来告诉我们有其他用户进行连接。
- `onclose`: 此监听器告诉我们有其他用户断开连接。

到现在为止，我们都是通过控制台来通知开发人员的。留一个练习给读者，来使这些监听事件有用。



由 扫描全能王 扫描创建



在写本书时，为了使这个例子能在 Chrome 浏览器中运行，我不得不添加 `{optional: [{RtpDataChannels: true}]}`  这个配置。对数据通道的支持工作还在进行中，如果应用没有运行，我们可能需要进行一些额外的设置。你可以搜索一下，对于数据通道和你选择的浏览器看看是否需要一些额外的配置。

现在，当用户单击发送按钮时，我们可以添加一个事件侦听器：

```
// 绑定文本输入框和消息接收区
sendButton.addEventListener("click", function (event) {
  var val = messageInput.value;
  received.innerHTML += "send: " + val + "<br />";
  received.scrollTop = received.scrollHeight;
  dataChannel.send(val);
});
```

这个监听器将获取文本字段的值。然后将它添加到本地消息框中。你应该还记得，我们可以在一个稳定的状态下运行我们的数据通道，以确保用户的消息框处于同步状态。我们不需要去关注其他用户是否已经接收到了消息，数据通道会帮你关注它。最后，我们让数据通道发送消息，它会触发其他用户的 `onmessage` 处理函数，并传递消息数据给他们。

给消息框添加一些样式，使它像常见的聊天应用程序一样，有滚动条，并且比较美观。

```
#received {
  display: block;
  width: 480px;
  height: 100px;
  background: white;
  padding: 10px;
  margin-top: 10px;
  color: black;
  overflow: scroll;
}
```

现在，运行上面的代码，你已经完成了一个基本的文本聊天应用！用户彼此之间可以进行视频、语音和文本聊天，拥有大部分聊天工具所具有的特点。然而，我们没有添加任



何文本校验，来让我们的应用更加容易使用。为了应用更加健壮，你需要对所实现的功能做一些改进。

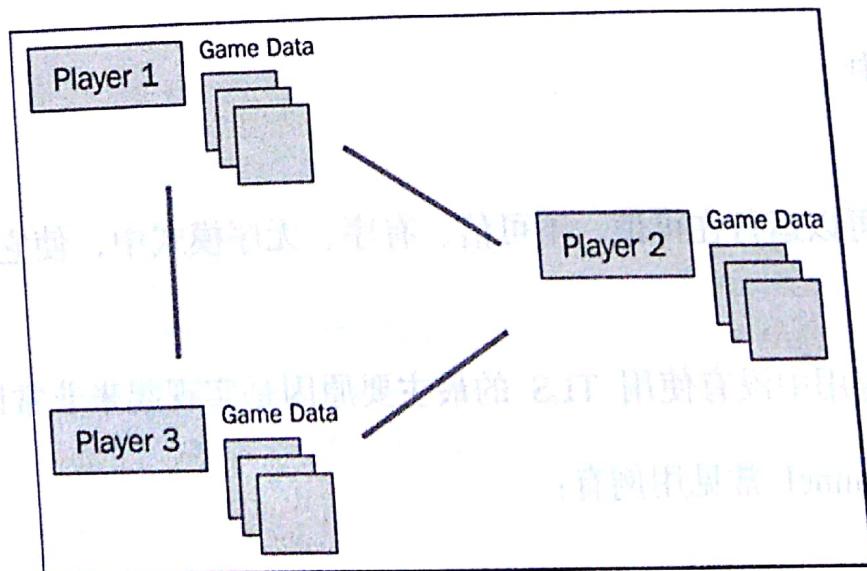
## 用例

我们接触到的仅仅是使用 WebRTC 数据通道的冰山一角。学到目前为止，添加文本聊天是对这个协议最简单的扩展。DTLS 规范表明，它支持用户之间传输任何类型的数据，只要你能够想得到。

游戏是数据通道协议最早的扩展之一。实际上，在多人网络游戏中，点对点通道是很常见的。以前，像雷神之锤 3 这类游戏，在游戏时，就是依靠点对点网络进行玩家之间的交流。背后的原因是考虑到服务器成本和传输速度。

玩家玩游戏时自己支付网络连接费用，这样游戏公司就不需要实现复杂的服务器用于来回传输数据。只要电脑能够通过因特网连接到另一台电脑，就可以进行游戏。有时因为拨号上网速度慢，也可以通过本地局域网来体验流畅的速度。随着时间的推移不断变化，也需要消除用户因为修改复杂游戏数据而领先于其他用户的问题。

现在许多游戏借助点对点的网络数据传输能力，来进行游戏的更新。在浏览器中进行文件传输，这个概念的提出来源于基于流的文件共享网络。在点对点网络中进行文件共享的概念是，用户通过互联网连接把文件给网络中的其他用户，就像是一台共享服务器。这意味着文件传输速度仅受限于用户的网络和有多少用户拥有这个文件，替代了昂贵可用的中心服务器。这种方式日趋流行在许多用户之间进行大数据传输。下面的图示例了常见的点对点游戏的网络布局：



这个概念甚至被扩展到浏览器中的文件的内容分发。一些开发者提出了新的理念，浏览器中传输脚本、图片等，使用强大的 WebRTC。这里想法是，用户间使用 WebRTC 连接来传输这些文件，替代昂贵的大型网络内容分发系统（CDN）。

CDNs 通过在全球范围内设置许多服务器来进行工作。这使得有用户接近网络中的台服务器的概率是非常高的，并且该用户得有很快的文件传输速度。当你将文件上传到网络中，全世界的网络节点在获得通知的第一时间对它进行复制，使它成为可用。不足之处在于这种方式网络维护成本很高。

使用 WebRTC 可以帮助缓解这种情况，假设用户可以用同样的方式，从附近的用户下载数据来替代从服务器下载。文件下载完成后，用户再次从一台服务器中通过对算法，来检查文件，以确保文件就是期望的那个。这可保证文件的完整性，同时使网络更加强大，但比 CDN 成本更低。

正是因为这些想法，提供了 WebRTC 数据通道背后强有力的支持。这种新的数据传输形式，使网页开发者原来不能实现的网络特性变成可能。利用强大的数据通道，在各行业中还将会有更多的创新想法被发现。

## 自测题

Q1. 数据通道不加密或者不安全，容易被黑客修改正在用户间传输的数据。对或错？

Q2. RTCDataChannel 不会处于以下哪种状态？

1. 重连
2. 关闭
3. 连接中
4. 开启

Q3. 数据通道可以运行在可信、不可信、有序、无序模式中，使它拥有强大的数据传输能力。对或错？

Q4. WebRTC 应用中没有使用 TLS 的最主要原因是实现起来非常困难。对或错？

Q5. RTCDataChannel 常见用例有：



由 扫描全能王 扫描创建

1. 多人游戏
2. 文件传输
3. 分发内容
4. 以上全部

## 小结

读完本章，你应该已经比较好地了解了 WebRTC 另一个难点——数据传输。这个经常被忽视，尽管它是 WebRTC 规范中非常强大的部分，它带了新的基于网络的轻型应用。本章，我们在 WebRTC 例子中，用数据通道创建了基于文本的聊天工具。

现在，你可以正式地告别我们的通信应用程序，这也是你在读本书时最后一次看到它。我们通过仅仅一百多页，就完成了这项惊人的壮举。不仅创建了一个多人应用，同时这个应用也拥有目前世界上流行的通信应用的大部分特性。这不仅仅是网络应用的能力，更主要归功于强大的 WebRTC。

如果你还没有这样做，这是另一个好机会来放下这本书，开始尝试我们的例子。在这里，还有一些特性没有在例子中实现，可以通过几个小时的实施来添加它们。最容易想到的是为用户的输入添加校验。通过这些例子，很容易实现数据的转移，甚至建立一个玩家可以控制的游戏。

接下来的章节，我们将开始进入更高级的应用程序中和一些 WebRTC 的例子。这将涉及以下话题，例如两个用户之间共享文件、开发手机端的 WebRTC 应用，也包括构建两个以上用户的网络。



由 扫描全能王 扫描创建

# 7

## 文件共享

结合浏览器中其他有用的技术，数据通道才能真正发挥出它的优势。如果能够使浏览器拥有点对点（peer-to-peer）的数据传输能力并且与文件 API 相结合，便能够让浏览器创造无限可能。这意味着，你能够在自己的应用中添加文件共享功能，并且能够面向所有拥有 internet 连接的用户。在这一章中，我们研究利用 WebRTC 的数据通道（Data Channel）以及文件 API 来构造一个简易的文件共享应用。

本章所构建的是在两个用户（peer）间共享数据的应用。该应用的基本要求是实时性（real-time），这意味着，两个用户必须同时在页面上，以共享一个文件。用户需要经历以下步骤来实现文件共享：

1. 用户 A 打开页面，输入一个唯一的 ID 号（unique ID）。
2. 用户 B 打开同样的页面，输入与用户 A 相同的 ID 号。
3. 两个用户使用 RTCPeerConnection 实现互联。
4. 一旦链接建立，其中的一个用户能够选择一个本地文件用于共享。
5. 另一个用户会在文件共享时收到通知，共享的文件可以通过链接传输到对方的计算机并且能够下载。

