Webrtc 教程

1 工具：

1.1 depot_tools:

chromium 自己整了一套构建系统，原来叫 gclient（名字好像让位给 google 桌面客户端了），现在改名 depot_tools。

1.1.1 目标：

Wrapper script for checking out and updating source code from multiple SCM repository locations.

chromium 使用了（目前 @159834）107 个代码仓库的代码，这些分散在多个代码仓库，chromiun 不需要某些仓库的东西，google 就封装个工具，这个工具既支持 svn，也支持 git，不光能 down 代码，也支持了

- patch
- cpplint，pylint
- apply_issue
- junction
- codereview

1.1.2 chromium·使用它来

- 更新 chromium·代码
- 生成工程文件，windows 上生产 sln，mac 生产 xcode 工程，linux 生成 scons 或者 makefile
- 其他的 patch，codereview，管理分散开发人员的修改

1.1.3 使用说明在这儿

http://www.chromium.org/developers/how-tos/depottools

1.1.4 下载：

1.1.4.1 linux 下：

```
sudo apt-get install git
git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
```

1.1.4.2 window 下：

- 已装 cygwin：

```
git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
```

- 无 cygwin：

https://src.chromium.org/svn/trunk/tools/depot_tools.zip

1.1.5 使用：

1.1.5.1 用 gclient 获取代码

- 首先会更新 depot_tools，有两种 bat 和 sh，目的都一样
  更新 depot_tools，然后运行 python 版 gclient.py,参数都传给 gclient.py
  这里解决了鸡生蛋还是蛋生鸡的问题，更新了 gclient.py
- 生成.gclient 文件，gclient 指定了某个版本的 chromium·代码
- 执行 gclient sync，更新代码，生成工程文件，这里使用了另一个工具 GYP

1.1.5.2 gclient 命令：

```
Commands are:
   cleanup      Cleans up all working copies.
   config       Create a .gclient file in the current directory.
   diff          Displays local diff for every dependencies.
   fetch         Fetches upstream commits for all modules.
   help          Prints list of commands or help for a specific command.
```

```
   hookinfo    Output the hooks that would be run by `gclientrunhooks`
   pack         Generate a patch which can be applied at the root of the tree.
   recurse      Operates on all the entries.
   revert       Revert all modifications in every dependencies.
   revinfo      Output revision info mapping for the client and its dependencies.
   runhooks    Runs hooks for files that have been modified in the local working copy.
   status       Show modification status for every dependencies.
   sync         Checkout/update all modules.
   update       Alias for the sync command. Deprecated.


Prints list of commands or help for a specific command.


Options:
   --version                    show program's version number and exit
   -h, --help                   show this help message and exit
   -j JOBS, --jobs=JOBS    Specify how many SCM commands can run in parallel;
                                default=8
   -v, --verbose              Produces additional output for diagnostics. Can be
                                used up to three times for more logging info.
   --gclientfile=CONFIG_FILENAME
                                Specify an alternate .gclient file
   --spec=SPEC                 create a gclient file containing the provided string.
                                Due to Cygwin/Python brokenness, it probably can't
                                contain any newlines.
```

1.1.6 具体使用例子:

1.1.6.1 安装工具

http://www.chromium.org/developers/how-tos/install-depot-tools

1.1.6.2 .配置

主要是写

.gclient 和 DEPS python 语法(精确点就是 json 语法+“#”型注释，list 最末元素可以有，执行时使用 python 的 eval 来解释的)

.gclient

1.2 Gyp 工具

Gyp 工具简介见附件。Google 自己搞的玩意。Webrtc 不是直接用的 gyp，而是又封装了一下。Webrtc 中的 gyp 工具是 build/gyp_chromium

2 Webrtc

2.1 下载、编译：

2.1.1 Windows 下：

1) 下载 depot_tools 工具

a) 先装 cygwin：

git clonehttps://chromium.googlesource.com/chromium/tools/depot_tools.git

b) 无 cygwin：

https://src.chromium.org/svn/trunk/tools/depot_tools.zip

2) 把路径设置到环境变量 PATH 中。

3) 下载 webrtc 代码，最好选择稳定代码下载，trunk 是当前开发代码库。

E:\source\muli\google>gclient confighttp://webrtc.googlecode.com/svn/trunk/

这一步主要下载 git、svn、python，得到稳定版本。和配置文件.gconfig。
默认配置下载与平台相应的代码，如果要下其它平台代码。修改.gconfig 文件，加入 target_os = ['windows', 'android', 'unix']

E:\source\muli\google>gclient sync --force
下载 webrtc 代码及相关工具，有 1G 多大小。注意：如果下载中卡住了，需要翻墙。
E:\source\muli\google>gclient runhooks --force
E:\source\muli\google\trunk>python build\gyp_chromium.py --depth . -G msvs_version=2008 all.gyp
在 trunk 目录下生 all.sln

4) 安装依赖库：

Windows SDK

Windows DDK

5) 编译：

2.1.2 ubuntu 下编译：

1) 安装 depot_tools:

svn co http://src.chromium.org/svn/trunk/tools/depot_tools
我的 depot_tools 下到了 /data/google/depot_tools 中。

2) 设置环境变量，把这个目录加入到 PATH 中：

export PATH=$PATH:/data/google/depot_tools

3) 下载 webrtc 代码，最好选择稳定代码下载，trunk 是当前开发代码库。

gclient config https://webrtc.googlecode.com/svn/trunk（生成.gconfig 文件）

默认配置下载与平台相应的代码，如果要下其它平台代码。修改.gconfig 文件，加入 target_os = ['windows', 'android', 'unix']

gclient sync --force（同步项目文件，要下载 1 个多 G 的文件，网速不好的，可以去玩一会再回来），注意：如果下载中卡住了，需要翻墙。

gclientrunhooks --force （下载相应的工程文件，Linux 的 MakeFile 文件）

4) 安装依赖开发库：

apt-get install libasound2-dev
apt-get install libpulse-dev
apt-get install libx11-dev
apt-get install libxext-dev

apt-get install libnss3-dev

5） 生成工程文件

a） 生成 make 工程

k@k-C410:/data/google/webrtc/trunk$ export GYP_GENERATORS=make
指定工程文件类型，如果没有这一步，在 linux 下默认使用 ninja
k@k-C410:/data/google/webrtc/trunk$ build/gyp_chromium --depth=. all.gyp
如果你没有安装依赖库，可能会出现下面错误:

```
Updating projects from gyp files...
Package nss was not found in the pkg-config search path.
Perhaps you should add the directory containing `nss.pc'
to the PKG_CONFIG_PATH environment variable
No package 'nss' found
gyp: Call to 'pkg-config --libs-only-L --libs-only-other nss' returned exit
status 1. while loading dependencies of all.gyp while trying to load all.gyp
```

安装 libnss 库:
k@k-C410:/data/google/webrtc/trunk$ sudo apt-get install libnss3-dev

然后再生成编译工程:
k@k-C410:/data/google/webrtc/trunk$ build/gyp_chromium --depth=. all.gyp

在当前目录下产生 Makefile 文件。

编译:
k@k-C410:/data/google/webrtc/trunk$ make peerconnection_server
k@k-C410:/data/google/webrtc/trunk$ make peerconnection_client
生成的文件放在 out 目录下。
k@k-C410:/data/google/webrtc/trunk/out/Debug$ ls

```
genmacrolibvpx_obj_int_extractobj.target              re2c
genmodulelibyuv.apeerconnection_clientyasm
genperflinker.lockpeerconnection_server
genstringobjprotoc
genversionobj.hostpyproto
```

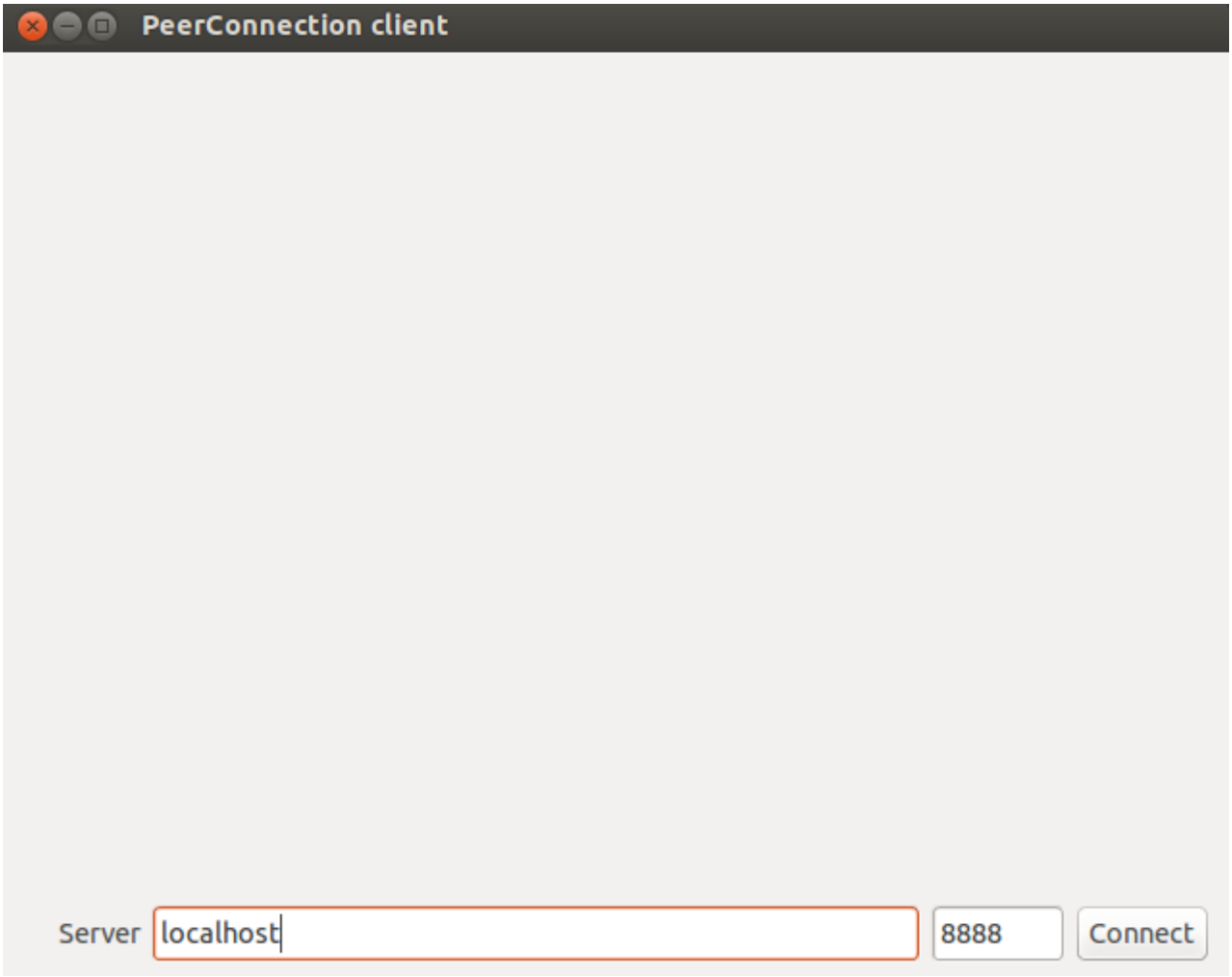
你可以看到:

peerconnection_client、 peerconnection_server 两个应用程序。
运行:
启动服务器:
k@k-C410:/data/google/webrtc/trunk/out/Debug$ ./peerconnection_server

