Android 平台 WebRTC 源码简析

1 PeerConnection 创建

音视频互通需要先创建 PeerConnection，然后通过创建及交换 Sdp，以完成通话建立。

1.1 创建 PeerConnectionFactory

```java
// CallActivity.java
CallActivity.onCreate() {
    // Create peer connection client.
    // 创建 peer connection client, 并调用 PeerConnectionFactory.initialize() 加载动态库及上下文环境等.
    peerConnectionClient = new PeerConnectionClient(
        getApplicationContext(), eglBase, peerConnectionParameters, CallActivity.this);

    /* 创建 PeerConnectionFactory */
    PeerConnectionFactory.Options options = new PeerConnectionFactory.Options();
    PeerConnectionClient.createPeerConnectionFactory(options)

    /* 开始通话建立 */
    startCall()
}
```

```java
// PeerConnectionClient.java
public void createPeerConnectionFactory(PeerConnectionFactory.Options options) {
    if (factory != null) {
        throw new IllegalStateException("PeerConnectionFactory has already been constructed");
    }
    executor.execute(() -> createPeerConnectionFactoryInternal(options));
}

private void createPeerConnectionFactoryInternal(PeerConnectionFactory.Options options) {
    /* 创建音频设备模块 */
    final AudioDeviceModule adm = createJavaAudioDevice();

    /* 视频编码是否使能 H264 high level profile */
    final boolean enableH264HighProfile =
        VIDEO_CODEC_H264_HIGH.equals(peerConnectionParameters.videoCodec);
    final VideoEncoderFactory encoderFactory;
    final VideoDecoderFactory decoderFactory;

    /* 创建视频编解码工厂, 软编或硬编 */
    if (peerConnectionParameters.videoCodecHwAcceleration) {
      encoderFactory = new DefaultVideoEncoderFactory(
          rootEglBase.getEglBaseContext(), true /* enableIntelVp8Encoder */, enableH264HighProfile);
      decoderFactory = new DefaultVideoDecoderFactory(rootEglBase.getEglBaseContext());
    } else {
      encoderFactory = new SoftwareVideoEncoderFactory();
```

```
        decoderFactory = new SoftwareVideoDecoderFactory();
    }

    /* 构建 PeerConnectionFactory */
    factory = PeerConnectionFactory.builder()
                  .setOptions(options)
                  .setAudioDeviceModule(adm)
                  .setVideoEncoderFactory(encoderFactory)
                  .setVideoDecoderFactory(decoderFactory)
                  .createPeerConnectionFactory();

    adm.release();
}

// sdk/android/api/org/webrtc/PeerConnectionFactory.java
// PeerConnectionFactory.Builder
public PeerConnectionFactory createPeerConnectionFactory() {
    // 通过 Jni 创建 Native 层 PeerConnectionFactory 对象
    return nativeCreatePeerConnectionFactory(ContextUtils.getApplicationContext(), options,
        /* 获取音频设备 Native C++指针 */
        audioDeviceModule.getNativeAudioDeviceModulePointer(),
        /* 创建音频编码工厂类 Native C++ 指针 */
        audioEncoderFactoryFactory.createNativeAudioEncoderFactory(),
        /* 创建音频解码工厂类 Native C++ 指针 */
        audioDecoderFactoryFactory.createNativeAudioDecoderFactory(),
        /* 视频编码器工厂类对象 */
        videoEncoderFactory,
        /* 视频解码器工厂类对象 */
        videoDecoderFactory,
        /* 音频增强处理，此处为 null */
        audioProcessingFactory == null ? 0 : audioProcessingFactory.createNative(),
        /* fec 控制器，此处为 null */
        fecControllerFactoryFactory == null ? 0 : fecControllerFactoryFactory.createNative(),
        /* 此处为 null */
        networkControllerFactoryFactory == null
            ? 0
            : networkControllerFactoryFactory.createNativeNetworkControllerFactory(),
        /* 此处为 null */
        networkStatePredictorFactoryFactory == null
            ? 0
            : networkStatePredictorFactoryFactory.createNativeNetworkStatePredictorFactory(),
        /* 此处为 null */
        mediaTransportFactoryFactory == null
            ? 0
            : mediaTransportFactoryFactory.createNativeMediaTransportFactory(),
```

```
        /* 此处为 null */
        neteqFactoryFactory == null ? 0 :
neteqFactoryFactory.createNativeNetEqFactory());
}
```

```
//
out/android_arm64_Debug/gen/sdk/android/generated_peerconnection_jni/PeerConnectio
n_factory_jni.h
// 该文件根据 sdk/android/api/org/webrtc/PeerConnectionFactory.java 在编译时动态生成
/* 创建 Native 层 PeerConnectionFactory 对象 */
JNI_GENERATOR_EXPORT jobject
    Java_org_webrtc_PeerConnectionFactory_nativeCreatePeerConnectionFactory(
    JNIEnv* env,
    jclass jcaller,
    jobject context,
    jobject options,
    jlong nativeAudioDeviceModule,
    jlong audioEncoderFactory,
    jlong audioDecoderFactory,
    jobject encoderFactory,
    jobject decoderFactory,
    jlong nativeAudioProcessor,
    jlong nativeFecControllerFactory,
    jlong nativeNetworkControllerFactory,
    jlong nativeNetworkStatePredictorFactory,
    jlong mediaTransportFactory,
    jlong neteqFactory) {
    return JNI_PeerConnectionFactory_CreatePeerConnectionFactory(env,
        base::android::JavaParamRef<jobject>(env, context),
        base::android::JavaParamRef<jobject>(env, options),
        nativeAudioDeviceModule,
        audioEncoderFactory, audioDecoderFactory,
        base::android::JavaParamRef<jobject>(env, encoderFactory),
        base::android::JavaParamRef<jobject>(env, decoderFactory),
        nativeAudioProcessor,
        nativeFecControllerFactory, nativeNetworkControllerFactory,
        nativeNetworkStatePredictorFactory, mediaTransportFactory,
neteqFactory).Release();
}

// sdk/android/src/jni/pc/peer_connection_factory.cc
static ScopedJavaLocalRef<jobject>
JNI_PeerConnectionFactory_CreatePeerConnectionFactory(
    JNIEnv* jni,
    const JavaParamRef<jobject>& jcontext,
    const JavaParamRef<jobject>& joptions,
    jlong native_audio_device_module,
    jlong native_audio_encoder_factory,
    jlong native_audio_decoder_factory,
```

```
    const JavaParamRef<jobject>& jencoder_factory,
    const JavaParamRef<jobject>& jdecoder_factory,
    jlong native_audio_processor,
    jlong native_fec_controller_factory,
    jlong native_network_controller_factory,
    jlong native_network_state_predictor_factory,
    jlong native_media_transport_factory,
    jlong native_neteq_factory) {
  rtc::scoped_refptr<AudioProcessing> audio_processor =
      reinterpret_cast<AudioProcessing*>(native_audio_processor);
  return CreatePeerConnectionFactoryForJava(
      jni, jcontext, joptions,
      // Native 音频设备对象
      reinterpret_cast<AudioDeviceModule*>(native_audio_device_module),
      TakeOwnershipOfRefPtr<AudioEncoderFactory>(native_audio_encoder_factory),
      TakeOwnershipOfRefPtr<AudioDecoderFactory>(native_audio_decoder_factory),
      // 视频编码器及解码器工厂类对象
      jencoder_factory, jdecoder_factory,
      // 创建音频增强处理对象
      audio_processor ? audio_processor : CreateAudioProcessing(),
      TakeOwnershipOfUniquePtr<FecControllerFactoryInterface>(
          native_fec_controller_factory),
      TakeOwnershipOfUniquePtr<NetworkControllerFactoryInterface>(
          native_network_controller_factory),
      TakeOwnershipOfUniquePtr<NetworkStatePredictorFactoryInterface>(
          native_network_state_predictor_factory),
      TakeOwnershipOfUniquePtr<MediaTransportFactory>(
          native_media_transport_factory),
      TakeOwnershipOfUniquePtr<NetEqFactory>(native_neteq_factory));
}

// Following parameters are optional:
// |audio_device_module|, |jencoder_factory|, |jdecoder_factory|,
// |audio_processor|, |media_transport_factory|, |fec_controller_factory|,
// |network_state_predictor_factory|, |neteq_factory|.
ScopedJavaLocalRef<jobject> CreatePeerConnectionFactoryForJava(
    JNIEnv* jni,
    const JavaParamRef<jobject>& jcontext,
    const JavaParamRef<jobject>& joptions,
    rtc::scoped_refptr<AudioDeviceModule> audio_device_module,
    rtc::scoped_refptr<AudioEncoderFactory> audio_encoder_factory,
    rtc::scoped_refptr<AudioDecoderFactory> audio_decoder_factory,
    const JavaParamRef<jobject>& jencoder_factory,
    const JavaParamRef<jobject>& jdecoder_factory,
    rtc::scoped_refptr<AudioProcessing> audio_processor,
    std::unique_ptr<FecControllerFactoryInterface> fec_controller_factory,
    std::unique_ptr<NetworkControllerFactoryInterface>
        network_controller_factory,
    std::unique_ptr<NetworkStatePredictorFactoryInterface>
```

```
        network_state_predictor_factory,
    std::unique_ptr<MediaTransportFactory> media_transport_factory,
    std::unique_ptr<NetEqFactory> neteq_factory) {
    // talk/ assumes pretty widely that the current Thread is ThreadManager'd, but
    // ThreadManager only WrapCurrentThread()s the thread where it is first
    // created.  Since the semantics around when auto-wrapping happens in
    // webrtc/rtc_base/ are convoluted, we simply wrap here to avoid having to
    // think about ramifications of auto-wrapping there.
    rtc::ThreadManager::Instance()->WrapCurrentThread();

    // 创建 network 线程
    std::unique_ptr<rtc::Thread> network_thread =
        rtc::Thread::CreateWithSocketServer();
    network_thread->SetName("network_thread", nullptr);
    RTC_CHECK(network_thread->Start()) << "Failed to start thread";

    // 创建 worker 线程
    std::unique_ptr<rtc::Thread> worker_thread = rtc::Thread::Create();
    worker_thread->SetName("worker_thread", nullptr);
    RTC_CHECK(worker_thread->Start()) << "Failed to start thread";

    // 创建 signaling 线程
    std::unique_ptr<rtc::Thread> signaling_thread = rtc::Thread::Create();
    signaling_thread->SetName("signaling_thread", NULL);
    RTC_CHECK(signaling_thread->Start()) << "Failed to start thread";

    rtc::NetworkMonitorFactory* network_monitor_factory = nullptr;

    const absl::optional<PeerConnectionFactoryInterface::Options> options =
        JavaToNativePeerConnectionFactoryOptions(jni, joptions);

    // Do not create network_monitor_factory only if the options are
    // provided and disable_network_monitor therein is set to true.
    if (!(options && options->disable_network_monitor)) {
        network_monitor_factory = new AndroidNetworkMonitorFactory();
        rtc::NetworkMonitorFactory::SetFactory(network_monitor_factory);
    }

    PeerConnectionFactoryDependencies dependencies;
    dependencies.network_thread = network_thread.get();
    dependencies.worker_thread = worker_thread.get();
    dependencies.signaling_thread = signaling_thread.get();
    dependencies.task_queue_factory = CreateDefaultTaskQueueFactory();
    dependencies.call_factory = CreateCallFactory();
    dependencies.event_log_factory = std::make_unique<RtcEventLogFactory>(
        dependencies.task_queue_factory.get());
    dependencies.fec_controller_factory = std::move(fec_controller_factory);
    dependencies.network_controller_factory =
        std::move(network_controller_factory);
```

```
    dependencies.network_state_predictor_factory =
        std::move(network_state_predictor_factory);
    dependencies.media_transport_factory = std::move(media_transport_factory);
    dependencies.neteq_factory = std::move(neteq_factory);

    cricket::MediaEngineDependencies media_dependencies;
    media_dependencies.task_queue_factory = dependencies.task_queue_factory.get();
    // 外部创建音频设备
    media_dependencies.adm = std::move(audio_device_module);
    media_dependencies.audio_encoder_factory = std::move(audio_encoder_factory);
    media_dependencies.audio_decoder_factory = std::move(audio_decoder_factory);
    media_dependencies.audio_processing = std::move(audio_processor);
    media_dependencies.video_encoder_factory =
        absl::WrapUnique(CreateVideoEncoderFactory(jni, jencoder_factory));
    media_dependencies.video_decoder_factory =
        absl::WrapUnique(CreateVideoDecoderFactory(jni, jdecoder_factory));
    // 创建 MediaEngine
    dependencies.media_engine =
        cricket::CreateMediaEngine(std::move(media_dependencies));

    // 创建 PeerConnectionFactory, 此调用与其他平台相同
    rtc::scoped_refptr<PeerConnectionFactoryInterface> factory =
        CreateModularPeerConnectionFactory(std::move(dependencies));

    RTC_CHECK(factory) << "Failed to create the peer connection factory; "
                          "WebRTC/libjingle init likely failed on this device";
    // TODO(honghaiz): Maybe put the options as the argument of
    // CreatePeerConnectionFactory.
    if (options)
        factory->SetOptions(*options);

    return NativeToScopedJavaPeerConnectionFactory(
        jni, factory, std::move(network_thread), std::move(worker_thread),
        std::move(signaling_thread), network_monitor_factory);
}

ScopedJavaLocalRef<jobject> NativeToScopedJavaPeerConnectionFactory(
    JNIEnv* env,
    rtc::scoped_refptr<webrtc::PeerConnectionFactoryInterface> pcf,
    std::unique_ptr<rtc::Thread> network_thread,
    std::unique_ptr<rtc::Thread> worker_thread,
    std::unique_ptr<rtc::Thread> signaling_thread,
    rtc::NetworkMonitorFactory* network_monitor_factory) {
    // OwnedFactoryAndThreads 保存维护 Native Factory 对象
    OwnedFactoryAndThreads* owned_factory = new OwnedFactoryAndThreads(
        std::move(network_thread), std::move(worker_thread),
        std::move(signaling_thread), network_monitor_factory, pcf);

    // 创建 Java 层的 PeerConnectionFactory 实例
```

```
    ScopedJavaLocalRef<jobject> j_pcf = Java_PeerConnectionFactory_Constructor(
        env, NativeToJavaPointer(owned_factory));

    // 回调 PeerConnectionFactory.onNetworkThreadReady()
    PostJavaCallback(env, owned_factory->network_thread(), RTC_FROM_HERE, j_pcf,
                     &Java_PeerConnectionFactory_onNetworkThreadReady);
    // 回调 PeerConnectionFactory.onWorkerThreadReady()
    PostJavaCallback(env, owned_factory->worker_thread(), RTC_FROM_HERE, j_pcf,
                     &Java_PeerConnectionFactory_onWorkerThreadReady);
    // 回调 PeerConnectionFactory.onSignalingThreadReady()
    PostJavaCallback(env, owned_factory->signaling_thread(), RTC_FROM_HERE, j_pcf,
                     &Java_PeerConnectionFactory_onSignalingThreadReady);

    return j_pcf;
}

static base::android::ScopedJavaLocalRef<jobject> Java_PeerConnectionFactory_Constructor(JNIEnv*
    env, jlong nativeFactory) {
    jclass clazz = org_webrtc_PeerConnectionFactory_clazz(env);
    CHECK_CLAZZ(env, clazz,
        org_webrtc_PeerConnectionFactory_clazz(env), NULL);

    jni_generator::JniJavaCallContextChecked call_context;
    call_context.Init<
        base::android::MethodID::TYPE_INSTANCE>(
            env,
            clazz,
            "<init>",
            "(J)V",
            &g_org_webrtc_PeerConnectionFactory_Constructor);

    // 构造 Java 层 PeerConnectionFactory 实例
    jobject ret =
        env->NewObject(clazz,
            call_context.base.method_id, nativeFactory);
    return base::android::ScopedJavaLocalRef<jobject>(env, ret);
}
```

1.2 创建 PeerConnection

1. 连接上房间服务器；
2. 若使能视频，则先创建视频采集源；
3. 创建 PeerConnection；

```
// CallActivity.java
/* 信令层连接房间服务器成功, 创建 PeerConnection, 若是主叫端则创建 offer sdp, 若是被叫端则需设置远端 Sdp 及创建本端 Sdp */
private void onConnectedToRoomInternal(final SignalingParameters params) {
```

```java
    final long delta = System.currentTimeMillis() - callStartedTimeMs;

    signalingParameters = params;
    logAndToast("Creating peer connection, delay=" + delta + "ms");
    /* 创建视频采集实例，可为：视频文件、屏幕共享、摄像头采集 */
    VideoCapturer videoCapturer = null;
    if (peerConnectionParameters.videoCallEnabled) {
        videoCapturer = createVideoCapturer();
    }
    /* 创建 PeerConnection */
    peerConnectionClient.createPeerConnection(
        localProxyVideoSink, remoteSinks, videoCapturer, signalingParameters);

    if (signalingParameters.initiator) {
        logAndToast("Creating OFFER...");
        // Create offer. Offer SDP will be sent to answering client in
        // PeerConnectionEvents.onLocalDescription event.
        peerConnectionClient.createOffer();
    } else {
        if (params.offerSdp != null) {
            peerConnectionClient.setRemoteDescription(params.offerSdp);
            logAndToast("Creating ANSWER...");
            // Create answer. Answer SDP will be sent to offering client in
            // PeerConnectionEvents.onLocalDescription event.
            peerConnectionClient.createAnswer();
        }
        if (params.iceCandidates != null) {
            // Add remote ICE candidates from room.
            for (IceCandidate iceCandidate : params.iceCandidates) {
                peerConnectionClient.addRemoteIceCandidate(iceCandidate);
            }
        }
    }
}
```

```java
// PeerConnectionClient.java
public void createPeerConnection(final VideoSink localRender, final VideoSink remoteSink,
      final VideoCapturer videoCapturer, final SignalingParameters signalingParameters) {
    if (peerConnectionParameters.videoCallEnabled && videoCapturer == null) {
        Log.w(TAG, "Video call enabled but no video capturer provided.");
    }
    createPeerConnection(
        localRender, Collections.singletonList(remoteSink), videoCapturer, signalingParameters);
}