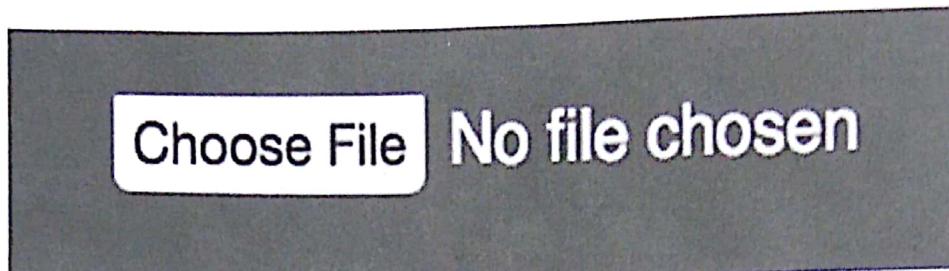
本章最关注的一个点是如何以一种新而有趣的方式使用数据通道。我们能够从浏览器中拾取文件，将文件分块并仅使用 RTCPeerConnection API 来传送给另一个用户。本章将重点阐述这种 API 所提升的交互性能，这种交互性能被应用在简单的项目中。在读完此书后，能够简单地对其进行扩展。



由 扫描全能王 扫描创建

## 使用文件 API 拾取文件

本节我们需要研究的是如何使用文件 API 从用户的计算中拾取文件。你可能有很多次接触文件 API 的经历，但是你丝毫没有意识到。这种 API 一般由标有“浏览（Browse）”或者“选择文件（Choose File）”的按钮和一个输入框表示。如图：



虽然文件 API 已经存在很长的时间，带来最深刻影响的依然是 1995 年那个最古老的版本。这个基于 HTML 表单的控件允许用户选择本机的一个文件，通过表单上传至服务器。在此之前，开发者只能指望使用第三方的工具来请求用户的本地文件。这个表单控件的问世，使得用户可以以一种统一的方式来上传本机文件至服务器，以此来提供下载、保存甚至交互的功能。当然，最初版本的文件 API 只能通过 HTML 表单实现。而 JavaScript 则无法实现文件 API 的交互。

在前端技术日新月异的今日，随着 HTML5 时代的到来，我们拥有了一个完全符合我们需求的文件 API，这个新的空间为 Web 应用实现对本机文件的操作处理打开了一扇门。这不仅意味着用户能上传文件至服务器，更意味着 Web 应用能够读取不同格式的本地文件，处理文件，甚至使用文件内容做很多事情。

虽然这些 API 有许多优秀的特性，本章只关注其中一小部分功能。即要求用户上传指定文件，并获取该文件的二进制内容。这与 Windows 系统提供的 notepad 应用比较相近——notepad 可以让用户交互地打开本地的文件并显示出来，文件 API 可以让用户在 Web 应用中打开和其他程序一样的二进制文件。

这便是使用文件 API 所带来的优越之处：它能够显示在一个 HTML 页面上，并且在大多数浏览器中都能够正常工作。就像是我们为 WebRTC 构建的演示（Demo）页面一样。在构造应用之前，需要将当前页面与一个简单的 Web 页面放在一起。这个例子很像上面讲到的一个，我们要将页面托管至一个静态页面服务器上。读完这本书，你将成为一个专业的单页面应用开发者！好，一起来看下面这段实现文件共享的 HTML 代码：

```
<!DOCTYPE html>
```



由 扫描全能王 扫描创建

```

<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Learning WebRTC - 第7章：文件共享</title>
  </head>
  <style>
    body {
      background-color: #404040;
      margin-top: 15px;
      font-family: sans-serif;
      color: white;
    }
    .thumb {
      height: 75px;
      border: 1px solid #000;
      margin: 10px 5px 0 0;
    }
    .page {
      position: relative;
      display: block;
      margin: 0 auto;
      width: 500px;
      height: 500px;
    }
    #byte_content {
      margin: 5px 0;
      max-height: 100px;
      overflow-y: auto;
      overflow-x: hidden;
    }
    #byte_range {
      margin-top: 5px;
    }
  </style>

```



```

        }
    </style>
</head>
<body>
    <div id="login-page" class="page">
        <h2>Login As</h2>
        <input type="text" id="username" />
        <button id="login">Login</button>
    </div>
    <div id="share-page" class="page">
        <h2>File Sharing</h2>
        <input type="text" id="their-username" />
        <button id="connect">Connect</button>
        <div id="ready">Ready!</div>
    </div>
    <br />
    <br />

    <input type="file" id="files" name="file" /> Read bytes:
    <button id="send">Send</button>
</div>

<script src="client.js"></script>
</body>
</html>

```

到这里，这个页面很容易读懂了。我们可以简单地实现基于 CSS 的显示和隐藏，就像之前叙述的那样。唯一的不同是文件输入样式的不同。我通过调换背景色来加以区分。

## 准备我们的页面

在开始编码之前，需要强调的是我们的应用基于上一章所开发的应用。你可以将代码复制一个副本到新的文件夹中，并将上一节开发的 HTML 加入这个项目。你还需要实现我们的 JavaScript 文件中的所有步骤来让两个用户登录、创建一个基于 WebRTC 的点对点连接和一条数据管道。要构建这个页面，请将下面的 JavaScript 代码复制至你的项目中：



由 扫描全能王 扫描创建

```

var name,
connectedUser;
var connection = new WebSocket('ws://localhost:8888');

connection.onopen = function () {
    console.log("Connected");
};

// 通过回调函数处理所有的消息
connection.onmessage = function (message) {
    console.log("Got message", message.data);

    var data = JSON.parse(message.data);
    switch(data.type) {
        case "login": onLogin(data.success); break;
        case "offer": onOffer(data.offer, data.name); break;
        case "answer": onAnswer(data.answer); break;
        case "candidate": onCandidate(data.candidate); break;
        case "leave": onLeave(); break;
        default: break;
    }
};

connection.onerror = function (err) {
}

```



```

        console.log("Got error", err);
    };

    // 以JSON形式发送消息
    function send(message) {
        if (connectedUser) {
            message.name = connectedUser;
        }

        connection.send(JSON.stringify(message));
    };

    var LoginPage = document.querySelector('#login-page'),
        usernameInput = document.querySelector('#username'),
        loginButton = document.querySelector('#login'),
        theirUsernameInput = document.querySelector('#their-username'),
        connectButton = document.querySelector('#connect'),
        sharePage = document.querySelector('#share-page'),
        sendButton = document.querySelector('#send'),
        readyText = document.querySelector('#ready'),
        statusText = document.querySelector('#status');

    sharePage.style.display = "none";
    readyText.style.display = "none";

    // 用户单击按钮时登录
    loginButton.addEventListener("click", function (event) {
        name = usernameInput.value;

        if (name.length > 0) {
            send({
                type: "login",
                name: name
            });
        }
    });
}

```



由 扫描全能王 扫描创建

```

function onLogin(success) {
  if (success === false) {
    alert("Login unsuccessful, please try a different name.");
  } else {
    LoginPage.style.display = "none";
    sharePage.style.display = "block";
  }
}

// 为每个请求建立连接
startConnection();

}

var yourConnection, connectedUser, dataChannel, currentFile,
currentFileSize, currentFileMeta;

function startConnection() {
  if (hasRTCPeerConnection()) {
    setupPeerConnection();
  } else {
    alert("Sorry, your browser does not support WebRTC.");
  }
}

function setupPeerConnection() {
  var configuration = {
    "iceServers": [{ "url": "stun:stun.1.google.com:19302" }];
  };
  yourConnection = new RTCPeerConnection(configuration, {optional: []});
}

// Setup ice handling
yourConnection.onicecandidate = function (event) {
  if (event.candidate) {
    send({
      type: "candidate",
      candidate: event.candidate
    });
  }
}

```



```

    });
}

};

openDataChannel();
}

function openDataChannel() {
    var dataChannelOptions = {
        ordered: true,
        reliable: true,
        negotiated: true,
        id: "myChannel"
    };

    dataChannel =
        yourConnection.createDataChannel("myLabel", dataChannelOptions);

    dataChannel.onerror = function (error) {
        console.log("Data Channel Error:", error);
    };

    dataChannel.onmessage = function (event) {
        // 文件接收到消息后的处理函数
    };

    dataChannel.onopen = function () {
        readyText.style.display = "inline-block";
    };

    dataChannel.onclose = function () {
        readyText.style.display = "none";
    };
}

function hasUserMedia() {
    navigator.getUserMedia = navigator.getUserMedia || "getUserMedia";
    navigator.webkit GetUserMedia || navigator.mozGetUserMedia || "getUserMedia";
}

```



```

navigator.msGetUserMedia;
    return !!navigator.getUserMedia;
}

function hasRTCPeerConnection() {
    window.RTCPeerConnection = window.RTCPeerConnection || 
window.webkitRTCPeerConnection || window.mozRTCPeerConnection;
    window.RTCSessionDescription = window.RTCSessionDescription || 
window.webkitRTCSessionDescription || 
window.mozRTCSessionDescription;
    window.RTCIceCandidate = window.RTCIceCandidate || 
window.webkitRTCIceCandidate || window.mozRTCIceCandidate;
    return !!window.RTCPeerConnection;
}

function hasFileApi() {
    return window.File && window.FileReader && window.FileList && 
window.Blob;
}

connectButton.addEventListener("click", function () {
    var theirUsername = theirUsernameInput.value;

    if (theirUsername.length > 0) {
        startPeerConnection(theirUsername);
    }
});

function startPeerConnection(user) {
    connectedUser = user;
    // 开始新建连接邀请
    yourConnection.createOffer(function (offer) {
        send({
            type: "offer",
            offer: offer
        });
    });
}

```



```

        yourConnection.setLocalDescription(offer);
    }, function (error) {
        alert("An error has occurred.");
    });
}

function onOffer(offer, name) {
    connectedUser = name;
    yourConnection.setRemoteDescription(new RTCSessionDescription(offer));
    yourConnection.createAnswer(function (answer) {
        yourConnection.setLocalDescription(answer);
        send({
            type: "answer",
            answer: answer
        });
    }, function (error) {
        alert("An error has occurred");
    });
}

function onAnswer(answer) {
    yourConnection.setRemoteDescription(new RTCSessionDescription(answer));
}

function onCandidate(candidate) {
    yourConnection.addIceCandidate(new RTCIceCandidate(candidate));
}

function onLeave() {
    connectedUser = null;
    yourConnection.close();
    yourConnection.onicecandidate = null;
}

