WebRTC 的模块处理机制

对于实时音视频应用来讲，媒体数据从采集到渲染，在数据流水线上依次完成一系列处理。流水线由不同的功能模块组成，彼此分工协作：数据采集模块负责从摄像头/麦克风采集音视频数据，编解码模块负责对数据进行编解码，RTP 模块负责数据打包和解包。数据流水线上的数据处理速度是影响应用实时性的最重要因素。与此同时，从服务质量保证角度讲，应用需要知道数据流水线的运行状态，如视频采集模块的实时帧率、当前网络的实时速率、接收端的数据丢包率，等等。各个功能模块可以基于这些运行状态信息作相应调整，从而在质量、速度等方面优化数据流水线的运行，实现更快、更好的用户体验。

WebRTC 采用模块机制，把数据流水线上功能相对独立的处理点定义为模块，每个模块专注于自己的任务，模块之间基于数据流进行通信。与此同时，专有线程收集和处理模块内部的运行状态信息，并把这些信息反馈到目标模块，实现模块运行状态监控和服务质量保证。本文在深入分析 WebRTC 源代码基础上，学习研究其模块处理机制的实现细节，从另一个角度理解 WebRTC 的技术原理。

1 WebRTC 数据流水线

我们可以把 WebRTC 看作是一个专注于实时音视频通信的 SDK。其对外的 API 主要负责 PeerConnection 建立、MediaStream 创建、NAT 穿透、SDP 协商等工作，对内则主要集中于音视频数据的处理，从数据采集到渲染的整个处理过程可以用一个数据流水线来描述，如图 1 所示。

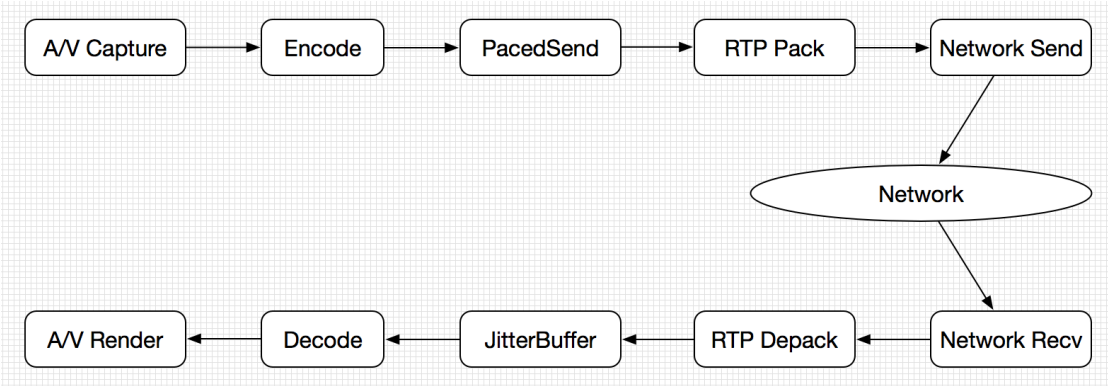

图 1 WebRTC 音视频数据流水线

音视频数据首先从采集端进行采集，一般来说音频数据来自麦克风，视频数据来自摄像头。在某些应用场景下，音频数据来自扬声器，视频数据来自桌面共享。采集端的输出是音视频 Raw Data。然后 Raw Data 到达编码模块，数据被编码器编码成符合语法规则的 NAL 单元，到达发送端缓冲区 PacedSender 处。接下来 PacedSender 把 NAL 单元发送到 RTP 模块打包为 RTP 数据包，最后经过网络模块发送到网络。

在接收端，RTP 数据包经过网络模块接收后进行解包得到 NAL 单元，接下来 NAL 单元到达接收端缓冲区(JitterBuffer 或者 NetEQ)进行乱序重排和组帧。一帧完整的数据接收并组帧之后，调用解码模块进行解码，得到该帧数据的 Raw Data。最后 Raw Data 交给渲染模块进行播放/显示。

