WebRTC getStats 详解－从标准、调用到实现

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**：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。
- **接收端延迟**：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 Javascript 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，分析这部分是如何协同工作的：

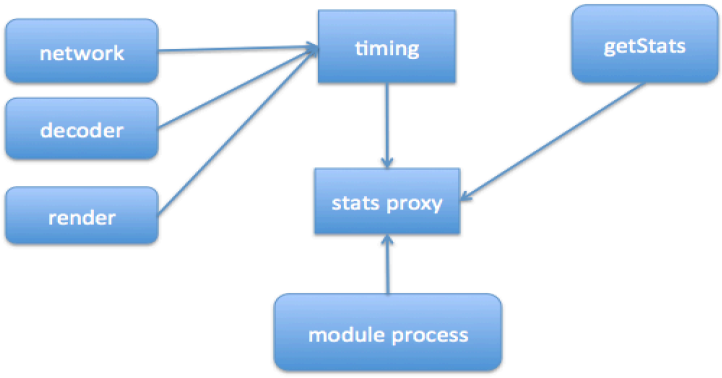

图 1 VideoReceiveStream 的数据更新和拉取

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