```



```
setupPeerConnection();
```

```
};
```

相当一部分代码和本书中其他代码如出一辙。首先设置好对屏幕上元素的引用，然后将用户连接（peer connection）设置为准备链接的状态。当用户决定登录，应用向服务器发送一个登录信息。该信息可以由第 4 章中的“识别用户”一节的方法构造。当用户成功登录，服务器会向用户返回一条成功信息告知。

到这里为止，我们实现了有一个用户连接到另一个给出了用户名的 WebRTC 用户。通过发送邀请和响应，将两个用户连接在一起。当连接建立，用户可以用过数据通道来回发送任意二进制信息。

这个过程非常简单，希望你能够在最快的时间完成编码并且运行起来，如果你还存在任何疑问，可以参考本书各章节中你遗漏的部分。这是本书最后一次提及这段代码，在进一步学习前，请做好充分的准备。

## 获取对文件的引用

我们已经能够完整地将一个页面跑起来。现在可以开始进行应用中文件共享的部分。在文件共享中，用户第一件需要做的事情是从一个本地的文件系统中选取一个文件。这个可以简单地通过输入控件（input element）来实现。浏览器允许用户在本地的文件系统中选择一个文件，并为接下来的应用保留对选择文件的引用。

当用户按下“发送（send）”按钮，我们希望得到对用户所选择文件的引用。为此，需要为此添加一个事件监听器，代码如下：

```
sendButton.addEventListener("click", function (event) {
  var files = document.querySelector('#files').files;

  if (files.length > 0) {
    dataChannelSend({
      type: "start",
      data: files[0]
    });

   .sendFile(files[0]);
  }
});
```



});

你或许会为了代码为何如此简单而感到惊讶。这便是使用浏览器完成工作的魅力所在，很多事情已经为你达成了。

这里，我们将通过 `input` 控件来获取对选择文件的引用。`Input` 控件能够支持单个文件的选取以及多个文件的选取。在选择文件之后，我们随后告诉另一方文件已经准确完毕可以随时传输，并调用 `sendFile` 函数。`sendFile` 的实现会在本章后面的内容中介绍。

你可能会认为我们所得到的文件对象包含了用户所选择的文件中的所有数据。实际上，得到的文件对象仅包含了文件的元数据。可以一起看下：

```
{
  lastModified: 1364868324000,
  lastModifiedDate: "2013-04-02T02:05:24.000Z",
  name: "example.gif",
  size: 1745559,
  type: "image/gif"
}
```

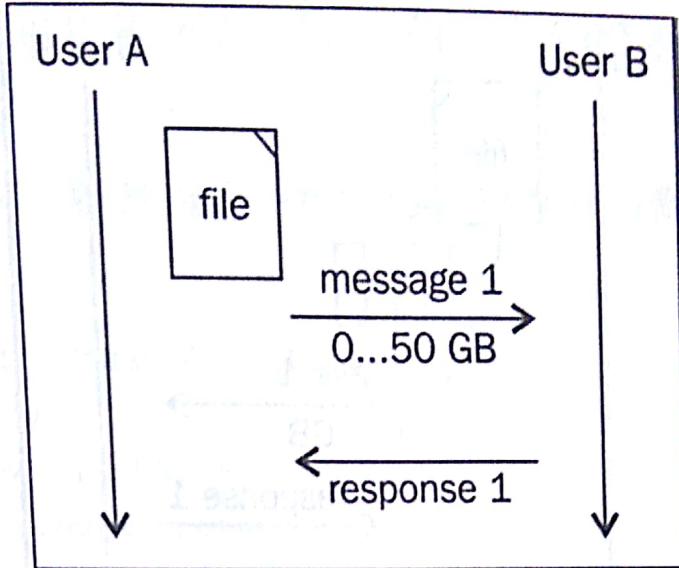
文件的元数据能够提供给我们所需告诉另一个用户的信息。以上面的元数据为例，我们可以告诉对方我们准备发送一个叫 `sample.gif` 的文件，以及其他的信息包括文件类型和最近修改的时间。接下来要做的就是读取文件的内容并且通过数据通道进行发送。这不是一个简单的过程，但是可以借助特殊的逻辑来实现。

## 文件分块

至此，我们成功地获得了对一个文件的引用。现在要做的是完成发送之前的准备工作。解决发送问题最简单的方式便是直接调用 `send` 函数来发送整个文件。这么做会使得整个文件以一个大体积消息的形式给到数据通道，即如下图的情况：



由 扫描全能王 扫描创建



一旦出现这种状况，数据通道只能机械地在用户间传送整个文件。假设我们的用户尝试传输他们保存在本地的整个音乐专辑，共 50GB。那么数据通道便会通知 SCTP 协议，本次要传输的消息地址从 0 字节开始，一直到 50GB 结束。

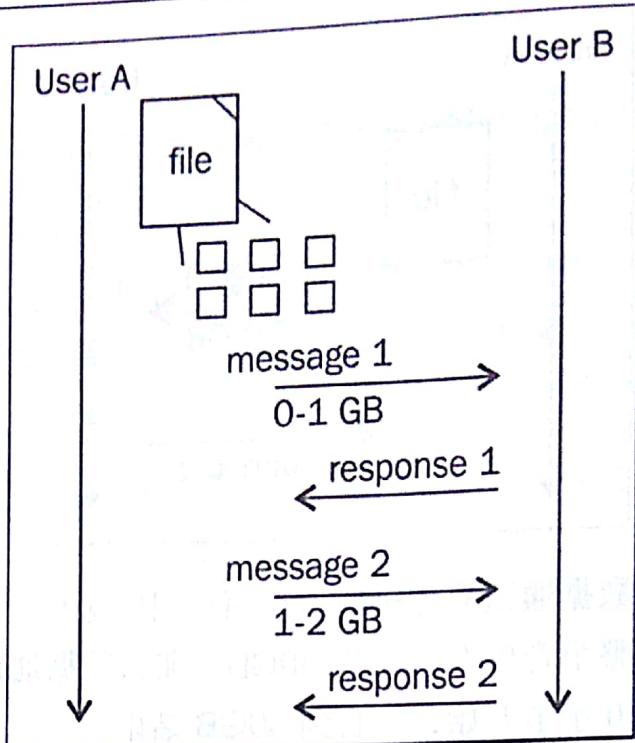
这便会引起一个很大的问题。即在网络条件不佳的情况下，任何一条消息都会存在发送失败的风险。如果我们将对数据通道设置了可重试，那么数据通道在发现消息发送失败后放弃整个 50GB 的消息传输，重头再来。也就是说，在以单一消息的形式发送 50GB 的文件时，两个用户必须在几个小时内都保持畅通及良好的网络条件，否则如此大体积的一个消息发送失败的风险是很大的！

为了能够解决这个问题，我们需要将文件分块发送。如果你曾经有使用 BitTorrent 的经历，你会很容易理解这种工作方式。BitTorrent 会将一个文件切分成几百甚至上千份小块，这样每一个小块都能很有效地在用户间相互传递。这个过程也能够被理解为集装箱式的货运，你只有将东西切分成大小差不多的小块，才能获取比运输整块东西更高的运输效率和更低的运输成本。

将这个理念应用到我们要传输的文件。将这个二进制文件理解为一行很长的 01 串。最简单的分块方法便是每次读取一行固定长度的二进制内容并且发送。分块与发送持续推进直至文件结尾。如果失去连接，我们只需要重试发送最近发送的小块即可，已经发送进度不会丢失。



由 扫描全能王 扫描创建



这是使用底层传输系统的一个弊端。很多简单的步骤没能够一步到位的实现。因为这个接口是为了保证对需求的通用性，所以在使用数据通道之前必须多实现一步。

## 使文件分块可读

另一个我们没有提及的话题就是具体怎么通过数据通道来发送二进制数据。在之前的例子中间，所有数据都是以单条文本的形式出现的。这是因为应用设定这些都是简单的字符串消息。而对于文件 API，情况就大不相同了。像图像、音频、视频及其他二进制文件很难直接转为人类可读的字符串，所以必须关注如何发送二进制文件。

到这里你可能会问，为什么不能直接通过文件 API 进行数据发送。这是很有道理的，因为可以在技术上实现双方客户端使用同样的语言，并且数据通道也能够支持这种语言。对么？

可是，生活是不会这么容易的！很不幸地，在 JavaScript 和底层网络协议之间有太多层。这些层次有极度特殊的要求，要求你将数据翻译成可读的字符串格式。

为解决这个棘手的问题，我们在之后的过程中采用 Base64 编码方式。Base64 编码是一种二进制转换为字符串的编码方式。这种编码方式可以使得二进制数据可以被简单地传送，因为它可以将任意二进制数据转为 ASCII 编码的字符。对于 JavaScript 和其他网络协议，ASCII 字符更容易在不同的平台上被传送。这种编码的使用方式是在发送文件之前将



由 扫描全能王 扫描创建

其转换为 Base64 编码格式的数据，传送至另一个客户端后对数据进行解码，即得到与源数据文件相同格式的数据。

在正式开始编码之前，我们先定义一些用于编码和解码的简单函数。可以在你的项目中的任何一处进行定义：

```
function arrayBufferToBase64(buffer) {
    var binary = '';
    var bytes = new Uint8Array( buffer );
    var len = bytes.byteLength;
    for (var i = 0; i < len; i++) {
        binary += String.fromCharCode( bytes[ i ] );
    }
    return btoa(binary);
}
```

