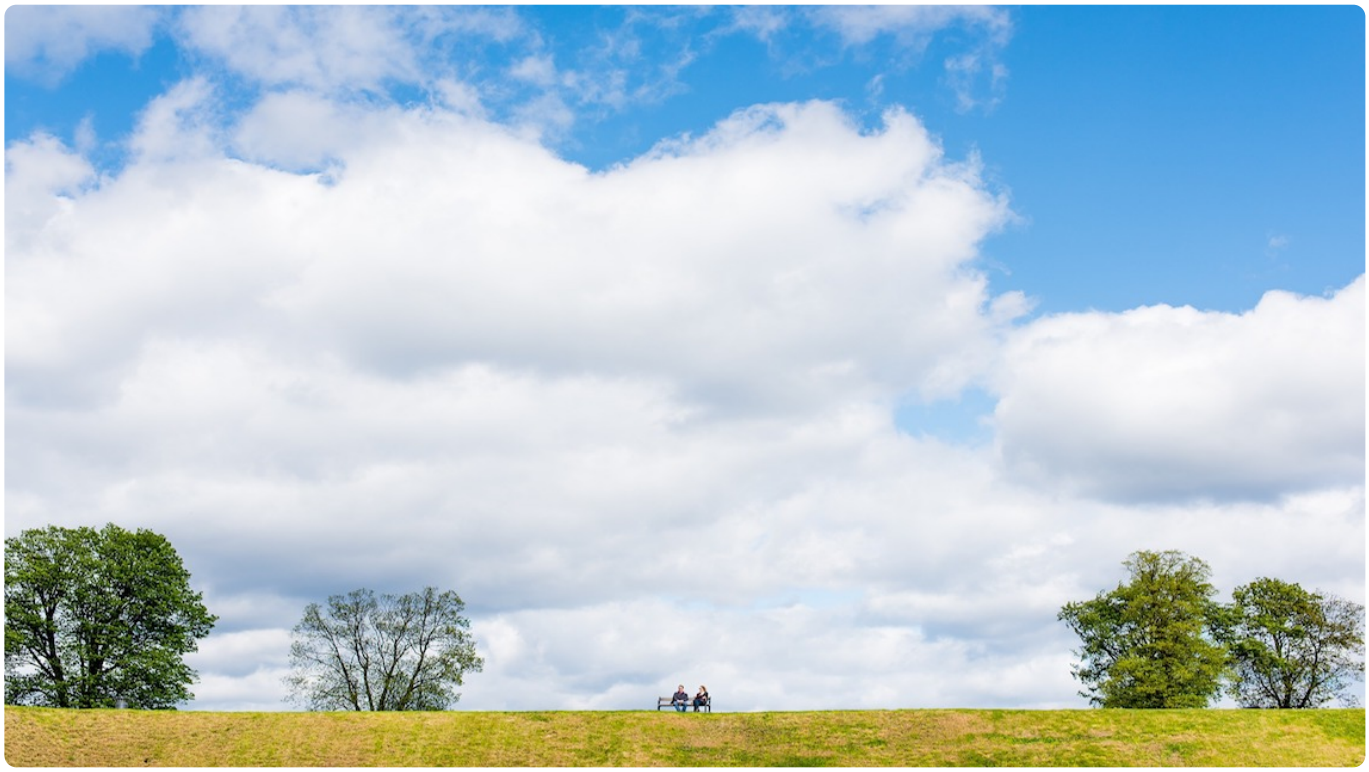
20 | 原来WebRTC还可以实时传输文件?

2019-08-29 李超



讲述：李超
时长 12:23 大小 14.18M

在上一篇文章中我向你介绍了在 WebRTC 中如何传输非音视频数据，并通过实现一个 1 对 1 实时聊天的例子向你展示了如何使用 RTCDataChannel 对象进行文本数据的传输。

其实利用 WebRTC 的 RTCDataChannel 对象，不光可以实现 1 对 1 的实时聊天，你还可以利用它进行**实时的文件传输**。

实时文件传输是一个非常有用的工具，尤其是通过浏览器进行实时文件传输就更加有价值，因为**它不会受到操作系统或开发语言的影响**，所以你可以在任何不同的操作系统上进行文件的传输，非常方便。

