

16 | WebRTC中的数据统计原来这么强大（下）

2019-08-20 李超

从0打造音视频直播系统

[进入课程 >](#)



讲述：李超

时长 12:40 大小 11.60M



在[上一篇文章](#)中我向你介绍了 WebRTC 可以获得哪些统计信息，以及如何使用 `RTCPeerConnection` 对象的 `getStats` 方法获取想要的统计信息。

那本文我们在[上一篇文章](#)的基础之上，继续对 WebRTC 中的统计信息做进一步的讨论，了解它更为详细的内容。

再论 `getStats`

现在你已经非常清楚，通过 `RTCPeerConnection` 对象的 `getStats` 方法可以很轻松地获取到各种统计信息，比如发了多少包、收了多少包、丢了多少包，等等。但实际上对于收发包这块儿的统计还可以从其他方法获取到，即通过 `RTCRtpSender` 的 `getStats` 方法和 `RTCRtpReceiver` 的 `getStats` 方法也能获取收发包的统计信息。

也就是说，除了 `RTCPeerConnection` 对象有 `getStats` 方法外，`RTCRtpSender` 和 `RTCRtpReceiver` 对象也有 `getStats` 方法，只不过它们只能获取到与传输相关的统计信息，而 `RTCPeerConnection` 还可以获取到其他更多的统计信息。

下面我们就来看一下它们三者之间的区别：


`RTCPeerConnection` 对象的 `getStats` 方法获取的是**所有的统计信息**，除了收发包的统计信息外，还有候选者、证书、编解码器等其他类型的统计信息。

`RTCRtpSender` 对象的 `getStats` 方法只统计**与发送相关**的统计信息。

`RTCRtpReceiver` 对象的 `getStats` 方法则只统计**与接收相关**的统计信息。

通过上面的描述，我想你已经非常清楚 `RTCPeerConnection` 中的 `getStats` 方法是获取到所有的统计信息，而 `RTCRtpSender` 和 `RTCRtpReceiver` 对象中的 `getStats` 方法则分别统计的是发包、收包的统计信息。所以 `RTCPeerConnection` 对象中的统计信息与 `RTCRtpSender` 和 `RTCRtpReceiver` 对象中的统计信息是**整体与局部**的关系。

下面咱们通过一段示例代码来详细看看它们之间的不同：

 复制代码

```
1 ...
2 var pc = new RTCPeerConnection(null);
3 ...
4
5 pc.getStats()
6   .then( reports => { // 得到相关的报告
7     reports.forEach( report => { // 遍历每个报告
8       console.log(report);
9     });
10  }).catch( err=>{
11    console.error(err);
12  });
13
14 // 从 PC 上获得 sender 对象
15 var sender = pc.getSenders()[0];
16
17 ...
18
19 // 调用 sender 的 getStats 方法
20 sender.getStats()
21   .then(reports => { // 得到相关的报告
22     reports.forEach(report =>{ // 遍历每个报告
23       if(report.type === 'outbound-rtp'){ // 如果是 rtp 输出流
```

```

24         ....
25     }
26 }
27 );
28 ...