```

```
public void createPeerConnection(final VideoSink localRender, final List<VideoSink> remoteSinks,
      final VideoCapturer videoCapturer, final SignalingParameters signalingParameters) {
    if (peerConnectionParameters == null) {
        Log.e(TAG, "Creating peer connection without initializing factory.");
        return;
    }
    // 本地预览的 sink
    this.localRender = localRender;
    // 远端渲染的 sink
    this.remoteSinks = remoteSinks;
    // 视频源实例
    this.videoCapturer = videoCapturer;
    this.signalingParameters = signalingParameters;
    executor.execute(() -> {
      try {
            createMediaConstraintsInternal();
            createPeerConnectionInternal();
            maybeCreateAndStartRtcEventLog();
      } catch (Exception e) {
            reportError("Failed to create peer connection: " + e.getMessage());
            throw e;
      }
    });
}

private void createPeerConnectionInternal() {
    if (factory == null || isError) {
      Log.e(TAG, "Peerconnection factory is not created");
      return;
    }
    Log.d(TAG, "Create peer connection.");

    queuedRemoteCandidates = new ArrayList<>();

    PeerConnection.RTCConfiguration rtcConfig =
        new PeerConnection.RTCConfiguration(signalingParameters.iceServers);
    // TCP candidates are only useful when connecting to a server that supports
    // ICE-TCP.
    rtcConfig.tcpCandidatePolicy = PeerConnection.TcpCandidatePolicy.DISABLED;
    rtcConfig.bundlePolicy = PeerConnection.BundlePolicy.MAXBUNDLE;
    rtcConfig.rtcpMuxPolicy = PeerConnection.RtcpMuxPolicy.REQUIRE;
    rtcConfig.continualGatheringPolicy = PeerConnection.ContinualGatheringPolicy.GATHER_CONTINUALLY;
    // Use ECDSA encryption.
    rtcConfig.keyType = PeerConnection.KeyType.ECDSA;
    // Enable DTLS for normal calls and disable for loopback calls.
    rtcConfig.enableDtlsSrtp = !peerConnectionParameters.loopback;
```

```
    /* Sdp 语法使用 Unified Plan 模式 */
    rtcConfig.sdpSemantics = PeerConnection.SdpSemantics.UNIFIED_PLAN;

    /* 创建 PeerConnection */
    peerConnection = factory.createPeerConnection(rtcConfig, pcObserver);

    if (dataChannelEnabled) {
        /* create data channel */
        ...
    }
    isInitiator = false;

    // Set INFO libjingle logging.
    // NOTE: this _must_ happen while |factory| is alive!
    Logging.enableLogToDebugOutput(Logging.Severity.LS_INFO);

    List<String> mediaStreamLabels = Collections.singletonList("ARDAMS");
    if (isVideoCallEnabled()) {
        /* 创建并添加视频源到 PeerConnection, 创建 VideoTrack 时将启动视频源采集 */
        peerConnection.addTrack(createVideoTrack(videoCapturer),
mediaStreamLabels);

        // We can add the renderers right away because we don't need to wait for an
        // answer to get the remote track.
        remoteVideoTrack = getRemoteVideoTrack();
        remoteVideoTrack.setEnabled(renderVideo);
        for (VideoSink remoteSink : remoteSinks) {
            remoteVideoTrack.addSink(remoteSink);
        }
    }

    // 创建并添加音频源 AudioTrack 到 PeerConnection
    peerConnection.addTrack(createAudioTrack(), mediaStreamLabels);
    if (isVideoCallEnabled()) {
        findVideoSender();
    }

    ...

    Log.d(TAG, "Peer connection created.");
}
```

```
// sdk/android/api/org/webrtc/PeerConnectionFactory.java
public PeerConnection createPeerConnection(
      PeerConnection.RTCConfiguration rtcConfig, PeerConnection.Observer observer)
{
    return createPeerConnection(rtcConfig, null /* constraints */, observer);
}

```

```
PeerConnection createPeerConnectionInternal(PeerConnection.RTCConfiguration
rtcConfig,
      MediaConstraints constraints/* null */, PeerConnection.Observer observer,
      SSLCertificateVerifier sslCertificateVerifier/* null */) {
    checkPeerConnectionFactoryExists();
    // 创建 PeerConnection.Observer 的 Native 层关联对象
    long nativeObserver =
PeerConnection.createNativePeerConnectionObserver(observer);
    if (nativeObserver == 0) {
        return null;
    }
    long nativePeerConnection = nativeCreatePeerConnection(
        nativeFactory, rtcConfig, constraints, nativeObserver,
sslCertificateVerifier);
    if (nativePeerConnection == 0) {
        return null;
    }
    return new PeerConnection(nativePeerConnection);
}

// sdk/android/api/org/webrtc/PeerConnection.java
PeerConnection(long nativePeerConnection) {
    this.nativePeerConnection = nativePeerConnection;
}
```

绑定 Java 层 PeerConnection Observer 对象，后续通过该 Observer 将 Native 层事件回调到 Java 层。其中，PeerConnectionObserverJni 为Android端 PeerConnectionObserver 的实现类。

```
//
out/android_arm64_Debug/gen/sdk/android/generated_peerconnection_jni/PeerConnection
_jni.h
// 该文件根据 PeerConnection.java 生成
JNI_GENERATOR_EXPORT jlong
Java_org_webrtc_PeerConnection_nativeCreatePeerConnectionObserver(
    JNIEnv* env,
    jclass jcaller,
    jobject observer) {
  return JNI_PeerConnection_CreatePeerConnectionObserver(env,
      base::android::JavaParamRef<jobject>(env, observer));
}

// sdk/android/src/jni/pc/peer_connection.cc
static jlong JNI_PeerConnection_CreatePeerConnectionObserver(
    JNIEnv* jni,
    const JavaParamRef<jobject>& j_observer) {
  return jlongFromPointer(new PeerConnectionObserverJni(jni, j_observer));
}

PeerConnectionObserverJni::PeerConnectionObserverJni(
```

```
    JNIEnv* jni,
    const JavaRef<jobject>& j_observer)
    // 创建 Observer Object 的 global 引用
    : j_observer_global_(jni, j_observer) {}
```

创建 Native 层 PeerConnection 对象

```
// out/android_arm64_Debug/gen/sdk/android/generated_peerconnection_jni/PeerConnection_factory_jni.h
// 该文件根据 sdk/android/api/org/webrtc/PeerConnectionFactory.java 在编译时动态生成
/* 创建 Native 层 PeerConnection 对象 */
JNI_GENERATOR_EXPORT jlong Java_org_webrtc_PeerConnectionFactory_nativeCreatePeerConnection(
    JNIEnv* env,
    jclass jcaller,
    jlong factory,
    jobject rtcConfig,
    jobject constraints, // null
    jlong nativeObserver,
    jobject sslCertificateVerifier) {
  return JNI_PeerConnectionFactory_CreatePeerConnection(env, factory,
      base::android::JavaParamRef<jobject>(env, rtcConfig),
      base::android::JavaParamRef<jobject>(env, constraints), nativeObserver,
      base::android::JavaParamRef<jobject>(env, sslCertificateVerifier));
}

// sdk/android/src/jni/pc/peer_connection_factory.cc
static jlong JNI_PeerConnectionFactory_CreatePeerConnection(
    JNIEnv* jni,
    jlong factory,
    const JavaParamRef<jobject>& j_rtc_config,
    const JavaParamRef<jobject>& j_constraints, /* null */
    jlong observer_p,
    const JavaParamRef<jobject>& j_sslCertificateVerifier /* null */) {
    /* PeerConnectionObserver */
    std::unique_ptr<PeerConnectionObserver> observer(
        reinterpret_cast<PeerConnectionObserver*>(observer_p));

    // RTCConfiguration 拷贝
    ...

    // rtc::RTCCertificate 生成
    ...

    // MediaConstraints 拷贝
    ...

    PeerConnectionDependencies peer_connection_dependencies(observer.get());
```

```

    // SSLCertificateVerifierWrapper 创建
    ...

    /* 创建 PeerConnection，该接口调用与各平台一致 */
    rtc::scoped_refptr<PeerConnectionInterface> pc =
        PeerConnectionFactoryFromJava(factory)->CreatePeerConnection(
            rtc_config, std::move(peer_connection_dependencies));
    if (!pc)
        return 0;

    /* 返回 PeerConnection 封装对象 */
    return jlongFromPointer(
        new OwnedPeerConnection(pc, std::move(observer), std::move(constraints)));
}

// pc/peer_connection_factory.cc
rtc::scoped_refptr<PeerConnectionInterface>
PeerConnectionFactory::CreatePeerConnection(
    const PeerConnectionInterface::RTCConfiguration& configuration,
    PeerConnectionDependencies dependencies) {

    ...
    // 创建 PeerConnection
    rtc::scoped_refptr<PeerConnection> pc(
        new rtc::RefCountedObject<PeerConnection>(this, std::move(event_log),
                                                  std::move(call)));
    ActionsBeforeInitializeForTesting(pc);
    // PeerConnection 初始化
    if (!pc->Initialize(configuration, std::move(dependencies))) {
        return nullptr;
    }
    return PeerConnectionProxy::Create(signaling_thread(), pc);
}
```

1.3 创建 VideoTrack

1. 创建视频渲染处理 SurfaceTextureHelper；
2. 创建 VideoSource;
3. 初始化视频采集源并启动，设置视频数据回调监听为 VideoSource 内实现的 CapturerObserver；
4. 创建 VideoTrack，并作为 VideoSink 注册到 VideoSource 内；
5. VideoTrack 使能视频渲染；
6. VideoTrack 添加本地预览Sink。

```
// PeerConnectionClient.java
private VideoTrack createVideoTrack(VideoCapturer capturer) {
    // 视频源纹理数据渲染实例，经过该实例纹理处理后的数据通过 CapturerObserver 回调出来用于编码或预览(Camera2)

```

```
    surfaceTextureHelper =
        SurfaceTextureHelper.create("CaptureThread",
rootEglBase.getEglBaseContext());

    // 创建 VideoSource
    videoSource = factory.createVideoSource(capturer.isScreencast());

    // 初始化及启动视频采集，摄像头的话，调用 CameraCapturer.initialize() 接口初始化，
CapturerObserver 为 VideoSource 创建
    capturer.initialize(surfaceTextureHelper, appContext,
videoSource.getCapturerObserver());

    capturer.startCapture(videoWidth, videoHeight, videoFps);

    // 创建 VideoTrack
    localVideoTrack = factory.createVideoTrack(VIDEO_TRACK_ID, videoSource);
    // 使能视频
    localVideoTrack.setEnabled(renderVideo);
    // 设置预览 Sink
    localVideoTrack.addSink(localRender);
    return localVideoTrack;
}

// sdk/android/api/org/webrtc/SurfaceTextureHelper.java
public static SurfaceTextureHelper create(
      final String threadName, final EglBase.Context sharedContext) {
    return create(threadName, sharedContext, /* alignTimestamps= */ false, new
YuvConverter(),
        /*frameRefMonitor=*/null);
}

private SurfaceTextureHelper(Context sharedContext, Handler handler, boolean
alignTimestamps,
      YuvConverter yuvConverter, FrameRefMonitor frameRefMonitor) {
    if (handler.getLooper().getThread() != Thread.currentThread()) {
        throw new IllegalStateException("SurfaceTextureHelper must be created on
the handler thread");
    }
    this.handler = handler;
    this.timestampAligner = alignTimestamps ? new TimestampAligner() : null;
    this.yuvConverter = yuvConverter;
    this.frameRefMonitor = frameRefMonitor;

    eglBase = EglBase.create(sharedContext, EglBase.CONFIG_PIXEL_BUFFER);
    try {
        // Both these statements have been observed to fail on rare occasions, see
BUG=webrtc:5682.
        eglBase.createDummyPbufferSurface();
        eglBase.makeCurrent();
```

```java
    } catch (RuntimeException e) {
        // Clean up before rethrowing the exception.
        eglBase.release();
        handler.getLooper().quit();
        throw e;
    }

    oesTextureId = GlUtil.generateTexture(GLES11Ext.GL_TEXTURE_EXTERNAL_OES);
    surfaceTexture = new SurfaceTexture(oesTextureId);
    setOnFrameAvailableListener(surfaceTexture, (SurfaceTexture st) -> {
        hasPendingTexture = true;
        tryDeliverTextureFrame();
    }, handler);
}
```

1.4 添加 VideoTrack

添加视频 VideoTrack 到 PeerConnection。
在此流程，会创建RtpSender 及 RtpReceiver 和收发器 RtpTransceiver。

简要代码流程：
peerConnection.addTrack(createVideoTrack(videoCapturer), mediaStreamLabels);

```java
// sdk/android/api/org/webrtc/PeerConnection.java
public RtpSender addTrack(MediaStreamTrack track) {
    return addTrack(track, Collections.emptyList());
}

public RtpSender addTrack(MediaStreamTrack track, List<String> streamIds) {
    if (track == null || streamIds == null) {
        throw new NullPointerException("No MediaStreamTrack specified in addTrack.");
    }

    // JNI 调用
    RtpSender newSender = nativeAddTrack(track.getNativeMediaStreamTrack(), streamIds);
    if (newSender == null) {
        throw new IllegalStateException("C++ addTrack failed.");
    }
    senders.add(newSender);
    return newSender;
}

// sdk/android/api/org/webrtc/RtpSender.java
// 从 C++ 层构造
@CalledByNative
public RtpSender(long nativeRtpSender) {
    this.nativeRtpSender = nativeRtpSender;
```

```
    long nativeTrack = nativeGetTrack(nativeRtpSender);
    // Java 层最终保存 VideoTrack 或 AudioTrack 的地方
    cachedTrack = MediaStreamTrack.createMediaStreamTrack(nativeTrack);

    long nativeDtmfSender = nativeGetDtmfSender(nativeRtpSender);
    dtmfSender = (nativeDtmfSender != 0) ? new DtmfSender(nativeDtmfSender) : null;
}
```

C++ 层调用

```
// out/android_arm64_Debug/gen/sdk/android/generated_peerconnection_jni/PeerConnection_jni.h
JNI_GENERATOR_EXPORT jobject Java_org_webrtc_PeerConnection_nativeAddTrack(
    JNIEnv* env,
    jobject jcaller,
    jlong track,
    jobject streamIds) {
    return JNI_PeerConnection_AddTrack(env, base::android::JavaParamRef<jobject>(env, jcaller), track,
        base::android::JavaParamRef<jobject>(env, streamIds)).Release();
}

// sdk/android/src/jni/pc/peer_connection.cc
static ScopedJavaLocalRef<jobject> JNI_PeerConnection_AddTrack(
    JNIEnv* jni,
    const JavaParamRef<jobject>& j_pc,
    const jlong native_track,
    const JavaParamRef<jobject>& j_stream_labels) {
    // 在此转入平台统一调用接口
    RTCErrorOr<rtc::scoped_refptr<RtpSenderInterface>> result =
        ExtractNativePC(jni, j_pc)->AddTrack(
            reinterpret_cast<MediaStreamTrackInterface*>(native_track),
            JavaListToNativeVector<std::string, jstring>(jni, j_stream_labels,
                                                         &JavaToNativeString));
    if (!result.ok()) {
        RTC_LOG(LS_ERROR) << "Failed to add track: " << result.error().message();
        return nullptr;
    } else {
        // 创建 Java 层 RtpSender
        return NativeToJavaRtpSender(jni, result.MoveValue());
    }
}

// pc/peer_connection.cc
RTCErrorOr<rtc::scoped_refptr<RtpSenderInterface>> PeerConnection::AddTrack(
    rtc::scoped_refptr<MediaStreamTrackInterface> track,
    const std::vector<std::string>& stream_ids) {

```

```
    ...

    /* 如果是 UnifiedPlan，会创建收发器 Transceiver，包括发送和接收。
     * 这样可以不用管是否已创建远端流，提前将远端渲染的sink 添加到收发器里面的接收 Track
（Android 是这样做的） */
    auto sender_or_error =
        (IsUnifiedPlan() ? AddTrackUnifiedPlan(track, stream_ids)
                         : AddTrackPlanB(track, stream_ids));
    if (sender_or_error.ok()) {
        UpdateNegotiationNeeded();
        stats_->AddTrack(track);
    }
    return sender_or_error;
}

/* Unified Plan 模式创建 RtpSender，若无收发器则创建收发器 Transeiver及 VideoRtpReceiver
*/
RTCErrorOr<rtc::scoped_refptr<RtpSenderInterface>>
PeerConnection::AddTrackUnifiedPlan(
    rtc::scoped_refptr<MediaStreamTrackInterface> track,
    const std::vector<std::string>& stream_ids) {
  /* 查找未设置 Track 且收发器 MediaType 与 track 相同，并且不处于发送模式
(sendonly,sendrecv)及未stop 的收发器 */
  auto transceiver = FindFirstTransceiverForAddedTrack(track);
  if (transceiver) { // 已存在收发器
    ...

  } else {
    /* 不存在收发器 */
    cricket::MediaType media_type =
        (track->kind() == MediaStreamTrackInterface::kAudioKind
             ? cricket::MEDIA_TYPE_AUDIO
             : cricket::MEDIA_TYPE_VIDEO);
    RTC_LOG(LS_INFO) << "Adding " << cricket::MediaTypeToString(media_type)
                     << " transceiver in response to a call to AddTrack.";
    std::string sender_id = track->id();
    // Avoid creating a sender with an existing ID by generating a random ID.
    // This can happen if this is the second time AddTrack has created a sender
    // for this track.
    if (FindSenderById(sender_id)) {
      sender_id = rtc::CreateRandomUuid();
    }
    /* 创建 RtpSender, 音频为 AudioRtpSender，视频为 VideoRtpSender，设置 Track，在调用
RtpSender::SetSend() 时会将编码的Sink(视频为VideoStreamEncoder) 添加到 track 内，最终也
是添加到对应的 VideoSource/AudioSource 里面 */
    auto sender = CreateSender(media_type, sender_id, track, stream_ids, {});
    /* 创建 RtpReceiver，音频为 AudioRtpReceiver，视频为 VideoRtpReceiver,
receiver_id 为随机产生 */
    auto receiver = CreateReceiver(media_type, rtc::CreateRandomUuid());
```

```
    /* 创建收发器 Transceiver */
    transceiver = CreateAndAddTransceiver(sender, receiver);
    /* 标记收发器创建原因 */
    transceiver->internal()->set_created_by_addtrack(true);
    /* 设置收发器收发模式 */
    transceiver->internal()->set_direction(RtpTransceiverDirection::kSendRecv);
  }
  return transceiver->sender();
}

/* 创建 RtpSender */
rtc::scoped_refptr<RtpSenderProxyWithInternal<RtpSenderInternal>>
PeerConnection::CreateSender(
    cricket::MediaType media_type,
    const std::string& id,
    rtc::scoped_refptr<MediaStreamTrackInterface> track,
    const std::vector<std::string>& stream_ids,
    const std::vector<RtpEncodingParameters>& send_encodings) {
  RTC_DCHECK_RUN_ON(signaling_thread());
  rtc::scoped_refptr<RtpSenderProxyWithInternal<RtpSenderInternal>> sender;
  if (media_type == cricket::MEDIA_TYPE_AUDIO) {
    /* 创建音频 AudioRtpSender */
    sender = RtpSenderProxyWithInternal<RtpSenderInternal>::Create(
        signaling_thread(),
        AudioRtpSender::Create(worker_thread(), id, stats_.get(), this));
    NoteUsageEvent(UsageEvent::AUDIO_ADDED);
  } else {
    /* 创建视频 VideoRtpSender */
    sender = RtpSenderProxyWithInternal<RtpSenderInternal>::Create(
        signaling_thread(), VideoRtpSender::Create(worker_thread(), id, this));
    NoteUsageEvent(UsageEvent::VIDEO_ADDED);
  }
  /* 设置 VideoTrack 或 AudioTrack，在此暂时不会将 Encoder Sink 添加到 track，因为还不存在 ssrc */
  bool set_track_succeeded = sender->SetTrack(track);

  ...

  return sender;
}

/* 创建 RtpReceiver */
rtc::scoped_refptr<RtpReceiverProxyWithInternal<RtpReceiverInternal>>
PeerConnection::CreateReceiver(cricket::MediaType media_type,
                               const std::string& receiver_id) {
  rtc::scoped_refptr<RtpReceiverProxyWithInternal<RtpReceiverInternal>>
      receiver;
  if (media_type == cricket::MEDIA_TYPE_AUDIO) {
    /* 创建音频 AudioRtpReceiver */
```

```
    receiver = RtpReceiverProxyWithInternal<RtpReceiverInternal>::Create(
        signaling_thread(), new AudioRtpReceiver(worker_thread(), receiver_id,
                                                  std::vector<std::string>({})));
    NoteUsageEvent(UsageEvent::AUDIO_ADDED);
  } else {
    /* 创建音频 VideoRtpReceiver */
    receiver = RtpReceiverProxyWithInternal<RtpReceiverInternal>::Create(
        signaling_thread(), new VideoRtpReceiver(worker_thread(), receiver_id,
                                                  std::vector<std::string>({})));
    NoteUsageEvent(UsageEvent::VIDEO_ADDED);
  }
  return receiver;
}

/* 创建收发器 */
rtc::scoped_refptr<RtpTransceiverProxyWithInternal<RtpTransceiver>>
PeerConnection::CreateAndAddTransceiver(
    rtc::scoped_refptr<RtpSenderProxyWithInternal<RtpSenderInternal>> sender,
    rtc::scoped_refptr<RtpReceiverProxyWithInternal<RtpReceiverInternal>>
        receiver) {
  // Ensure that the new sender does not have an ID that is already in use by
  // another sender.
  // Allow receiver IDs to conflict since those come from remote SDP (which
  // could be invalid, but should not cause a crash).
  RTC_DCHECK(!FindSenderById(sender->id()));
  auto transceiver = RtpTransceiverProxyWithInternal<RtpTransceiver>::Create(
      signaling_thread(),
      new RtpTransceiver(
          sender, receiver, channel_manager(),
          sender->media_type() == cricket::MEDIA_TYPE_AUDIO
              ? channel_manager()->GetSupportedAudioRtpHeaderExtensions()
              : channel_manager()->GetSupportedVideoRtpHeaderExtensions()));
  /* 保存收发器 */
  transceivers_.push_back(transceiver);
  /* 信号连接 */
  transceiver->internal()->SignalNegotiationNeeded.connect(
      this, &PeerConnection::OnNegotiationNeeded);
  return transceiver;
}

// sdk/android/src/jni/pc/rtp_sender.cc
// Native 层创建 Java 层 RtpSender 实例
ScopedJavaLocalRef<jobject> NativeToJavaRtpSender(
    JNIEnv* env,
    rtc::scoped_refptr<RtpSenderInterface> sender) {
    if (!sender)
        return nullptr;
    // Sender is now owned by the Java object, and will be freed from
    // RtpSender.dispose(), called by PeerConnection.dispose() or getSenders().
```

```
    return Java_RtpSender_Constructor(env, jlongFromPointer(sender.release()));
}

// out/android_arm64_Debug/gen/sdk/android/generated_peerconnection_jni/RtpSender_jni.h
static base::android::ScopedJavaLocalRef<jobject> Java_RtpSender_Constructor(JNIEnv* env, jlong
    nativeRtpSender) {
    jclass clazz = org_webrtc_RtpSender_clazz(env);
    CHECK_CLAZZ(env, clazz,
        org_webrtc_RtpSender_clazz(env), NULL);

    jni_generator::JniJavaCallContextChecked call_context;
    call_context.Init<
        base::android::MethodID::TYPE_INSTANCE>(
            env,
            clazz,
            "<init>",
            "(J)V",
            &g_org_webrtc_RtpSender_Constructor);

    jobject ret =
        env->NewObject(clazz,
            call_context.base.method_id, nativeRtpSender);
    return base::android::ScopedJavaLocalRef<jobject>(env, ret);
}
```