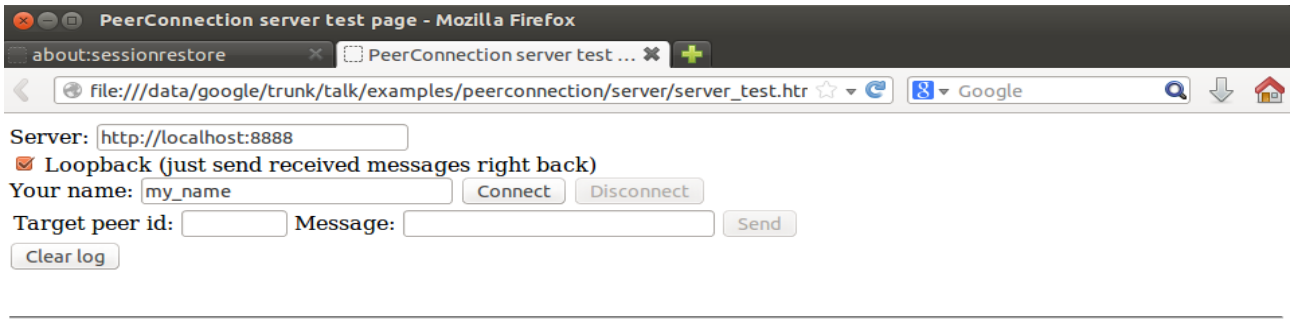
```
Server listening on port 8888
```

启动客户端:

k@k-C410:/data/google/webrtc/trunk/out/Debug$ ./peerconnection_client



也可以用 trunk/talk/examples/peerconnection/server/server_test.html 目录下的页面进行测试。



a) 生成默认工程，默认为 ninja

k@k-C410:/data/google/webrtc/trunk$ build/gyp_chromium --depth=. all.gyp

如果你没有安装依赖库，可能会出现下面错误:

Updating projects from gyp files...

Package nss was not found in the pkg-config search path.

Perhaps you should add the directory containing `nss.pc'
to the PKG_CONFIG_PATH environment variable
No package 'nss' found
gyp: Call to 'pkg-config --libs-only-L --libs-only-other nss' returned exit status 1. while loading dependencies of all.gyp while trying to load all.gyp

安装 libnss 库:
k@k-C410:/data/google/webrtc/trunk$ sudo apt-get install libnss3-dev

然后再生成编译工程:
k@k-C410:/data/google/webrtc/trunk$ build/gyp_chromium --depth=. all.gyp

在当前目录下产生 out 目录，在 out 目录中有 Debug、Release 两个子目录，在子目录中有 ninja 工程文件 build.ninja

编译指定的目标:
k@k-C410:/data/google/webrtc/trunk$ ninja -C outpeerconnection_server
k@k-C410:/data/google/webrtc/trunk$ ninja -C outpeerconnection_client
编译所有工程目标：
k@k-C410:/data/google/webrtc/trunk$ ninja -C out All
在 Debug 下可以看到许多的测试程序。

2.1.3 编译 Android:

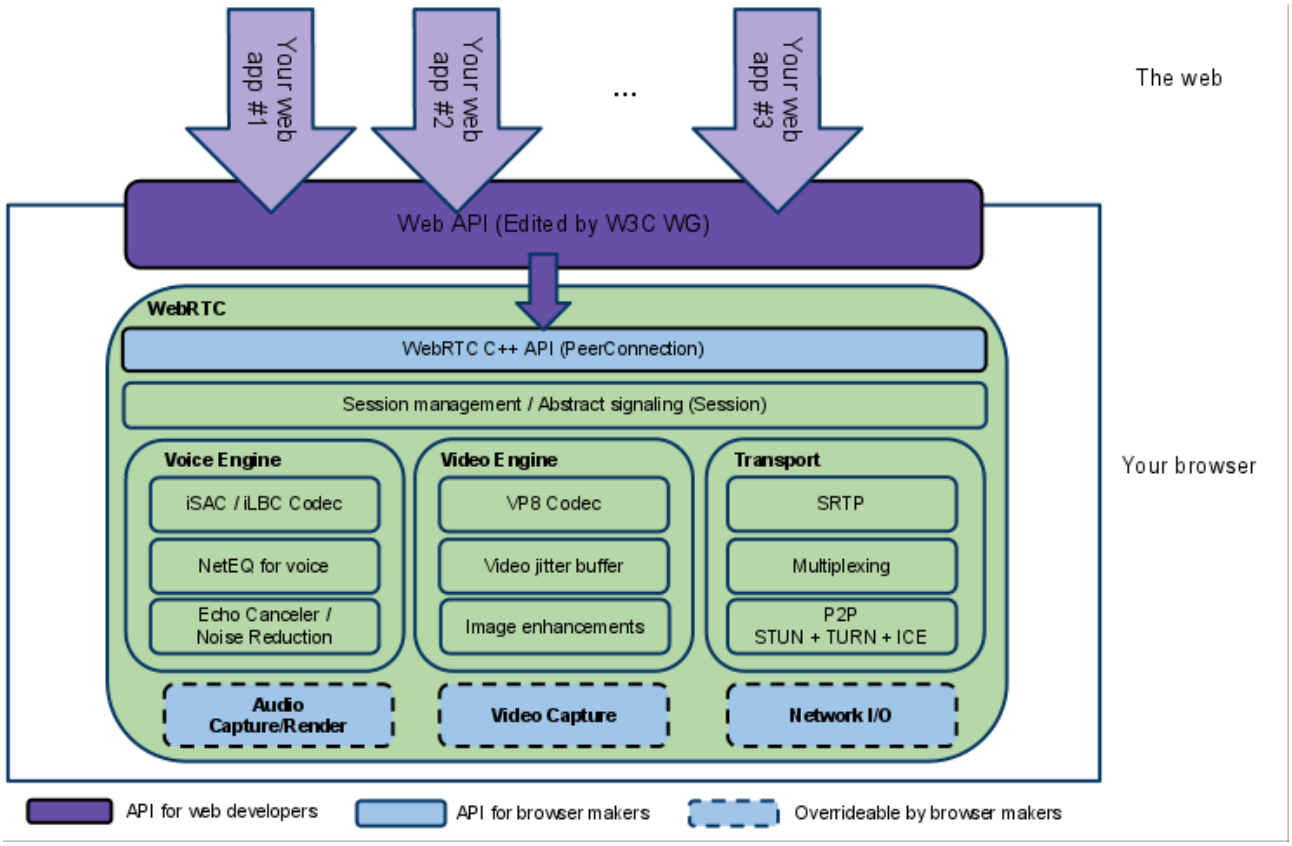
设置环境变量：ANDROID_NDK_ROOT、ANDROID_SDK_ROOT、JAVA_HOME
k@k-C410:/data/google/trunk$ export JAVA_HOME=/data/jdk1.7.0_45
k@k-C410:/data/google/trunk$ export ANDROID_SDK_ROOT=/data/adt-bundle-linux-x86_64-20130917/sdk
k@k-C410:/data/google/trunk$ export ANDROID_NDK_ROOT=/data/android-ndk-r9
k@k-C410:/data/google/trunk$ source build/android/envsetup.sh
k@k-C410:/data/google/trunk$ build/gyp_chromium --depth=. all.gyp
k@k-C410:/data/google/trunk$ ninja -C out/Debug All
在编译过程中会有些类实例调用静态方法的警告错误。需要提示修改代码。
生成所有程序。
*k@k-C410:/data/google/trunk/out/Debug$ ls *.apk*
AppRTCDemo-debug.apkOpenSlDemo-debug.apkWebRTCDemo-debug.apk

在 Debug 下可以看到许多的测试程序。

3 webrtc 开发：

在编译环境搭建完成后，可以开始干活了。我们用 webrtc 的主要目的是应用它的音 / 视频捕获、视频显示、音频播放、音视频压缩、网络通信。如果你只是想在 HTML5 中用音 / 视频解决方案，可以跳过本教程，因为 webrtc 的最终目标是实现 HTML5 的音视频解决方案。本教程主要讲解在应用程序中如何使用 webrtc 库。

3.1 webrtc 整个架构：



（1）紫色部分是 Web 开发者 API 层；
（2）蓝色实线部分是面向浏览器厂商的 API 层（本教程主要讲解的部分）
（3）蓝色虚线部分浏览器厂商可以自定义实现

3.1.1 WebRTC 架构组件介绍

(1) Your Web App
Web 开发者开发的程序，Web 开发者可以基于集成 WebRTC 的浏览器提供的 web API 开发基于视频、音频的实时通信应用。

(2) Web API
面向第三方开发者的 WebRTC 标准 API（Javascript），使开发者能够容易地开发出类似于网络视频聊天的 web 应用，最新的标准化进程可以查看这里。

(3) WebRTC Native C++ API
本地 C++ API 层，使浏览器厂商容易实现 WebRTC 标准的 Web API，抽象地对数字信号过程进行处理。