以上的函数接受一个 `ArrayBuffer` 对象作为参数。该对象是文件 API 读取文件内容时的返回值。编码函数首先分配一个新的数组，然后遍历二进制数据块并转换为字符，然后使用浏览器内置的 `btoa` 函数对字符加以修改翻译。在获取了编码函数之后需要在另一端建立解码函数：

```
function base64ToBlob(b64Data, contentType) {
    contentType = contentType || '';
    var byteArrays = [], byteNumbers, slice;
    for (var i = 0; i < b64Data.length; i++) {
        slice = b64Data[i];
        byteNumbers = new Array(slice.length);
        byteNumbers = new Array(slice.length);
        for (var n = 0; n < slice.length; n++) {
            byteNumbers[n] = slice.charCodeAt(n);
        }
        var byteArray = new Uint8Array(byteNumbers);
        byteArrays.push(byteArray);
    }
}
```



```
var byteArrays = new Uint8Array(translatedText);
var blob = new Blob(byteArrays, {type: contentType});
return blob;
```

这个函数比编码函数稍微复杂一些。该函数的第一步做的是与编码函数相反的，即遍历数组并将中间的每一个字符转化成二进制数据。在得到翻译后的数组之后，需要将它转换为 blob。如此，JavaScript 可以简单地与该数据进行交互，并且可以保存为文件。这点在本章的后面几节会介绍。

## 文件读取与发送

本节介绍具体如何从文件中读取二进制数据并且发送给另一个用户。在这里我们将数据通道和 Base64 编码有效地结合。首先，我们实现在“获取文件引用”一节中间提到的 sendFile 函数：

```
var CHUNK_MAX = 16000;
function sendFile(file) {
    var reader = new FileReader();

    reader.onloadend = function(evt) {
        if (evt.target.readyState == FileReader.DONE) {
            var buffer = reader.result,
                start = 0,
                end = 0,
                last = false;

            function sendChunk() {
                end = start + CHUNK_MAX;
                if (end > file.size) {
                    end = file.size;
                    last = true;
                }
                dataChannel.send(arrayBufferToBase64(buffer.slice(start,
end)));
            }
        }
    }
}
```



由 扫描全能王 扫描创建

```

        // 如果是最后一块数据的话就发送消息，不然继续发送数据
        if (last === true) {
            dataChannelSend({
                type: "end"
            });
        } else {
            start = end;
            // 防止数据溢出
            setTimeout(function () {
                sendChunk();
            }, 100);
        }
    }
    sendChunk();
}
}

reader.readAsArrayBuffer(file);
}
}

```

一个需要注意到的地方是我们实例化了一个 `FileReader` 对象。这是一个文件 API 专属的对象。这个对象封装了 JavaScript 中使用不同格式读取文件的方法。在这个例子中，我们以 `ArrayBuffer` 的形式读取文件。这种方式适用于从文件中读取底层的二进制数据。

当整个文件被读取，需要走以下的流程：

1. 确认 `FileReader` 对象在 `DONE` 的状态。
2. 初始化并获取文件数据的缓冲区引用。
3. 建立一个递归函数，实现发送文件块的功能。
4. 在函数中，我们从 0 开始读取一个文件块的字节。
5. 在确保没有超过文件尾，否则没有数据可以读取。
6. 将数据通过 `Base64` 格式进行编码，并且进行发送。



7. 如果是最后一个文件块，告诉另一个用户我们已经完成文件发送。

8. 如果还有数据需要传输，在固定的时间后发送另一个分块防止 API 发生泛洪 (flooding)。

9. 最后通过调用 `sendChunk` 函数开始递归过程。

到此为止，可以尝试使用文件 API 实现文件的发送。你可以连接两个用户，选择文件，然后在另一端查看接收的文件块。可以尝试通过这个例子发送大型的文件和小文件，体会区别。你同样可以尝试发送不同格式的文件，例如 PNG 和 TXT。

## 在“另一端”组合文件块

接收完成后，在另一端的用户收到了一系列的文件分块。因为我们在数据通道中使用了 `Ordered` 选项，用户所接收到的文件分块是有序的。这意味着这些区块可以在“另一端”被组合并还原成原文件。

为了存储数据块，我们可以将其放入一个简单数据结构。可以在数据通道报告消息已经收到的时候直接进行这一步。还会用到在本章前两节中使用到的解码函数，将文件块组合成正确的格式：

```
// 新建一些全局变量
var currentFile = [], // 当前正在处理的文件块
    currentFileMeta; // 当前正在处理的文件元数据

// 把以下功能绑定到openDataChannel函数
dataChannel.onmessage = function (event) {
  try {
    var message = JSON.parse(event.data);

    switch (message.type) {
      case "start":
        currentFile = [];
        currentFileMeta = message.data;
        console.log("Receiving file", currentFileMeta);
        break;
    }
  } catch (e) {
    console.error(e);
  }
}
```



```

        case "end":
            saveFile(currentFileMeta, currentFile);
            break;
    }
} catch (e) {
    // 假定这是文件数据
    currentFile.push(atob(event.data));
}
};

```

你可以看到我们加入了两个全局变量来存储当前要发送的文件数据，以及对应描述文件的元数据。现在我们可以在 `openDataChannel` 函数中填写 `onmessage` 句柄。首先，尝试使用 `JSON` 来对消息的内容进行解码。名为 `JSON.parse` 的 API 在无法对数据进行解码的情况下将抛出一个错误。这意味着数据不是 `JSON` 格式的。使用这个机制来识别消息是一个文件还是来自于另一个用户的命令。

我们有两个 `JSON` 命令，分别是 `start` 和 `end` 触发器。`start` 触发器告知我们需要设置我们的局部变量来接受新的文件数据并且传递文件的信息。`end` 触发器告知我们文件传输已经结束，可以将文件保存至用户的硬盘中。

如果这条消息不是 `JSON` 格式的，我们就假设这是一个文件的分块。这个数据将被解码并且插入到数据列表中。直到稍后将这个文件保存，这个列表将包含所有获得的文件分块。

当最终获取了结束的事件，需要将文件保存到用户的系统中。浏览器的一个很大的短板在于它是一个沙盒环境。这意味着任何在浏览器环境中发生的事情都不可能以任何形式来影响用户的操作系统。这是一种防止恶意网站通过某种方式来修改和删除用户系统中文件的方法。相反，我们将使用一种安全的方式来将文件保存至系统中。

我们的做法是将文件以一种和用户在站点上下载文件一样的方式保存到系统中。文件下载的典型方式是站点将在你不想整天听自己说话的时候存放于服务器端，然后向用户提供一个指向文件的 URL。当浏览器识别出这个链接是指向一个文件的时候，根据文件的类型，它将会弹出对话框让用户选择是保存还是直接在浏览器中查看。使用相同的策略，我们将函数 `saveFile` 加到我们的项目中：

```
var blob = base64ToBlob(data, meta.type);
```



由 扫描全能王 扫描创建

```

    var link = document.createElement('a');
    link.href = window.URL.createObjectURL(blob);
    link.download = meta.name;
    link.click();
}

```

我们的函数做的第一件事情就是获取我们的文件数据然后转化成几个块（blob）。这个转化是通过我们在“文件分块”一节下的“使区块可读”子节中介绍的 Base64 编码方法来完成的。接下来，我们可以创建一个新的 link 对象，可以在用户单击该对象的时候模拟下载。我们可以将链接的位置设置为一个伪 URL，指向我们的数据。

我们使用另一个叫作 `createObjectURL` 的浏览器 API 来创建这个伪的位置。这是一个特定的函数，用来为文件和基于块的 JavaScript 对象来创建位置。当我们有了这个伪的位置，我们可以将它赋值给链接对象，这样当用户单击它的时候浏览器会弹出对这个位置的导航。因为我们的位置指向的是一个文件，弹出的对话框会询问是否需要下载文件。

## 向用户展示进度

最后一个能向我们的应用添加的功能是一个能够显示传输文件大小和剩余大小的进度条。这个进度条能够显示传输时的进度变化，提升传输时的用户体验。我们需要向用户展示文件发送了多少以及文件接收了多少。

为了展示文件的多少部分已经被发送，可以向 `sendChunk` 函数中间添加如下代码：

```

if (end > file.size) {
  end = file.size;
  last = true;
} // 已存在的代码

var percentage = Math.floor((end / file.size) * 100);
statusText.innerHTML = "Sending... " + percentage + "%";

```

这里我们仅仅获取了当前发送的字节数，然后除以整个文件的大小来得到一个百分比。可以在接受端做同样的事情：



```

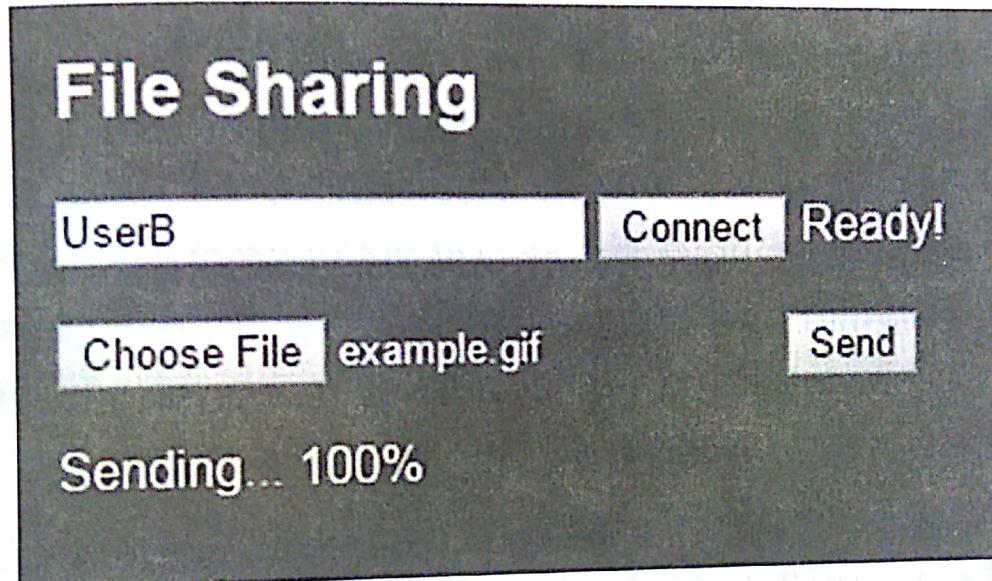
currentFile.push(atob(event.data)); // 已存在的代码

currentFileSize += currentFile[currentFile.length - 1].length;

var percentage = Math.floor((currentFileSize /
currentFileMeta.size) * 100);
statusText.innerHTML = "Receiving... " + percentage + "%";