2 音频设备模块(AudioDeviceModule)

外部音频设备仅支持 java 层级，不支持 OpenSLES. 借由 JavaAudioDeviceModule 创建音频采集和音频播放实例，并通过 Jni 创建 native C++ 层 AudioDeviceModule 对象。

2.1 创建 JavaAudioDeviceModule

```
// PeerConnectionClient.java
AudioDeviceModule createJavaAudioDevice() {
    /* 采样频率在 Builder 构造时从 WebRtcAudioManager 获取设置 */
    return JavaAudioDeviceModule.builder(appContext)
        /* 音频采集数据回调，用于保存到文件 */
        .setSamplesReadyCallback(saveRecordedAudioToFile)
        /* 是否开启系统回声消除，需设备支持 */
        .setUseHardwareAcousticEchoCanceler(!peerConnectionParameters.disableBuiltInAEC)
        /* 是否开启系统噪声抑制，需设备支持 */
        .setUseHardwareNoiseSuppressor(!peerConnectionParameters.disableBuiltInNS)
        /* 设置错误回调及状态回调 */
        .setAudioRecordErrorCallback(audioRecordErrorCallback)
```

```java
            .setAudioTrackErrorCallback(audioTrackErrorCallback)
            .setAudioRecordStateCallback(audioRecordStateCallback)
            .setAudioTrackStateCallback(audioTrackStateCallback)
            /* 构建音频设备模块 */
            .createAudioDeviceModule();
}

// sdk/android/api/org/webrtc/JavaAudioDeviceModule.java
// JavaAudioDeviceModule.Builder
public AudioDeviceModule createAudioDeviceModule() {
    /* 创建音频采集实例 sdk/android/src/java/org/webrtc/audio/WebRtcAudioRecord.java */
    final WebRtcAudioRecord audioInput = new WebRtcAudioRecord(context, audioManager, audioSource,
        audioFormat, audioRecordErrorCallback, audioRecordStateCallback, samplesReadyCallback,
        useHardwareAcousticEchoCanceler, useHardwareNoiseSuppressor);

    /* 创建音频播放实例 sdk/android/src/java/org/webrtc/audio/WebRtcAudioTrack.java */
    final WebRtcAudioTrack audioOutput = new WebRtcAudioTrack(
        context, audioManager, audioTrackErrorCallback, audioTrackStateCallback);

    /* 创建音频设备模块 */
    return new JavaAudioDeviceModule(context, audioManager, audioInput, audioOutput,
        inputSampleRate, outputSampleRate, useStereoInput, useStereoOutput);
}

// JavaAudioDeviceModule 是 AudioDeviceModule 接口的派生类
private JavaAudioDeviceModule(Context context, AudioManager audioManager,
      WebRtcAudioRecord audioInput, WebRtcAudioTrack audioOutput, int inputSampleRate,
      int outputSampleRate, boolean useStereoInput, boolean useStereoOutput) {
    this.context = context;
    this.audioManager = audioManager;
    this.audioInput = audioInput;
    this.audioOutput = audioOutput;
    this.inputSampleRate = inputSampleRate;
    this.outputSampleRate = outputSampleRate;
    this.useStereoInput = useStereoInput;
    this.useStereoOutput = useStereoOutput;
}

/* 创建音频设备 Native 对象，在创建 PeerConnectionFactory 时调用 */
public long getNativeAudioDeviceModulePointer() {
    synchronized (nativeLock) {
        if (nativeAudioDeviceModule == 0) {
            nativeAudioDeviceModule = nativeCreateAudioDeviceModule(context, audioManager, audioInput,
```

```
                    audioOutput, inputSampleRate, outputSampleRate, useStereoInput, useStereoOutput);
          }
          return nativeAudioDeviceModule;
      }
}
```

2.2 创建 Native 层 AudioDeviceModule

通过JNI 调用创建 AudioDeviceModule

```
// out/android_arm64_Debug/gen/sdk/android/generated_java_audio_jni/JavaAudioDeviceModule_jni.h
// 此文件为编译过程中动态生成
JNI_GENERATOR_EXPORT jlong
    Java_org_webrtc_audio_JavaAudioDeviceModule_nativeCreateAudioDeviceModule(
    JNIEnv* env,
    jclass jcaller,
    jobject context,
    jobject audioManager,
    jobject audioInput,
    jobject audioOutput,
    jint inputSampleRate,
    jint outputSampleRate,
    jboolean useStereoInput,
    jboolean useStereoOutput) {
  return JNI_JavaAudioDeviceModule_CreateAudioDeviceModule(env,
        /* Java 层 Context 引用 */
        base::android::JavaParamRef<jobject>(env, context),
        /* Java 层 AudioManager 引用*/
        base::android::JavaParamRef<jobject>(env, audioManager),
        /* Java 层 WebRtcAudioRecord 引用 */
        base::android::JavaParamRef<jobject>(env, audioInput),
        /* Java 层 WebRtcAudioTrack 引用 */
        base::android::JavaParamRef<jobject>(env, audioOutput),
        inputSampleRate, outputSampleRate,
        useStereoInput, useStereoOutput);
}

// sdk/android/src/jni/audio_device/java_audio_device_module.cc
static jlong JNI_JavaAudioDeviceModule_CreateAudioDeviceModule(
    JNIEnv* env,
    const JavaParamRef<jobject>& j_context,
    const JavaParamRef<jobject>& j_audio_manager,
    const JavaParamRef<jobject>& j_webrtc_audio_record,
    const JavaParamRef<jobject>& j_webrtc_audio_track,
    int input_sample_rate,
    int output_sample_rate,
```

```
    jboolean j_use_stereo_input,
    jboolean j_use_stereo_output) {
    AudioParameters input_parameters;
    AudioParameters output_parameters;
    /* 通过 AudioManager 获取音频采集及播放的相关参数 */
    GetAudioParameters(env, j_context, j_audio_manager, input_sample_rate,
                        output_sample_rate, j_use_stereo_input,
                        j_use_stereo_output, &input_parameters,
                        &output_parameters);

    /* 创建音频采集 AudioRecordJni 对象 */
    auto audio_input = std::make_unique<AudioRecordJni>(
        env, input_parameters, kHighLatencyModeDelayEstimateInMilliseconds,
        j_webrtc_audio_record);

    /* 创建音频播放 AudioTrackJni 对象 */
    auto audio_output = std::make_unique<AudioTrackJni>(env, output_parameters,
                                                          j_webrtc_audio_track);


    return jlongFromPointer(CreateAudioDeviceModuleFromInputAndOutput(
                                // AudioDeviceModule::AudioLayer
                                AudioDeviceModule::kAndroidJavaAudio,
                                // 是否双声道
                                j_use_stereo_input, j_use_stereo_output,
                                // 播放延时估计, 150ms
                                kHighLatencyModeDelayEstimateInMilliseconds,
                                // 音频采集及播放对象
                                std::move(audio_input), std::move(audio_output))
                                .release());
}

// sdk/android/src/jni/audio_device/audio_device_module.cc
// 创建 AudioDeviceModule(modules/audio_device/include/audio_device.h),
// AndroidAudioDeviceModule 继承于该类.
rtc::scoped_refptr<AudioDeviceModule> CreateAudioDeviceModuleFromInputAndOutput(
    AudioDeviceModule::AudioLayer audio_layer,
    bool is_stereo_playout_supported,
    bool is_stereo_record_supported,
    uint16_t playout_delay_ms,
    std::unique_ptr<AudioInput> audio_input,
    std::unique_ptr<AudioOutput> audio_output) {
  RTC_LOG(INFO) << __FUNCTION__;
  return new rtc::RefCountedObject<AndroidAudioDeviceModule>(
      audio_layer, is_stereo_playout_supported, is_stereo_record_supported,
      playout_delay_ms, std::move(audio_input), std::move(audio_output));
}

AndroidAudioDeviceModule(AudioDeviceModule::AudioLayer audio_layer,
```

```
                           bool is_stereo_playout_supported,
                           bool is_stereo_record_supported,
                           uint16_t playout_delay_ms,
                           std::unique_ptr<AudioInput> audio_input,
                           std::unique_ptr<AudioOutput> audio_output)
      : audio_layer_(audio_layer),
        is_stereo_playout_supported_(is_stereo_playout_supported),
        is_stereo_record_supported_(is_stereo_record_supported),
        playout_delay_ms_(playout_delay_ms),
        task_queue_factory_(CreateDefaultTaskQueueFactory()),
        input_(std::move(audio_input)),
        output_(std::move(audio_output)),
        initialized_(false) {
    RTC_CHECK(input_);
    RTC_CHECK(output_);
    RTC_LOG(INFO) << __FUNCTION__;
    thread_checker_.Detach();
}
```

2.3 Native 获取音频采集播放参数

```
// sdk/android/src/jni/audio_device/audio_device_module.cc
void GetAudioParameters(JNIEnv* env,
                        const JavaRef<jobject>& j_context,
                        const JavaRef<jobject>& j_audio_manager,
                        int input_sample_rate,
                        int output_sample_rate,
                        bool use_stereo_input,
                        bool use_stereo_output,
                        AudioParameters* input_parameters,
                        AudioParameters* output_parameters) {
  const int output_channels = use_stereo_output ? 2 : 1;
  const int input_channels = use_stereo_input ? 2 : 1;
  const size_t output_buffer_size = Java_WebRtcAudioManager_getOutputBufferSize(
      env, j_context, j_audio_manager, output_sample_rate, output_channels);
  const size_t input_buffer_size = Java_WebRtcAudioManager_getInputBufferSize(
      env, j_context, j_audio_manager, input_sample_rate, input_channels);
  output_parameters->reset(output_sample_rate,
                           static_cast<size_t>(output_channels),
                           static_cast<size_t>(output_buffer_size));
  input_parameters->reset(input_sample_rate,
                          static_cast<size_t>(input_channels),
                          static_cast<size_t>(input_buffer_size));
  RTC_CHECK(input_parameters->is_valid());
  RTC_CHECK(output_parameters->is_valid());
}

```

```
//out/android_arm64_Debug/gen/sdk/android/generated_audio_device_module_base_jni/WebRTCAudioManager_jni.h
// 该文件根据 sdk/android/src/java/org/webrt/audio/WebRtcAudioManager.java 编译时生成
// 获取 AudioTrack 的 getMinBufferSize
static jint Java_WebRtcAudioManager_getOutputBufferSize(JNIEnv* env, const
    base::android::JavaRef<jobject>& context,
    const base::android::JavaRef<jobject>& audioManager,
    JniIntWrapper sampleRate,
    JniIntWrapper numberOfOutputChannels) {
    jclass clazz = org_webrtc_audio_WebRtcAudioManager_clazz(env);
    CHECK_CLAZZ(env, clazz,
        org_webrtc_audio_WebRtcAudioManager_clazz(env), 0);

    jni_generator::JniJavaCallContextChecked call_context;
    call_context.Init<
        base::android::MethodID::TYPE_STATIC>(
            env,
            clazz,
            "getOutputBufferSize",
            "(Landroid/content/Context;Landroid/media/AudioManager;II)I",
            &g_org_webrtc_audio_WebRtcAudioManager_getOutputBufferSize);

    /* 调用 sdk/android/src/java/org/webrt/audio/WebRtcAudioManager.java 的 getOutputBufferSize() 方法 */
    jint ret =
        env->CallStaticIntMethod(clazz,
            call_context.base.method_id, context.obj(), audioManager.obj(), as_jint(sampleRate),
                as_jint(numberOfOutputChannels));
    return ret;
}

// 获取 AudioRecord 的 getMinBufferSize
static jint Java_WebRtcAudioManager_getInputBufferSize(JNIEnv* env, const
    base::android::JavaRef<jobject>& context,
    const base::android::JavaRef<jobject>& audioManager,
    JniIntWrapper sampleRate,
    JniIntWrapper numberOfInputChannels) {
    jclass clazz = org_webrtc_audio_WebRtcAudioManager_clazz(env);
    CHECK_CLAZZ(env, clazz,
        org_webrtc_audio_WebRtcAudioManager_clazz(env), 0);

    jni_generator::JniJavaCallContextChecked call_context;
    call_context.Init<
        base::android::MethodID::TYPE_STATIC>(
            env,
            clazz,
            "getInputBufferSize",
```

```
            "(Landroid/content/Context;Landroid/media/AudioManager;II)I",
            &g_org_webrtc_audio_WebRtcAudioManager_getInputBufferSize);

    /* 调用 sdk/android/src/java/org/webrt/audio/WebRtcAudioManager.java 的 getInputBufferSize() 方法 */
    jint ret =
        env->CallStaticIntMethod(clazz,
            call_context.base.method_id, context.obj(), audioManager.obj(), as_jint(sampleRate),
                as_jint(numberOfInputChannels));
    return ret;
}
```

2.4 音频采集 AudioRecordJni

Native 层音频采集对象，在构建时绑定 Java 层的 WebRtcAudioRecord 实例，以实现音频采集启动及读取音频采集数据进行编码发送。

```
AudioRecordJni::AudioRecordJni(JNIEnv* env,
                               const AudioParameters& audio_parameters,
                               int total_delay_ms,
                               const JavaRef<jobject>& j_audio_record)
    : j_audio_record_(env, j_audio_record), // 保存 Java 层音频采集 WebRtcAudioRecord 实例引用
      audio_parameters_(audio_parameters),
      total_delay_ms_(total_delay_ms),
      direct_buffer_address_(nullptr),
      direct_buffer_capacity_in_bytes_(0),
      frames_per_buffer_(0),
      initialized_(false),
      recording_(false),
      audio_device_buffer_(nullptr) {
    RTC_LOG(INFO) << "ctor";
    RTC_DCHECK(audio_parameters_.is_valid());

    /* 调用 Java层WebRtcAudioRecord的setNativeAudioRecord 方法设置 Native 对象，在启动音频采集和将采集数据回调C++时使用 */
    Java_WebRtcAudioRecord_setNativeAudioRecord(env, j_audio_record_,
                                                jni::jlongFromPointer(this));
    // Detach from this thread since construction is allowed to happen on a
    // different thread.
    thread_checker_.Detach();
    thread_checker_java_.Detach();
}

// out/android_arm64_Debug/gen/sdk/android/generated_java_audio_device_module_native_jni/WebRtcAudioRecord_jni.h
// 该文件根据 org/webrtc/audio/WebRtcAudioRecord.java 在编译时生成
```

```
static void Java_WebRtcAudioRecord_setNativeAudioRecord(JNIEnv* env, const
    base::android::JavaRef<jobject>& obj, jlong nativeAudioRecord) {
  jclass clazz = org_webrtc_audio_WebRtcAudioRecord_clazz(env);
  CHECK_CLAZZ(env, obj.obj(),
      org_webrtc_audio_WebRtcAudioRecord_clazz(env));

  jni_generator::JniJavaCallContextChecked call_context;
  call_context.Init<
      base::android::MethodID::TYPE_INSTANCE>(
          env,
          clazz,
          "setNativeAudioRecord",
          "(J)V",
          &g_org_webrtc_audio_WebRtcAudioRecord_setNativeAudioRecord);

     env->CallVoidMethod(obj.obj(),
          call_context.base.method_id, nativeAudioRecord);
}
```

Java 层调用

```
// sdk/android/src/java/org/webrtc/audio/WebRtcAudioRecord.java
@CalledByNative
public void setNativeAudioRecord(long nativeAudioRecord) {
    this.nativeAudioRecord = nativeAudioRecord;
}
```

2.5 音频播放 AudioTrackJni

Native 层音频播放对象，在构建时绑定到 Java 层的 WebRtcAudioTrack 实例，实现音频播放初始化及音频播放数据读取。

```
AudioTrackJni::AudioTrackJni(JNIEnv* env,
                             const AudioParameters& audio_parameters,
                             const JavaRef<jobject>& j_webrtc_audio_track)
    : j_audio_track_(env, j_webrtc_audio_track), // 保存 Java 层音频采集 WebRtcAudioTrack 实例引用
      audio_parameters_(audio_parameters),
      direct_buffer_address_(nullptr),
      direct_buffer_capacity_in_bytes_(0),
      frames_per_buffer_(0),
      initialized_(false),
      playing_(false),
      audio_device_buffer_(nullptr) {
    RTC_LOG(INFO) << "ctor";
    RTC_DCHECK(audio_parameters_.is_valid());