(4) Transport / Session
传输/会话层
会话层组件采用了 libjingle 库的部分组件实现，无须使用 xmpp/jingle 协议
参见：https://developers.google.com/talk/talk_developers_home

a. RTP Stack 协议栈
Real Time Protocol

b. STUN/ICE
可以通过 STUN 和 ICE 组件来建立不同类型网络间的呼叫连接。

c. Session Management

一个抽象的会话层，提供会话建立和管理功能。该层协议留给应用开发者自定义实现。

(5) VoiceEngine

音频引擎是包含一系列音频多媒体处理的框架，包括从视频采集卡到网络传输端等整个解决方案。

PS：VoiceEngine 是 WebRTC 极具价值的技术之一，是 Google 收购 GIPS 公司后开源的。在 VoIP 上，技术业界领先，后面的文章会详细了解

a. iSAC

Internet Speech Audio Codec

针对 VoIP 和音频流的宽带和超宽带音频编解码器，是 WebRTC 音频引擎的默认的编解码器

采样频率：16khz，24khz，32khz；（默认为 16khz）

自适应速率为 10kbit/s ~ 52kbit/；

自适应包大小：30~60ms；

算法延时：frame + 3ms

b. iLBC

Internet Low Bitrate Codec

VoIP 音频流的窄带语音编解码器

采样频率：8khz；

20ms 帧比特率为 15.2kbps

30ms 帧比特率为 13.33kbps

标准由 IETF RFC3951 和 RFC3952 定义

c. NetEQ for Voice

针对音频软件实现的语音信号处理元件

NetEQ 算法：自适应抖动控制算法以及语音包丢失隐藏算法。使其能够快速且高解析度地适应不断变化的网络环境，确保音质优美且缓冲延迟最小。

是 GIPS 公司独步天下的技术，能够有效的处理由于网络抖动和语音包丢失时候对语音质量产生的影响。

PS：NetEQ 也是 WebRTC 中一个极具价值的技术，对于提高 VoIP 质量有明显效果，加以 AEC\NR\AGC 等模块集成使用，效果更好。

d. Acoustic Echo Canceler (AEC)

回声消除器是一个基于软件的信号处理元件，能实时的去除 mic 采集到的回声。

e. Noise Reduction (NR)

噪声抑制也是一个基于软件的信号处理元件，用于消除与相关 VoIP 的某些类型的背景噪声（嘶嘶声，风扇噪音等等… …）

(6) VideoEngine

WebRTC 视频处理引擎

VideoEngine 是包含一系列视频处理的整体框架，从摄像头采集视频到视频信息网络传输再到视频显示整个完整过程的解决方案。

a. VP8

视频图像编解码器，是 WebRTC 视频引擎的默认的编解码器

VP8 适合实时通信应用场景，因为它主要是针对低延时而设计的编解码器。

PS:VPx 编解码器是 Google 收购 ON2 公司后开源的，VPx 现在是 WebM 项目的一部分，而 WebM 项目是 Google 致力于推动的 HTML5 标准之一

b. Video Jitter Buffer

视频抖动缓冲器，可以降低由于视频抖动和视频信息包丢失带来的不良影响。

c. Image enhancements

图像质量增强模块

对网络摄像头采集到的图像进行处理，包括明暗度检测、颜色增强、降噪处理等功能，用来提升视频质量。

3.1.2 WebRTC 核心模块 API

(1)、网络传输模块：libjingle

WebRTC 重用了 libjingle 的一些组件，主要是 network 和 transport 组件，关于 libjingle 的文档资料可以查看**这里**。

参见：https://developers.google.com/talk/talk_developers_home

(2)、音频、视频图像处理的主要数据结构

常量\VideoEngine\VoiceEngine

注意：以下所有的方法、类、结构体、枚举常量等都在 webrtc 命名空间里

类、结构体、枚举常量	头文件	说明
Structures	common_types.h	Lists the structures common to the VoiceEngine&VideoEngine
Enumerators	common_types.h	List the enumerators common to the VoiceEngine&VideoEngine
Classes	common_types.h	List the classes common to VoiceEngine&VideoEngine
class VoiceEngine	voe_base.h	How to allocate and release resources for the VoiceEngine using factory methods in the VoiceEngine class. It also lists the APIs which are required to enable file tracing and/or traces as callback messages
class VideoEngine	vie_base.h	How to allocate and release resources for the VideoEngine using factory methods in the VideoEngine class. It also lists the APIs which are required to enable file tracing and/or traces as callback messages

(3)、音频引擎（VoiceEngine）模块 APIs

下表列的是目前在 VoiceEngine 中可用的 sub APIs

sub-API	头文件	说明
VoEAudioProcessing	voe_audio_processing.h	Adds support for Noise Suppression (NS), Automatic Gain Control (AGC) and Echo Control (EC). Receiving side VAD is also included.
VoEBase	voe_base.h	Enables full duplex VoIP using G.711. **NOTE:** This API must always be created.
VoECallReport	voe_call_report.h	Adds support for call reports which contains number of dead-or-alive detections, RTT measurements, and Echo metrics.

VoECodec	voe_codec.h	Adds non-default codecs (e.g. iLBC, iSAC, G.722 etc.), Voice Activity Detection (VAD) support.
VoEDTMF	voe_dtmf.h	Adds telephone event transmission, DTMF tone generation and telephone event detection. (Telephone events include DTMF.)
VoEEncryption	voe_encryption.h	Adds external encryption/decryption support.
VoEErrors	voe_errors.h	Error Codes for the VoiceEngine
VoEExternalMedia	voe_external_media.h	Adds support for external media processing and enables utilization of an external audio resource.
VoEFile	voe_file.h	Adds file playback, file recording and file conversion functions.
VoEHardware	voe_hardware.h	Adds sound device handling, CPU load monitoring and device information functions.
VoENetEqStats	voe_neteq_stats.h	Adds buffer statistics functions.
VoENetwork	voe_network.h	Adds external transport, port and address filtering, Windows QoS support and packet timeout notifications.
VoERTP_RTCP	voe_rtp_rtcp.h	Adds support for RTCP sender reports, SSRC handling, RTP/RTCP statistics, Forward Error Correction (FEC), RTCP APP, RTP capturing and RTP keepalive.
VoEVideoSync	voe_video_sync.h	Adds RTP header modification support, playout-delay tuning and monitoring.
VoEVolumeControl	voe_volume_control.h	Adds speaker volume controls, microphone volume controls, mute support, and additional stereo scaling methods.

(4)、视频引擎（VideoEngine）模块 APIs

下表列的是目前在 VideoEngine 中可用的 sub APIs

sub-API	头文件	说明
ViEBase	vie_base.h	Basic functionality for creating a VideoEngine instance, channels and VoiceEngine interaction. **NOTE:** This API must always be created.
ViECapture	vie_capture.h	Adds support for capture device allocation as well as capture device capabilities.
ViECodec	vie_codec.h	Adds non-default codecs, codec settings and packet loss functionality.
ViEEncryption	vie_encryption.h	Adds external encryption/decryption support.
ViEErrors	vie_errors.h	Error codes for the VideoEngine
ViEExternalCodec	vie_external_codec.h	Adds support for using external codecs.
ViEFile	vie_file.h	Adds support for file recording, file playout, background images and snapshot.
ViEImageProcess	vie_image_process.h	Adds effect filters, deflickering, denoising and color enhancement.
ViENetwork	vie_network.h	Adds send and receive functionality, external transport, port and address filtering, Windows QoS support, packet timeout notification and

		changes to network settings.
ViERender	vie_render.h	Adds rendering functionality.
ViERTP_RTCP	vie_rtp_rtcp.h	Adds support for RTCP reports, SSRS handling RTP/RTCP statistics, NACK/FEC, keep-alive functionality and key frame request methods.

3.1.3 webRTC 本地 API

webRTC 本地 API：http://www.webrtc.org/reference/native-apis。
翻译：http://www.cnblogs.com/longrenle/archive/2012/03/04/2378433.html 。
webRTC 本地 API 是基于 WebRTC spec 的实现。webRTC 的实现代码（包括流和 PeerConnection API）是在 libjingle 中实现。

线程模型

WebRTC native APIs 拥有两个全局线程：信令线程（signaling thread）和工作者线程（worker thread）。取决于 PeerConnection factory 被创建的方式，应用程序可以提供这两个线程或者直接使用内部创建好的线程。

Stream APIs 和 PeerConnection APIs 的调用会被代理到信令线程，这就意味着应用程序可以在任何线程调用这些 APIs。

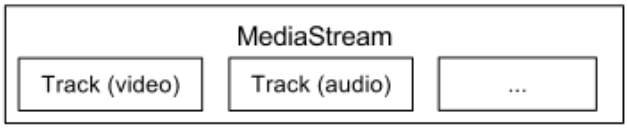
所有的回调函数都在信令线程调用。应用程序应当尽快地跳出回调函数以避免阻塞信令线程。严重消耗资源的过程都应当其他的线程执行。