在数据流水线上，还有一系列模块负责服务质量监控，如丢帧策略，丢包策略，编码器过度使用保护，码率估计，前向纠错，丢包重传，等等。

WebRTC 数据流水线上的功能单元被定义为模块，每个模块从上游模块获取输入数据，在本模块进行加工后得到输出数据，交给下游模块进行下一步处理。WebRTC 的模块处理机制包括模块和模块处理线程，前者把 WebRTC 数据流水线上的功能部件封装为模块，后者驱动模块内部状态更新和模块之间状态传递。

模块一般挂载到模块处理线程上，由处理线程驱动模块的处理函数。下面分别描述之。

2 WebRTC 模块

WebRTC 模块虚基类 Module 定义在 webrtc/modules/include/modue.h 中，如图 2 所示。

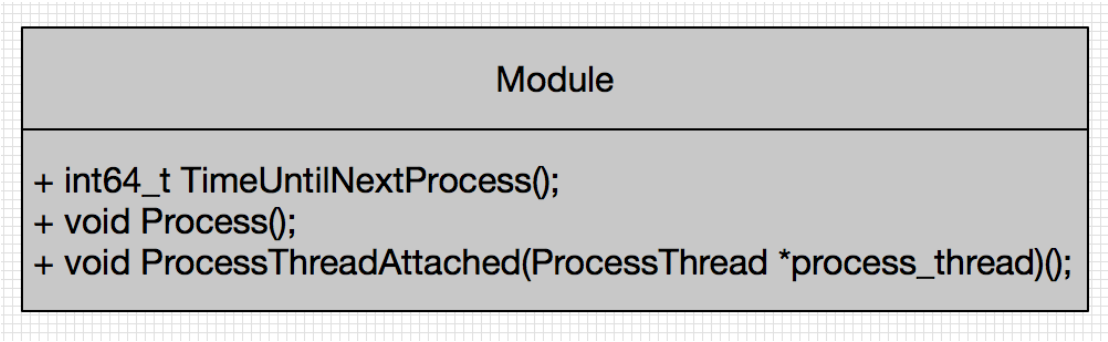

图 2 模块虚基类 Module 定义

Module 虚基类对外提供三个函数作为 API：TimeUntilNextProcess()用来计算距下次调用处理函数 Process()的时间间隔；Process()是模块的处理函数，负责模块内部运行监控、状态更新和模块间通信；ProcessThreadAttached()用来把模块挂载到模块处理线程，或者从模块处理线程分离出来，实际实现中这个函数暂时没有被用到。