```

在上面的代码中生成了两段统计信息，一段是通过 `RTCPeerConnection` 对象的 `getStats` 方法获取到的，其结果如下：

```

{id: "RTCCertificate_86:10:AE:6C:AE:68:D4:B1:97:28:8D:BA..2:4A:3D:0E:FC:9E:83:F8:47:7B:9C:EB:CE:02:A9:2E:1E", timestamp: 1565719486676.13, type: "certificate", fingerprint: "86:10:AE:6C:AE:68:D4:B1:97:28:8D:BA:48:93:70:62:4A:3D:0E:FC:9E:83:F8:47:7B:9C:EB:CE:02:A9:2E:1E", fingerprintAlgorithm: "sha-256", ...}
main bw.js:433
{id: "RTCCertificate_8B:4F:0D:E2:79:DE:3C:D7:7A:78:42:CE..3:B9:0B:F5:E5:88:A8:00:CE:69:A9:93:C4:12:40:16:0E", timestamp: 1565719486676.13, type: "certificate", fingerprint: "8B:4F:0D:E2:79:DE:3C:D7:7A:78:42:CE:D2:D3:A1:C3:B9:0B:F5:E5:88:A8:00:CE:69:A9:93:C4:12:40:16:0E", fingerprintAlgorithm: "sha-256", ...}
main bw.js:433
▶ {id: "RTCCodec_0_Inbound_100", timestamp: 1565719486676.13, type: "codec", payloadType: 100, mimeType: "video/VP9", ...}
main bw.js:433
▶ {id: "RTCCodec_0_Inbound_101", timestamp: 1565719486676.13, type: "codec", payloadType: 101, mimeType: "video/rtx", ...}
main bw.js:433
▶ {id: "RTCCodec_0_Inbound_102", timestamp: 1565719486676.13, type: "codec", payloadType: 102, mimeType: "video/H264", ...}
main bw.js:433
▶ {id: "RTCCodec_0_Inbound_107", timestamp: 1565719486676.13, type: "codec", payloadType: 107, mimeType: "video/rtx", ...}
main bw.js:433
▶ {id: "RTCCodec_0_Inbound_108", timestamp: 1565719486676.13, type: "codec", payloadType: 108, mimeType: "video/H264", ...}
main bw.js:433
▶ {id: "RTCCodec_0_Inbound_109", timestamp: 1565719486676.13, type: "codec", payloadType: 109, mimeType: "video/rtx", ...}
main bw.js:433
▶ {id: "RTCCodec_0_Inbound_114", timestamp: 1565719486676.13, type: "codec", payloadType: 114, mimeType: "video/red", ...}
main bw.js:433
▶ {id: "RTCCodec_0_Inbound_115", timestamp: 1565719486676.13, type: "codec", payloadType: 115, mimeType: "video/rtx", ...}
main bw.js:433
▶ {id: "RTCCodec_0_Inbound_116", timestamp: 1565719486676.13, type: "codec", payloadType: 116, mimeType: "video/ulpfec", ...}
main bw.js:433
▶ {id: "RTCCodec_0_Inbound_119", timestamp: 1565719486676.13, type: "codec", payloadType: 119, mimeType: "video/rtx", ...}

```

另一段是通过 `RTCRtpSender` 对象的 `getStats` 方法获取到的，其结果如下：

```

{id: "RTCCertificate_E3:24:28:5A:63:AE:ED:46:5B:EE:58:CB..2:51:2A:1A:55:AC:1A:CB:75:B7:C1:C9:A5:57:E2:1F:B6", timestamp: 1565719635455.195, type: "certificate", fingerprint: "E3:24:28:5A:63:AE:ED:46:5B:EE:58:CB:CF:D5:E7:92:51:2A:1A:55:AC:1A:CB:75:B7:C1:C9:A5:57:E2:1F:B6", fingerprintAlgorithm: "sha-256", ...}
main bw.js:461
{id: "RTCCertificate_E5:CB:5B:E6:E8:FE:3D:61:4F:17:53:47..4:8D:52:37:39:01:BB:68:78:7E:21:12:E8:FE:EA:E0:A2", timestamp: 1565719635455.195, type: "certificate", fingerprint: "E5:CB:5B:E6:E8:FE:3D:61:4F:17:53:47:5B:07:5D:74:8D:52:37:39:01:BB:68:78:7E:21:12:E8:FE:EA:E0:A2", fingerprintAlgorithm: "sha-256", ...}
main bw.js:461
▶ {id: "RTCCodec_0_Outbound_96", timestamp: 1565719635455.195, type: "codec", payloadType: 96, mimeType: "video/VP8", ...}
main bw.js:461
▶ {id: "RTCIceCandidatePair_ENOFqJLg_qVMQcJ4R", timestamp: 1565719635455.195, type: "candidate-pair", transportId: "RTCTransport_0_1", localCandidateId: "RTCIceCandidate_ENOFqJLg", ...}
main bw.js:461
▶ {id: "RTCIceCandidate_ENOFqJLg", timestamp: 1565719635455.195, type: "local-candidate", transportId: "RTCTransport_0_1", isRemote: false, ...}
main bw.js:461
▶ {id: "RTCIceCandidate_qVMQcJ4R", timestamp: 1565719635455.195, type: "remote-candidate", transportId: "RTCTransport_0_1", isRemote: true, ...}
main bw.js:461
▶ {id: "RTCMediaStreamTrack_sender_26", timestamp: 1565719635455.195, type: "track", trackIdentifier: "df52e244-e19f-4292-8989-5dba2c62dd52", mediaSourceId: "RTCVideoSource_26", ...}
main bw.js:461
▶ {id: "RTCOutboundRTPVideoStream_973203781", timestamp: 1565719635455.195, type: "outbound-rtp", ssrc: 973203781, isRemote: false, ...}
main bw.js:461
▶ {id: "RTCTransport_0_1", timestamp: 1565719635455.195, type: "transport", bytesSent: 17239, bytesReceived: 16727, ...}
main bw.js:461
▶ {id: "RTCVideoSource_26", timestamp: 1565719635455.195, type: "media-source", trackIdentifier: "df52e244-e19f-4292-8989-5dba2c62dd52", kind: "video", ...}