工作者线程被用来处理资源消耗量大的过程，比如说数据流传输。

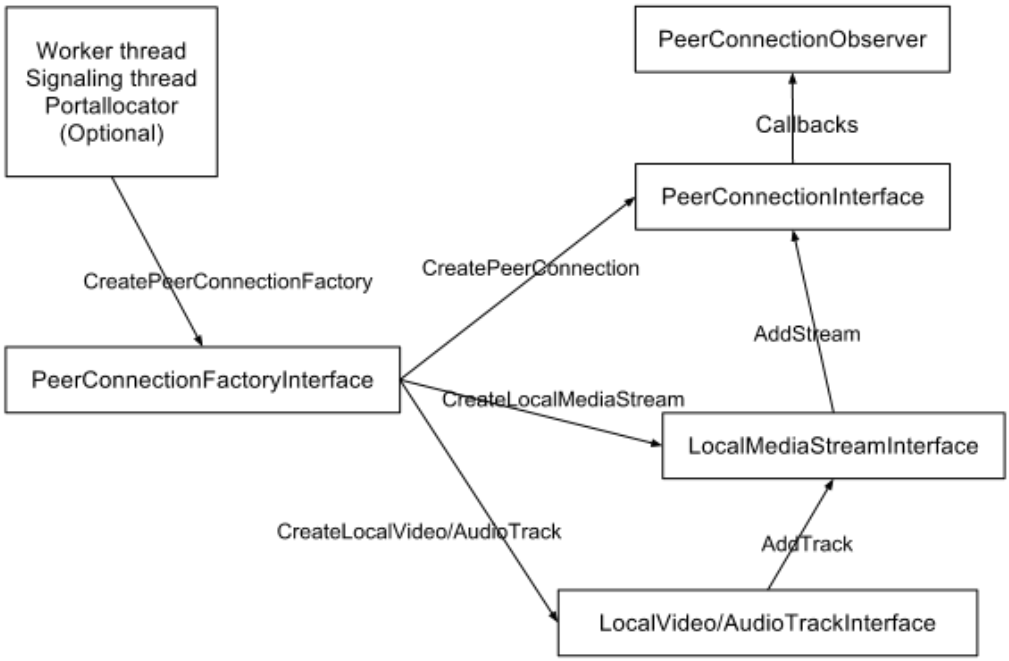
3.1.4 libjingle_peerconnection

块图：

Stream

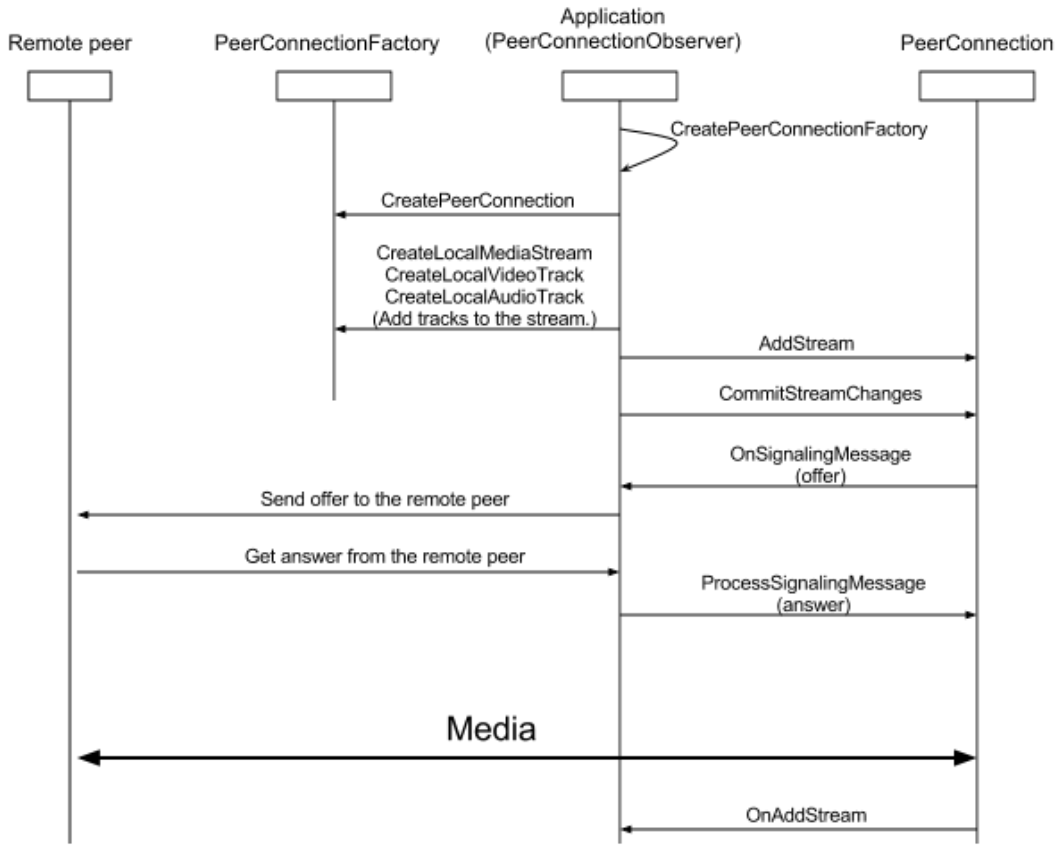


PeerConnection



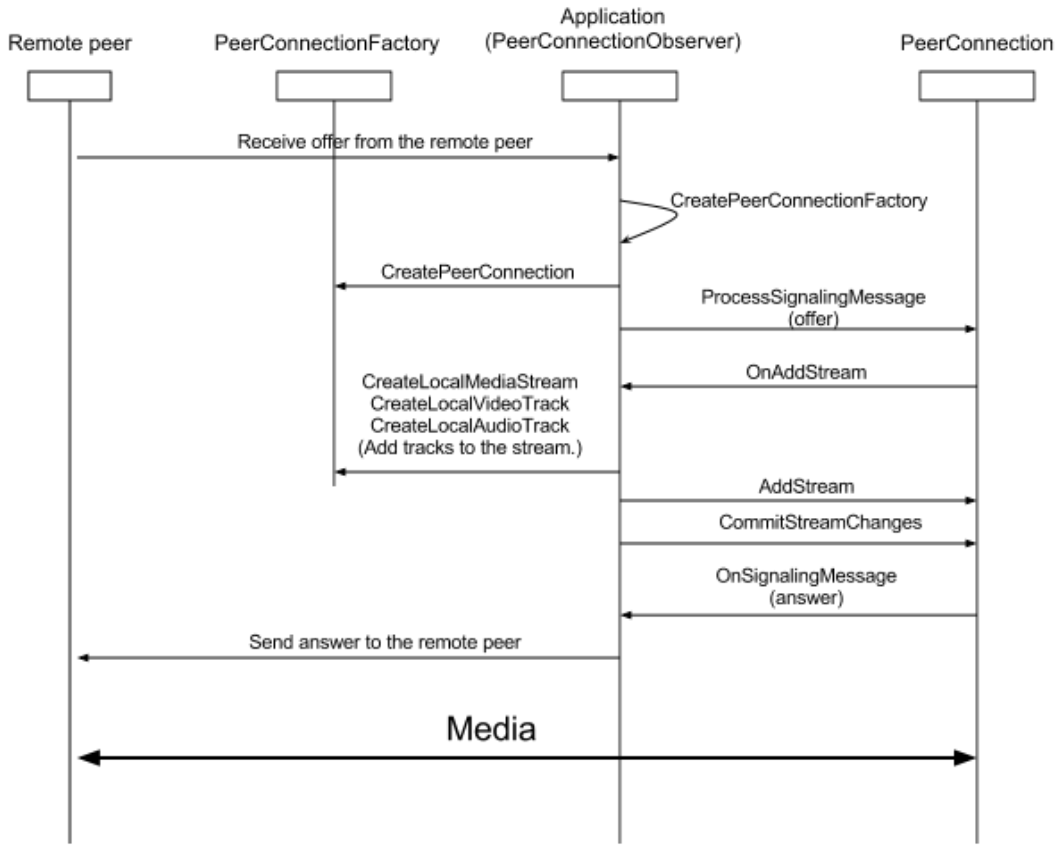
调用顺序：

安装调用：



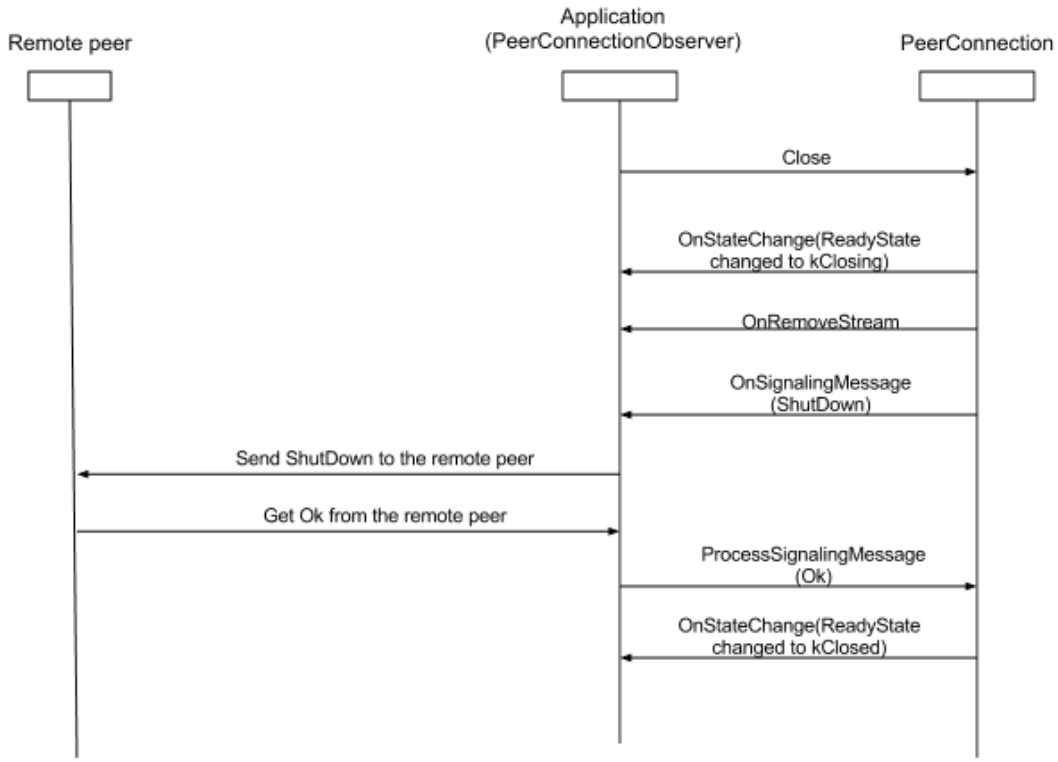

```
// The Following steps are needed to setup a typical call using Jsep.
// 1. Create a PeerConnectionFactoryInterface. Check constructors for more
// information about input parameters.
// 2. Create a PeerConnection object. Provide a configuration string which
// points either to stun or turn server to generate ICE candidates and provide
// an object that implements the PeerConnectionObserver interface.
// 3. Create local MediaStream and MediaTracks using the PeerConnectionFactory
// and add it to PeerConnection by calling AddStream.
// 4. Create an offer and serialize it and send it to the remote peer.
// 5. Once an ice candidate have been found PeerConnection will call the
// observer function OnIceCandidate. The candidates must also be serialized and
// sent to the remote peer.
// 6. Once an answer is received from the remote peer, call
// SetLocalSessionDescription with the offer and SetRemoteSessionDescription
// with the remote answer.
// 7. Once a remote candidate is received from the remote peer, provide it to
// the peerconnection by calling AddIceCandidate.
```

