WebRTC getStats详解 - 从标准、调用到实现

前言

getStats是WebRTC一个非常重要的API，用来向开发者和用户导出WebRTC运行时状态信息，包括网络数据接收和发送状态、P2P客户端媒体数据采集和渲染状态等[1]。这些信息对于监控WebRTC运行状态、排除程序错误等非常重要。

本文首先描述W3C定义的getStats标准，然后展示如何在JS层调用getStats，最后深入分析WebRTC源代码中getStats的实现。全文从标准到实现，全方位透彻展示getStats的细节。

一 getStats标准

getStats的标准由W3C定义，其接口很简单，但是却返回丰富的WebRTC运行时信息。其返回信息的主要内容如下[2]：

1. 发送端采集统计：对应于媒体数据的产生，包括帧率，帧大小，媒体数据源的时钟频率，编解码器名称，等等。
2. 发送端RTP统计：对应于媒体数据的发送，包括发送数据包数，发送字节数，往返时间RTT，等等。
3. 接收端RTP统计：对应于媒体数据的接收，包括接收数据包数，接收字节数，丢弃数据包数，丢失数据包数，网络抖动jitter，等等。
4. 接收端渲染统计：对应于媒体数据的渲染，包括丢弃帧数，丢失帧数，渲染帧数，渲染延迟，等等。

另外还有一些杂项统计，如DataChannel度量，网络接口度量，证书统计等等。在众多信息中，有一些反映WebRTC运行状态的核心度量，包括往返时间RTT，丢包率和接收端延迟等，分别表述如下：

- 往返时间RTT：表示数据在网络上传输所用的时间，一般通过RTCP 的SR/RR数据包中的相关域进行计算。该度量直接反映网络状况的好坏。
- 丢包率影响接收端音视频质量，在严重的情况下可能导致声音跳变或者视频马赛克，从侧面反映网络状况的好坏。
- 音视频数据到达接收端之后，要经历收包、解码、渲染等过程，该过程会带来延迟。接收端延迟是数据从采集到渲染单向延迟的重要组成部分。

通过以上分析可知，getStats的返回信息包含WebRTC数据管线的各个阶段的统计信息，从数据采集、编码到发送，再到数据接收、解码和渲染。这为监控WebRTC应用的运行状态提供第一手数据。

二 使用JS调用getStats

getStats的JS API很简单， W3C规定getStats的JS API函数PTCPeerConnection.getStats需要三个参数：一个可为空的MediaStreamTrack对象，一个调用成功时的回调函数和一个调用失败时的回调函数。

成功回调函数的参数为getStats得到的RTCStatsReport，主要工作就发生在解析RTCStatsReport上，拿到我们感兴趣的参数，进而分析应用的运行状态。

下面我们选取W3C标准上给出的例子作简单讲解[1]。假设当前会话的通话质量很差，我们想知道是不是由于丢包率过大引起的。因此，我们可以通过getStats返回结果的outbound-rtp中的丢包数和收包数计算丢包率，然后进行判断。具体代码实现如下：

```
var baselineReport, currentReport;
var selector = pc.getRemoteStreams()[0].getAudioTracks()[0];
pc.getStats(selector, function (report) {
   baselineReport = report;
}, logError);

setTimeout(function () {
   pc.getStats(selector, function (report) {
      currentReport = report;
      processStats();
   }, logError);
}, aBit);

function processStats() {
   for (var i in currentReport) {
      var now = currentReport[i];
      if (now.type != "outbund-rtp")
         continue;

      base = baselineReport[now.id];
      if (base) {
         remoteNow = currentReport[now.associateStatsId];
         remoteBase = baselineReport[base.associateStatsId];
         var packetsSent = now.packetsSent - base.packetsSent;
         var packetsReceived = remoteNow.packetsReceived –
               remoteBase.packetsReceived;

         // if fractionLost is > 0.3, we have probably found the culprit
         var fractionLost = (packetsSent - packetsReceived) / packetsSent;
      }
   }
}
function logError(error) {
log(error.name + ": " + error.message);
}
```

通过上述例子，我们可以体会到从JS层调用getStats分析应用运行状态的基本流程。值得注意的是，

Chrome和Firefox两款浏览器在调用方面有稍微差别，具体请参考文档[3]。

三 getStats在WebRTC内部的实现

JS层的getStats调用如何传递到到WebRTC内部的实现函数，涉及到浏览器的内部工作原理，具体到Chrome浏览器来讲，是由WebKit，V8，Content，libjingle等模块一起协同工作实现。本节我们不讨论这里面的细节，我们只关注getStats在WebRTC内部的实现。

WebRTC模块对外提供两个重要对象：PeerConnectionFactory和PeerConnection，前者负责一系列重要对象的创建，如MediaStream，MediaSource，MediaTrack等等，后者则负责P2P连接的建立和维护，包括CreateOffer/Answer，AddStream等操作。监控P2P连接运行状态GetStats函数，自然在PeerConnection对象中实现，而该对象把任务委托给成员变量StatsCollector对象的UpdateStats函数来实现:

```
void StatsCollector::UpdateStats(PeerConnectionInterface::StatsOutputLevel level) {
  RTC_DCHECK(pc_->session()->signaling_thread()->IsCurrent());  // 由signal线程调用；
  double time_now = GetTimeNow();
  const double kMinGatherStatsPeriod = 50;
  if (stats_gathering_started_ != 0 &&
      stats_gathering_started_ + kMinGatherStatsPeriod > time_now) {
    return;  // 调用间隔不低于50ms；
  }

  stats_gathering_started_ = time_now;
  if (pc_->session()) {
    ExtractSessionInfo();     // 收集传输信息；
    ExtractVoiceInfo();       // 收集VoiceChannel信息；
    ExtractVideoInfo(level);  // 收集VideoChannel信息；
    ExtractSenderInfo();      // 收集PeerConnection的sender信息；
    ExtractDataInfo();        // 收集DataChannel信息；
    UpdateTrackReports();     // 更新Track报告；
  }
}
```