```

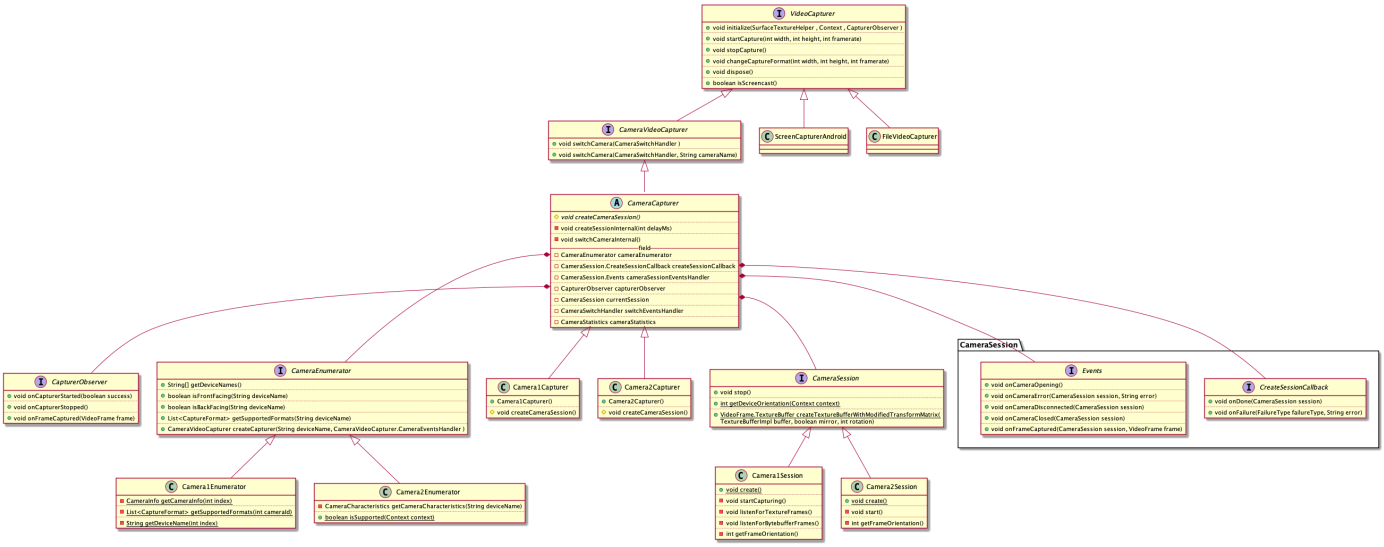
```
    /* 调用 Java层 WebRtcAudioTrack 的 setNativeAudioTrac 方法设置 Native 对象，在启动音频播放和从C++层获取播放数据时使用 */
    Java_WebRtcAudioTrack_setNativeAudioTrack(env, j_audio_track_,
                                              jni::jlongFromPointer(this));
    // Detach from this thread since construction is allowed to happen on a
    // different thread.
    thread_checker_.Detach();
    thread_checker_java_.Detach();
}

// out/android_arm64_Debug/gen/sdk/android/generated_java_audio_device_module_native_jni/WebRtcAudioTrack_jni.h
// 该文件根据 org/webrtc/audio/WebRtcAudioTrack.java 在编译时生成
static void Java_WebRtcAudioTrack_setNativeAudioTrack(JNIEnv* env, const
    base::android::JavaRef<jobject>& obj, jlong nativeAudioTrack) {
  jclass clazz = org_webrtc_audio_WebRtcAudioTrack_clazz(env);
  CHECK_CLAZZ(env, obj.obj(),
      org_webrtc_audio_WebRtcAudioTrack_clazz(env));

  jni_generator::JniJavaCallContextChecked call_context;
  call_context.Init<
      base::android::MethodID::TYPE_INSTANCE>(
          env,
          clazz,
          "setNativeAudioTrack",
          "(J)V",
          &g_org_webrtc_audio_WebRtcAudioTrack_setNativeAudioTrack);

     env->CallVoidMethod(obj.obj(),
          call_context.base.method_id, nativeAudioTrack);
}
```

Java 层调用

```
// sdk/android/src/java/org/webrtc/audio/WebRtcAudioTrack.java
@CalledByNative
public void setNativeAudioTrack(long nativeAudioTrack) {
    this.nativeAudioTrack = nativeAudioTrack;
}
```

3 视频图像采集模块(VideoCapturer)

视频采集源分为视频文件，屏幕共享及摄像头，这几类视频源统一继承了 VideoCapturer 类。其中摄像头采集源的通过 CameraEnumerator、CameraSession、CameraCapturer 连接不同 Camera API， 其中 Camera1Enumerator、Camera1Session， Camera1Capturer 连接 Camera V1 Api(即Android 5之前的摄像头接口)； 而 Camera2Enumerator、Camera2Session， Camera2Capturer 连接 Camera V2 接口。这两套接口封装了新旧 Camera Api的差异性，使外部调用保持一致。



Android 平台 VideoCapturer 类图

3.1 创建摄像头采集源(CameraVideoCapturer)

创建视频采集源， Demo 端调用入口。

```java
// CallActivity.java
private @Nullable VideoCapturer createVideoCapturer() {
    final VideoCapturer videoCapturer;
    String videoFileAsCamera = getIntent().getStringExtra(EXTRA_VIDEO_FILE_AS_CAMERA);
    /* 视频文件作为视频采集源 */
    if (videoFileAsCamera != null) {
        try {
            videoCapturer = new FileVideoCapturer(videoFileAsCamera);
        } catch (IOException e) {
            reportError("Failed to open video file for emulated camera");
            return null;
        }
    } else if (screencaptureEnabled) {
        /* 屏幕共享 */
        return createScreenCapturer();
    } else if (useCamera2()) {
        /* Camera2 Api */
        if (!captureToTexture()) {
            reportError(getString(R.string.camera2_texture_only_error));
            return null;
        }

        Logging.d(TAG, "Creating capturer using camera2 API.");
        videoCapturer = createCameraCapturer(new Camera2Enumerator(this));
    } else {
        /* Camera Api */
        Logging.d(TAG, "Creating capturer using camera1 API.");
```

```
        videoCapturer = createCameraCapturer(new Camera1Enumerator(captureToTexture()));
    }
    if (videoCapturer == null) {
        reportError("Failed to open camera");
        return null;
    }
    return videoCapturer;
}

// 创建摄像头采集源
private @Nullable VideoCapturer createCameraCapturer(CameraEnumerator enumerator) {
    /* 获取设备摄像头信息 */
    final String[] deviceNames = enumerator.getDeviceNames();

    // First, try to find front facing camera
    Logging.d(TAG, "Looking for front facing cameras.");
    // 创建前置摄像头
    for (String deviceName : deviceNames) {
        if (enumerator.isFrontFacing(deviceName)) {
            Logging.d(TAG, "Creating front facing camera capturer.");
            VideoCapturer videoCapturer = enumerator.createCapturer(deviceName, /* CameraEventsHandler */null);

            if (videoCapturer != null) {
                return videoCapturer;
            }
        }
    }

    // Front facing camera not found, try something else
    Logging.d(TAG, "Looking for other cameras.");
    // 若无前置摄像头，则创建后置摄像头
    for (String deviceName : deviceNames) {
        if (!enumerator.isFrontFacing(deviceName)) {
            Logging.d(TAG, "Creating other camera capturer.");
            VideoCapturer videoCapturer = enumerator.createCapturer(deviceName, /* CameraEventsHandler */null);

            if (videoCapturer != null) {
                return videoCapturer;
            }
        }
    }

    return null;
}
```

3.1.1 摄像头 Camera1 采集(Camera1Capturer)

1. 创建 Camera1Enumerator；
2. 通过 Camera1Enumerator 接口 createCapturer() 创建 Camera1Capturer

```
// sdk/android/api/org/webrtc/Camera1Enumerator.java
public Camera1Enumerator(boolean captureToTexture) {
    this.captureToTexture = captureToTexture;
}

@Override
public CameraVideoCapturer createCapturer(
    String deviceName, CameraVideoCapturer.CameraEventsHandler eventsHandler) {
    return new Camera1Capturer(deviceName, eventsHandler, captureToTexture);
}
```

3. Camera1Capturer 构造函数调用父类 CameraCapturer 构造函数，保存相关参数等操作

```
// sdk/android/api/org/webrtc/Camera1Capturer.java
public Camera1Capturer( String cameraName, CameraEventsHandler eventsHandler,
boolean captureToTexture) {
    /* 调用父类构造函数，注意此处重建了 Camera1Enumerator  */
    super(cameraName, eventsHandler, new Camera1Enumerator(captureToTexture));

    this.captureToTexture = captureToTexture;
}

// sdk/android/src/java/org/webrt/CameraCapturer.java
public CameraCapturer(String cameraName, @Nullable CameraEventsHandler
eventsHandler,
      CameraEnumerator cameraEnumerator) {
    // 此处 eventsHandler 为 null
    if (eventsHandler == null) {
        eventsHandler = new CameraEventsHandler() {
            @Override
            public void onCameraError(String errorDescription) {}
            @Override
            public void onCameraDisconnected() {}
            @Override
            public void onCameraFreezed(String errorDescription) {}
            @Override
            public void onCameraOpening(String cameraName) {}
            @Override
            public void onFirstFrameAvailable() {}
            @Override
            public void onCameraClosed() {}
        };
    }

```

```
    this.eventsHandler = eventsHandler;
    this.cameraEnumerator = cameraEnumerator;
    this.cameraName = cameraName;
    List<String> deviceNames = Arrays.asList(cameraEnumerator.getDeviceNames());
    uiThreadHandler = new Handler(Looper.getMainLooper());
}
```

3.1.2 摄像头 Camera2 采集(Camera2Capturer)

1. 创建 Camera2Enumerator；
2. 通过 Camera2Enumerator 接口 createCapturer() 创建 Camera2Capturer

```
// sdk/android/api/org/webrtc/Camera2Enumerator.java
public Camera2Enumerator(Context context) {
    this.context = context;
    /* 获取系统摄像头管理服务 */
    this.cameraManager = (CameraManager)
context.getSystemService(Context.CAMERA_SERVICE);
}

@Override
public CameraVideoCapturer createCapturer(String deviceName,
CameraVideoCapturer.CameraEventsHandler eventsHandler) {
    return new Camera2Capturer(context, deviceName, eventsHandler);
}
```

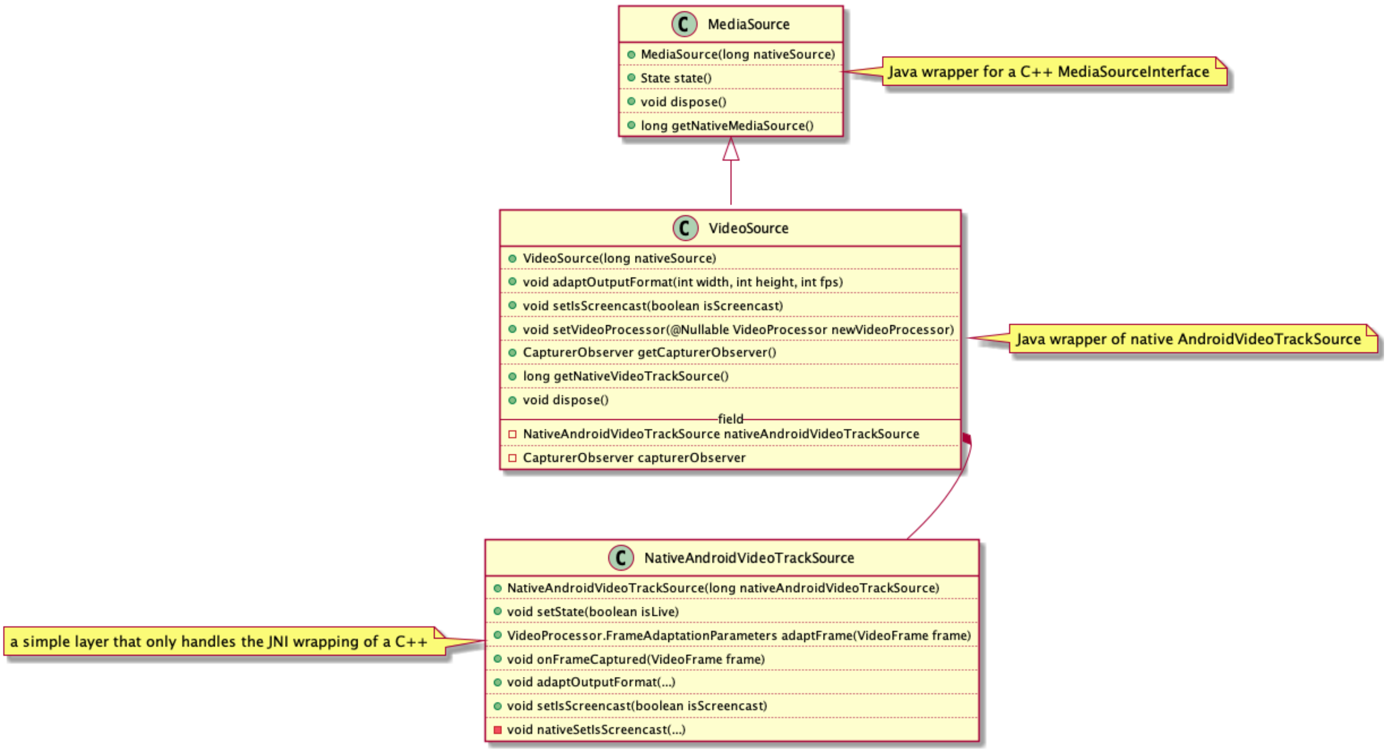
3. Camera2Capturer 构造函数调用父类 CameraCapturer 构造函数，保存相关参数等操作

```
// sdk/android/api/org/webrtc/Camera1Capturer.java
public Camera2Capturer(Context context, String cameraName, CameraEventsHandler
eventsHandler) {
    /* 调用父类构造函数，注意此处新建了 Camera2Enumerator  */
    super(cameraName, eventsHandler, new Camera2Enumerator(context));

    this.context = context;
    /* 获取系统摄像头管理服务 */
    cameraManager = (CameraManager)
context.getSystemService(Context.CAMERA_SERVICE);
}

// sdk/android/src/java/org/webrt/CameraCapturer.java
public CameraCapturer(String cameraName, @Nullable CameraEventsHandler
eventsHandler,
      CameraEnumerator cameraEnumerator) {
    // 此处 eventsHandler 为 null
    if (eventsHandler == null) {
        eventsHandler = new CameraEventsHandler() {
            @Override
```

```java
            public void onCameraError(String errorDescription) {}
            @Override
            public void onCameraDisconnected() {}
            @Override
            public void onCameraFreezed(String errorDescription) {}
            @Override
            public void onCameraOpening(String cameraName) {}
            @Override
            public void onFirstFrameAvailable() {}
            @Override
            public void onCameraClosed() {}
        };
    }

    this.eventsHandler = eventsHandler;
    this.cameraEnumerator = cameraEnumerator;
    this.cameraName = cameraName;
    List<String> deviceNames = Arrays.asList(cameraEnumerator.getDeviceNames());
    uiThreadHandler = new Handler(Looper.getMainLooper());
}
```

3.2 创建 VideoSource

1. 创建 Java 层 VideoSource，由 PeerConnectionClient.createVideoTrack() 内调用。



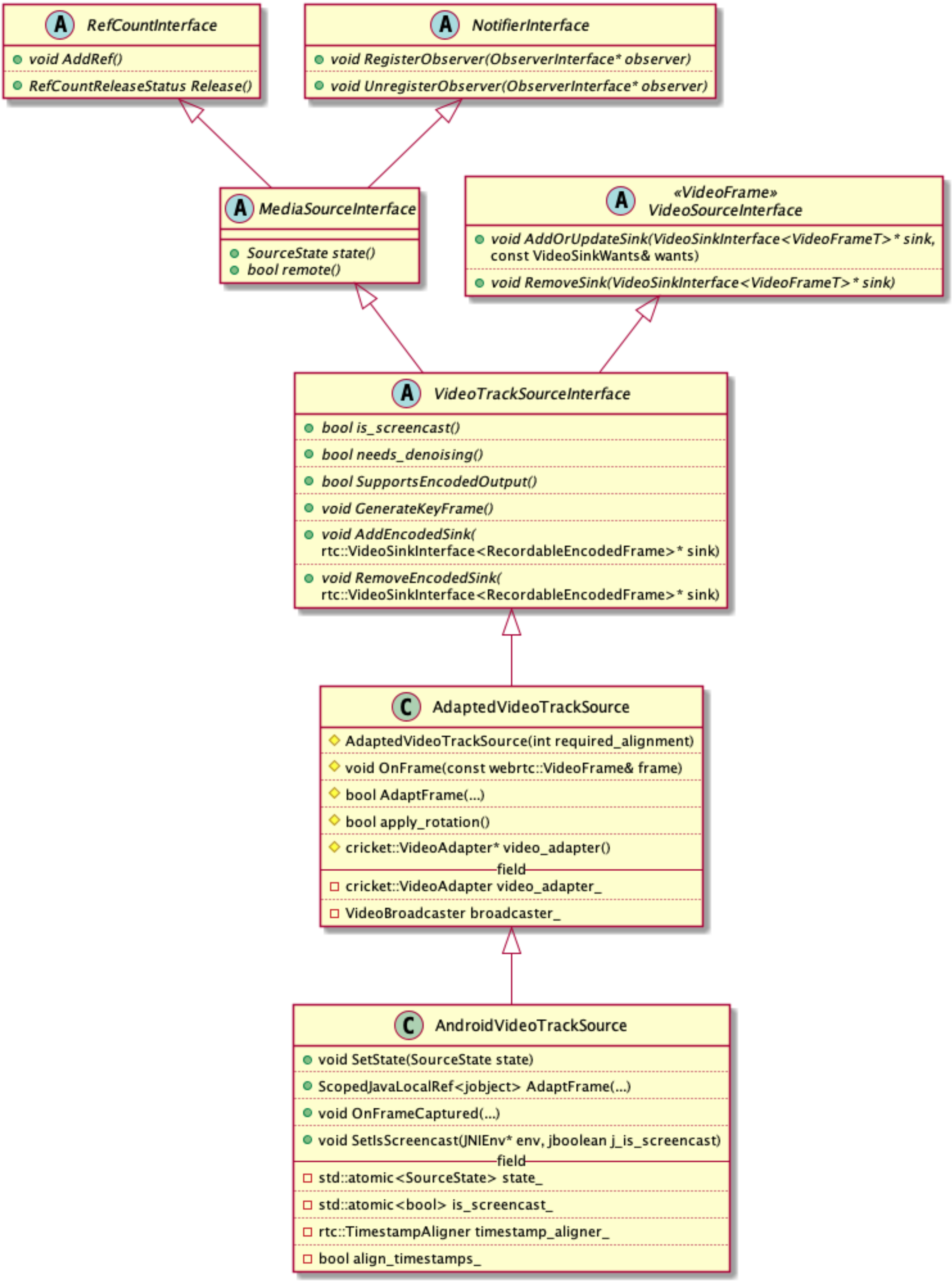
Java 层 VideoSource 类图

简要代码流程：

```java
// sdk/android/api/org/webrtc/PeerConnectionFactory.java
```

```java
public VideoSource createVideoSource(boolean isScreencast) {
    return createVideoSource(isScreencast, /* alignTimestamps= */ true);
}

public VideoSource createVideoSource(boolean isScreencast, boolean alignTimestamps) {
    checkPeerConnectionFactoryExists();
    // 需先创建 Native 层 VideoSource，进行绑定
    return new VideoSource(nativeCreateVideoSource(nativeFactory, isScreencast, alignTimestamps));
}

// sdk/android/api/org/webrtc/VideoSource.java
public VideoSource(long nativeSource) {
    // 调用父类 MediaSource 构造
    super(nativeSource);
    // 创建 AndroidVideoTrackSource 的 java 层映射
    this.nativeAndroidVideoTrackSource = new NativeAndroidVideoTrackSource(nativeSource);
}

// // sdk/android/src/java/org/webrtc/NativeAndroidVideoTrackSource.java
public NativeAndroidVideoTrackSource(long nativeAndroidVideoTrackSource) {
    this.nativeAndroidVideoTrackSource = nativeAndroidVideoTrackSource;
}

// sdk/android/api/org/webrtc/MediaSource.java
public MediaSource(long nativeSource) {
    refCountDelegate = new RefCountDelegate(() -> JniCommon.nativeReleaseRef(nativeSource));
    this.nativeSource = nativeSource;
}
```