接收调用：

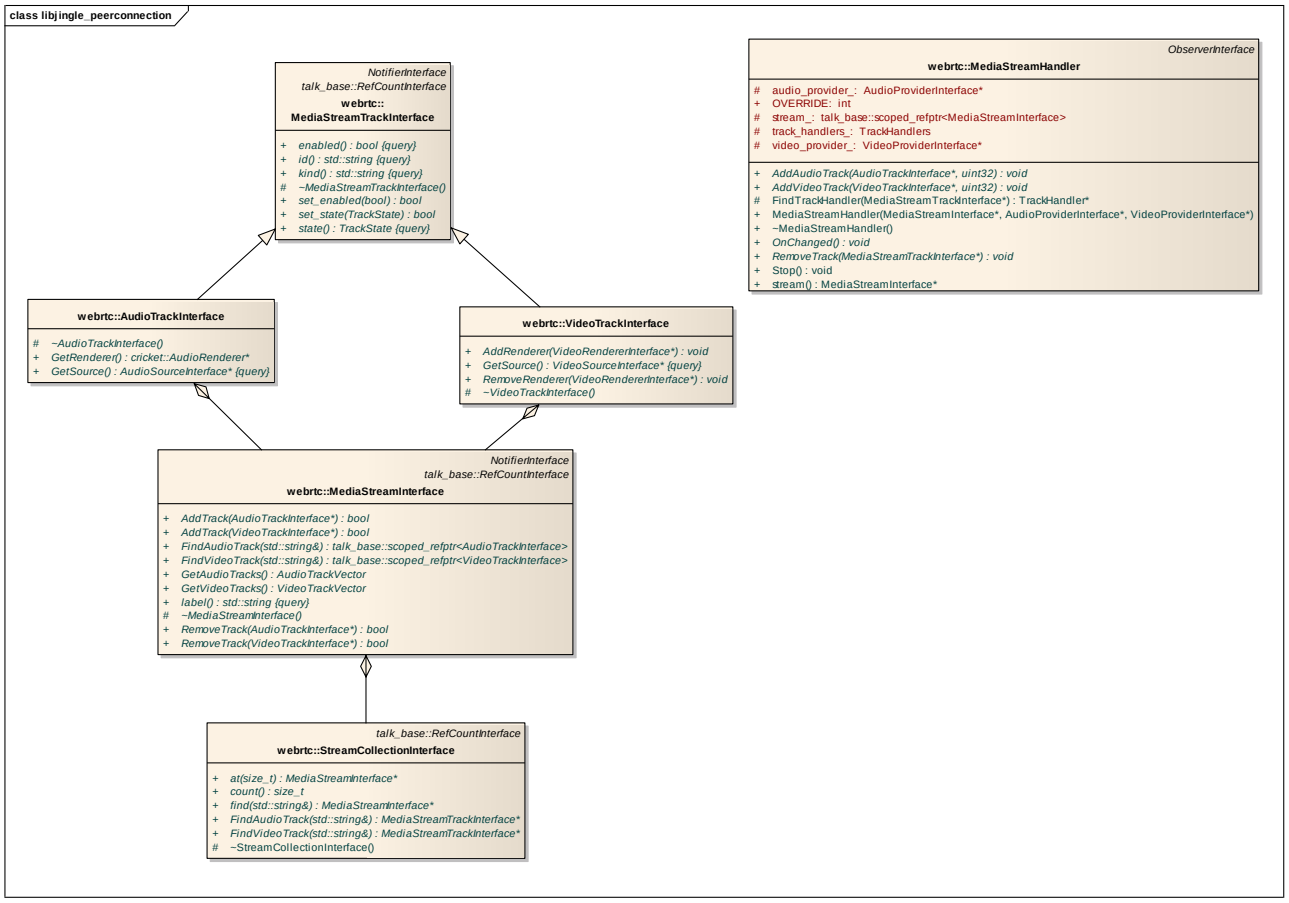



```
// The Receiver of a call can decide to accept or reject the call.
// This decision will be taken by the application not peerconnection.
// If application decides to accept the call
// 1. Create PeerConnectionFactoryInterface if it doesn't exist.
// 2. Create a new PeerConnection.
// 3. Provide the remote offer to the new PeerConnection object by calling
// SetRemoteSessionDescription.
// 4. Generate an answer to the remote offer by calling CreateAnswer and send it
// back to the remote peer.
// 5. Provide the local answer to the new PeerConnection by calling
// SetLocalSessionDescription with the answer.
// 6. Provide the remote ice candidates by calling AddIceCandidate.
// 7. Once a candidate have been found PeerConnection will call the observer
// function OnIceCandidate. Send these candidates to the remote peer.
```

关闭调用：


Remote peer
Application
(PeerConnectionObserver)
PeerConnection
Close
OnStateChange(ReadyState
changed to kClosing)
OnRemoveStream
OnSignalingMessage
(ShutDown)
Send ShutDown to the remote peer
Get Ok from the remote peer
ProcessSignalingMessage
(Ok)
OnStateChange(ReadyState
changed to kClosed)

webrtc 本地 API：talk/app/webrtc/mediastreaminterface.h

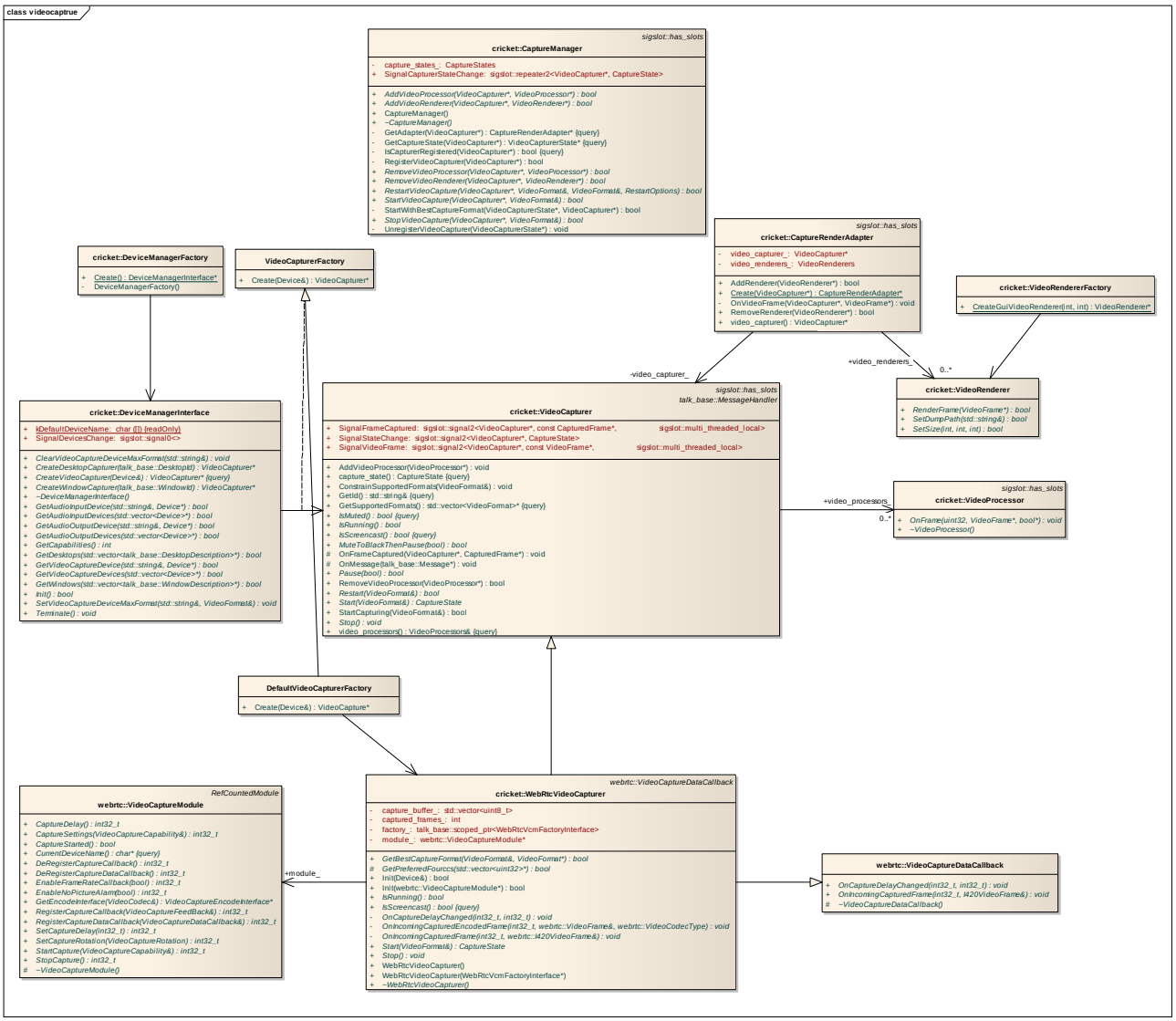


peerconnectionAPI:talk/app/webrtc/peerconnectioninterface.h

3.1.5 libjingle_media 库：

3.1.5.1 视频采集，处理、渲染类：

视频捕获类：



- 得到 VideoCapture 过程：

```
cricket::VideoCapturer* Conductor::OpenVideoCaptureDevice() {
talk_base::scoped_ptr<cricket::DeviceManagerInterface>dev_manager(
cricket::DeviceManagerFactory::Create());
if (!dev_manager->Init()) {
LOG(LS_ERROR) <<"Can't create device manager";
returnNULL;
  }
std::vector<cricket::Device>devs;
if (!dev_manager->GetVideoCaptureDevices(&devs)) {
LOG(LS_ERROR) <<"Can't enumerate video devices";
returnNULL;
  }
std::vector<cricket::Device>::iteratordev_it = devs.begin();
cricket::VideoCapturer* capturer = NULL;
for (; dev_it != devs.end(); ++dev_it) {
capturer = dev_manager->CreateVideoCapturer(*dev_it);
if (capturer != NULL)
break;
```

```
  }
returncapturer;
}
```