```

在接收端，当前接收的比特数可以由第一个文件分块到最近接受的文件分块的大小加和得到。你现在可以得到如下简单但是有效的文件共享应用：



两方的用户现在都可以看到文件传输完成的百分比。在两台不同的计算机之间进行文件传输来观察数据通道的性能是一种非常有趣的实验。同时，你可以尝试不同大小的文件来观察不同的结果。

## 自测题

Q1. 最新版本的文件 API 在 HTML5 中的升级提供了比以往多得多的功能。对或错？

Q1. 最新版本的文件 API 在 HTML5 中的升级提供了比以往多得多的功能。对或错？

Q2. 将文件分块能够使如下哪种情况获益：

1. 在较差的网络条件下发送文件
2. 恢复传输中丢失的文件块
3. 发送超大文件
4. 上述所有



Q3. 既然发送和接受双方的计算机使用同样的二进制语言，我们能够简单地通过数据通道进行发送。对或错？

Q4. 在本章的例子中间，保存一个文件更像是：

1. 在 Microsoft Word 中间保存文档
2. 从一个链接中下载文件
3. 复制/粘贴文件

Q5. 文件 API 在一个沙盒环境中间运行，这意味着它不能像其他原生应用一样工作。对或错？

## 小结

我们能够在一个小的章节中间结束对文件 API 的介绍真的很神奇。本章介绍的这个例子和其他的文件共享应用的工作原理基本一致。可以仅仅使用几百行代码来将 API 组合在一起，创建一个功能完备的应用。这个应用还能在一些方面进行可优化，譬如较差的网络条件和超大文件。

这是一个非常值得你继续研究和扩展的例子。有很多地方值得你继续尝试，让这个应用变得更加好用。第一步是提供更多的信息给用户，例如在百分比之外提供文件名以及文件大小。还可以提供进度以外的传输速度等其他信息。用户甚至能够提供下载文件的时间而不是立即进行下载，或者是在下载后对文件进行重命名。

本章介绍的应用在处理大文件时有一个缺点。当前在传输大文件时会将整个文件载入内存并且转化为一个大文件下载。所以其在传输大文件时会遇到问题，因为这会消耗大量的内存，甚至超过计算机的内存上限。所以可以通过一些方法来挑战 10GB 甚至 15GB 的大文件。

在本书的最后一些内容中，我们将介绍一些在数据通道使用中的进阶技术。这些技术是你在将本书介绍的数据通道应用于产品级别的环境中非常值得借鉴的。我们将要深挖 WebRTC 究竟能够支持多大规模的应用，以及在面对大规模应用时将会遇到的问题。



由 扫描全能王 扫描创建

# 8

## 高安全性与大规模优化

读到这里，我们已经能够覆盖 WebRTC 的基本内容了。但是这些内容仅仅局限于在一台本地的计算机上实现某个章节介绍的内容，而没有接触到一项真正的服务。对于一个仅涉及你和你朋友的应用来说，这些技术已经很不错了。但事实上，它们不能引导你去开发一个能够连接来自世界各地上千个用户的应用。为了使本书能够达到这个效果，本章将研究一些更深层次的、更高级的问题，比如安全以及支持大规模网络。

本章的主要目标是提供一些与这些高级问题有关的信息，以便你在结束本书的学习后利用剩余时间进行深层次的研究。绝大部分信息是概念性的，我们不会在本章展示一个示例应用。在本章的最后，你会对如何建设一个大规模的基于 WebRTC 的服务有一定的基本认识以及如何获取更多的信息。

边阅读边研究，是学习不断发展的最新技术的法宝。当你阅读本章时，可以在获得一些新的概念时停下来，尝试使用你学到的 WebRTC 知识来建立原型并对这些概念进行研究。当你能在实现自己的 WebRTC 应用时能够爆出自己的一些想法时，你便成功了。

### 保护信令服务器

之前所构建的一些应用的主要目的是学习一个单一的 WebRTC 功能。这意味着很多细枝末节的东西被一些捷径所替代。整体的效率也因为需要更清楚展示的需求而被牺牲掉。对于我们的信令服务器，虽然它能够工作，但是这个原始的环境需要进一步升级以及添加大量的特性才能够实际使用。



由 扫描全能王 扫描创建

## 使用编码

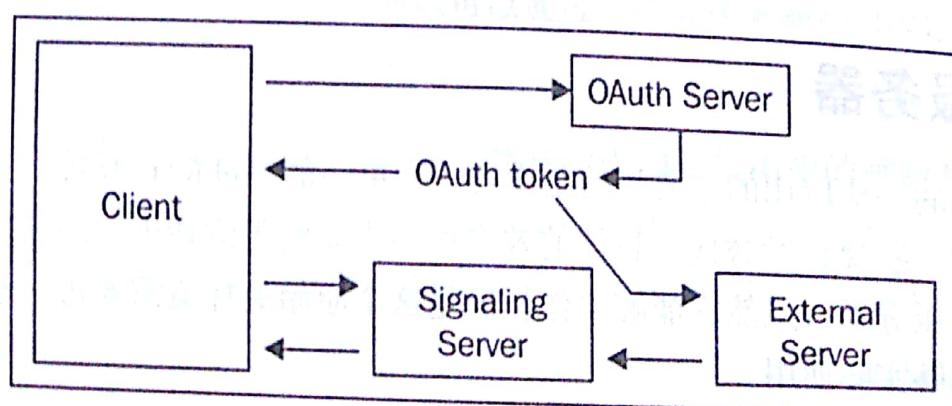
应用最需要添加的部分便是强制地加密。对信令服务器的消息进行加密可以保证没人可以对服务器的消息进行拦截并以此分析是哪个客户端与哪客户端进行通信的。这是目前构建的信令服务器最大的一个缺陷。当然这个缺陷也是最容易被填补的：加密技术是当前标准化程度最高、通用性最佳的一种方式。

我们的信令服务器将要使用两种标准的加密方式：HTTPS 和 WSS。你应该能够认出 HTTPS 是当前 HTTP 站点通行的标准 SSL 加密方式。这使加密方式可以很简单地通过互联网中提供的资源来实现。WSS 是一种运行于 TLS 协议上的基于 WebSocket 的 SSL 加密方式。这种加密方式在你的 web 服务器提供正确的证书时的工作方式与 HTTPS 是一致的。对这两种加密方式进行深刻的了解和研究对你使用 WebRTC 开发真实应用大有益处。

## 使用 OAuth 提供器

另一个对我们应用进行的升级是集成一个身份提供器。身份提供器是一个可以让你利用第三方的用户身份数据来标识和跟踪当前应用的用户的软件。一个典型的例子就是 Google Login。它可以为每一个个体的用户提供一个唯一的 ID 以及他们的联系人列表。同时它会将信令服务器的安全等级提升一个档次，因为只有两个相互认识对方身份的用户才能相互联系。

当今开放的网络使得用户可以有很多选择来对用户进行身份认证。OAuth 是目前最流行的身份认证技术之一。OAuth 是一个开放标准的集合，使用户可以授权对他们所属数据的访问。在例子中，希望我们的用户能够给我们一个唯一的身份标识，以及一个他们希望联系的联系人列表。这种方式可以提升系统安全性，因为只有通过认证的用户才能使用我们的应用：



我们需要研究的第一步是如何以及在服务的哪个位置添加身份提供器。在之前的章节中，我们通过一个框图展示了信令服务器和 WebRTC 通过交互来在两个用户之间建立链接。在实现的时候有如下考虑：

- 双方用户在形成链接之前都必须登录。
- 双方用户必须相互认识才能被允许进行对话。
- 双方用户在对话时不会受到网络攻击的愚弄。

为了保证双方用户在对话之前进行身份认证，身份认证机制必须放在用户与信令服务器交互之前。如果用户必须在登录之后才能做所需要的操作，服务器在用户面前曝光，让用户能够入侵的机会大大减少。

当你选择了一个 OAuth 服务时，你可以将其集成在自己的信令服务器中。如此一来，用户会在第一时间看到一个登录界面——使用你自己的或是第三方的身份认证服务来登录到你提供的服务。登录后，客户端将获取对你的 WebRTC 应用中相应数据的访问权限。

其背后的工作原理是口令（token）的使用。一个“口令”是一个包含数字与字母的随机字符串，用以表示对某一组数据访问权限的密钥。当客户端通过 OAuth 获得了权限，JavaScript 脚本从第三方获取一个口令。该口令通过 JS 脚本发送至信令服务器来通过验证并获取服务器端中用户需要的数据。

一个可以用于 OAuth 快速上手的类比是，OAuth 是 web 资源的一把贴身钥匙。这把钥匙可以给你一个访问在另一个服务器上的有限资源集合的权限。

任何一种第三方服务都会提供一个关于如何在用户自己的平台上接入认证服务的教程。记住，这里的第三方身份认证服务同样可以指你自己公司提供的服务。这种说法其实是鼓励读者们将自己的公司的 OAuth 登录服务与多个产品线分离开来，以获得更好的可维护性和可伸缩性（scalability）。在此不赘述关于 OAuth 的更多细节，而是留给用户做进一步的自主研究。

使用加密手段以及集成基本 OAuth 服务将赋予你的应用最基本的安全特性，保证只有获得认证的用户才能连接到你的应用。当然，对于用户认证和安全性而言，这只是冰山一角。