1. 创建 VideoSource 对应 C++ 层 AndroidVideoTrackSource。



C++ 层 VideoSource 类图

简要代码流程：

```
// out/android_arm64_Debug/gen/sdk/android/generated_peerconnection_jni/PeerConnectionFactory_jni.h
JNI_GENERATOR_EXPORT jlong Java_org_webrtc_PeerConnectionFactory_nativeCreateVideoSource(
    JNIEnv* env,
    jclass jcaller,
    jlong factory,
    jboolean is_screencast,
    jboolean alignTimestamps) {
  return JNI_PeerConnectionFactory_CreateVideoSource(env, factory, is_screencast, alignTimestamps);
}

// sdk/android/src/jni/pc/peer_connection_factory.cc
static jlong JNI_PeerConnectionFactory_CreateVideoSource(
    JNIEnv* jni,
    jlong native_factory,
    jboolean is_screencast,
    jboolean align_timestamps) {
  OwnedFactoryAndThreads* factory =
      reinterpret_cast<OwnedFactoryAndThreads*>(native_factory);
  return jlongFromPointer(CreateVideoSource(jni, factory->signaling_thread(),
                                            factory->worker_thread(),
                                            is_screencast, align_timestamps));
}

// sdk/android/src/jni/pc/video.cc
void* CreateVideoSource(JNIEnv* env,
                        rtc::Thread* signaling_thread,
                        rtc::Thread* worker_thread,
                        jboolean is_screencast,
                        jboolean align_timestamps) {
    rtc::scoped_refptr<AndroidVideoTrackSource> source(
        new rtc::RefCountedObject<AndroidVideoTrackSource>(
            signaling_thread, env, is_screencast, align_timestamps));
    return source.release();
}

// sdk/android/src/jni/android_video_track_source.cc
// AndroidVideoTrackSource 继承于 AdaptedVideoTrackSource, AdaptedVideoTrackSource 又继承于 VideoTrackSourceInterface
// AdaptedVideoTrackSource 实现了对视频裁剪，角度旋转，丢帧等处理
AndroidVideoTrackSource::AndroidVideoTrackSource(rtc::Thread* signaling_thread,
                                                 JNIEnv* jni,
                                                 bool is_screencast,
                                                 bool align_timestamps)
    : AdaptedVideoTrackSource(kRequiredResolutionAlignment),
      signaling_thread_(signaling_thread),
```

```
        is_screencast_(is_screencast),
        align_timestamps_(align_timestamps) {
    RTC_LOG(LS_INFO) << "AndroidVideoTrackSource ctor";
}
```

3.3 摄像头采集初始化及启动

```
// sdk/android/src/java/org/webrtc/CameraCapturer.java
@Override
public void initialize(SurfaceTextureHelper surfaceTextureHelper, Context
applicationContext,
      org.webrtc.CapturerObserver capturerObserver) {
    this.applicationContext = applicationContext;
    // capturerObserver 视频数据回调接口
    this.capturerObserver = capturerObserver;
    this.surfaceHelper = surfaceTextureHelper;
    this.cameraThreadHandler = surfaceTextureHelper.getHandler();
}

@Override
public void startCapture(int width, int height, int framerate) {
    Logging.d(TAG, "startCapture: " + width + "x" + height + "@" + framerate);
    if (applicationContext == null) {
        throw new RuntimeException("CameraCapturer must be initialized before
calling startCapture.");
    }

    synchronized (stateLock) {
        if (sessionOpening || currentSession != null) {
        Logging.w(TAG, "Session already open");
        return;
        }

        this.width = width;
        this.height = height;
        this.framerate = framerate;

        sessionOpening = true;
        // 摄像头启动尝试次数
        openAttemptsRemaining = MAX_OPEN_CAMERA_ATTEMPTS;
        createSessionInternal(0);
    }
}

private void createSessionInternal(int delayMs) {
    // 启动摄像头启动超时定时器，以便重启摄像头
    uiThreadHandler.postDelayed(openCameraTimeoutRunnable, delayMs +
OPEN_CAMERA_TIMEOUT);
```

```
    cameraThreadHandler.postDelayed(new Runnable() {
        @Override
        public void run() {
            // 调用子类 Camera1Capturer 或 Camera2Capturer 的 createCameraSession() 实现
            createCameraSession(createSessionCallback, cameraSessionEventsHandler, applicationContext,
                surfaceHelper, cameraName, width, height, framerate);
        }
    }, delayMs);
}
```

3.3.1 启动 Camera1 采集

在调用 CameraCapturer 的 startCapture() 时启动，先创建 Camera1Session，并在 Camera1Session 内启动摄像头。

1. 创建 Camera1Session，并打开摄像头，设置摄像头参数。

```
// sdk/android/api/org/webrtc/Camera1Capturer.java
@Override
protected void createCameraSession(CameraSession.CreateSessionCallback createSessionCallback,
    CameraSession.Events events, Context applicationContext,
    SurfaceTextureHelper surfaceTextureHelper, String cameraName, int width, int height,
    int framerate) {
    // 创建 Camera1Session 并启动摄像头
    Camera1Session.create(createSessionCallback, events, captureToTexture, applicationContext,
        surfaceTextureHelper, Camera1Enumerator.getCameraIndex(cameraName), width, height,
        framerate);
}

// sdk/android/src/java/org/webrtc/Camera1Session.java
public static void create(final CreateSessionCallback callback, final Events events,
      final boolean captureToTexture, final Context applicationContext,
      final SurfaceTextureHelper surfaceTextureHelper, final int cameraId, final int width,
      final int height, final int framerate) {
    final long constructionTimeNs = System.nanoTime();
    Logging.d(TAG, "Open camera " + cameraId);
    // 状态上报
    events.onCameraOpening();

    final android.hardware.Camera camera;
    try {
```

```java
        // 打开摄像头
        camera = android.hardware.Camera.open(cameraId);
    } catch (RuntimeException e) {
        // 摄像头打开失败，回调触发重启
        callback.onFailure(FailureType.ERROR, e.getMessage());
        return;
    }

    if (camera == null) {
        // 摄像头打开失败，回调触发重启
        callback.onFailure(FailureType.ERROR,
            "android.hardware.Camera.open returned null for camera id = " + cameraId);
        return;
    }

    try {
        // 设置摄像头预览
        camera.setPreviewTexture(surfaceTextureHelper.getSurfaceTexture());
    } catch (IOException | RuntimeException e) {
        camera.release();
        // 摄像头预览设置失败，回调触发重启
        callback.onFailure(FailureType.ERROR, e.getMessage());
        return;
    }

    // 摄像头 参数设置
    ...

    // 若以非纹理方式捕捉，则通过 Camera.PreviewCallback 回调采集数据，需设置缓冲区
    if (!captureToTexture) {
        final int frameSize = captureFormat.frameSize();
        for (int i = 0; i < NUMBER_OF_CAPTURE_BUFFERS; ++i) {
            final ByteBuffer buffer = ByteBuffer.allocateDirect(frameSize);
            camera.addCallbackBuffer(buffer.array());
        }
    }

    // Calculate orientation manually and send it as CVO insted.
    camera.setDisplayOrientation(0 /* degrees */);

    callback.onDone(new Camera1Session(events, captureToTexture, applicationContext,
        surfaceTextureHelper, cameraId, camera, info, captureFormat, constructionTimeNs));
}

private Camera1Session(Events events, boolean captureToTexture, Context applicationContext,
```

```
        SurfaceTextureHelper surfaceTextureHelper, int cameraId, android.hardware.Camera camera,
        android.hardware.Camera.CameraInfo info, CaptureFormat captureFormat,
        long constructionTimeNs) {
    Logging.d(TAG, "Create new camera1 session on camera " + cameraId);

    this.cameraThreadHandler = new Handler();
    this.events = events;
    this.captureToTexture = captureToTexture;
    this.applicationContext = applicationContext;
    this.surfaceTextureHelper = surfaceTextureHelper;
    this.cameraId = cameraId;
    this.camera = camera;
    this.info = info;
    this.captureFormat = captureFormat;
    this.constructionTimeNs = constructionTimeNs;

    // 设置采集数据输出分辨率
    surfaceTextureHelper.setTextureSize(captureFormat.width, captureFormat.height);

    // 开始摄像头采集
    startCapturing();
}
```

2. 启动摄像头采集，设置视频采集数据回调。

```
// sdk/android/src/java/org/webrtc/Camera1Session.java
private void startCapturing() {
    Logging.d(TAG, "Start capturing");
    checkIsOnCameraThread();

    state = SessionState.RUNNING;

    camera.setErrorCallback(new android.hardware.Camera.ErrorCallback() {
        @Override
        public void onError(int error, android.hardware.Camera camera) {
            String errorMessage;
            if (error == android.hardware.Camera.CAMERA_ERROR_SERVER_DIED) {
                errorMessage = "Camera server died!";
            } else {
                errorMessage = "Camera error: " + error;
            }
            Logging.e(TAG, errorMessage);
            stopInternal();
            if (error == android.hardware.Camera.CAMERA_ERROR_EVICTED) {
                events.onCameraDisconnected(Camera1Session.this);
            } else {
                events.onCameraError(Camera1Session.this, errorMessage);
            }
```

```java
            }
        });

        // 设置采集数据监听
        if (captureToTexture) {
            listenForTextureFrames();
        } else {
            listenForBytebufferFrames();
        }
        try {
            // 正式启动摄像头
            camera.startPreview();
        } catch (RuntimeException e) {
            stopInternal();
            events.onCameraError(this, e.getMessage());
        }
    }
```

3.3.2 启动 Camera2 采集

在调用 CameraCapturer 的 startCapture() 时启动，先创建 Camera2Session，并在 Camera2Session 内启动摄像头。

1. 创建 Camera2Session

```java
// sdk/android/api/org/webrtc/Camera2Capturer.java
@Override
protected void createCameraSession(CameraSession.CreateSessionCallback createSessionCallback,
      CameraSession.Events events, Context applicationContext,
      SurfaceTextureHelper surfaceTextureHelper, String cameraName, int width, int height,
      int framerate) {
    Camera2Session.create(createSessionCallback, events, applicationContext, cameraManager,
        surfaceTextureHelper, cameraName, width, height, framerate);
}

// sdk/android/src/java/org/webrtc/Camera2Session.java
public static void create(CreateSessionCallback callback, Events events,
      Context applicationContext, CameraManager cameraManager,
      SurfaceTextureHelper surfaceTextureHelper, String cameraId, int width, int height,
      int framerate) {
    new Camera2Session(callback, events, applicationContext, cameraManager, surfaceTextureHelper,
        cameraId, width, height, framerate);
}

```

```java
private Camera2Session(CreateSessionCallback callback, Events events, Context applicationContext,
        CameraManager cameraManager, SurfaceTextureHelper surfaceTextureHelper, String cameraId,
        int width, int height, int framerate) {
    Logging.d(TAG, "Create new camera2 session on camera " + cameraId);

    constructionTimeNs = System.nanoTime();

    this.cameraThreadHandler = new Handler();
    this.callback = callback;
    this.events = events;
    this.applicationContext = applicationContext;
    this.cameraManager = cameraManager;
    this.surfaceTextureHelper = surfaceTextureHelper;
    this.cameraId = cameraId;
    this.width = width;
    this.height = height;
    this.framerate = framerate;

    start();
}
```

2. 打开摄像头

```java
private void start() {
    checkIsOnCameraThread();
    Logging.d(TAG, "start");

    try {
        cameraCharacteristics = cameraManager.getCameraCharacteristics(cameraId);
    } catch (final CameraAccessException e) {
        reportError("getCameraCharacteristics(): " + e.getMessage());
        return;
    }
    cameraOrientation = cameraCharacteristics.get(CameraCharacteristics.SENSOR_ORIENTATION);
    isCameraFrontFacing = cameraCharacteristics.get(CameraCharacteristics.LENS_FACING)
        == CameraMetadata.LENS_FACING_FRONT;

    findCaptureFormat();
    openCamera();
}

private void openCamera() {
    checkIsOnCameraThread();

    Logging.d(TAG, "Opening camera " + cameraId);
```

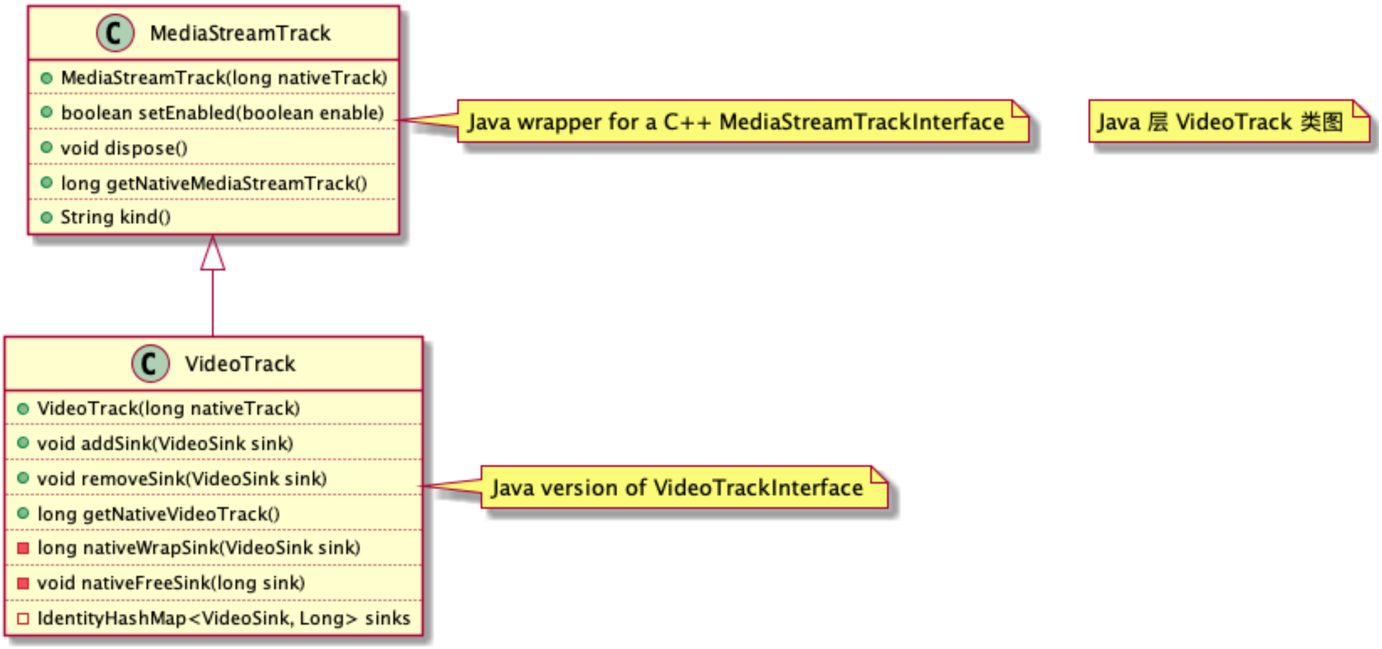
```
    events.onCameraOpening();

    try {
        // 设置摄像头状态回调，打开摄像头
        cameraManager.openCamera(cameraId, new CameraStateCallback(),
cameraThreadHandler);
    } catch (CameraAccessException e) {
        reportError("Failed to open camera: " + e);
        return;
    }
}
```

3.4 创建 VideoTrack

调用入口为 PeerConnectionClient.createVideoTrack()， 详见 1.3 创建 VideoTrack。
在创建 VideoSource 及打开摄像头之后，调用 PeerConnectionFactory.createVideoTrack()。

3.4.1 创建 VideoTrack

localVideoTrack = factory.createVideoTrack(VIDEO_TRACK_ID, videoSource);

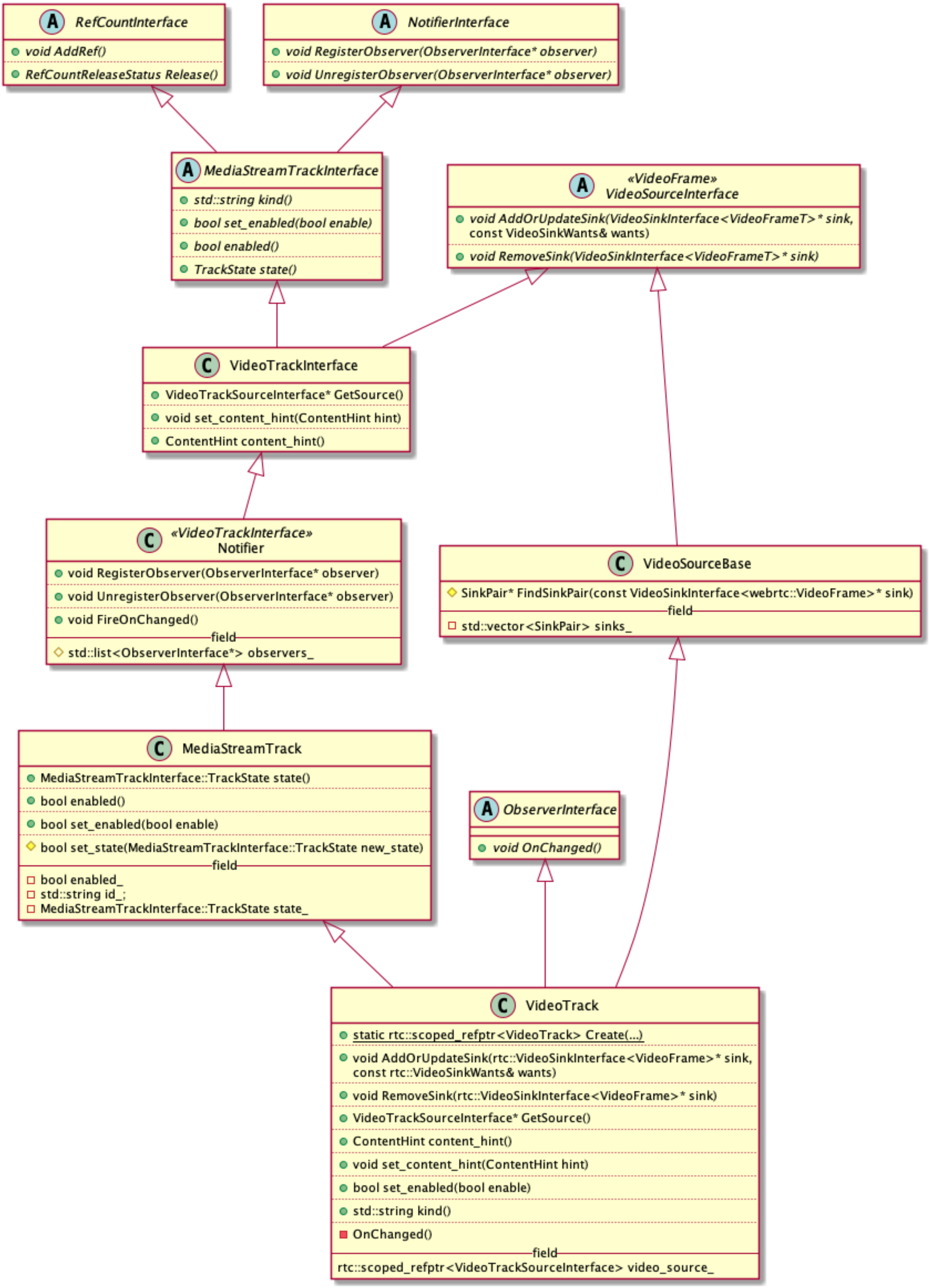


Java 层 VideoTrack 类图

创建流程分析：

```java
// sdk/android/api/org/webrtc/PeerConnectionFactory.java
// id: "ARDAMSv0"
public VideoTrack createVideoTrack(String id, VideoSource source) {
    checkPeerConnectionFactoryExists();
    return new VideoTrack(
        nativeCreateVideoTrack(nativeFactory, id, source.getNativeVideoTrackSource()/* Native VideoSource 指针地址 */));
}

// sdk/android/api/org/webrtc/VideoTrack.java
public VideoTrack(long nativeTrack) {
    // 调用父类 MediaStreamTrack 构造函数，保存 native 指针
    super(nativeTrack);
}
```