```

通过对上面两幅图的对比你可以发现，RTCPeerConnection 对象的 getStats 方法获取到的统计信息明显要比 RTCRtpSender 对象的 getStats 方法获取到的信息多得多。这也证明了我们上面的结论，即 RTCPeerConnection 对象的 getStats 方法获取到的信息与 RTCRtpSender 对象的 getStats 方法获取的信息之间是**整体与局部**的关系。

RTCStatsReport

我们通过 getStats API 可以获取到 WebRTC 各个层面的统计信息，它的返回值的类型是 RTCStatsReport。

RTCStatsReport 的结构如下：

 复制代码

```
1 interface RTCStatsReport {  
2     readonly maplike<DOMString, object>;  
3 };
```

即 RTCStatsReport 中有一个 Map，Map 中的 key 是一个字符串，object 是 RTCStats 的继承类。

RTCStats 作为基类，它包括以下三个字段。

id：对象的唯一标识，是一个字符串。

timestamp：时间戳，用来标识该条 Report 是什么时间产生的。

type：类型，是 RTCStatsType 类型，它是各种类型 Report 的基类。

而继承自 RTCStats 的子类就特别多了，下面我挑选其中的一些子类向你做下介绍。

第一种，编解码器相关的统计信息，即 RTCCodecStats。其类型定义如下：

 复制代码

```
1 dictionary RTCCodecStats : RTCStats {  
2     unsigned long payloadType; // 数据负载类型  
3     RTCCodecType codecType;    // 编解码类型  
4     DOMString    transportId;  // 传输 ID
```




```

5         DOMString      mimeType;
6         unsigned long clockRate;    // 采样时钟频率
7         unsigned long channels;     // 声道数，主要用于音频
8         DOMString      sdpFmtpLine;
9         DOMString      implementation;
10    };

```

通过 `RTCCodecStats` 类型的统计信息，你就可以知道现在直播过程中都支持哪些类型的编解码器，如 AAC、OPUS、H264、VP8/VP9 等等。

第二种，输入 RTP 流相关的统计信息，即 `RTCI inboundRtpStreamStats`。其类型定义如下：

 复制代码

```

1 dictionary RTCInboundRtpStreamStats : RTCReceivedRtpStreamStats {
2     ...
3     unsigned long      frameWidth;    // 帧宽度
4     unsigned long      frameHeight;   // 帧高度
5     double              framesPerSecond; // 每秒帧数
6     ...
7     unsigned long long  bytesReceived; // 接收到的字节数
8     ....
9     unsigned long      packetsDuplicated; // 重复的包数
10    ...
11    unsigned long      nackCount;      // 丢包数
12    ....
13    double              jitterBufferDelay; // 缓冲区延迟
14    ....
15    unsigned long      framesReceived;  // 接收的帧数
16    unsigned long      framesDropped;   // 丢掉的帧数
17    ...
18 };