角。我们能够确定的是，安全性对于任何应用而言都是非常重要的。

## 支持移动设备

当通过移动设备访问 Web 的大潮袭来，几乎所有的开发者必须要应对来自于智能手机与平板的 WebRTC 请求。这是一种与基于 Web 的通信的想法配对的媒介。而依据流量套餐收费标准进行优化，并且坚持使用一个数据服务是一种非常具有吸引力的概念。同时，移动设备上展现的 Web 页面可以很容易地访问与用户移动设备集成的相机，而不是像 PC 应用一样要求用户在使用前需要购买昂贵的摄像头。在另一方面，紧缺的电池资源，以及基于蜂窝网络的链接将是基于流（streaming）的应用的挑战。

当我们一台 PC 与智能电话之间传输数据时，第一个遇到的问题将会是数据链接。即使运营商提供的网络再优秀，其传输速度仍然会逊于通过双绞线进行 internet 连接的 PC。当智能手机处于一个不佳的网络环境中时，WebRTC 提供的通话质量将会远不及运营商提供的手机通话服务。

一个在智能手机上经常会被提及和研究的问题便是连通性问题。我们来比较一下 WebRTC 提供的通话服务以及你手机默认的通话套餐。即使你有一百万的用户一天只进行一次通话，而 99% 的接通成功率仍会造成一天一万次的通话失败。而由于需要兼容大量的不同设备的现实，以及需要大量服务器和实际进行通话所需要的过程，这种情况会变得无比复杂。更不用说手机生产商被强制要求提供对紧急通信的服务以及支持多个网络的备用模式。这是一个非常严峻的问题。

为了解决数据链接问题，我们需要尽可能地减少通过网络发送的数据总量。发送的数据量由我们拍摄和发送的视频的帧数和每一帧的大小决定。这些数据是会被加密并且分包的，并一帧一帧地发送到另一个客户端。为了减少数据量，需要在传输之前减少用户拍摄的视频大小。以下是我们的应用所需要用到的代码：

```
var mobile = {
  video: {
    mandatory: {
      maxWidth: 640,
      maxHeight: 360
    }
  }
}
```



由 扫描全能王 扫描创建

```

};

var desktop = {
  video: {
    mandatory: {
      minWidth: 1280,
      minHeight: 720
    }
  }
};

var constraints;

if(/Android|webOS|iPhone|iPad|iPod|BlackBerry|IEMobile|Opera
Mini/i.test(navigator.userAgent)) {
  constraints = mobile;
} else {
  constraints = desktop;
}

navigator.getUserMedia(constraints, success, function (error) {
  // 有可能的话在另外一种分辨率下尝试一下
});

```

需要对每一种设备定义一个集合的约束 (constraint)。在这个例子中，我们仅仅对移动设备和 PC 分别设定一个约束集合。只需要通过改变 UA (user agent) 值来查看约束集合是否使用正确。UA 字符串被传给 getUserMedia 函数作为参数，该函数通过浏览器来判断这些约束是否能够被支持。如果不是，函数会调用错误回调函数。在错误回调函数中，你可以尝试另一种备用的约束，看是否能够成功。

你可能注意到，当一个移动设备正与一个 PC 进行通话时，移动设备会发送一个低分辨率的流而同时却收到一个高分辨率的流。这意味着我们可以根据对方客户端所使用的设备来决定发送的视频流是否可以被进一步压缩。我们可以通过在信令服务器中添加额外的命令的方式来实现。只需在客户端开始连接之前交换双方的设备类型。

这只是支持移动设备连接的一个很好的开端，还有很多的优化可以实现。例如当网络

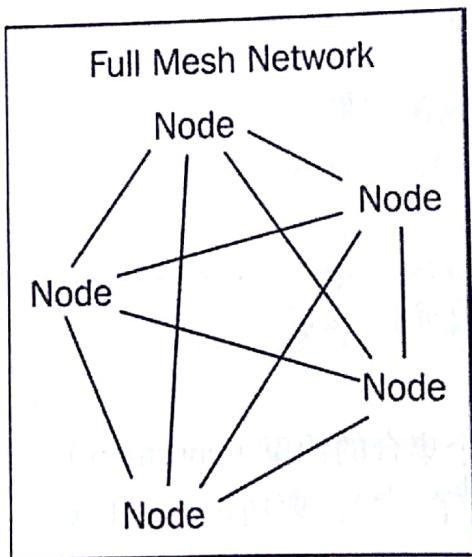


出现故障时自动重连以及根据网络条件自适应地调节发送视频的分辨率。对移动设备连接进行何种优化依赖于你的 WebRTC 应用的实际需求。同时你需要尽可能地在许多移动设备上进行测试。

## 网格网络简介

当你的 WebRTC 应用已经能够支持多种设备在安全保障下的互联时，你可能要问，我该如何将这种方式扩展到多个用户的连接？一对（one-to-one）的连接是很不错的，但是如何在一个通话中同时包含若干用户？本节我们深入介绍网格网络。

你会在实际生活中找到很多网格网络的例子。例如 Internet 就是一个很好的网格网络的例子。当你发出一个 web 页面的请求，网络上的每一个节点通过协作来向你的计算机传输这个页面。网格网络能够使得每一个节点和其他所有节点在一个全网格进行对话或者使一个子集的节点在一个部分网格中进行互联。



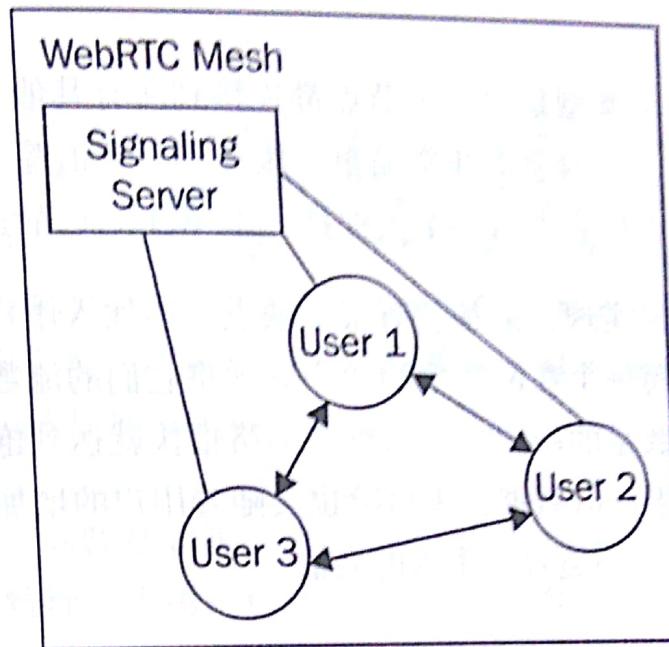
其实在你的生活中会遇到很多的网格网络，这是一个很好的机会。例如，互联网本身就是网格网络形式的。所以，当你向某一个 web 页面发送请求时，网络中的每一个节点会共同协作将这个页面传送到你的计算机上。网格网络能够在一个完全网格中让每一个节点都能与其他节点对话，而在一个部分网格中，每一个节点只能与一个子集中的节点对话。

你已经可以在点对点（peer-to-peer）技术中体会这一类概念，例如 WebRTC。当前，每一个通话都是一个全互联的网格网络：每一个节点都是一个用户，而每一次通话都是两个节点之间的连接。



由 扫描全能王 扫描创建

现在可以对这个想法进行扩展，并在网络中加入更多的节点。既然两个用户可以互联，为什么不能是三个。在 WebRTC 中，一个浏览器在同一时间所能支持的链接数量是无限的。所以任何一个在网络中间的用户都能够同时连接多个用户，即在一个通话中创建一个用户的网格：



在上面的图中每一个用户都是从连接信令服务器开始的，接下来请求对所有希望进行联系的用户的通话。每一个用户都需要通过与他们要连接的用户进行请求/响应的过程。这样便会产生 3 个 WebRTC 的连接，同时允许所有人与所有人之间的通信。

 尽管我们可以想象一个发生在 100 个人之间的通话，这并不意味着当前的技术可以支持这样的规模。每一个链接都会附加一个开销，所以当一个用户加入的时候链接的总数会加倍。在本章的后面我们会介绍如何突破简单网格规模限制。

## 网格类型

现在我们已经能够弄明白什么是网格网络，下面可以探讨各种不同网格网络类型。和之前的章节一样，我们事先定义一系列的约束条件，然后分析解释每一种网格网络在不同的约束条件下的优劣。主要目标是使用一种效率最高的方式尽可能地连接所有用户，使每个用户都能够得到最佳的体验。

这里有一系列评价每一种网格类型的简单准则：



由 扫描全能王 扫描创建

- 给定一个用户，其最低或最高的带宽是什么？
- 我希望连接的最多的用户是什么？
- 对每一个给定的用户，多少数据丢失量是可以接受的？

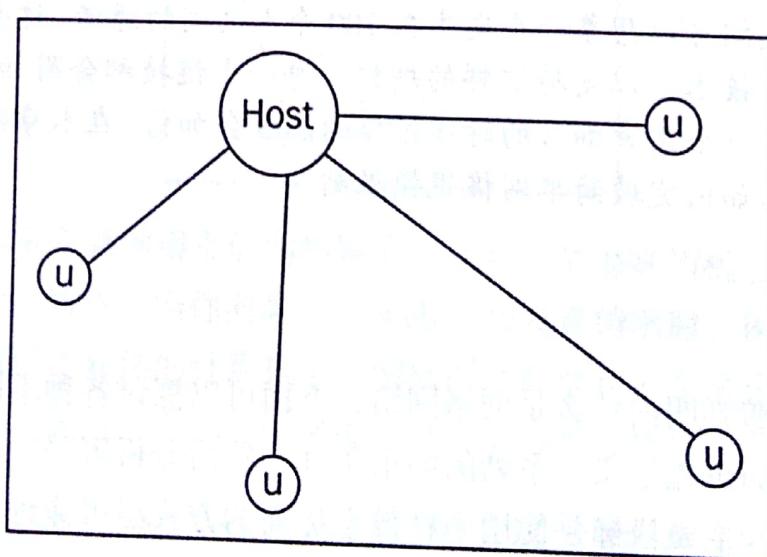
## 所有人对所有人

我们介绍的第一种网格类型是每一个节点都连接到所有其他节点。到目前为止，这种方式是实现起来最简单的，因为逻辑非常简单。服务器只要追踪每一个加入通话的每一个人并且每一个人都和其他在列表中每一个人保持一个 WebRTC 的连接。