举个例子，由于工作的需要，你可能经常需要在不同的操作系统间切来切去（一会儿在 Windows 系统上开发，一会儿在 Linux 系统上开发，一会儿又在 Mac 系统上开发），当

你想将 Windows 上的文件传到 Linux 系统上时就特别不方便，但如果可以通过浏览器传输文件的话，那将会大大提高你的工作效率。

基本原理

在 WebRTC 中，**实时文件的传输与实时文本消息传输的基本原理是一样的，都是使用 RTCDataChannel 对象进行传输。**但它们之间还是有一些差别的，一方面是**传输数据的类型**不一样，另一方面是**数据的大小**不一样。下面我们就从这两方面来具体讨论一下。

在上一篇文章中，我曾向你介绍过 RTCDataChannel 对象支持很多数据类型，包括字符串、Blob、ArrayBuffer 以及 ArrayBufferView，由于文本聊天发送的数据都比较小，而且都是普通文本，所以在实时文本聊天中只需要使用**字符串**类型的数据就可以了。但文件的传输就不一样了，它一般都是以二进制的方式传输数据的，所以更多的是使用 **Blob 和 ArrayBuffer** 类型。

相对于文本消息来说，文件的大小要大得多，因此在传输文件过程中失败的概率也会大很多。虽然 WebRTC 为了达到传输的可靠性，对非音视频数据可以选择将数据通道配置成可靠的、有序的传输，但在真实的场景中还是有可能因为各种原因导致文件传输过程中半途被中止，比如说，在文件传输的过程中网线掉了或者电脑没电了等情况发生时。

为了解决这个问题，一般情况下我们都会采用**断点续传**的方式进行文件的传输。那么如何进行断点续传呢？这就要用到信令服务器了。比较简单的方式是将一个大文件划分成固定大小的“块”，然后按“块”进行传输，并且每传输完一“块”数据，接收端就发一个通知消息告知发送端该块数据它已经接收到了。发送端收到该消息后，就会更新被传输文件的描述信息，这样一旦文件传输过程中发生中断，**下次启动时还可以从上次发生断点的地方继续传输。**

文件传输的具体实现

了解完实时文件传输的基本原理后，接下来我们就来看一下它的具体实现吧！

利用 WebRTC 的 RTCDataChannel 对象进行实时文件的传输与实时文本消息的传输在代码逻辑上是很相似的。接下来，我们就按照其实施步骤（创建 RTCDataChannel 对象、接收数据、文件读取与发送、传递文件信息）做下详细讲解。

1. RTCDataChannel 对象的创建

RTCDataChannel 对象的创建与在实时文本聊天中 RTCDataChannel 对象的创建基本是一致的，具体示例代码如下：

复制代码

```
...
// 创建 RTCDataChannel 对象的选项
var options = {
  ordered: true,
  maxRetransmits : 30
};

// 创建 RTCPeerConnection 对象
var pc = new RTCPeerConnection();

// 创建 RTCDataChannel 对象
var dc = pc.createDataChannel("dc", options);

...
```


通过对比，你可以看到它们之间的不同是：在实时文件传输中创建 RTCDataChannel 对象带了 options 参数，而实时文本聊天中并没有带该参数。

在这个示例中之所以要带 options 参数，是因为**在端与端之间传输文件时，必须要保证文件内容的有序和完整**，所以在创建 RTCDataChannel 对象时，你需要给它设置一些参数，即需要设置 **ordered** 和 **maxRetransmits** 选项。当 ordered 设置为真后，就可以保证数据的有序到达；而 maxRetransmits 设置为 30，则保证在有丢包的情况下可以对丢包进行重传，并且最多尝试重传 30 次。

通过实践证明，如果你在创建 RTCDataChannel 对象时不设置这两个参数的话，那么在传输大文件（如 800M 大小的文件）时就很容易失败。而设置了这两个参数后，传输大文件时基本就没再失败过了，由此可见这两个参数的重要性了。

2. 通过 RTCDataChannel 对象接收数据

创建好 RTCDataChannel 对象后，你仍然要实现 RTCDataChannel 对象的 4 个重要事件（打开、关闭、接收到消息以及出错时接收到事件）的回调函数，代码如下：