说明：

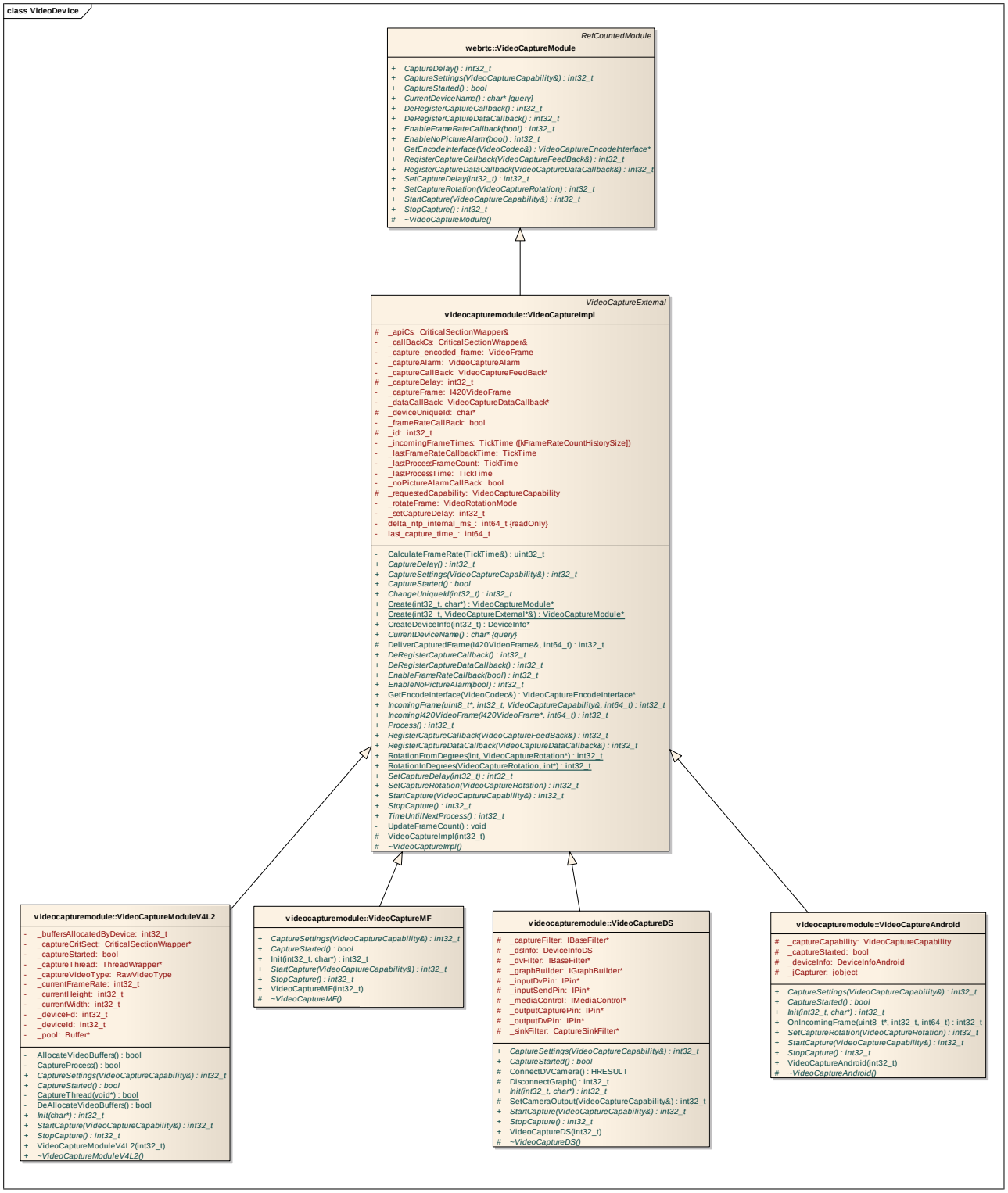
- 捕获设备的建立：

建立 DeviceManagerFactory 实例。Windows 下在实例初始化时并实例化 DefaultVideoCapturerFactory。
调用 dev_manager->GetVideoCaptureDevices 得到设备。
调用 dev_manager->CreateVideoCapturer 建立一个视频捕获对象。
在 dev_manager->CreateVideoCapturer 中，
先检查是否是文件捕获对象。如果，则返回文件捕获对象指针。
如果不是，调用 device_video_capturer_factory_->Create(device);建立视频捕获对象。这里 device_video_capturer_factory_=DefaultVideoCapturerFactory

```
classDefaultVideoCapturerFactory : publicVideoCapturerFactory {
public:
DefaultVideoCapturerFactory() {}
virtual ~DefaultVideoCapturerFactory() {}

VideoCapturer* Create(constDevice&device) {
#ifdefined(VIDEO_CAPTURER_NAME)
VIDEO_CAPTURER_NAME* return_value = newVIDEO_CAPTURER_NAME;
if (!return_value->Init(device)) {
deletereturn_value;
returnNULL;
    }
returnreturn_value;
#else
    return NULL;
#endif
  }
};
```

这个 DefaultVideoCapturerFactory 类厂很简单，在 Create 函数中建立 WebRtcVideoCapturer 对象。并调用 Init 初始化此对象。在这个初始化中，会建立调用 module_ = factory_->Create(0, vcm_id);其中 factory_建立过程如下：
WebRtcVcmFactory-> VideoCaptureFactory-> videocapturemodule::VideoCaptureImpl-> VideoCaptureDS
类关系详见捕获设备类：



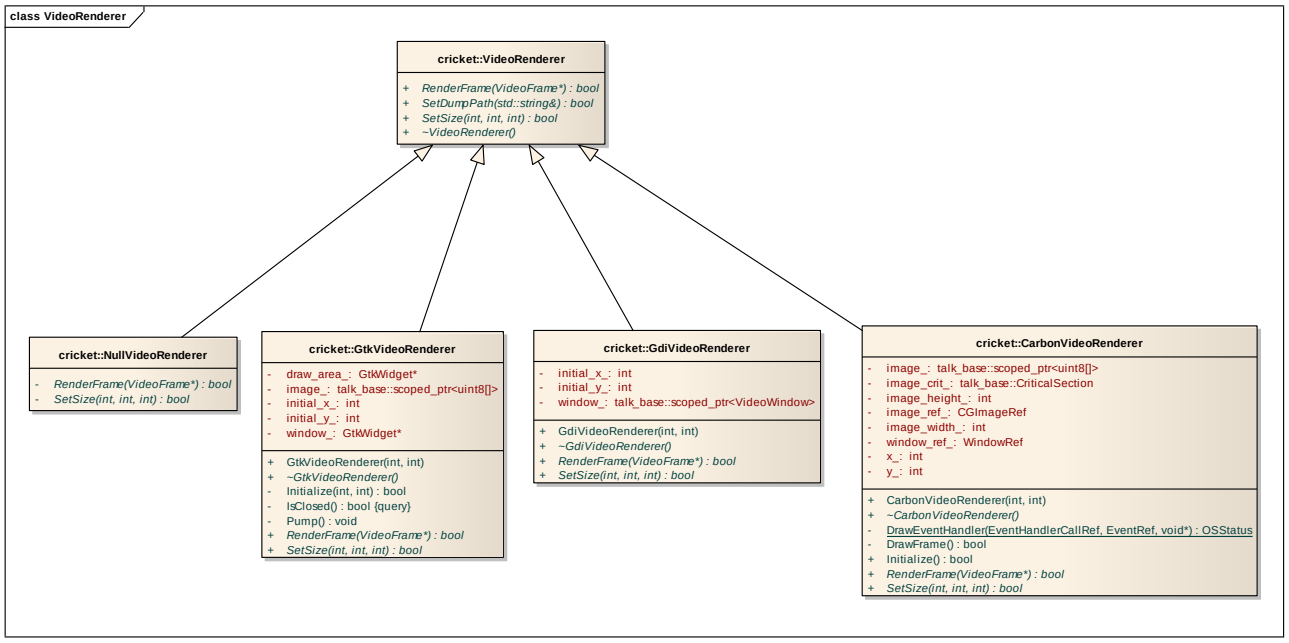
VideoCaptureModuleV4L2：linux 下 v4l2 捕获设备。
VideoCaptureDS：windows 下 DirectxShow 捕获设备。
VideoCaptureAndroid：android 下捕获设备。

- 捕获数据流：

在 WebRtcVideoCapturer 中的 Start 函数中，会把回调处理对象 VideoCaptureDataCallback 与视频捕获设备对象 VideoCaptureModule 进行关联。这样，当视频捕获设备对象有视频帧被捕获时。会调用 VideoCaptureDataCallback 中的 OnIncomingCapturedFrame。WebRtcVideoCapturer 继承 VideoCaptureDataCallback，所以会调用 WebRtcVideoCapturer::OnIncomingCapturedFrame。在其中调用 SignalFrameCaptured(this, &frame);发送捕获信息。会调用基类 VideoCapturer::OnFrameCaptured。

在这里会对视频格式由 BITMAP 转换为 I420。并且会调用已注册的 VideoProcessors 对象。还会 SignalVideoFrame(this, &i420_frame);发送视频帧信号。

- 渲染类：



GdiVideoRenderer:windowsgdi 渲染
GtkVideoRenderer: GTK 库渲染，这个用在 linux 平台。

3.1.6 Webrtc 库介绍：

trunk\webrtc\modules

视频采集---video_capture

源代码在 webrtc/modules/video_capture/main 目录下，包含接口和各个平台的源代码。

在 windows 平台上，WebRTC 采用的是 dshow 技术，来实现枚举视频的设备信息和视频数据的采集，这意味着可以支持大多数的视频采集设备；对那些需要单独驱动程序的视频采集卡（比如海康高清卡）就无能为力了。

视频采集支持多种媒体类型，比如 I420、YUY2、RGB、UYUY 等，并可以进行帧大小和帧率控制。

视频编解码---video_coding

源代码在 webrtc/modules/video_coding 目录下。

WebRTC 采用 I420/VP8 编解码技术。VP8 是 google 收购 ON2 后的开源实现，并且也用在 WebM 项目中。VP8 能以更少的数据提供更高质量的视频，特别适合视频会议这样的需求。

视频加密--video_engine_encryption

视频加密是 WebRTC 的 video_engine 一部分，相当于视频应用层面的功能，给点对点的视频双方提供了数据上的安全保证，可以防止在 Web 上视频数据的泄漏。