```

通过 `RTCI inboundRtpStreamStats` 类型的统计信息，你就可以从中取出接收到字节数、包数、丢包数等信息了。

第三种，输出 RTP 流相关的统计信息，即 `RTCO outboundRtpStreamStats`。其类型定义如下：

```
1 dictionary RTCOutboundRtpStreamStats : RTCSentRtpStreamStats {
2     ...
3     unsigned long long    retransmittedPacketsSent; // 重传包数
4     unsigned long long    retransmittedBytesSent; // 重传字节数
5     double                targetBitrate; // 目标码率
6     ...
7 .
8     unsigned long        frameWidth; // 帧的宽度
9     unsigned long        frameHeight; // 帧的高度
10    double                framesPerSecond; // 每秒帧数
11    unsigned long         framesSent; // 发送的总帧数
12    ...
13    unsigned long         nackCount; // 丢包数
14    ....
15 };
```

通过 `RTCOutboundRtpStreamStats` 类型的统计信息，你就可以从中得到目标码率、每秒发送的帧数、发送的总帧数等内容了。

在 WebRTC 1.0 规范中，一共定义了 17 种 `RTCStats` 类型的子类，这里我们就不一一进行说明了。关于这 17 种子类型，你可以到文末的[参考](#)中去查看。实际上，这个表格在[上一篇文章](#)中我已经向你做过介绍了，这里再重新温习一下。

若你对具体细节很感兴趣的话，可以通过《WebRTC1.0 规范》去查看每个 `RTCStats` 的详细定义，[相关链接在这里](#)。

RTCP 交换统计信息

在[上一篇文章](#)中，我给你留了一道思考题，不知你是否已经找到答案了？实际上在 WebRTC 中，上面介绍的输入 / 输出 RTP 流报告中的统计数据都是通过 RTCP 协议中的 SR、RR 消息计算而来的。

关于 RTCP 以及 RTCP 中的 SR、RR 等相关协议内容记不清的同学可以再重新回顾一下[《06 | WebRTC 中的 RTP 及 RTCP 详解》](#)一文的内容。

在 RTCP 协议中，SR 是发送方发的，记录的是 RTP 流从发送到现在一共发了多少包、发送了多少字节数据，以及丢包率是多少。RR 是接收方发的，记录的是 RTP 流从接收到现在一共收了多少包、多少字节的数据等。

通过 SR、RR 的不断交换，在通讯的双方就很容易计算出每秒钟的传输速率、丢包率等统计信息了。

在使用 RTCP 交换信息时有一个主要原则，就是 RTCP 信息包在整个数据流的传输中占带宽的百分比不应超过 5%。也就是说你的媒体包发送得越多，RTCP 信息包发送得也就越多。你的媒体包发得少，RTCP 包也会相应减少，它们是一个联动关系。

绘制图形

通过 `getStats` 方法我们现在可以获取到各种类型的统计数据了，而且在上面的 **RTCP 交换统计信息**中，我们也知道了 WebRTC 底层是如何获取到传输相关的统计数据的了，那么接下来我们再来看一下如何利用 `RTCStatsReport` 中的信息来绘制出各种分析图形，从而使监控的数据更加直观地展示出来。

在本文的例子中，我们以绘制每秒钟发送的比特率和每秒钟发送的包数为例，向你展示如何将 `RTCStats` 信息转化为图形。

要将 Report 转化为图形大体上分为以下几个步骤：

- 引入第三方库 `graph.js`；

- 启动一个定时器，每秒钟绘制一次图形；

- 在定时器的回调函数中，读取 `RTCStats` 统计信息，转化为可量化参数，并将其传给 `graph.js` 进行绘制。


了解了上面的步骤后，接下来我们就来实操一下吧！

第三方库 `graph.js` 是由 WebRTC 项目组开发的，是专门用于绘制各种图形的，它底层是通过 `Canvas` 来实现的。这个库非常短小，只有 600 多行代码，使用起来也非常方便，在下面的代码中会对它的使用做详细的介绍。