创建 Native 层 VideoTrack。



Native 层 VideoTrack 类图

创建流程分析：

```
//
out/android_arm64_Debug/gen/sdk/android/generated_peerconnection_jni/PeerConnection
Factory_jni.h
JNI_GENERATOR_EXPORT jlong
Java_org_webrtc_PeerConnectionFactory_nativeCreateVideoTrack(
    JNIEnv* env,
    jclass jcaller,
    jlong factory,
    jstring id,
    jlong nativeVideoSource) {
    return JNI_PeerConnectionFactory_CreateVideoTrack(env, factory,
        base::android::JavaParamRef<jstring>(env, id), nativeVideoSource);
}

// sdk/android/src/jni/pc/peer_connection_factory.cc
static jlong JNI_PeerConnectionFactory_CreateVideoTrack(
    JNIEnv* jni,
    jlong native_factory,
    const JavaParamRef<jstring>& id,
    jlong native_source) {
    rtc::scoped_refptr<VideoTrackInterface> track =
        PeerConnectionFactoryFromJava(native_factory)
            ->CreateVideoTrack(
                JavaToStdString(jni, id),
                reinterpret_cast<VideoTrackSourceInterface*>(native_source));
    return jlongFromPointer(track.release());
}

// pc/peer_connection_factory.cc
rtc::scoped_refptr<VideoTrackInterface> PeerConnectionFactory::CreateVideoTrack(
    const std::string& id,
    VideoTrackSourceInterface* source) {
  RTC_DCHECK(signaling_thread_->IsCurrent());
  rtc::scoped_refptr<VideoTrackInterface> track(
      VideoTrack::Create(id, source, worker_thread_));
  return VideoTrackProxy::Create(signaling_thread_, worker_thread_, track);
}

// pc/video_track.cc
rtc::scoped_refptr<VideoTrack> VideoTrack::Create(
    const std::string& id,
    VideoTrackSourceInterface* source,
    rtc::Thread* worker_thread) {
    // 构造 VideoTrack
    rtc::RefCountedObject<VideoTrack>* track =
```

```
        new rtc::RefCountedObject<VideoTrack>(id, source, worker_thread);
    return track;
}

VideoTrack::VideoTrack(const std::string& label,
                       VideoTrackSourceInterface* video_source,
                       rtc::Thread* worker_thread)
    : MediaStreamTrack<VideoTrackInterface>(label),
      worker_thread_(worker_thread),
      // 保存 VideoSource，在 set_enabled() 的时候添加 sink。
      video_source_(video_source),
      content_hint_(ContentHint::kNone) {

    // 注册 VideoSource 观察者
    video_source_->RegisterObserver(this);
}

// api/notifier.h
// AndroidVideoTrackSource 继承于 AdaptedVideoTrackSource,
// AdaptedVideoTraceSource 继承于 webrtc::Notifier<webrtc::VideoTrackSourceInterface>
virtual void RegisterObserver(ObserverInterface* observer) {
    RTC_DCHECK(observer != nullptr);
    observers_.push_back(observer);
}
```

3.4.2 使能 VideoTrack

localVideoTrack.setEnabled(renderVideo);

```
// sdk/android/api/org/webrtc/MediaStreamTrack.java
// VideoTrack 是 MediaStreamTrack 的扩展类，setEnabled() 的实现位于 MediaStreamTrack
public boolean setEnabled(boolean enable) {
    checkMediaStreamTrackExists();
    // 调用 Native 层 VideoTrack 函数
    return nativeSetEnabled(nativeTrack, enable);
}
```

转入 Native 层调用

```
// out/android_arm64_Debug/gen/sdk/android/generated_peerconnection_jni/MediaStreamTrack_jni.h
JNI_GENERATOR_EXPORT jboolean Java_org_webrtc_MediaStreamTrack_nativeSetEnabled(
    JNIEnv* env,
    jclass jcaller,
    jlong track,
    jboolean enabled) {
  return JNI_MediaStreamTrack_SetEnabled(env, track, enabled);
```

```
}

// sdk/android/src/jni/pc/media_stream_track.cc
static jboolean JNI_MediaStreamTrack_SetEnabled(JNIEnv* jni,
                                                jlong j_p,
                                                jboolean enabled) {
    return reinterpret_cast<MediaStreamTrackInterface*>(j_p)->set_enabled(enabled);
}

// pc/video_track.cc
bool VideoTrack::set_enabled(bool enable) {
    RTC_DCHECK(signaling_thread_checker_.IsCurrent());
    worker_thread_->Invoke<void>(RTC_FROM_HERE, [enable, this] {
        RTC_DCHECK(worker_thread_->IsCurrent());
        for (auto& sink_pair : sink_pairs()) {
            rtc::VideoSinkWants modified_wants = sink_pair.wants;
            modified_wants.black_frames = !enable;
            video_source_->AddOrUpdateSink(sink_pair.sink, modified_wants);
        }
    });

    // 调用基类函数
    return MediaStreamTrack<VideoTrackInterface>::set_enabled(enable);
}

// pc/media_stream_track.h
bool set_enabled(bool enable) override {
    bool fire_on_change = (enable != enabled_);
    // 保存使能状态
    enabled_ = enable;
    if (fire_on_change) {
        // 此处 T 为 VideoTrackInterface
        Notifier<T>::FireOnChanged();
    }
    return fire_on_change;
}
```

3.4.3 添加渲染 Sink

添加预览的 VideoSink 到 VideoTrack。

localVideoTrack.addSink(localRender);

```
// sdk/android/api/org/webrtc/VideoTrack.java
public void addSink(VideoSink sink) {
    if (sink == null) {
        throw new IllegalArgumentException("The VideoSink is not allowed to be null");
    }
```

```
    // We allow calling addSink() with the same sink multiple times. This is similar to the C++
    // VideoTrack::AddOrUpdateSink().
    if (!sinks.containsKey(sink)) {
        /* Java 层 VideoSink 与 Native 层 VideoSink 的映射 */
        final long nativeSink = nativeWrapSink(sink);
        /* 保存映射关系，在 删除Sink时，释放 Native 层的 VideoSink */
        sinks.put(sink, nativeSink);
        nativeAddSink(getNativeMediaStreamTrack(), nativeSink);
    }
}
```

1. 映射 Java 层 VideoSink 到 Native 层 VideoSink（VideoSinkWrapper）。

```
// out/android_arm64_Debug/gen/sdk/android/generated_video_jni/VideoTrack_jni.h
JNI_GENERATOR_EXPORT jlong Java_org_webrtc_VideoTrack_nativeWrapSink(
    JNIEnv* env,
    jclass jcaller,
    jobject sink) {
    return JNI_VideoTrack_WrapSink(env, base::android::JavaParamRef<jobject>(env, sink));
}

// sdk/android/src/jni/video_track.cc
static jlong JNI_VideoTrack_WrapSink(JNIEnv* jni,
                                     const JavaParamRef<jobject>& sink) {
  return jlongFromPointer(new VideoSinkWrapper(jni, sink));
}

// sdk/android/src/jni/video_sink.cc
// 保存 Java 层 VideoSink 实例，后续视频帧通过该实例回调给 Java 层。
// VideoSinkWrapper 父类是 rtc::VideoSinkInterface<VideoFrame>，重写了 OnFrame(VideoFrame)
VideoSinkWrapper::VideoSinkWrapper(JNIEnv* jni, const JavaRef<jobject>& j_sink)
    : j_sink_(jni, j_sink) {}
```

2. 添加 VideoSink 到 VideoTrack 上

```
// out/android_arm64_Debug/gen/sdk/android/generated_video_jni/VideoTrack_jni.h
JNI_GENERATOR_EXPORT void Java_org_webrtc_VideoTrack_nativeAddSink(
    JNIEnv* env,
    jclass jcaller,
    jlong track,
    jlong nativeSink) {
    return JNI_VideoTrack_AddSink(env, track, nativeSink);
}

// sdk/android/src/jni/video_track.cc
```

```
static void JNI_VideoTrack_AddSink(JNIEnv* jni,
                                   jlong j_native_track,
                                   jlong j_native_sink) {
    reinterpret_cast<VideoTrackInterface*>(j_native_track)
        ->AddOrUpdateSink(
            reinterpret_cast<rtc::VideoSinkInterface<VideoFrame>*>(j_native_sink),
            rtc::VideoSinkWants());
}

// pc/video_track.cc
void VideoTrack::AddOrUpdateSink(rtc::VideoSinkInterface<VideoFrame>* sink,
                                 const rtc::VideoSinkWants& wants) {
    RTC_DCHECK(worker_thread_->IsCurrent());
    /* 基类保存 Sink */
    VideoSourceBase::AddOrUpdateSink(sink, wants);
    rtc::VideoSinkWants modified_wants = wants;

    // 是否已使能 VideoTrack，未使能的话用黑屏帧代替，因之前已设置 enable，所以此处 black_frames 为false
    modified_wants.black_frames = !enabled();
    // VideoSource 添加回调 Sink，即调用 AdaptedVideoTrackSource::AddOrUpdateSink()
    video_source_->AddOrUpdateSink(sink, modified_wants);
}

// media/base/video_source_base.cc
void VideoSourceBase::AddOrUpdateSink(
    VideoSinkInterface<webrtc::VideoFrame>* sink,
    const VideoSinkWants& wants) {
    RTC_DCHECK(sink != nullptr);

    SinkPair* sink_pair = FindSinkPair(sink);
    if (!sink_pair) {
        sinks_.push_back(SinkPair(sink, wants));
    } else {
        sink_pair->wants = wants;
    }
}

// 添加 VideoSink 到 VideoSource
// AndroidVideoTrackSource 继承于 AdaptedVideoTrackSource,
void AdaptedVideoTrackSource::AddOrUpdateSink(
    rtc::VideoSinkInterface<webrtc::VideoFrame>* sink,
    const rtc::VideoSinkWants& wants) {
    broadcaster_.AddOrUpdateSink(sink, wants);
    OnSinkWantsChanged(broadcaster_.wants());
}
```

3.4.4 添加视频编码 Sink

1. 创建本地 Offer Sdp；

```
// CallActivity.java
peerConnectionClient.createOffer();
```

2. 设置本地 Offer Sdp，会逐步创建 VideoMediaChannel, VideoSendStream, 添加 VideoStreamEncoder 到 VideoSource;

```
// PeerConnectionClient.java
// 设置本地 Offer Sdp 是在创建 Offer 成功的回调里面:
private class SDPObserver implements SdpObserver {
    @Override
    public void onCreateSuccess(final SessionDescription origSdp) {
      if (localSdp != null) {
        reportError("Multiple SDP create.");
        return;
      }
      String sdpDescription = origSdp.description;
      if (preferIsac) {
        sdpDescription = preferCodec(sdpDescription, AUDIO_CODEC_ISAC, true);
      }
      if (isVideoCallEnabled()) {
        sdpDescription =
            preferCodec(sdpDescription,
getSdpVideoCodecName(peerConnectionParameters), false);
      }
      final SessionDescription sdp = new SessionDescription(origSdp.type,
sdpDescription);
      localSdp = sdp;
      executor.execute(() -> {
        if (peerConnection != null && !isError) {
          Log.d(TAG, "Set local SDP from " + sdp.type);
          /* 设置 Offer Sdp */
          peerConnection.setLocalDescription(sdpObserver, sdp);
        }
      });
    }

    ...
}

// PeerConnection.java
public void setLocalDescription(SdpObserver observer, SessionDescription sdp) {
    /* 调用 Native 函数 */
    nativeSetLocalDescription(observer, sdp);
}
```

Native 层调用

```
// out/android_arm64_Debug/gen/sdk/android/generated_peerconnection_jni/PeerConnection_jni.h
JNI_GENERATOR_EXPORT void Java_org_webrtc_PeerConnection_nativeSetLocalDescription(
    JNIEnv* env,
    jobject jcaller,
    jobject observer,
    jobject sdp) {
  return JNI_PeerConnection_SetLocalDescription(env, base::android::JavaParamRef<jobject>(env,
      jcaller), base::android::JavaParamRef<jobject>(env, observer),
      base::android::JavaParamRef<jobject>(env, sdp));
}

// sdk/android/src/jni/pc/peer_connection.cc
static void JNI_PeerConnection_SetLocalDescription(
    JNIEnv* jni,
    const JavaParamRef<jobject>& j_pc,
    const JavaParamRef<jobject>& j_observer,
    const JavaParamRef<jobject>& j_sdp) {
  rtc::scoped_refptr<SetSdpObserverJni> observer(
      new rtc::RefCountedObject<SetSdpObserverJni>(jni, j_observer, nullptr));
  ExtractNativePC(jni, j_pc)->SetLocalDescription(
      observer, JavaToNativeSessionDescription(jni, j_sdp).release());
}

// pc/peer_connection.cc
void PeerConnection::SetLocalDescription(
    SetSessionDescriptionObserver* observer,
    SessionDescriptionInterface* desc_ptr) {
  RTC_DCHECK_RUN_ON(signaling_thread());
  // Chain this operation. If asynchronous operations are pending on the chain,
  // this operation will be queued to be invoked, otherwise the contents of the
  // lambda will execute immediately.
  /* 如上面注释所述，operations_chain_ 其实是一个半异步操作，内部是一个先进先出的队列，缓存着要执行的任务，
   * 但如果插入任务时刚好是第一个，则立即执行，否则，只做插入操作就返回，该次插入的任务等待前面任务完成后自动调用执行。
   */
  operations_chain_->ChainOperation(
      [this_weak_ptr = weak_ptr_factory_.GetWeakPtr(),
       observer_refptr =
           rtc::scoped_refptr<SetSessionDescriptionObserver>(observer),
       desc = std::unique_ptr<SessionDescriptionInterface>(desc_ptr)](
          std::function<void()> operations_chain_callback) mutable {
        // Abort early if |this_weak_ptr| is no longer valid.
```

```
        if (!this_weak_ptr) {
          // For consistency with DoSetLocalDescription(), we DO NOT inform the
          // |observer_refptr| that the operation failed in this case.
          // TODO(hbos): If/when we process SLD messages in ~PeerConnection,
          // the consistent thing would be to inform the observer here.
          // 通知 operations_chain_，当前任务已经执行完成，operations_chain_ 会从队列中删除该任务，并执行下一个任务
          operations_chain_callback();
          return;
        }
        /* 设置 Offer Sdp，此操作当前为同步执行，因为 operations_chain_ 内还没有其他任务 */
        this_weak_ptr->DoSetLocalDescription(std::move(desc),
                                             std::move(observer_refptr));
        // DoSetLocalDescription() is currently implemented as a synchronous
        // operation but where the |observer|'s callbacks are invoked
        // asynchronously in a post to OnMessage().
        // For backwards-compatability reasons, we declare the operation as
        // completed here (rather than in OnMessage()). This ensures that
        // subsequent offer/answer operations can start immediately (without
        // waiting for OnMessage()).
        operations_chain_callback();
      });
}

void PeerConnection::DoSetLocalDescription(
    std::unique_ptr<SessionDescriptionInterface> desc,
    rtc::scoped_refptr<SetSessionDescriptionObserver> observer) {

  /* 一堆校验，忽略 */
  ...

  // Grab the description type before moving ownership to ApplyLocalDescription,
  // which may destroy it before returning.
  const SdpType type = desc->GetType();

  error = ApplyLocalDescription(std::move(desc));
  // |desc| may be destroyed at this point.

  if (!error.ok()) {
    /* Local Sdp 设置失败，发送 MSG_SET_SESSIONDESCRIPTION_FAILED 消息，再通过 observer 回调上层 */
    ...

    return;
  }

  /* Local Sdp 设置成功，发送 MSG_SET_SESSIONDESCRIPTION_SUCCESS 消息，再通过 observer 回调上层 */
```

```
  PostSetSessionDescriptionSuccess(observer);

  // MaybeStartGathering needs to be called after posting
  // MSG_SET_SESSIONDESCRIPTION_SUCCESS, so that we don't signal any candidates
  // before signaling that SetLocalDescription completed.
  /* 开始 ICE 地址收集 */
  transport_controller_->MaybeStartGathering();

  ...
}

RTCError PeerConnection::ApplyLocalDescription(
    std::unique_ptr<SessionDescriptionInterface> desc) {

  ...

  /* 根据是否已存在 remote sdp 判断是否发起方 */
  if (!is_caller_) {
    if (remote_description()) {
      // Remote description was applied first, so this PC is the callee.
      is_caller_ = false;
    } else {
      // Local description is applied first, so this PC is the caller.
      is_caller_ = true;
    }
  }

  ...

  /* Android RTC demo 使用的是 Unified Plan */
  if (IsUnifiedPlan()) {
    /* 1. 创建 MediaChannel, 更新到收发器 */
    RTCError error = UpdateTransceiversAndDataChannels(
        cricket::CS_LOCAL, *local_description(), old_local_description,
        remote_description());
    if (!error.ok()) {
      return error;
    }

    ...

  } else {
    // Media channels will be created only when offer is set. These may use new
    // transports just created by PushdownTransportDescription.
    if (type == SdpType::kOffer) {
      // TODO(bugs.webrtc.org/4676) - Handle CreateChannel failure, as new local
      // description is applied. Restore back to old description.
      RTCError error = CreateChannels(*local_description()->description());
      if (!error.ok()) {
```

```
        return error;
      }
    }
    // Remove unused channels if MediaContentDescription is rejected.
    RemoveUnusedChannels(local_description()->description());
  }

  /* 2. 创建 VideoSendStream 或 AudioSendStream */
  error = UpdateSessionState(type, cricket::CS_LOCAL,
                             local_description()->description());
  if (!error.ok()) {
    return error;
  }

  ...

  if (IsUnifiedPlan()) {
    for (const auto& transceiver : transceivers_) {
      const ContentInfo* content =
          FindMediaSectionForTransceiver(transceiver, local_description());
      if (!content) {
        continue;
      }
      cricket::ChannelInterface* channel = transceiver->internal()->channel();
      if (content->rejected || !channel || channel->local_streams().empty()) {
        // 0 is a special value meaning "this sender has no associated send
        // stream". Need to call this so the sender won't attempt to configure
        // a no longer existing stream and run into DCHECKs in the lower
        // layers.
        transceiver->internal()->sender_internal()->SetSsrc(0);
      } else {
        // Get the StreamParams from the channel which could generate SSRCs.
        const std::vector<StreamParams>& streams = channel->local_streams();
        transceiver->internal()->sender_internal()->set_stream_ids(
            streams[0].stream_ids());

        /* 3. 设置 RtpSender 的 ssrc, 同时触发添加 VideoStreamEncoder 到 VideoSource */
        transceiver->internal()->sender_internal()->SetSsrc(
            streams[0].first_ssrc());
      }
    }
  } else {
    // Plan B semantics.

    // Update state and SSRC of local MediaStreams and DataChannels based on the
    // local session description.
    const cricket::ContentInfo* audio_content =
        GetFirstAudioContent(local_description()->description());
```

```
    if (audio_content) {
      if (audio_content->rejected) {
        RemoveSenders(cricket::MEDIA_TYPE_AUDIO);
      } else {
        const cricket::AudioContentDescription* audio_desc =
            audio_content->media_description()->as_audio();
        UpdateLocalSenders(audio_desc->streams(), audio_desc->type());
      }
    }

    const cricket::ContentInfo* video_content =
        GetFirstVideoContent(local_description()->description());
    if (video_content) {
      if (video_content->rejected) {
        RemoveSenders(cricket::MEDIA_TYPE_VIDEO);
      } else {
        const cricket::VideoContentDescription* video_desc =
            video_content->media_description()->as_video();
        UpdateLocalSenders(video_desc->streams(), video_desc->type());
      }
    }
  }

  ...

  return RTCError::OK();
}

/*********************************************************/
/* 1. 创建 MediaChannel, 更新到收发器, 限于 Unified Plan 模式 */
RTCError PeerConnection::UpdateTransceiversAndDataChannels(
    cricket::ContentSource source,
    const SessionDescriptionInterface& new_session,
    const SessionDescriptionInterface* old_local_description,
    const SessionDescriptionInterface* old_remote_description) {

  ...

  const ContentInfos& new_contents = new_session.description()->contents();
  for (size_t i = 0; i < new_contents.size(); ++i) {
    const cricket::ContentInfo& new_content = new_contents[i];
    cricket::MediaType media_type = new_content.media_description()->type();
    mid_generator_.AddKnownId(new_content.name);
    if (media_type == cricket::MEDIA_TYPE_AUDIO ||
        media_type == cricket::MEDIA_TYPE_VIDEO) {

      ...