Module 的派生类分布在 WebRTC 数据流水线上的不同部分，各自承担自己的数据处理和服务质量保证任务。

3 WebRTC 模块处理线程

WebRTC 模块处理线程是模块处理机制的驱动器，它的核心作用是对所有挂载在本线程下的模块，周期性调用其 Process()处理函数处理模块内部事务，并处理

异步任务。其虚基类 ProcessThread 和派生类 ProcessThreadImpl 如图 3 所示。

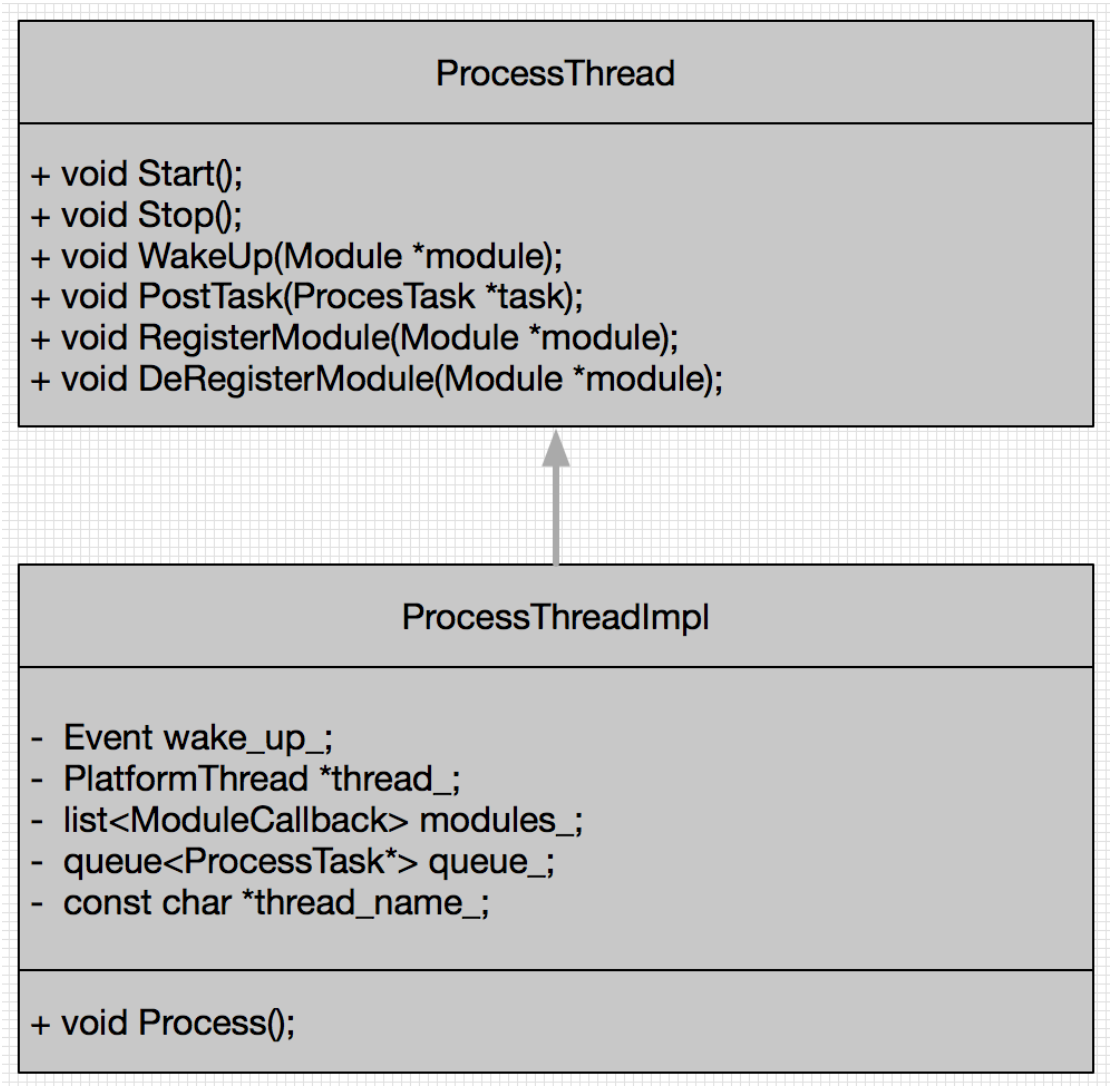

图 3 模块处理线程虚基类 ProcessThread 及派生类 ProcessThreadImpl

ProcessThread 基类提供一系列 API 完成线程功能：Start()/Stop()函数用来启动和结束线程；WakeUp()函数用来唤醒挂载在本线程下的某个模块，使得该模块有机会马上执行其 Process()处理函数；PostTask()函数用来邮递一个任务给本线程。RegisterModule()/DeRegisterModule()用来向线程注册/注销模块。

WebRTC 基于 ProcessThread 线程实现派生类 ProcessThreadImpl，如图 3 所示。在成员变量方面：wake_up_用来唤醒处于等待状态的线程；thread_是平台相关

的线程实现如 POSIX 线程；modules_是注册在本线程下的模块集合；queue_是邮递给本线程的任务集合；thread_name_是线程名字。在成员函数方面，Process()完成 ProcessThread 的核心任务，其伪代码如下所示。

```
bool ProcessThreadImpl::Process() {
    for (ModuleCallback& m : modules_) {
      if (m.next_callback == 0)
        m.next_callback = GetNextCallbackTime(m.module, now);

      if (m.next_callback <= now || m.next_callback == kCallProcessImmediately) {
        m.module->Process();
        m.next_callback = GetNextCallbackTime(m.module, rtc::TimeMillis(););
      }

      if (m.next_callback < next_checkpoint)
        next_checkpoint = m.next_callback;
    }

    while (!queue_.empty()) {
      ProcessTask* task = queue_.front();
      queue_.pop();
      task->Run();
      delete task;
    }
  }

  int64_t time_to_wait = next_checkpoint - rtc::TimeMillis();
  if (time_to_wait > 0)
    wake_up_->Wait(static_cast<unsigned long>(time_to_wait));

  return true;
}
```