视频加密在发送端和接收端进行加解密视频数据，密钥由视频双方协商，代价是会影响视频数据处理

的性能；也可以不使用视频加密功能，这样在性能上会好些。

视频加密的数据源可能是原始的数据流，也可能是编码后的数据流。估计是编码后的数据流，这样加密代价会小一些，需要进一步研究。

视频媒体文件--media_file

源代码在 webrtc/modules/media_file 目录下。

该功能是可以用本地文件作为视频源，有点类似虚拟摄像头的功能；支持的格式有 Avi。

另外，WebRTC 还可以录制音视频到本地文件，比较实用的功能。

视频图像处理--video_processing

源代码在 webrtc/modules/video_processing 目录下。

视频图像处理针对每一帧的图像进行处理，包括明暗度检测、颜色增强、降噪处理等功能，用来提升视频质量。

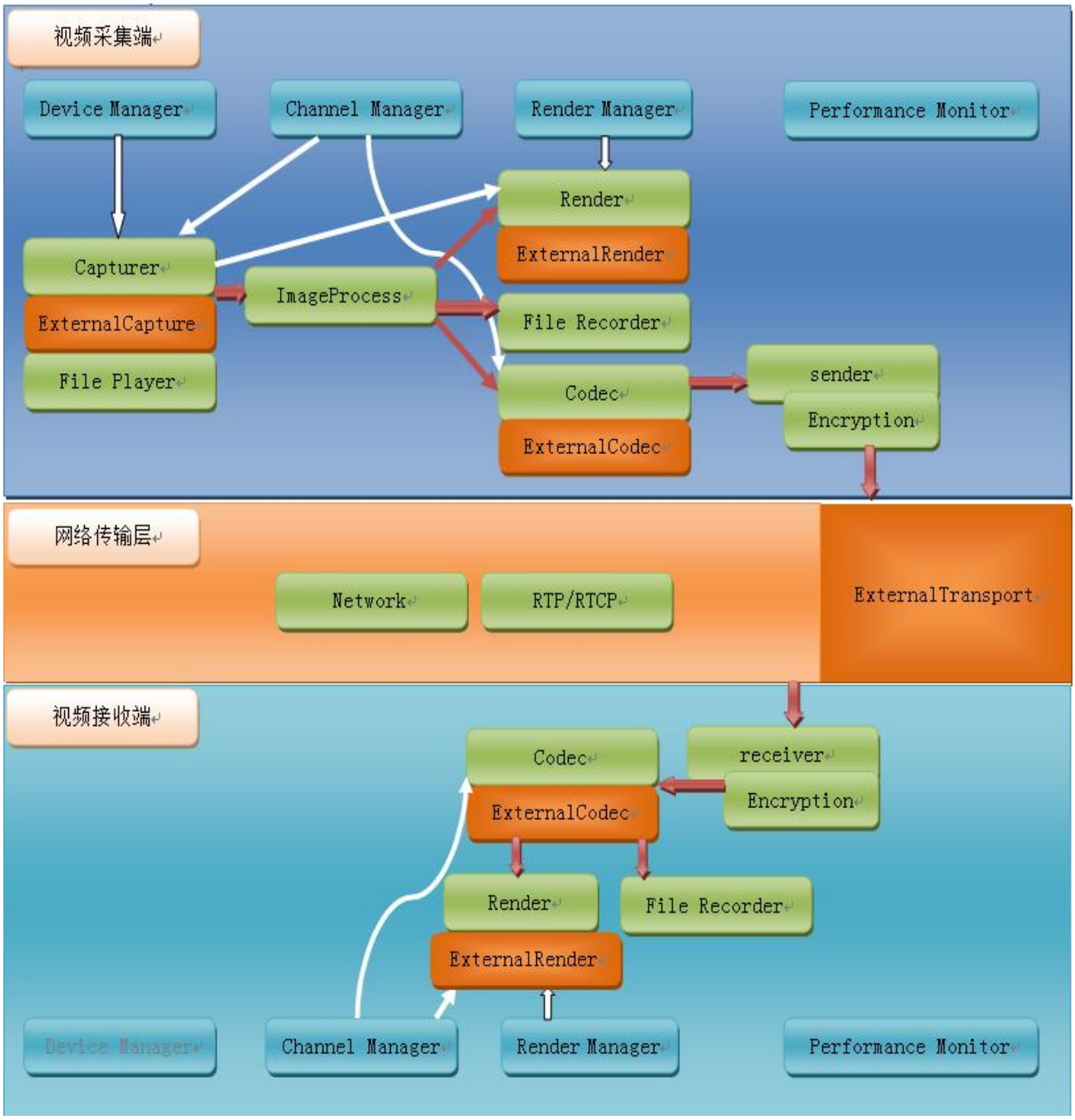
视频显示--video_render

源代码在 webrtc/modules/video_render 目录下。

在 windows 平台，WebRTC 采用 direct3d9 和 directdraw 的方式来显示视频，只能这样，必须这样。

网络传输与流控

对于网络视频来讲，数据的传输与控制是核心价值。WebRTC 采用的是成熟的 RTP/RTCP 技术。



webrtc.eap

4 附件：

4.1 Gyp 工具

GYP 简介:转载自：http://blog.xiaogaozi.org/2011/10/29/introduction-to-gyp/

说起项目构建工具，Linux 用户最熟悉的恐怕就是 Autotools，它将编译安装这个步骤大大简化。但对于项目作者来说，想要使用 Autotools 生成有效的配置文件着实需要下一番功夫，用现在流行的话来说就是用户体验不够友好。对 Unix shell 的依赖，也使得 Autotools 天生对于跨平台支持不佳。

与其类似的有 CMake，CMake 使用 C++ 编写，原生支持跨平台，不需要像 Autotools 那样写一堆的配置文件，只需一个 CMakeLists.txt 文件即可。简洁的使用方式，强大的功能使得我立马对 CMake 情有独钟。在后来的使用过程中，虽然会遇到一些因为使用习惯带来的小困扰，但我对于 CMake 还是基本满意的。直到我发现了 GYP。

GYP（Generate Your Projects）是由 Chromium 团队开发的跨平台自动化项目构建工具，Chromium 便是通过 GYP 进行项目构建管理。为什么我要选择 GYP，而放弃 CMake 呢？功能上 GYP 和 CMake 很是相似，在我看来，它们的最大区别在于配置文件的编写方式和其中蕴含的思想。

编写 CMake 配置文件相比 Autotools 来说已经简化很多，一个最简单的配置文件只需要写上源文件及生成类型（可执行文件、静态库、动态库等）即可。对分支语句和循环语句的支持也使得 CMake 更加灵活。但是，CMake 最大的问题也是在这个配置文件，请看下面这个示例文件：

```
1  cmake_minimum_required(VERSION 2.8)
2  project(VP8 CXX)
3
4  add_definitions(-Wall)
5  cmake_policy(SET CMP0015 NEW)
6  include_directories("include")
7  link_directories("lib")
8  set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY "../lib")
9  set(VP8SRC VP8Encoder.cpp VP8Decoder.cpp)
10
11 if(X86)
12    set(CMAKE_SYSTEM_NAME Darwin)
13    set(CMAKE_SYSTEM_PROCESSOR i386)
14    set(CMAKE_OSX_ARCHITECTURES "i386")
15
16 add_library(vp8 STATIC ${VP8SRC})
17 elseif(IPHONE)
18    if(SIMULATOR)
19       set(PLATFORM "iPhoneSimulator")
20       set(PROCESSOR i386)
21       set(ARCH "i386")
22    else()
23       set(PLATFORM "iPhoneOS")
24       set(PROCESSOR arm)
25       set(ARCH "armv7")
26 endif()
27
28    set(SDKVER "4.0")
29    set(DEVROOT "/Developer/Platforms/${PLATFORM}.platform/Developer")
30    set(SDKROOT "${DEVROOT}/SDKs/${PLATFORM}${SDKVER}.sdk")
31    set(CMAKE_OSX_SYSROOT "${SDKROOT}")
32    set(CMAKE_SYSTEM_NAME Generic)
33    set(CMAKE_SYSTEM_PROCESSOR ${PROCESSOR})
34    set(CMAKE_CXX_COMPILER "${DEVROOT}/usr/bin/g++")
35    set(CMAKE_OSX_ARCHITECTURES ${ARCH})
36
37 include_directories(SYSTEM "${SDKROOT}/usr/include")
38 link_directories(SYSTEM "${SDKROOT}/usr/lib")
```

```
39
40 add_definitions(-D_PHONE)
41 add_library(vp8-armv7-darwin STATIC ${VP8SRC})
42 endif()
```

你能一眼看出这个配置文件干了什么吗？其实这个配置文件想要产生的目标（target）只有一个，就是通过 ${VP8SRC} 编译生成的静态库，但因为加上了条件判断，及各种平台相关配置，使得这个配置文件看起来很是复杂。在我看来，编写 CMake 配置文件是一种线性思维，对于同一个目标的配置可能会零散分布在各个地方。而 GYP 则相当不同，GYP 的配置文件更多地强调模块化、结构化。看看下面这个示例文件：

```
1  {
2    'targets': [
3      {
4        'target_name': 'foo',
5        'type': '<(library)',
6        'dependencies': [
7          'bar',
8        ],
9        'defines': [
10         'DEFINE_FOO',
11         'DEFINE_A_VALUE=value',
12       ],
13       'include_dirs': [
14         '..',
15       ],
16       'sources': [
17         'file1.cc',
18         'file2.cc',
19       ],
20       'conditions': [
21         ['OS=="linux"', {
22           'defines': [
23             'LINUX_DEFINE',
24           ],
25           'include_dirs': [
26             'include/linux',
27           ],
28         }],
29         ['OS=="win"', {
30           'defines': [
31             'WINDOWS_SPECIFIC_DEFINE',
32           ],
33         }, { # OS != "win",
34           'defines': [
35             'NON_WINDOWS_DEFINE',
36           ],
37         }]
38       ],
39     }
40   ],
41 }
```

我们可以立马看出上面这个配置文件的输出目标只有一个，也就是 foo，它是一个库文件

（至于是静态的还是动态的这需要在生成项目时指定），它依赖的目标、宏定义、包含的头文件路径、源文件是什么，以及根据不同平台设定的不同配置等。这种定义配置文件的方式相比 CMake 来说，让我觉得更加舒服，也更加清晰，特别是当一个输出目标的配置越来越多时，使用 CMake 来管理可能会愈加混乱。
配置文件的编写方式是我区分 GYP 和 CMake 之间最大的不同点，当然 GYP 也有一些小细节值得注意，比如支持跨平台项目工程文件输出，Windows 平台默认是 Visual Studio，Linux 平台默认是 Makefile，Mac 平台默认是 Xcode，这个功能 CMake 也同样支持~~，只是缺少了 Xcode~~。Chromium 团队成员也撰文详细比较了 GYP 和 CMake 之间的优缺点，在开发 GYP 之前，他们也曾试图转到 SCons（这个我没用过，有经验的同学可以比较一下），但是失败了，于是 GYP 就诞生了。
当然 GYP 也不是没有缺点，相反，我觉得它的「缺点」一大堆：
文档不够完整，项目不够正式，某些地方还保留着 Chromium 的影子，看起来像是还没有完全独立出来。
大量的括号嵌套，很容易让人看晕，有过 Lisp 使用经验的同学可以对号入座。对于有括号恐惧症，或者不使用现代编辑器的同学基本可以绕行。
为了支持跨平台，有时不得不加入某些特定平台的配置信息，比如只适用于 Visual Studio 的 RuntimeLibrary 配置，这不利于跨平台配置文件的编写，也无形中增加了编写复杂度。
不支持 make clean，唯一的方法就是将输出目录整个删除或者手动删除其中的某些文件。
如果你已经打算尝试 GYP，那一定记得在生成项目工程文件时加上 --depth 参数，譬如：
$ gyp --depth=. foo.gyp
这也是一个从 Chromium 项目遗留下来的历史问题。