虽然这种形式非常容易实现，但是它有很多缺点。当加入用户增加到一定数量，整个网络的带宽将急剧提升。每一个在网络中的用户必须将它们的流数据发送到网络中的其他所有用户。这意味着在一定数量的用户加入后整个网络很快就达到饱和极限，也就是这种网络的最大用户数较低，且潜在的数据丢失风险也会随着用户的增加而增加。这种策略适合在 3~4 人间进行的会话，但不适合更多人的会话。

## 星形网络

为了能够支持更多的用户，我们需要一种更聪明的方式去连接他们。一个简单的实现方法是减少一个通话中的总连接数量。可以使用一个星形的拓扑结构来减轻我们需要连接会话中所有人的问题。在一个星形的网络中，所有的用户都连接到一个主节点。该节点负责将正确的视频流发送到网络中的其他节点：



在我们例子中，主节点可以是通话中的任何一个节点。所有其他的节点都通过 WebRTC 连接到主节点。而主节点则作为中继为其他节点传递流数据。因为 WebRTC 对于浏览器是



透明的，代码可以从一个连接中获取流数据然后添加到另一个连接中。这种方式下，网络中的连接数量比完全图网络要少，因为连接数等于所有的用户数量减一。另一方面，我们仍然需要传输和完全图网络一样多，即使能够获得更多的连接数量和更少的数据丢失，但是整体收益不会太多。



这是另外一个不幸的时刻，我必须停下来然后谈谈浏览器支持问题。

不幸的是，在一个 WebRTC 连接中添加多个流在一些浏览器中仍然是一个还未实现的功能。流技术是一个广阔而复杂的技术，所以还需要一些时间才能让星型网格使用客户端真正实现。说到这里，我们将在本章下面谈论多点控制单元（Multipoint Control Unit, MCU），这将帮助我们解决这个问题。

我们需要解决的一个问题是如何选择一个节点作为此次通话的主节点。这是一个可以加入到信令服务器上的特性。当用户登录，他们可以收集一定量的用户信息并且传送给信令服务器。此时，当一个用户发起一个通话，信令服务器比较所有用户的数据并且依据以下规则选出一个作为主节点：

- 设备类型可以作为依据处理能力和带宽选择主节点用户的依据。桌面 PC 的用户会比移动设备用户具有更高的优先级。
- 视频编码的支持是另一个参考：如果一个用户具有更快、更多样的能力来对视频进行编解码，则会获得加分。
- 我们可以通过下载一个很小的文件对用户的实际带宽进行测试，据此来判断用户在什么类型的网络。

上述的最后一个规则是在当前的情况下最有用的，因为这种方式可以直接衡量用户所在的网络环境。如果一个用户在使用一个笔记本电脑，这意味着他比使用移动设备的用户具有更多处理能力，但是如果笔记本电脑是和一个连接到移动网络的手机连接在一起，他的网速不会快于一个连接到高性能的 Wi-Fi 路由器的手机。在我们的例子中实现时，尝试下载一个已知大小的图片，并且计算下载图片所需的时间：

```
var src = "example-image.jpg",
    size = 500000,
    image = new Image(),
```



由 扫描全能王 扫描创建

```

        startTime,
        endTime,
        totalTime = 0,
        speed = 0;

image.onload = function () {
    endTime = (new Date()).getTime();
    totalTime = (endTime - startTime) / 1000;
    speed = (size * 8 / totalTime); // bytes per second
};

startTime = (new Date()).getTime();
image.src = src + "?cacheBust=" + startTime;

```

这里的逻辑比较直接：

- 事先准备好一个已知大小的图片文件。
- 设置一个载入事件处理器，在图片完成下载时计算下载的速度。
- 在 JavaScript 中将下载的 startTime 属性设置为当前的日期时间。
- 设置图片的源数据，开始下载。

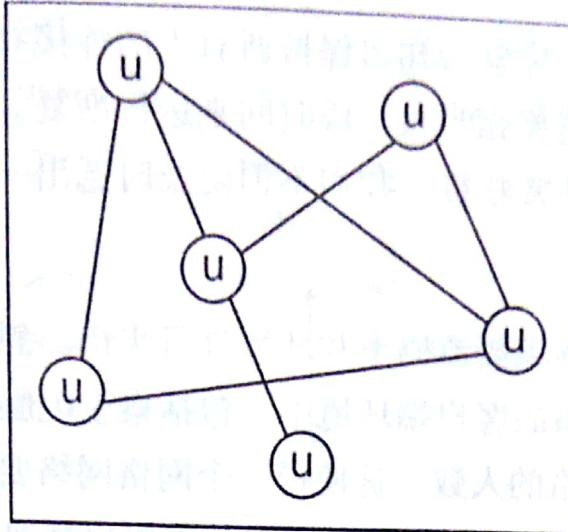
这会给我们提供关于用户网络连接有多快、质量有多高的参考信息。因为用户的网络条件会随时变化，多次重复这种测试是比较有用的。当我们收集了一定的关于用户网络条件的数据之后，可以将这些数据发送至服务器。我们的信令服务器接下来可以使用这些信息来决定最佳的节点来担任主节点。

## 部分网格

最后介绍的一种网格类型是部分网格网络。部分网格在多用户通信领域外的应用中具有良好的口碑。这种网格在用户使用数据通道时非常有用，因为网格中的每一个节点不是一直需要连接到其他每一个节点的：



由 扫描全能王 扫描创建



我们之前提到过一个很好的例子，即一个基于容错的文件共享网络。共享一个文件，你需要一定数量的数据分块，不然你是从那个节点获取的。在这种情况下，我们可以选择一组潜在的用户进行连接或者轮流尝试进行连接。这些需要连接的节点可以从网络中区别于前者节点的其他节点获取数据。这意味着我们可以依据用户拥有的连接数和带宽对网络中的用户进行组织和排序。

部分网格的基本策略是获取每个用户最优的连接数。例如，部分网格可以让每个移动的用户只连接到一个或者两个节点，而让桌面 PC 的用户连接至十个或者十五个节点。我们可以根据用户设备信息进行排序和分析，据此优化数据传输。

总体来说，部分网格如果能够被正确地使用，将会是一个非常有用的工具。当网络中具有足够用户时，我们可以获得 100% 的实时性以及最少的数据丢失。如果一个节点断开连接，你可以简单地与网络中的另一个节点进行连接并请求同样的数据。但是，这种网格的能力与每个节点传输数据的相似程度以及网络中用户的数量息息相关。解决这种问题的最佳方法是建立一个可以对连接进行排序并且指派节点的服务器，并且交替优化排序算法和连接个数直到整个网络对每一个用户都达到最佳状态。

## 网格网络的缺陷

虽然网格网络是一个实现多用户通信的简单路径，但是缺点仍然存在。在实际情况中，网格网络是一种直接的、易懂的技术。在一个产品级别的环境下，实现这种技术是非常简单的，因为绝大部分的逻辑可以在客户端进行实现。没有必要去添加大容量的服务器，而是提供决定哪个节点连接哪个节点的服务器。

网格网络模型第一个产生影响的缺点是如何应对当一个用户掉出通话的情况。如果你



由 扫描全能王 扫描创建

与很多的人在同一个通话中，你就会知道保持所有人的连接有多难。如果一个人主持通话然后掉出通话，其他的用户需要相当长一段时间来进行恢复。在星形网格结构中，如果主节点掉出通话，情况将变得非常糟糕。我们不但要及时选出一个新的主节点，还要让其他的用户展开新的连接。

另一主要的缺点是对多种视频流码率和环境进行支持。想象一种情况，当每一个在通话中的人都在非常不同的网络的客户端环境中。包括桌上电脑、移动电话、平板以及在咖啡店中使用同一个 Wi-Fi 网络的人数。这使得一个网格网络要平等地满足所有客户端并且给每一个客户端都提供良好的体验。最后总会有某些用户遭殃。

网格网络的本质问题是在开发的简易程度和用户体验之间进行权衡。这种权衡的一个典型的例子已经发生在视频游戏领域中很多年了。刚开始的时候，很多游戏基于连接所有用户的目的选用了星形网格网络。一个用户可以创建一个游戏然后其他用户便可加入，本质上就是我们曾经提到过的那种模式。随着时间的推移，用户游戏变得越来越流行，用户体验中的瑕疵变得越来越明显。当一个主节点掉出通话，整个游戏都必须转移给另一个用户，造成延迟。不仅仅是这个问题，主节点用户经常作弊并且修改游戏数据，导致主节点用户更倾向于作弊。

今天，许多游戏都转型成服务器-客户端的网络类型。以能够同时处理几百个用户的高性能服务器及硬件作为代价，这种转变会变得非常诱人。一台服务器基本可以确定成为星形网络的主节点，确保游戏永不断线。另外，这些服务器由开发方拥有，这样可以大大降低用户作弊以及没有经过开发方的同意篡改游戏数据。类似这种理论已经出现在通信的世界中，被叫作 MCU。