Process()函数首先处理挂载在本线程下的模块，这也是模块处理线程的核心任务：针对每个模块，计算其下次调用模块 Process()处理函数的时刻(调用该模块的TimeUntilNextProcess()函数)。如果时刻是当前时刻，则调用模块的 Process()处理函数，并更新下次调用时刻。需要注意，不同模块的执行频率不一样，线程在本轮调用末尾的等待时间和本线程下所有模块的最近下次调用时刻相关。

接下来线程 Process()函数处理 ProcessTask 队列中的异步任务，针对每个任务调用 Run()函数，然后任务出队列并销毁。等模块调用和任务都处理完后，则把本次时间片的剩余时间等待完毕，然后返回。如果在等待期间其他线程向本线程 Wakeup 模块或者邮递一个任务，则线程被立即唤醒并返回，进行下一轮时间片的执行。

至此，关于 WebRTC 的模块和模块处理线程的基本实现分析完毕，下一节将对 WebRTC SDK 内模块实例和模块处理线程实例进行详细分析。

4 WebRTC 模块处理线程实例

WebRTC 关于模块和处理线程的实现在 webrtc/modules 目录下，该目录汇集了所有派生类模块和模块处理线程的实现及实例分布。本节对这些内容进行总结。

WebRTC 目前创建三个 ProcessThreadImpl 线程实例，分别是负责处理音频的 VoiceProcessTread，负责处理视频和音视频同步的 ModuleProcessThread，以及负责数据平滑发送的 PacerThread。这三个线程和挂载在线程下的模块如图 4 所示。