由该函数我们可以看到，信息的收集是分模块进行的，其中最重要的是四个模块的信息：Transport，VoiceChannel，VideoChannel，DataChannel。顾名思义，Transport是和网络相关的统计信息，而其余三个是和各自MediaChannel相关的统计信息。

Extract系列函数从相应模块收集到信息后，执行后处理操作，把不同类型的信息重新组织为类型相同的StatsReport对象，存储到StatsCollector的列表中。StatsReport对象结构基本定义如下：

```
struct StatsReport {
  const Id id_;      // 包括类型，唯一标示符等信息；
```

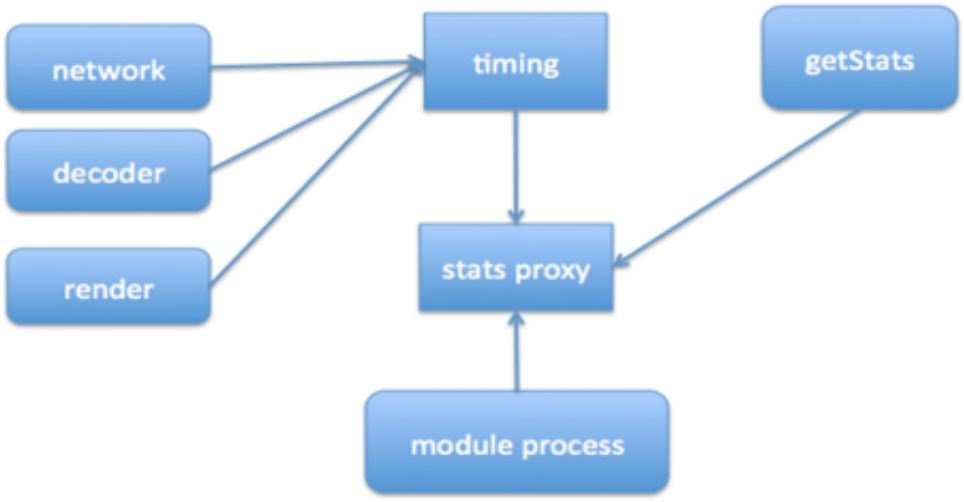
```
  double timestamp_;  // 本次信息收集的开始时间；
  Values values_;     // 信息集合，可存储int, int64, string, bool, double等类型
};
```

下面以ExtractVideoInfo为例分析信息收集过程：

```
void StatsCollector::ExtractVideoInfo(PeerConnectionInterface::StatsOutputLevel level) {
  cricket::VideoMediaInfo video_info;
  // 从video channel收集信息，包括发送端，接收端和带宽估计信息；
  if (!pc_->session()->video_channel()->GetStats(&video_info)) {
    return;
  }
  // 收集到的信息归一化为StatsReport对象；
  ExtractStatsFromList(video_info.receivers, transport_id, this,
      StatsReport::kReceive);
  ExtractStatsFromList(video_info.senders, transport_id, this,
      StatsReport::kSend);
    ExtractStats(video_info.bw_estimations[0], stats_gathering_started_, level, report);
}
```

从videochannel收集到的数据来自三个模块：VideoSendStream，VideoReceiveStream 和Call，这三个模块分别从自己的信息统计对象中获得统计数据，最后汇总为VideoMediaInfo对象，由ExtractStatsFromXX系列函数归一化为StatsReport对象。

以上分析的即为getStats函数的内部实现细节。需要注意的是，getStats只负责拉取统计数据，而统计数据本身则由WebRTC内部各个模块周期性更新，这个过程是异步的。例如，传输层的RTT是由网络线程收到数据包后实时更新，而带宽估计信息则是在受到RTCP报文后解析计算得到。下面以VideoReceiveStream统计信息的更新过程为例，深入分析这部分是如何协同工作的：



VideoReceiveStream的数据更新和拉取.png

在Video接收端，network/decoder/render三个线程在各自工作完成后，都会更新相应的统计数据到timing对象中。而module process线程则周期性更新Stats proxy对象，该对象从timing对象中拉取数据，保存在自己的stats成员变量中。最后，getstats线程调用流程到达stats proxy对象，获取stats数据而返。工作线程、更新线程和拉取线程共同协同工作完成统计数据的产生、更新和拉取。

四 总结

本文从标准、使用和实现三个方面全方位分析了WebRTC的getStats API，这对WebRTC应用的运行时监控和状态分析排错具有重要意义，我们从另一角度对WebRTC有了更深入的理解。

参考文献

[1] Identifiers for WebRTC's Statistics API:
https://www.w3.org/TR/webrtc-stats/
[2] Basics of WebRTC getStats() API:
https://www.callstats.io/2015/07/06/basics-webrtc-getstats-api/
[3] RTCPeerConnection.getStats: Chrome VS Firefox:
http://blog.telenor.io/webrtc/2015/06/11/getstats-chrome-vs-firefox.html