另外，该库的代码链接我已经放到了文章的末尾，供你参考。

1. 引入第三方库

在 JavaScript 中引入第三方库也非常简单，只要使用 `<script>` 就可以将第三方库引入进来了。具体代码如下：

 复制代码

```
1 <html>
2   ...
3   <body>
4     ...
5     <script src="js/client.js"></script>
6
7     // 引入第三方库 graph.js
8     <script src="js/third_party/graph.js"></script>
9     ...
10  </body>
11 </html>
```

2. client.js 代码的实现

client.js 是绘制图形的核心代码，具体代码如下所示：

 复制代码

```
1 ...
2
3 var pc = null;
4
5 // 定义绘制比特率图形相关的变量
6 var bitrateGraph;
7 var bitrateSeries;
8
9 // 定义绘制发送包图形相关的变量
10 var packetGraph;
11 var packetSeries;
12 ...
13
14 pc = new RTCPeerConnection(null);
15
16 ...
17
18 //bitrateSeries 用于绘制点
19 bitrateSeries = new TimelineDataSeries();
20 //bitrateGraph 用于将 bitrateSeries 绘制的点展示出来
21 bitrateGraph = new TimelineGraphView('bitrateGraph', 'bitrateCanvas');
22 bitrateGraph.updateEndDate(); // 绘制时间轴
23
24 // 与上面一样，只是不是用于绘制包相关的图
```



```

77         lastResult.get(report.id).packetsSent);
78         packetGraph.setDataSet([packetSeries]);
79         packetGraph.updateEndDate();
80     }
81 }
82 });
83
84 // 记录上一次的报告
85 lastResult = res;
86
87 });
88
89 }, 1000); // 每秒钟触发一次
90
91 ...

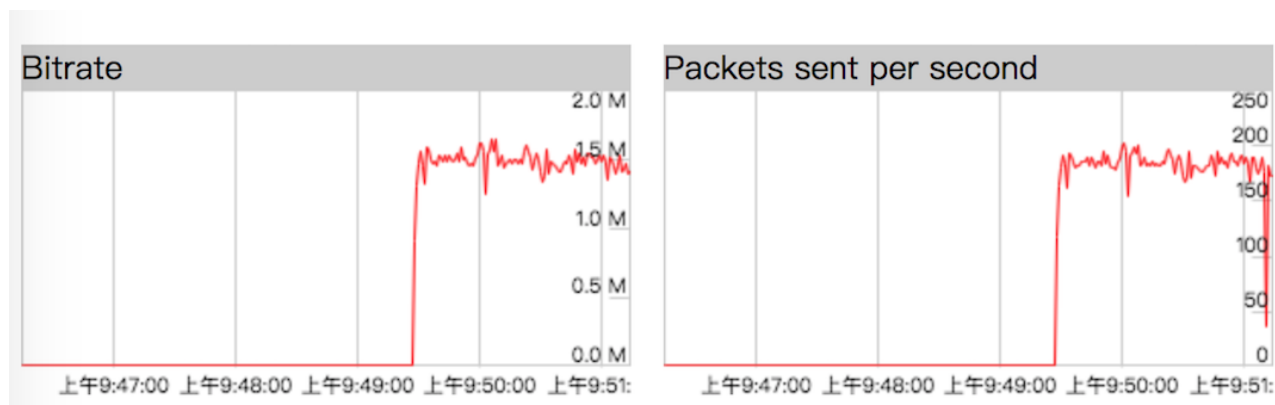
```

在该代码中，最重要的是 32 ~ 89 行的代码，因为这其中实现了一个定时器——每秒钟执行一次。每次定时器被触发时，都会调用 sender 的 getStats 方法获取与传输相关的统计信息。

然后对获取到的 RTCStats 类型做判断，只取 RTCStats 类型为 outbound-rtp 的统计信息。最后将本次统计信息的数据与上一次信息的数据做差值，从而得到它们之间的增量，并将增量绘制出来。

3. 最终的结果

当运行上面的代码时，会绘制出下面的结果，这样看起来就一目了然了。通过这张图你可以看到，当时发送端的码率为 1.5Mbps 的带宽，每秒差不多发送小 200 个数据包。