复制代码

```
dc.onerror = (error)=> {
```

```
  ...
};

dc.onopen = ()=> {
  ...
};

dc.onclose = () => {
  ...
};

dc.onmessage = (event)=>{
  ...
}
...
```

这四个事件的作用如下：

onerror，是指当发生连接失败时的处理逻辑；

onopen，是指当 datachannel 打开时的处理逻辑；

onclose，是指当 datachannel 关闭时的处理逻辑；

onmessage，是指当收到消息时的处理逻辑。

其中最重要的是 **onmessage 事件**，当有数据到达时就会触发该事件。那接下来，我们就看一下到底该如何实现这个事件处理函数，具体代码如下：

复制代码

```
...
var receiveBuffer = []; // 存放数据的数组
var receiveSize = 0; // 数据大小
...
onmessage = (event) => {

  // 每次事件被触发时，说明有数据来了，将收到的数据放到数组中
  receiveBuffer.push(event.data);
  // 更新已经收到的数据的长度
  receivedSize += event.data.byteLength;

  // 如果接收到的字节数与文件大小相同，则创建文件
  if (receivedSize === fileSize) { //fileSize 是通过信令传过来的
    // 创建文件
    var received = new Blob(receiveBuffer, {type: 'application/octet-stream'});
```

```
    // 将 buffer 和 size 清空，为下一次传文件做准备
    receiveBuffer = [];
    receiveSize = 0;

    // 生成下载地址
    downloadAnchor.href = URL.createObjectURL(received);
    downloadAnchor.download = fileName;
    downloadAnchor.textContent =
      `Click to download '${fileName}' (${fileSize} bytes)`;
    downloadAnchor.style.display = 'block';
  }
}
```

上面这段代码的逻辑还是非常简单的，每当该函数被调用时，说明被传输文件的一部分数据已经到达了。这时你只需要简单地将收到的这块数据 push 到 receiveBuffer 数组中即可。

当文件的所有数据都收到后，即`receivedSize === fileSize`条件成立时，你就可以以 receiveBuffer[] 数组为参数创建一个 Blob 对象了。紧接着，再给这个 Blob 对象创建一个下载地址，这样接收端的用户就可以通过该地址获取到文件了。

3. 文件的读取与发送

前面讲完了文件的接收，现在我们再来看一下文件的读取与发送。实际上这块逻辑也非常简单，代码如下：

复制代码

```
...
function sendData(){

  var offset = 0; // 偏移量
  var chunkSize = 16384; // 每次传输的块大小
  var file = fileInput.files[0]; // 要传输的文件，它是通过 HTML 中的 file 获取的
  ...

  // 创建 fileReader 来读取文件
  fileReader = new FileReader();
  ...
  fileReader.onload = e => { // 当数据被加载时触发该事件
    ...
    dc.send(e.target.result); // 发送数据
    offset += e.target.result.byteLength; // 更改已读数据的偏移量
    ...
    if (offset < file.size) { // 如果文件没有被读完
```

```
      readSlice(offset); // 读取数据
    }
  }

  var readSlice = o => {
    const slice = file.slice(offset, o + chunkSize); // 计算数据位置
    fileReader.readAsArrayBuffer(slice); // 读取 16K 数据
  };

  readSlice(0); // 开始读取数据

}
...
```

在这段示例代码中，数据的读取是通过 sendData 函数实现的。在该函数中，使用 FileReader 对象每次从文件中读取 16K 的数据，然后再调用 RTCDataChannel 对象的 send 方法将其发送出去。

这段代码中有**两个关键点**：一是 sendData 整个函数的执行是 readSlice(0) 开始的；二是 FileReader 对象的 onload 事件是在有数据被读入到 FileReader 的缓冲区之后才触发的。掌握了这两个关键点，你就非常容易理解 sendData 函数的逻辑了。

那该怎么理解这两个关键点呢？实际上， sendData 函数在创建了 FileReader 对象之后，下面的代码都不会执行，直到调用 readSlice(0) 才开始从文件中读取数据；当数据被读到 FileReader 对象的缓冲区之后，就会触发 onload 事件，从而开始执行 onload 事件的回调函数。而在这个回调函数中是一个循环，不断地从文件中读取数据、发送数据，直到读到文件结束为止。以上就是 sendData 函数的逻辑。