4.2 Google test 程序

玩转 Google 开源 C++单元测试框架 Google Test 系列(gtest)(总)

玩转 Google 开源 C++单元测试框架 Google Test 系列(gtest)之一 - 初识 gtest

一、前言

本篇将介绍一些 gtest 的基本使用，包括下载，安装，编译，建立我们第一个测试 Demo 工程，以及编写一个最简单的测试案例。

二、下载

如果不记得网址，直接在 google 里搜 gtest ,第一个就是。目前 gtest 的最新版本为 1.3.0，从下列地址可以下载到该最新版本：

http://googletest.googlecode.com/files/gtest-1.3.0.zip

http://googletest.googlecode.com/files/gtest-1.3.0.tar.gz

http://googletest.googlecode.com/files/gtest-1.3.0.tar.bz2

三、编译

下载解压后， 里面有个 msvc 目录：

4.3 libjingle 源码分析
http://blog.csdn.net/chenyufei1013/article/category/1248211

libjingle 源码分析之一：Signal 机制

分类： libjingle 2012-10-18 17:18 1133 人阅读 评论(3) 收藏 举报

signalclass 文档 parametersfunctionstring

- 摘要

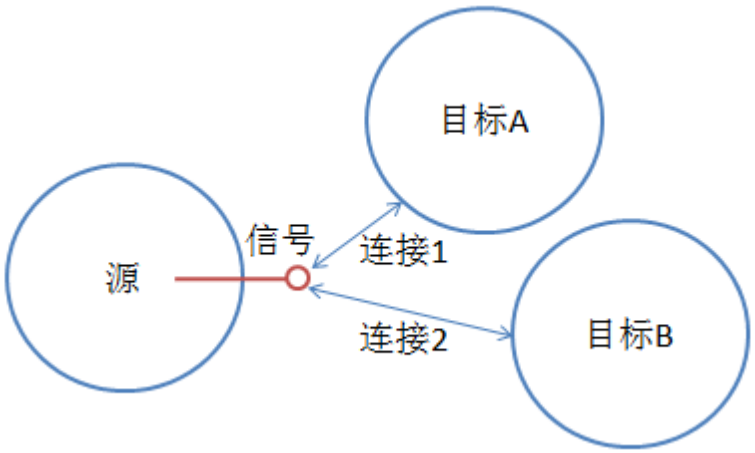
本文主要分析了 libjingle 中的 Signal（信号）机制，它实际上是基于 sigslot 开源库。本文开始描述了 Signal 机制是什么；然后，给出一个 libjingle 文档中的例子，来描述它是如何使用的。最后，介绍了 Signal 机制的具体实现。

- 概述

按照 libjingle 文档关于 Signal（https://developers.google.com/talk/libjingle/important_concepts#signals）的介绍，Signal 机制实际上采用的是 sigslot 开源库（http://sourceforge.net/projects/sigslot/?source=directory）。sigslot 是一个开源的回调框架，它可以使得类之间的回调使用的简单化，下面是 libjingle 文档中对 sigslot 的描述。

sigslot is a generic framework that enables you to connect a calling member to a receiving function in any class (including the same class) very simply.

Signal 机制的工作方式参见下图的描述。源中设置一个或多个信号，目标为了在源的信号触发时获获得通知，需要连接到信号上。可以有多个目标发起连接，也可以同一个目标发起多个连接。连接创建好之后，源触发信号时，目标 A 和目标 B 就可以收到信号触发的消息了。



4.4 Stun 协议:

STUN（Session Traversal Utilities for NAT， NAT 会话传输应用程序）是一种网络协议，它允许位于 NAT（或多重 NAT）后的客户端找出自己的公网地址，查出自己位于哪种类型的 NAT 之后以及 NAT 为某一个本地端口所绑定的 Internet 端端口。这些信息被用来在两个同时处于 NAT 路由器之后的主机之间建立 UDP 通信。该协议由 RFC 5389 定义。

一旦客户端得知了 Internet 端的 UDP 端口，通信就可以开始了。如果 NAT 是完全圆锥型的，那么双方中的任何一方都可以发起通信。如果 NAT 是受限圆锥型或端口受限圆锥型，双方必须一起开始传输。

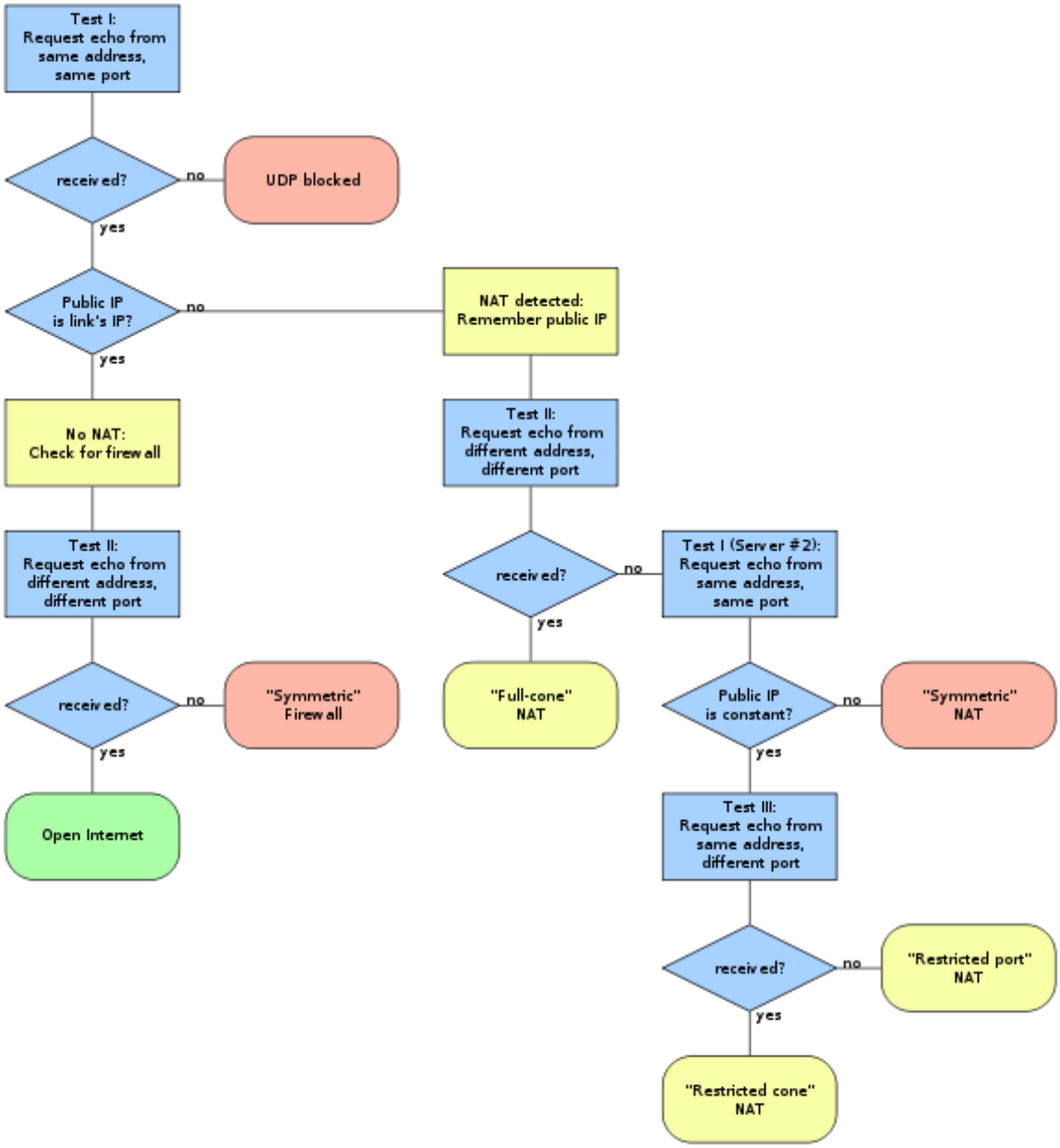
需要注意的是，要使用 STUN RFC 中描述的技术并不一定需要使用 STUN 协议——还可以另外设计一个协议并把相同的功能集成到运行该协议的服务器上。

SIP 之类的协议是使用 UDP 分组在 Internet 上传输音频和 / 或视频数据的。不幸的是，由于通信的两个末端往往位于 NAT 之后，因此用传统的方法是无法建立连接的。这也就是 STUN 发挥作用的地方。

STUN 是一个客户机－服务器协议。一个 VoIP 电话或软件包可能会包括一个 STUN 客户端。这个客户端会向 STUN 服务器发送请求，之后，服务器就会向 STUN 客户端报告 NAT 路由器的公网 IP 地址以及 NAT 为允许传入流量传回内网而开通的端口。

以上的响应同时还使得 STUN 客户端能够确定正在使用的 NAT 类型——因为不同的 NAT 类型处理传入的 UDP 分组的方式是不同的。四种主要类型中有三种是可以使用的：完全圆锥型 NAT、受限圆锥型 NAT 和端口受限圆锥型 NAT——但大型公司网络中经常采用的对称型 NAT（又称为双向 NAT）则不能使用。

STUN 使用下列的算法（取自 RFC 3489）来发现 NAT gateways 以及防火墙（firewalls）：



一旦路径通过红色箱子的终点时，UDP 的连通是沒有可能性的。一旦通过黄色或是绿色的箱子，就有连接的可能。

互動式連接建立（Interactive Connectivity Establishment），一種綜合性的 NAT 穿越的技術。

互動式連接建立是由 IETF 的 MMUSIC 工作組開發出來的一種 framework，可整合各種 NAT 穿透技術，如 STUN、TURN（Traversal Using Relay NAT，中繼 NAT 實現的穿透）、RSIP（Realm Specific IP，特定域 IP）等。該 framework 可以讓 SIP 的客戶端利用各種 NAT 穿透方式打穿遠程的防火牆。

IETF 規格

- Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols RFC 5245

- Session Traversal Utilities for NAT (STUN): RFC 5389
- Traversal Using Relays around NAT (TURN): Relay Extensions to STUN RFC 5766

源码：

PJNATH - Open Source ICE, STUN, and TURN Library

libnice: GLib ICE library