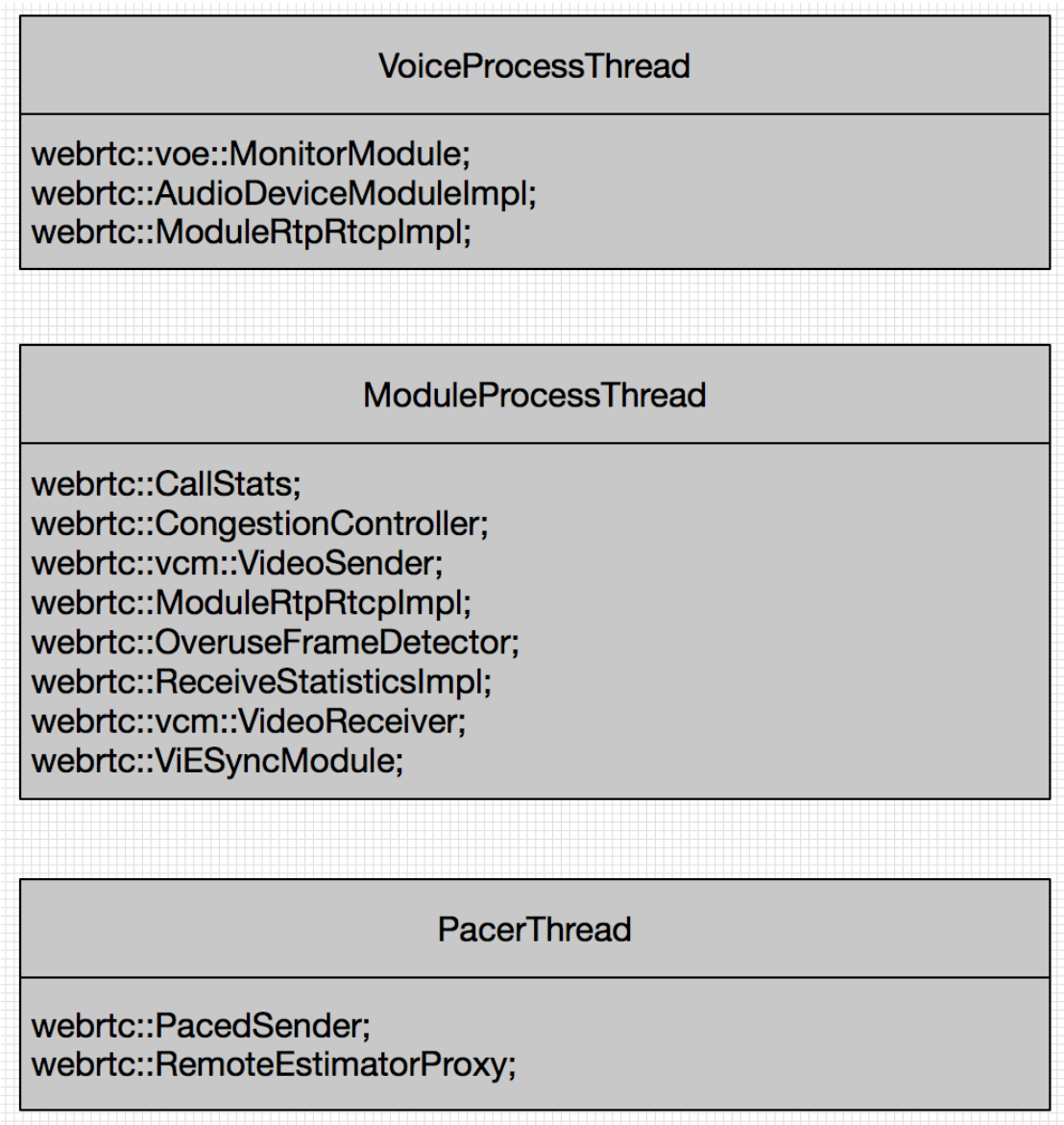

图 4 模块处理线程实例

VoiceProcessThread 线程由 Worker 线程在创建 VoiceEngine 时创建，负责音频端模块的处理。挂载在该线程下的模块如图 4 所示，其中 MonitorModule 负责对音频数据混音处理过程中产生的警告和错误进行处理，AudioDeviceModuleImpl 负责对音频设备采集和播放音频数据时产生的警告和错误进行处理，ModuleRtpRtcpImpl 负责音频 RTP 数据包发送过程中的码率计算、RTT 更新、RTCP 报文发送等内容。

ModuleProcessThread 线程由 Worker 线程在创建 VideoChannel 时创建，负责视

频端模块的处理。挂载在该线程下的模块如图 4 所示，其中 CallStats 负责 Call 对象统计数据(如 RTT)的更新，CongestionController 负责拥塞控制[1][2]，VideoSender 负责视频发送端统计数据的更新，VideoReceiver 负责视频接收端统计数据更新和处理状态反馈(如请求关键帧)，ModuleRtpRtcpImpl 负责视频 RTP 数据包发送过程中的码率计算、RTT 更新、RTCP 报文发送等内容，OveruseFrameDetector 负责视频帧采集端过载监控，ReceiveStatisticsImpl 负责由接收端统计数据触发的码率更新过程，ViESyncModule 负责音视频同步。

PacerThread 线程由 Worker 线程在创建 VideoChannel 时创建，负责数据平滑发送。挂载在该线程下的 PacedSender 负责发送端数据平滑发送；RemoteEstimatorProxy 派生自 RemoteBitrateEstimator，负责在启用发送端码率估计的情况下把接收端收集到的反馈信息发送回发送端。

由以上分析可知，WebRTC 创建的模块处理线程实例基本上涵盖了音视频数据从采集到渲染过程中的大部分数据操作。但还有一些模块不依赖于模块线程工作，这部分模块是少数，本文不展开具体的描述。

5 总结

本文在深入分析 WebRTC 源代码基础上，学习研究其模块处理机制的实现细节，为进一步全面理解 WebRTC 的技术原理奠定基础。

参考文献

[1] WebRTC 基于 GCC 的拥塞控制(上) – 算法分析

http://www.jianshu.com/p/0f7ee0e0b3be

[2] WebRTC 基于 GCC 的拥塞控制(下) - 实现分析

http://www.jianshu.com/p/5259a8659112