小结

在本文中，我首先向你介绍了除了可以通过 `RTCPeerConnection` 对象的 `getStats` 方法获取到各种统计信息之外，还可以通过 `RTCRtpSender` 或 `RTCRtpReceiver` 的 `getStats` 方法获得与传输相关的统计信息。WebRTC 对这些统计信息做了非常细致的分类，按类型可细分为 17 种，关于这 17 种类型你可以查看文末[参考](#)中的表格。

在文中我还向你重点介绍了**编解码器、输入 RTP 流以及输出 RTP 流**相关的统计信息。

除此之外，在文中我还向你介绍了**网络传输**相关的统计信息是如何获得的，即通过 RTCP 协议中的 SR 和 RR 消息进行交换而来的。实际上，对于 RTCP 的知识我在前面[《06 | WebRTC 中的 RTP 及 RTCP 详解》](#)一文中已经向你讲解过了，而本文所讲的内容则是 RTCP 协议的具体应用。

最后，我们通过使用第三方库 `graph.js` 与 `getStats` 方法结合，就可以将统计信息以图形的方式绘制出来，使你可以清晰地看出这些报告真正表达的意思。

思考时间

今天你要思考的问题是：当使用 RTCP 交换 SR/RR 信息时，如果 SR/RR 包丢失了，会不会影响数据的准确性呢？为什么呢？

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。

参考

统计信息类型	说明
candidate-pair	候选者对
certificate	证书
codec	当前流正在使用的编解码器
csrc	输入流中共享源的ID
data-channel	数据通道统计信息
inbound-rtp	当前连接RTP输入流的统计信息
local-candidate	ICE 本地候选者
outbound-rtp	当前连接RTP输出流的统计信息
peer-connection	RTCPeerConnection 相关的统计信息
receiver	RTP接收者的统计信息，如果kind是音频，则统计音频的信息；如果是视频，则统计的是视频的信息。
remote-candidate	ICE 远程候选者，它包括网络类型、协议、URL、relay类型等。
remote-inbound-rtp	远端RTP输入流统计信息，这与本地的 outbound rtp对应。
remote-outbound-rtp	远端RTP Sender的统计信息，它与本地的inbound rtp对应。
sender	RTP Sender 的统计信息，它又可以根据 kind 类型做不同的统计。
stream	流相关的统计信息
track	音视频轨相关的统计信息
transport	传输相关的统计信息

[例子代码地址，戳这里](#)

[第三方库地址，戳这里](#)

从 0 打造音视频直播系统

手把手教你打造实时互动音视频直播系统

李超

新东方音视频直播技术专家
前沪江音视频架构师



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | WebRTC中的数据统计原来这么强大（上）

下一篇 17 | 如何使用Canvas绘制统计图表（上）？

精选留言 (3)

写留言



峰

2019-08-22

老师还是昨天问题，在Firefox，IE浏览器上是可以播放的，只是Google Chrome上播放一点就报错，提示视频问题或浏览器某些特征不支持，如果真的是视频问题，这种现象，暂时无法理解！

作者回复: 你用video 标签播的吗？如果是 video标签各浏览器的实现不一样，在chrome下，浏览器对vp8/vp9支持的更好。你可能通过第三方库来播MP4文件



峰

2019-08-21

老师，这个问题你遇到吗，能否帮帮我

Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'F:/mp4/convert/041.mp4': Metadata:
major_brand : isom minor_version : 512 compatible_brands: isomiso2avc1mp41 en
coder : Lavf57.76.100 Duration: 00:00:21.04, start: 0.033008, bitrate: 3656 kb/s Strea
m #0:0(und): Video: h264 (Main) (avc1 / 0x31637661), yuv420p(tv, bt709), 1920x1...

展开 ∨

作者回复: 提示上有这个信息 “Invalid NAL unit 8” 说明你的视频数据有问题



许童童

2019-08-20

思考题：

不会影响准确性，因为每一次传输都是全量的，丢失只会丢失这一次的值，在下一次又会全量带过来。

展开 ∨

作者回复: 赞！