4. 通过信令传递文件的基本信息

上面我已经将 RTCDataChannel 对象的创建、数据发送与接收的方法以及 JavaScript 对文件进行读取的操作向你做了详细的介绍。但还有一块儿重要的知识需要向你讲解，那就是：接收端是如何知道发送端所要传输的文件大小、类型以及文件名的呢？

其实这个问题也不麻烦，解决的办法就是在传输文件之前，发送端先通过信令服务器将要传输文件的基本信息发送给接收端。

那我们就来看一下发送端是如何处理的，具体代码如下：

> 需要说明的是，在本例中 JavaScript 脚本是通过 socket.io 库与信令服务器通信的。

复制代码

```
...
// 获取文件相关的信息
fileName = file.name;
fileSize = file.size;
fileType = file.type;
lastModifyTime = file.lastModified;

// 向信令服务器发送消息
sendMessage(roomid,
  {
    // 将文件信息以 JSON 格式发磅
    type: 'fileinfo',
    name: file.name,
    size: file.size,
    filetype: file.type,
    lastmodify: file.lastModified
  }
);
```

在本段代码中，发送端首先获得被传输文件的基本信息，如文件名、文件类型、文件大小等，然后再通过 socket.io 以 JSON 的格式将这些信息发给信令服务器。

信令服务器收到该消息后不做其他处理，直接转发到接收端。下面是接收端收到消息后的处理逻辑，代码如下：

复制代码

```
...
socket.on('message', (roomid, data) => {
  ...
  // 如果是 fileinfo 类型的消息
  if(data.hasOwnProperty('type') && data.type === 'fileinfo'){
    // 读出文件的基本信息
    fileName = data.name;
    fileType = data.filetype;
    fileSize = data.size;
    lastModifyTime = data.lastModify;
    ...
  }
  ...
```

```
14 });
15 ...
```

在接收端的 socket.io 一直在侦听 message 消息，当收到 message 消息且类型（type）为 fileinfo 时，说明对方已经将要传输文件的基本信息发送过来了。

接收到文件的基本信息后，接收端的处理逻辑也非常简单，只需要将传过来的基本信息保存起来就好了，等整个文件传输完成后，再根据这些信息生成对应的文件，这样接收端就可以将传输的文件拿到手了。

小结

本文我首先向你介绍了使用 RTCDataChannel 对象进行文件传输的基本工作原理，实际上它与实时文本消息的用法基本相同，只不过由于文件传输对数据的有序性和完整性有特别的要求，所以你在创建 RTCDataChannel 对象时一定要将其 options 选项中的 ordered 和 maxRetransmits 设置好。

之后，我又向你详细讲解了如何实现一个端到端文件的传输，其整个实现主要包括四个步骤，即 RTCDataChannel 对象的创建、数据的发送、数据的接收以及文件基本信息的传输。

在实际的应用中，由于某些特殊原因，文件在传输过程中还是会被中断掉，如网络连接断了、传输被人为暂停了等等。对于这类情况，你可以使用断点续传的方法来恢复被中断的文件传输。这个并不麻烦，如果你感兴趣，可以自行学习和研究下。

思考时间

在上面描述中我们有提到，当文件在传输过程中被中断后，可以通过断点续传的方式恢复被中断的文件传输，这种情况一般是在浏览器页面没有被关闭的情况下才可以做到。但如果页面被关闭了，或者说浏览器被关了，是否还有办法恢复被中断的传输？

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

上一篇　19 | WebRTC能不能进行文本聊天呢?

下一篇　21 | 如何保证数据传输的安全（上）？

精选留言 (2)

写留言

xw616525957
2019-08-29

页面被关掉的情况可以考虑用浏览器做个缓存，比如indexedDB.将发送端和接收端的状态和file数据都保存下来，下次进room的时候再恢复上传。

这里有个问题，在一对多的情况下，发送端是不是要维护每个接收端的收发状态。这种情况是不是通过server中转一层比较好，这样可以利用cdn来加速接收端的下载速度。

展开

作者回复: 如果是一对多，最好是使用服务器中转的方式

1

Jason
2019-08-29

有办法恢复被中断的传输：将传输的连接、状态信息保存下来，浏览器再次启动时加载这些信息。

作者回复: 有没有想过是否还可以通过 Blob 进行断点续传呢?