```

```
      /* sdp 与 收发器 Transceiver 进行关联(通过 mline_index)，更新 Transceiver 的
mid，返回 Transceiver. */
      auto transceiver_or_error =
          AssociateTransceiver(source, new_session.GetType(), i, new_content,
                               old_local_content, old_remote_content);
      if (!transceiver_or_error.ok()) {
        return transceiver_or_error.MoveError();
      }
      auto transceiver = transceiver_or_error.MoveValue();
      RTCError error =
          UpdateTransceiverChannel(transceiver, new_content, bundle_group);
      if (!error.ok()) {
        return error;
      }
    } else if (media_type == cricket::MEDIA_TYPE_DATA) {
      ...
    } else {
      LOG_AND_RETURN_ERROR(RTCErrorType::INTERNAL_ERROR,
                           "Unknown section type.");
    }
  }

  return RTCError::OK();
}

/* 创建及关联 MediaChannel */
RTCError PeerConnection::UpdateTransceiverChannel(
    rtc::scoped_refptr<RtpTransceiverProxyWithInternal<RtpTransceiver>>
        transceiver,
    const cricket::ContentInfo& content,
    const cricket::ContentGroup* bundle_group) {
  RTC_DCHECK(IsUnifiedPlan());
  RTC_DCHECK(transceiver);
  cricket::ChannelInterface* channel = transceiver->internal()->channel();
  if (content.rejected) {
    if (channel) {
      transceiver->internal()->SetChannel(nullptr);
      DestroyChannelInterface(channel);
    }
  } else {
    if (!channel) {
      if (transceiver->media_type() == cricket::MEDIA_TYPE_AUDIO) {
        /* 创建音频 Channel */
        channel = CreateVoiceChannel(content.name);
      } else {
        /* 创建视频 Channel */
        channel = CreateVideoChannel(content.name);
      }
      if (!channel) {
```

```
        LOG_AND_RETURN_ERROR(
            RTCErrorType::INTERNAL_ERROR,
            "Failed to create channel for mid=" + content.name);
      }

      /* 设置 Channel 到收发器内，并将 MediaChannel 设置到收发器内的每个 RtpSender 和
RtpReceiver */
      transceiver->internal()->SetChannel(channel);
    }
  }
  return RTCError::OK();
}

/* 创建 VideoChannel，保存着 MediaChannel */
cricket::VideoChannel* PeerConnection::CreateVideoChannel(
    const std::string& mid) {
  /* Rtp/Rtcp 数据包发送接口 */
  RtpTransportInternal* rtp_transport = GetRtpTransport(mid);
  /* rtp 包最大包长配置 */
  MediaTransportConfig media_transport_config =
      transport_controller_->GetMediaTransportConfig(mid);

  /* 创建视频 VideoChannel */
  cricket::VideoChannel* video_channel = channel_manager()->CreateVideoChannel(
      call_ptr_, configuration_.media_config, rtp_transport,
      media_transport_config, signaling_thread(), mid, SrtpRequired(),
      GetCryptoOptions(), &ssrc_generator_, video_options_,
      video_bitrate_allocator_factory_.get());
  if (!video_channel) {
    return nullptr;
  }

  video_channel->SignalDtlsSrtpSetupFailure.connect(
      this, &PeerConnection::OnDtlsSrtpSetupFailure);
  video_channel->SignalSentPacket.connect(this,
                                          &PeerConnection::OnSentPacket_w);
  video_channel->SetRtpTransport(rtp_transport);

  return video_channel;
}

// pc/channel_manager.cc
// 创建 VideoMediaChannel, 基类是 MediaChannel
VideoChannel* ChannelManager::CreateVideoChannel(
    webrtc::Call* call,
    const cricket::MediaConfig& media_config,
    webrtc::RtpTransportInternal* rtp_transport,
    const webrtc::MediaTransportConfig& media_transport_config,
    rtc::Thread* signaling_thread,
```

```
    const std::string& content_name,
    bool srtp_required,
    const webrtc::CryptoOptions& crypto_options,
    rtc::UniqueRandomIdGenerator* ssrc_generator,
    const VideoOptions& options,
    webrtc::VideoBitrateAllocatorFactory* video_bitrate_allocator_factory) {

  ...

  /* 通过 WebRtcVideoEngine 创建 WebRtcVideoChannel, 基类是 VideoMediaChannel */
  VideoMediaChannel* media_channel = media_engine_->video().CreateMediaChannel(
      call, media_config, options, crypto_options,
      video_bitrate_allocator_factory);
  if (!media_channel) {
    return nullptr;
  }

  /* 创建 VideoChannel, 保存 media_channel 到 VideoChannel 的基类 BaseChannel 中 */
  auto video_channel = std::make_unique<VideoChannel>(
      worker_thread_, network_thread_, signaling_thread,
      absl::WrapUnique(media_channel), content_name, srtp_required,
      crypto_options, ssrc_generator);

  /* 初始化 VideoChannel, 实际是调用基类 BaseChannel::Init_w(), 设置 media_channel 的 Rtp 发送接口 */
  video_channel->Init_w(rtp_transport, media_transport_config);
    ---> BaseChannel::Init_w()
      --> WebRtcVideoChannel::SetInterface()
        --> MediaChannel::SetInterface(iface, media_transport_config)

  /* 保存 VideoChannel */
  VideoChannel* video_channel_ptr = video_channel.get();
  video_channels_.push_back(std::move(video_channel));
  return video_channel_ptr;
}

/* 设置 Channel */
void RtpTransceiver::SetChannel(cricket::ChannelInterface* channel) {
  // Cannot set a non-null channel on a stopped transceiver.
  if (stopped_ && channel) {
    return;
  }

  ...

  channel_ = channel;

  if (channel_) {
    channel_->SignalFirstPacketReceived().connect(
```

```
        this, &RtpTransceiver::OnFirstPacketReceived);
  }

  for (const auto& sender : senders_) {
    sender->internal()->SetMediaChannel(channel_ ? channel_->media_channel()
                                                 : nullptr);
  }

  for (const auto& receiver : receivers_) {
    if (!channel_) {
      receiver->internal()->Stop();
    }

    receiver->internal()->SetMediaChannel(channel_ ? channel_->media_channel()
                                                   : nullptr);
  }
}

/**********************************************/
/* 2. 创建 VideoSendStream 或 AudioSendStream */

RTCError PeerConnection::UpdateSessionState(
    SdpType type,
    cricket::ContentSource source,
    const cricket::SessionDescription* description) {

  ...

  // Update internal objects according to the session description's media
  // descriptions.
  RTCError error = PushdownMediaDescription(type, source);
  if (!error.ok()) {
    return error;
  }

  return RTCError::OK();
}

RTCError PeerConnection::PushdownMediaDescription(
    SdpType type,
    cricket::ContentSource source) {
  const SessionDescriptionInterface* sdesc =
      (source == cricket::CS_LOCAL ? local_description()
                                   : remote_description());
  RTC_DCHECK(sdesc);

  // Push down the new SDP media section for each audio/video transceiver.
  for (const auto& transceiver : transceivers_) {
    const ContentInfo* content_info =
```

```
        FindMediaSectionForTransceiver(transceiver, sdesc);
    cricket::ChannelInterface* channel = transceiver->internal()->channel();

    ...

    std::string error;
    bool success = (source == cricket::CS_LOCAL)
                       ? channel->SetLocalContent(content_desc, type, &error)
                       : channel->SetRemoteContent(content_desc, type, &error);
    if (!success) {
      LOG_AND_RETURN_ERROR(RTCErrorType::INVALID_PARAMETER, error);
    }
  }

  ...

  return RTCError::OK();
}

bool VideoChannel::SetLocalContent_w(const MediaContentDescription* content,
                                     SdpType type,
                                     std::string* error_desc) {

  ...

  if (!media_channel()->SetRecvParameters(recv_params)) {
    SafeSetError("Failed to set local video description recv parameters.",
                 error_desc);
    return false;
  }

  ...

  last_recv_params_ = recv_params;

  if (needs_send_params_update) {
    if (!media_channel()->SetSendParameters(send_params)) {
      SafeSetError("Failed to set send parameters.", error_desc);
      return false;
    }
    last_send_params_ = send_params;
  }

  // TODO(pthatcher): Move local streams into VideoSendParameters, and
  // only give it to the media channel once we have a remote
  // description too (without a remote description, we won't be able
  // to send them anyway).
  if (!UpdateLocalStreams_w(video->streams(), type, error_desc)) {
    SafeSetError("Failed to set local video description streams.", error_desc);
```

```
    return false;
  }

  set_local_content_direction(content->direction());
  UpdateMediaSendRecvState_w();
  return true;
}

bool BaseChannel::UpdateLocalStreams_w(const std::vector<StreamParams>& streams,
                                       SdpType type,
                                       std::string* error_desc) {
  // In the case of RIDs (where SSRCs are not negotiated), this method will
  // generate an SSRC for each layer in StreamParams. That representation will
  // be stored internally in |local_streams_|.
  // In subsequent offers, the same stream can appear in |streams| again
  // (without the SSRCs), so it should be looked up using RIDs (if available)
  // and then by primary SSRC.
  // In both scenarios, it is safe to assume that the media channel will be
  // created with a StreamParams object with SSRCs. However, it is not safe to
  // assume that |local_streams_| will always have SSRCs as there are scenarios
  // in which niether SSRCs or RIDs are negotiated.

  // Check for streams that have been removed.
  bool ret = true;

  ...

  // Check for new streams.
  std::vector<StreamParams> all_streams;
  for (const StreamParams& stream : streams) {
    StreamParams* existing = GetStream(local_streams_, StreamFinder(&stream));
    if (existing) {
      // Parameters cannot change for an existing stream.
      all_streams.push_back(*existing);
      continue;
    }

    all_streams.push_back(stream);
    StreamParams& new_stream = all_streams.back();

    ...

    // At this point we use the legacy simulcast group in StreamParams to
    // indicate that we want multiple layers to the media channel.
    if (!new_stream.has_ssrcs()) {
      // TODO(bugs.webrtc.org/10250): Indicate if flex is desired here.
      new_stream.GenerateSsrcs(new_stream.rids().size(), /* rtx = */ true,
                               /* flex_fec = */ false, ssrc_generator_);
    }
```

```

    /* 创建 VideoSendStream */
    media_channel()->AddSendStream(new_stream);
    --> WebRtcVideoChannel::AddSendStream(const StreamParams& sp) //
webrtc_video_engine.cc
      --> WebRtcVideoChannel::WebRtcVideoSendStream::WebRtcVideoSendStream()
        --> WebRtcVideoChannel::WebRtcVideoSendStream::SetCodec()
          --> WebRtcVideoChannel::WebRtcVideoSendStream::RecreateWebRtcStream()
            --> webrtc::VideoSendStream* Call::CreateVideoSendStream() //
call/call.cc
              --> VideoSendStream::VideoSendStream() // video/video_send_stream.cc
                --> video_stream_encoder_ = CreateVideoStreamEncoder()
  }

  local_streams_ = all_streams;
  return ret;
}

/**********************************************************************/
/* 3. 设置 RtpSender 的 ssrc, 同时触发添加 VideoStreamEncoder 到 VideoSource */
// pc/rtp_sender.cc

void RtpSenderBase::SetSsrc(uint32_t ssrc) {
  TRACE_EVENT0("webrtc", "RtpSenderBase::SetSsrc");
  if (stopped_ || ssrc == ssrc_) {
    return;
  }
  // If we are already sending with a particular SSRC, stop sending.
  if (can_send_track()) {
    ClearSend();
    RemoveTrackFromStats();
  }
  ssrc_ = ssrc;
  // can_send_track() 判断 track_ 及 ssrc_ 是否存在。此时已满足条件。
  if (can_send_track()) {
    // 调用子类的 SetSend()
    SetSend();
    AddTrackToStats();
  }

  ...
}


//
void VideoRtpSender::SetSend() {
  ...

  cricket::VideoOptions options;
```

```
  /* 获取 VideoSource, 该创建详见 */
  VideoTrackSourceInterface* source = video_track()->GetSource();
  if (source) {
    options.is_screencast = source->is_screencast();
    options.video_noise_reduction = source->needs_denoising();
  }
  options.content_hint = cached_track_content_hint_;
  switch (cached_track_content_hint_) {
    case VideoTrackInterface::ContentHint::kNone:
      break;
    case VideoTrackInterface::ContentHint::kFluid:
      options.is_screencast = false;
      break;
    case VideoTrackInterface::ContentHint::kDetailed:
    case VideoTrackInterface::ContentHint::kText:
      options.is_screencast = true;
      break;
  }

  /* 设置 VideoSource, 将 VideoStreamEncoder 作为 VideoSink 添加到 VideoSource */
  bool success = worker_thread_->Invoke<bool>(RTC_FROM_HERE, [&] {
    return video_media_channel()->SetVideoSend(ssrc_, &options, video_track());
  });
  RTC_DCHECK(success);
}

// media/engine/webrt_video_engine.cc
bool WebRtcVideoChannel::SetVideoSend(
    uint32_t ssrc,
    const VideoOptions* options,
    rtc::VideoSourceInterface<webrtc::VideoFrame>* source) {

  ...

  // 查找 VideoSendStream
  const auto& kv = send_streams_.find(ssrc);
  if (kv == send_streams_.end()) {
    // Allow unknown ssrc only if source is null.
    RTC_CHECK(source == nullptr);
    RTC_LOG(LS_ERROR) << "No sending stream on ssrc " << ssrc;
    return false;
  }

  return kv->second->SetVideoSend(options, source);
}

bool WebRtcVideoChannel::WebRtcVideoSendStream::SetVideoSend(
    const VideoOptions* options,
    rtc::VideoSourceInterface<webrtc::VideoFrame>* source) {
```

```

  ...

  /* WebRtcVideoSendStream 创建时 source_ 为空，所以在创建时和这里都不会调用 SetSource */
  if (source_ && stream_) {
    stream_->SetSource(nullptr, webrtc::DegradationPreference::DISABLED);
  }
  // Switch to the new source.
  source_ = source;
  /* source_ 不为空，及VideoSendStream 已经创建，调用 SetSource() */
  if (source && stream_) {
    // 此处参数 this 为 WebRtcVideoSendStream，即后面流程添加 Sink 的 source
    stream_->SetSource(this, GetDegradationPreference());
    --> VideoSendStream::SetSource(source, degradation_preference) // video/video_send_stream.cc
      --> VideoStreamEncoder::SetSource(source, degradation_preference) // video/video_stream_encoder.cc
        --> VideoSourceSinkController::SetSource(source, degradation_preference) // video/video_source_sink_controller.cc
              // 此处 sink_ 为 VideoStreamEncoder， 是 VideoSourceSinkController 在 VideoStreamEncoder 创建时传入
          --> source->AddOrUpdateSink(sink_, wants);
            --> WebRtcVideoChannel::WebRtcVideoSendStream::AddOrUpdateSink(sink, wants) // media/engine/webrt_video_engine.cc
              --> encoder_sink_ = sink; // 保存 sink
                  // 此处 source_ 实为 VideoTrack，后续流程与添加视频预览 sink 基本一致。可回看上一节 《3.4.3》。
              --> source_->AddOrUpdateSink(encoder_sink_, wants)
                --> VideoTrack::AddOrUpdateSink(sink, wants)
                  --> video_source_->AddOrUpdateSink(sink, modified_wants)
  }
  return true;
}


```

3.5 摄像头视频数据流

```
(Camera2 或 Camera1 使能 captureToTexture) (SurfaceTextureHelper.java) SurfaceTextureHelper.listener.onFrame() -->
Camera1Session/Camera2Session --> (CameraCapturer.java) CameraSession.Events.onFrameCaptured()
--> (VideoSource.java) CapturerObserver.onFrameCaptured()
--> (NativeAndroidVideoTrackSource.java) NativeAndroidVideoTrackSource.onFrameCaptured()
--> (android_video_track_source.cc) AndroidVideoTrackSource.onFrameCaptured()
--> (adapted_video_track_source.cc) AdaptedVideoTrackSource::OnFrame() --> broadcaster_.OnFrame(frame)
```

```
// 分支1: 视频预览
--> (sdk/android/src/jni/video_sink.cc) VideoSinkWrapper:::OnFrame(VideoFrame)
--> (gen/sdk/android/generated_video_jni/VideoSink_jni.h)
Java_VideoSink_onFrame(VideoFrame) // 映射到 Java 层 VideoFrame
--> (CallActivity.java) ProxyVideoSink::onFrame() --> target.onFrame(VideoFrame)
--> (SurfaceViewRenderer.java) SurfaceViewRenderer::onFrame(VideoFrame)

// 分支2: 视频编码
--> (video/video_stream_encoder.cc) VideoStreamEncoder::OnFrame(VideoFrame)
```

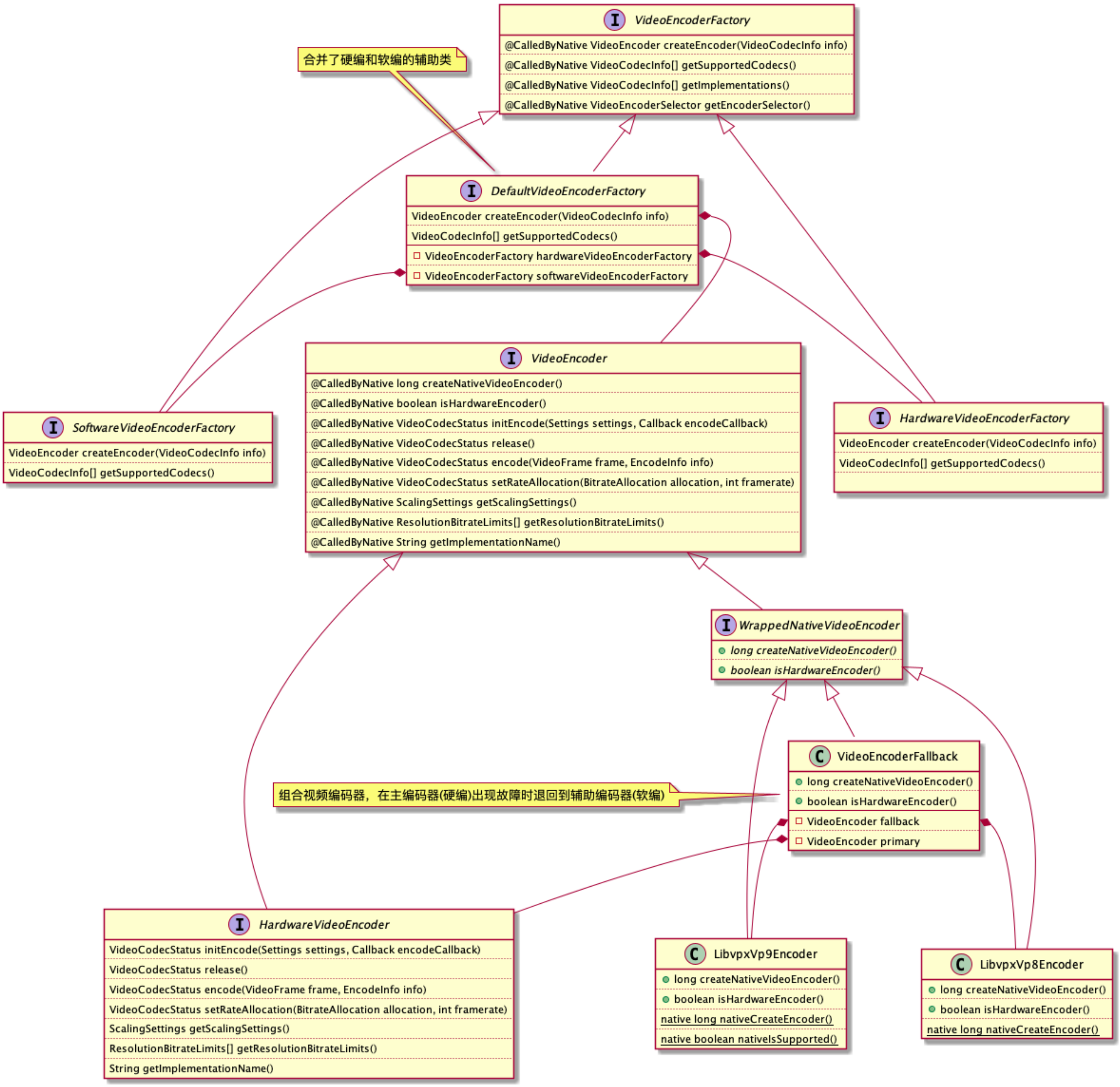
4 视频编解码器

```
// examples/androidapp/src/org/appspot/apprtc/PeerConnectionClient.java
private void createPeerConnectionFactoryInternal(PeerConnectionFactory.Options
options) {
  ...

  /* 创建视频编解码工厂, 软编或硬编 */
  if (peerConnectionParameters.videoCodecHwAcceleration) {
    /* 硬编+软编 */
    encoderFactory = new DefaultVideoEncoderFactory(
      rootEglBase.getEglBaseContext(), true /* enableIntelVp8Encoder */,
enableH264HighProfile);
    /* 硬解 + 软解 */
    decoderFactory = new
DefaultVideoDecoderFactory(rootEglBase.getEglBaseContext());
  } else {
    /* 软编 */
    encoderFactory = new SoftwareVideoEncoderFactory();
    /* 软解 */
    decoderFactory = new SoftwareVideoDecoderFactory();
  }
  /** 创建 PeerConnectionFactory */
  factory = PeerConnectionFactory.builder()
                .setOptions(options)
                .setAudioDeviceModule(adm)
                .setVideoEncoderFactory(encoderFactory)
                .setVideoDecoderFactory(decoderFactory)
                .createPeerConnectionFactory();
    Log.d(TAG, "Peer connection factory created.");
    ...
}
```