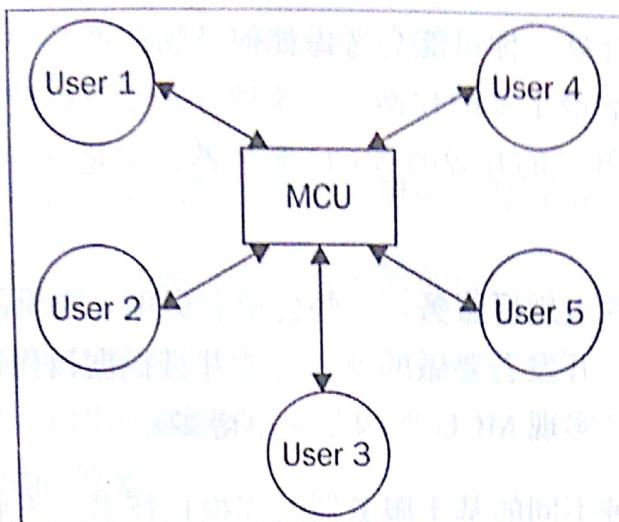
## 更多用户的视频会议

很多支持大规模通信的公司都进行了到服务器-客户端的转变，以此来连接大量的用户。虽然已经存在大量不同的解决方案，但是它们主要都是基于将服务器而不是客户端作为网络节点的假设来建立的。这些 MCU 以提升成本作为代价，给予网络更高的稳定性、效率以及用户体验。

使用 MCU 的网络类型有着和星形网格网络相似的工作方式：将服务器作为主节点而不是任何一个单一的用户。这种方式允许从一个具有很高的稳定性和处理能力的中心位置控制通话。同时它还允许应用的开发者们提升网络带宽，满足用户的基本需求：



由 扫描全能王 扫描创建



这同时也给了网络对用户如何连接和交互的控制能力。这对用户体验是一种细粒度的控制。下面详细阐述了一个典型程度 MCU 连接是怎么工作的：

- 所有的用户都成功地连接到应用开发者搭建的 MCU 服务器。
- 在通话中的所有用户都将他们的视频和音频流发送至服务器。
- MCU 将对每一个输入的流进行单独的解码和处理。
- MCU 可以选择将所有独立的视频流重新编码成一个单独的流以节约带宽。
- MCU 将含有其他用户视频数据的流回传给用户。

你可能注意到这里主要的不同是将其他的视频流打包并且只提供一个流回传给每一个用户。这是一个非常强大的特性，因为可以在很大程度上减少带宽以支持大量的用户进行通话。

与必须要相互传输多个流不同，这种方式只需要带宽支持一个像服务器传送的流和一个回传流。这种方式在极大程度上提升了用户体验，并且允许你的网络规模远远大于传统的网格网络。

当你对网络流有了完全的控制之后，我们可以将这个概念推广得更远。如果你有类似于 Google Hangouts 一类的东西，你可能会发现有这样一个流。当这个流展示的时候，其他的流会以小方块的形式显示在屏幕的下方。这样的方式允许网络基于单个用户来切换速度。这样可以将最高的带宽给每台机器上活跃的用户。这只是一个可以在使用类似于 MCU 的服务器时被用来进行增强的策略。



对于这里提到的所有好处，你可能会考虑如何开始在产品中实现一个 MCU 的架构。这是使用这种技术时面临的最主要的权衡——这种方式很难构建并且相当昂贵。当前网络上有一些致力于 WebRTC 开发的开源的 MCU 服务器，但是安装和编译这些服务器仅仅针对高级用户。

另一种则是选择已有的大规模服务。一些公司会提供大规模的 WebRTC 应用当作一个提供给开发者的高级服务。开发者要做的就是付款并且根据教程在他们的服务器上配置应用。这在某种程度上比自己实现 MCU 架构要简单得多。

强烈建议你多了解多种不同的基于服务器的规模化技术。工业界是一个包含不同技术的广阔领域，包括开源的（open source）和非开源的（closed source）。所以，这里有很多东西需要筛选。你需要准备好花很多时间和金钱在这条路上。在另一方面，如果你的应用需要你提供一个最佳的用户体验，使用基于服务器的架构将满足你的要求。

## 视频会议的未来

这仅仅是大规模 WebRTC 应用的冰山一角。随着通信技术的全球化，任何平台都不会很快过时。这是一个将会被大集团持续投资的产业。这意味着这里介绍的技术将会变得更加强大及可用。

需要记住的是，本章介绍的所有技术都不是 WebRTC 带来的新元素。所有提到的概念都是在 WebRTC 面世之前存在多年的技术。这意味着你有一个很大的宝藏，包含大量已经被使用和测试过的技术可以被转化并且使用在 WebRTC 的应用中。

站在大量已经存在的技术的最高处，多用户通信领域有很多条路可以去尝试进行提升以及创新。整个 web 的框架都可以由你支配，有很多新的方式可以将这种技术和其他已有技术进行结合，提供更好的用户体验。

## 自测题

Q1. 你可以用来保护你的信令服务器的技术包括：

1. https
2. WSS



由 扫描全能王 扫描创建

3. OAuth
4. 以上所有

Q2. 支持移动设备的最大问题在于设备的分辨率。对或错？

Q3. 以下哪个是构建一个网格网络需要考虑的重点：

1. 用户计算机的操作系统
2. 每一个用户所使用的浏览器
3. 用户网络可用的带宽
4. 用户用过哪个 OAuth 方式进行登录

Q4. 一个星形的网络是一个所有用户都连接到一个主节点的网络，主节点给所有用户提供服务。对或错？

Q5. 网格网络可以提供最佳的用户体验，并且对于开发者来说开发容易且成本低。对或错？

Q6. 基于 MCU 的网络与以下那个网格网络最相似：

1. 洪流网络
2. 星形网络网格
3. 完全网络网格
4. 部分网格网络

## 小结

到目前为止，你的脑子将会充斥着 WebRTC 能够做到的令人惊讶的事情。我们介绍了大量可以用于提升你应用的可用性和效率的技术。每一项都有大量的信息值得你去查找更多的资料并且自己发现。如果你准备继续 WebRTC 学习，在空闲的时间中花时间对每一项进行研究将会是一个不错的选择。

本章所介绍的每一项都是针对学习如何发布一个大规模的 WebRTC 应用的。当学习如何使用 WebRTC 时，开发者们可能不会想到从两个用户扩展到一百个用户的时候会发生什么。如果你没有准备好当你的应用加入更多用户时会发生什么，你将会变得越来越痛苦。



由 扫描全能王 扫描创建

学习的最佳方式是尝试使用该技术进行实验并且选择你需求中最紧迫的东西。

在上面的内容中构建的例子是进入 WebRTC 世界非常好的切入点。这里展示的所有东西都可以进行重构和改进，从而变得比上述章节中介绍的更加强大。这便是使用 web 技术，更精细一些，WebRTC 的精美的地方。使用 web 标准打开了通往不单是集成更多的库和框架，而且是第三方应用的大门。WebRTC 不但包含了在通信中连接用户，更包含了通过提供更多的跨平台特性来连接应用。这是一个值得你在接下来几年中持续关注的技术，你可以看到人们对 web 开发观念的变化。



由 扫描全能王 扫描创建

## 附录 自测题答案

这个附录包含了每章最后的自测题的答案。现在让我们看看每个问题所对应的答案。

### 第1章 开启WebRTC之旅

Q1	True
Q2	3
Q3	False
Q4	4

### 第2章 获取User Media

Q1	False
Q2	3
Q3	True
Q4	1
Q5	True



由 扫描全能王 扫描创建

## 第 3 章 创建简单的 WebRTC 应用

Q1	True
Q2	False
Q3	3
Q4	True
Q5	4

## 第 4 章 创建信令服务器

Q1	True
Q2	2
Q3	4
Q4	True

## 第 5 章 把客户端连接到一起

Q1	False
Q2	4
Q3	False
Q4	3

## 第 6 章 使用 WebRTC 发送数据

Q1	False
Q2	1
Q3	True
Q4	False
Q5	4



由 扫描全能王 扫描创建

## 第7章 文件共享

Q1	True
Q2	4
Q3	False
Q4	2
Q5	True

## 第8章 高安全性与大规模优化

Q1	4
Q2	False
Q3	3
Q4	True
Q5	True
Q6	2



由 扫描全能王 扫描创建

本书从教你如何在网页中使用Media Capture和Streams API来展现音频和视频流开始。

你将创建一个可用的WebRTC应用，用来进行语音和视频通话。本书还会带给你一些深入的知识，关于信令和使用Node.js来创建一个信令服务器。当介绍到RTCDataChannel对象时，你将学习到它和WebRTC的关系，以及如何在你的应用中添加基于文本的聊天室。另外，通过一些不同的技术，你还将进一步学习到如何让应用支持多用户，以及如何提升应用的性能和安全性。本书还涉及一些使用全网状网络、局部网状网络及多点控制单元的理论知识。本书的最后，你将会对实时通信、WebRTC协议及它的API有一个全方位的了解。

### 本书写给谁

如果你是一个网站开发人员，同时希望创建一个好用的WebRTC应用给你的用户，那么这本书非常适合。即使你已经从事网络开发很多年，本书也可以让你对WebRTC API有一个完整的了解。本书假定你之前有使用HTML5和JavaScript这些技术进行网站开发的经验。

### 你可以从这本书中学到什么

- 了解创建WebRTC的底层平台
- 利用网络摄像头和麦克风创建应用程序
- 从零开始创建信令服务器
- 使应用程序可以进行多用户通信
- 使用WebRTC点对点连接来共享数据和文件
- 在WebRTC应用运行时进行一些安全的最佳实践
- 针对多用户场景进一步了解多点网状网络
- 在网络、信号、安全以及数据传输的理论基础上学习WebRTC应用的最佳实践
- 使用全网状网络、局部网状网络、多点控制单元来完成你的应用

上架建议：移动终端>程序设计

ISBN 978-7-121-28817-3



由 扫描全能王 扫描创建