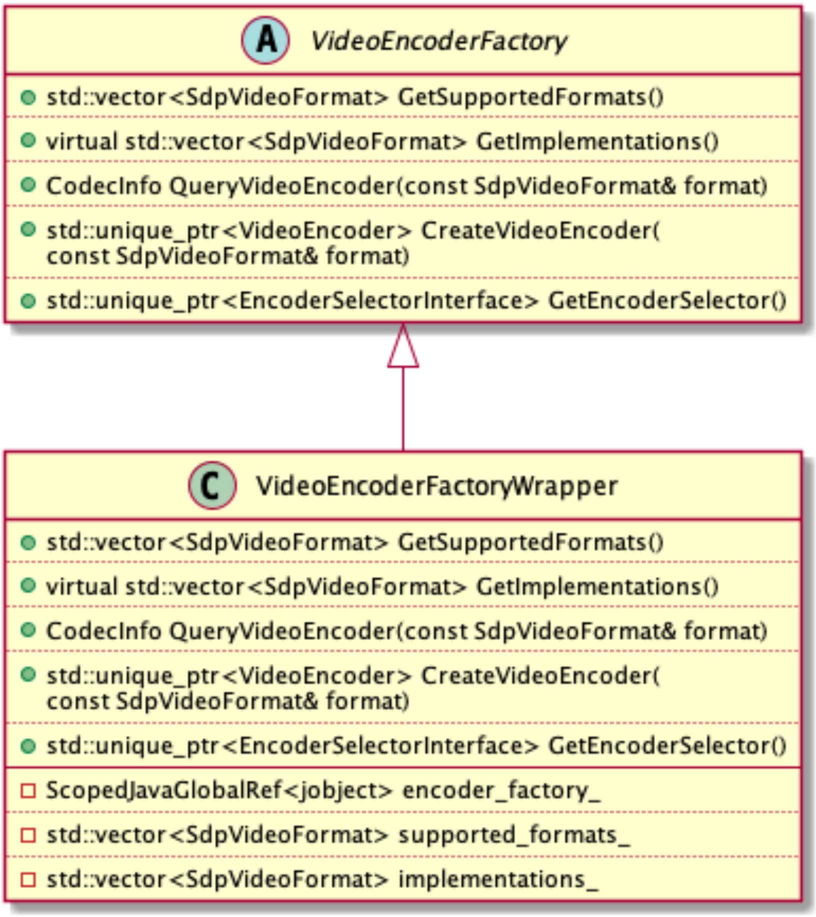
4.1 视频编码器

4.1.1 类图

Android 端视频编码器工厂类为 DefaultVideoEncoderFactory/SoftwareVideoEncoderFactory，其中 DefaultVideoEncoderFactory 创建视频硬编+软编，而 SoftwareVideoEncoderFactory 为VP8或VP9软编。Java 层视频编码器工厂类会传入到Native 层，在Native层通过相应的 VideoEncoderFactoryWrapper 类保存，该 VideoEncoderFactoryWrapper 类将作为Native层 PeerConnectionFactory 的外部视频编码器工厂类，之后视频编码器的创建将通过该 Wrapper 类回调Java层工厂类实例接口完成。



Java 层视频编码器类图



Native 层视频编码器工厂类图

4.1.2 编码器工厂创建

Java 层代码:

```java
// sdk/android/api/org/webrtc/DefaultVideoEncoderFactory.java
/** 默认创建视频软编工厂类 */
private final VideoEncoderFactory softwareVideoEncoderFactory = new SoftwareVideoEncoderFactory();
/** Create encoder factory using default hardware encoder factory. */
public DefaultVideoEncoderFactory(
    EglBase.Context eglContext, boolean enableIntelVp8Encoder, boolean enableH264HighProfile) {
    /** 创建视频硬编工厂类 */
    this.hardwareVideoEncoderFactory =
        new HardwareVideoEncoderFactory(eglContext, enableIntelVp8Encoder, enableH264HighProfile);
}
```

Native 层代码:

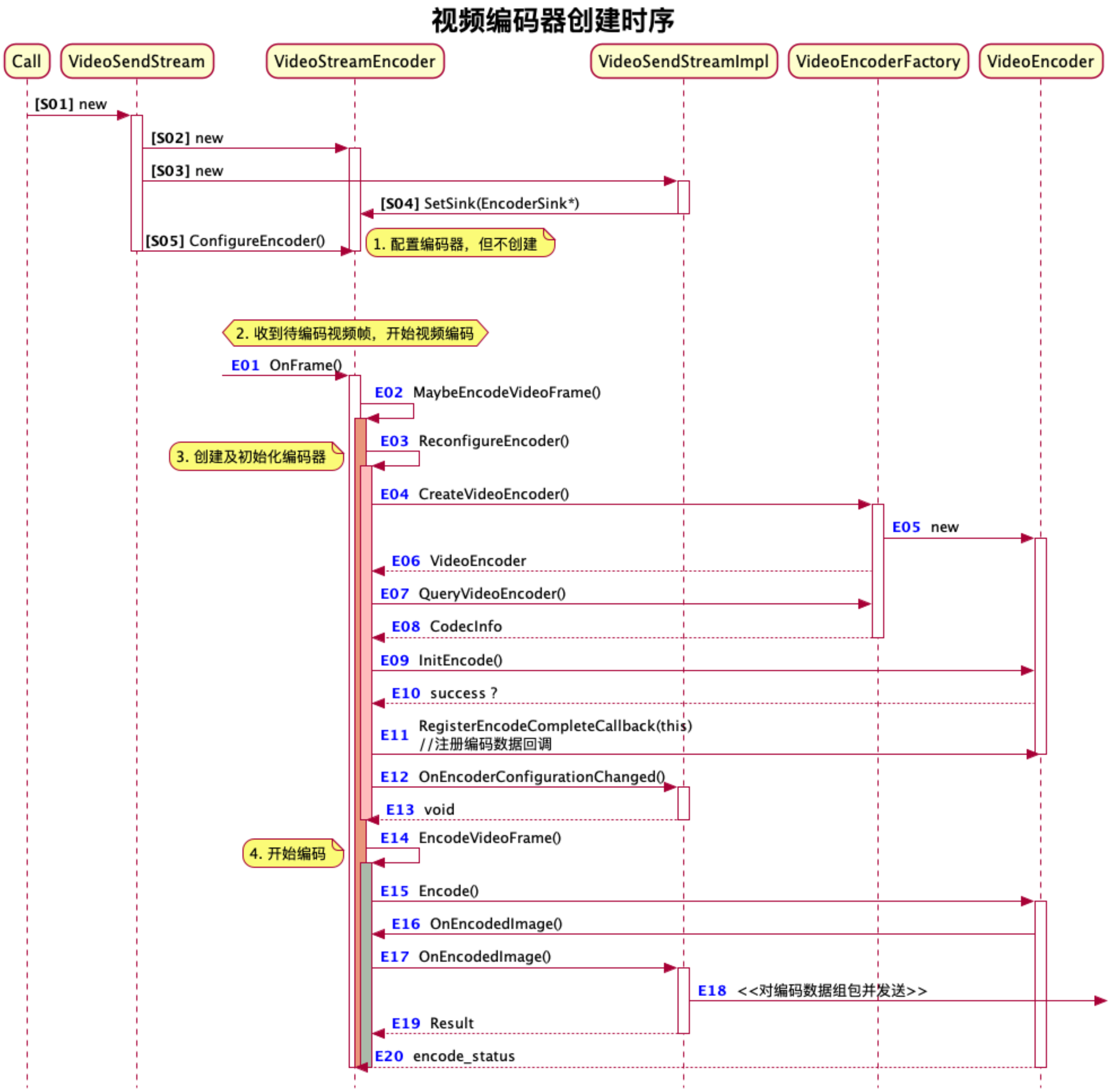
```cpp
// sdk/android/src/jni/pc/peer_connection_factory.cc
static ScopedJavaLocalRef<jobject>
JNI_PeerConnectionFactory_CreatePeerConnectionFactory(
```

```
    JNIEnv* jni,
    const JavaParamRef<jobject>& jcontext,
    const JavaParamRef<jobject>& joptions,
    jlong native_audio_device_module,
    jlong native_audio_encoder_factory,
    jlong native_audio_decoder_factory,
    const JavaParamRef<jobject>& jencoder_factory,
    const JavaParamRef<jobject>& jdecoder_factory,
    jlong native_audio_processor,
    jlong native_fec_controller_factory,
    jlong native_network_controller_factory,
    jlong native_network_state_predictor_factory,
    jlong native_media_transport_factory,
    jlong native_neteq_factory) {
  ...

  // 创建 Native 层视频编码及解码器工厂类
  // jencoder_factory 为 Java 层视频编码器工厂类实例
  media_dependencies.video_encoder_factory =
      absl::WrapUnique(CreateVideoEncoderFactory(jni, jencoder_factory));
  // jdecoder_factory 为Java层视频解码器工厂类实例
  media_dependencies.video_decoder_factory =
      absl::WrapUnique(CreateVideoDecoderFactory(jni, jdecoder_factory));

  ...

}

// sdk/android/src/jni/pc/video.cc
VideoEncoderFactory* CreateVideoEncoderFactory(
    JNIEnv* jni,
    const JavaRef<jobject>& j_encoder_factory) {
  return IsNull(jni, j_encoder_factory)
             ? nullptr
              /* 创建视频编码器工厂 wrapper 类 */
             : new VideoEncoderFactoryWrapper(jni, j_encoder_factory);
}

// sdk/android/src/jni/video_encoder_factory_wrapper.cc
VideoEncoderFactoryWrapper::VideoEncoderFactoryWrapper(
    JNIEnv* jni,
    const JavaRef<jobject>& encoder_factory)
    // 保存 Java 层视频编码器工厂实例
    : encoder_factory_(jni, encoder_factory) {
  // 获取支持的视频编码器
  const ScopedJavaLocalRef<jobjectArray> j_supported_codecs =
      Java_VideoEncoderFactory_getSupportedCodecs(jni, encoder_factory);
  // 将 java层数组转换成Native层Vector
  supported_formats_ = JavaToNativeVector<SdpVideoFormat>(
```

```
        jni, j_supported_codecs, &VideoCodecInfoToSdpVideoFormat);
    // 获取 支持的编码器
    const ScopedJavaLocalRef<jobjectArray> j_implementations =
        Java_VideoEncoderFactory_getImplementations(jni, encoder_factory);
    implementations_ = JavaToNativeVector<SdpVideoFormat>(
        jni, j_implementations, &VideoCodecInfoToSdpVideoFormat);
}
```

4.1.3 创建视频编码器

根据代码流程，视频编码器是在第一帧视频编码的时候创建的，而不是在创建视频流的时候。



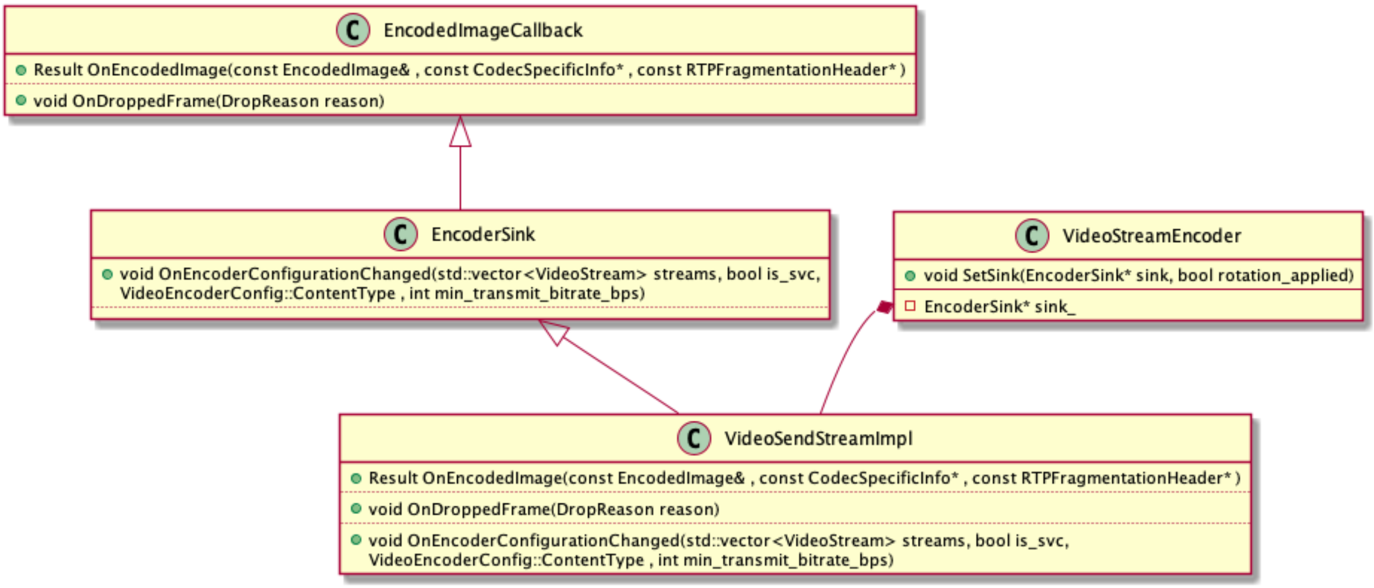
1. 创建视频流编码器；

```
// video/video_send_stream.cc
```

```
VideoSendStream::VideoSendStream(
  ...
  VideoSendStream::Config config,
  VideoEncoderConfig encoder_config,
  ...
) {
  ...

  // 1. 创建视频流编码器
  video_stream_encoder_ =
      CreateVideoStreamEncoder(clock, task_queue_factory, num_cpu_cores,
                               &stats_proxy_, config_.encoder_settings);
  ...

  // 2. 设置 VideoSendStreamImpl 为视频编码后的数据接收端
  send_stream_.reset(new VideoSendStreamImpl(
            ...
            video_stream_encoder_.get(),
            ...
            );

  ...
  // 3. 配置视频编码器，但条件不足，不执行创建
  ReconfigureVideoEncoder(std::move(encoder_config));
}

void VideoSendStream::ReconfigureVideoEncoder(VideoEncoderConfig config) {
  ...
  video_stream_encoder_->ConfigureEncoder(
      std::move(config),
      config_.rtp.max_packet_size - CalculateMaxHeaderSize(config_.rtp));
}
```

2. 设置视频编码数据接收端。*VideoSendStreamImpl* 继承了编码数据回调的类 *VideoStreamEncoderInterface::EncoderSink* 。

```
// video/video_send_stream_impl.cc
VideoSendStreamImpl::VideoSendStreamImpl(
...
VideoStreamEncoderInterface* video_stream_encoder,
...
){
  ...
  video_stream_encoder_->SetSink(this, rotation_applied);
  ...
}
```



3. 配置编码器

```
// video/video_stream_encoder.cc
VideoStreamEncoder::VideoStreamEncoder(
    ...
    const VideoStreamEncoderSettings& settings,
    ...)
  : ...
    settings_(settings), // settings_ 内保存了 VideoEncoderFactory 工厂类对象, 用于创建编码器
  ... {
  ...
}

void VideoStreamEncoder::ConfigureEncoder(VideoEncoderConfig config,
                                          size_t max_data_payload_length) {
  encoder_queue_.PostTask(
      [this, config = std::move(config), max_data_payload_length]() mutable {
        ...

        // 因为尚未创建 encoder_, 所以 pending_encoder_creation_ = true
        pending_encoder_creation_ =
            (!encoder_ || encoder_config_.video_format != config.video_format ||
             max_data_payload_length_ != max_data_payload_length);
        // 保存编码器配置
        encoder_config_ = std::move(config);
        max_data_payload_length_ = max_data_payload_length;
        // 等待重新配置 encoder
        pending_encoder_reconfiguration_ = true;

```

```
        // Reconfigure the encoder now if the encoder has an internal source or
        // if the frame resolution is known. Otherwise, the reconfiguration is
        // deferred until the next frame to minimize the number of
        // reconfigurations. The codec configuration depends on incoming video
        // frame size.
        // last_frame_info_ 当前不存在，为空，故不执行 ReconfigureEncoder()
        if (last_frame_info_) {
          ReconfigureEncoder();
        } else {
          codec_info_ = settings_.encoder_factory->QueryVideoEncoder(
              encoder_config_.video_format);
          // HasInternalSource() 当前恒为false，故也不执行 ReconfigureEncoder()
          if (HasInternalSource()) {
            last_frame_info_ = VideoFrameInfo(kDefaultInputPixelsWidth,
                                              kDefaultInputPixelsHeight, false);
            ReconfigureEncoder();
          }
        }
      });
}



```

4. 创建编码器并初始化

```
// video/video_stream_encoder.cc
void VideoStreamEncoder::ReconfigureEncoder() {
  ...
  // pending_encoder_creation_ 已在 ConfigureEncoder() 中设置为 true
  if (pending_encoder_creation_) {
    // Destroy existing encoder instance before creating a new one. Otherwise
    // attempt to create another instance will fail if encoder factory
    // supports only single instance of encoder of given type.
    encoder_.reset();

    // 通过编码器工厂类创建指定 format 的编码器
    encoder_ = settings_.encoder_factory->CreateVideoEncoder(
        encoder_config_.video_format);

    ...

```

```
    // 获取编码器参数
    codec_info_ = settings_.encoder_factory->QueryVideoEncoder(
        encoder_config_.video_format);

    encoder_reset_required = true;
  }

  ...

   bool success = true;
  if (encoder_reset_required) {
    // 若之前已初始化同一编码器，需先去初始化编码器
    ReleaseEncoder();
    const size_t max_data_payload_length = max_data_payload_length_ > 0
                                           ? max_data_payload_length_
                                           : kDefaultPayloadSize;
    // 初始化编码器
    if (encoder_->InitEncode(
            &send_codec_,
            VideoEncoder::Settings(settings_.capabilities,
number_of_cores_,
                                   max_data_payload_length)) != 0) {
      RTC_LOG(LS_ERROR) << "Failed to initialize the encoder associated
with "
                           "codec type: "
                        << CodecTypeToPayloadString(send_codec_.codecType)
                        << " (" << send_codec_.codecType << ")";
      ReleaseEncoder();
      success = false;
    } else {
      encoder_initialized_ = true;
      // 注册编码数据回调为当前 VideoStreamEncoder 对象，编码完成后，
      // 将回调 EncodedImageCallback::Result
VideoStreamEncoder::OnEncodedImage() 接口
      encoder_->RegisterEncodeCompleteCallback(this);
      frame_encode_metadata_writer_.OnEncoderInit(send_codec_,
                                                  HasInternalSource());
    }

    ...
  }

  ...
}
```

4.2 视频解码器

4.2.1 类图

4.2.2 解码器工厂类创建