Linphone 分析

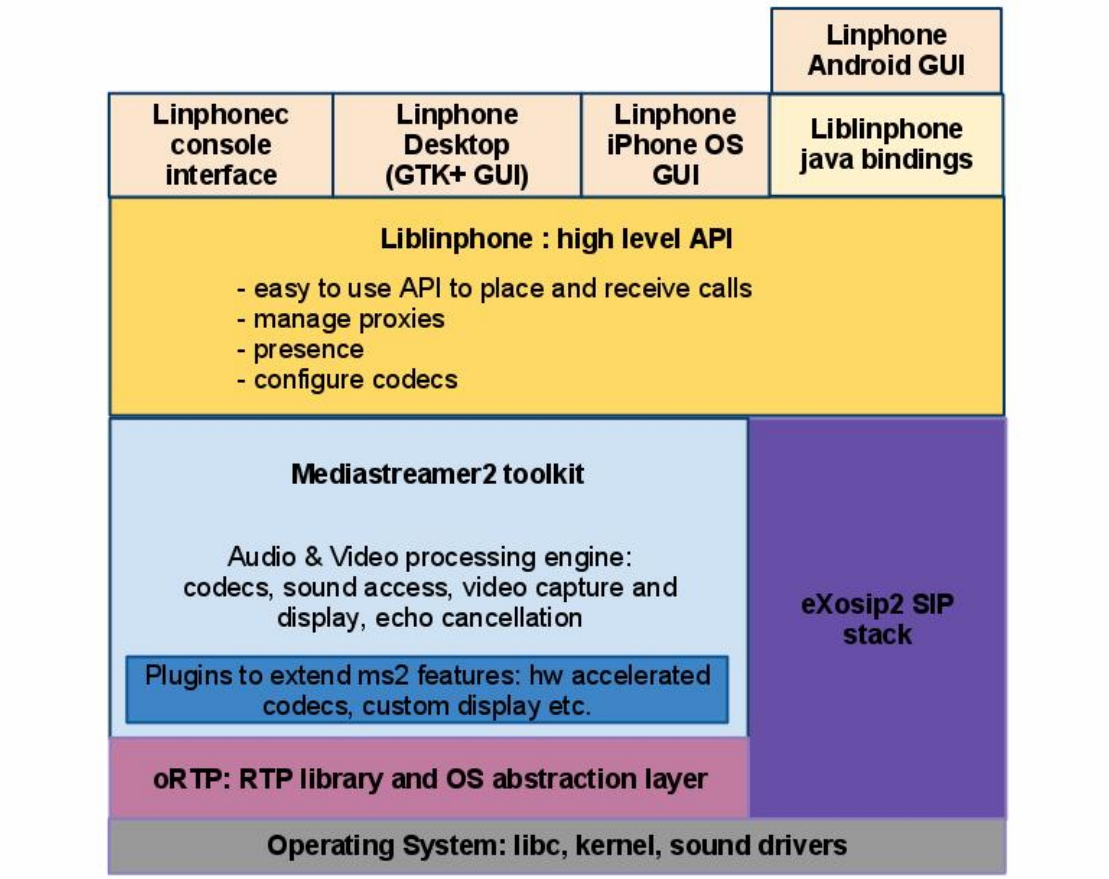
基于 linphone-3.3.2 版本，新版本 linphone-3.4.3 支持同时有多路 call，所以，相比之前版本会有不少变化。

一 linphone 架构及组成模块

Linphone 是一款跨平台的可视电话客户端软件，同时支持视频通话功能。Linphone 可以在 Linux，windows 等主流操作系统平台上运行。

Linphone 基于开源软件构建，本身也是开源软件。Linphone 架构中 sip 协议的处理基于 osip 以及 exosip 两个开源库实现，媒体数据的选择整合处理使用 mediastream2 完成，该软件使用 ffmepg、speedx 等多款开源软件完成音视频的编解码，并通过 ortp 完成基于 rtp 协议的音视频数据传输。ortp 是一款处理 RTP 会话的开源软件。

1 整体架构图如下：



整个软件分为两层，上层为用户接口前端（user interface frontends），下层为 linphone 核心引擎（linphone core engine）。

2 功能模块说明：

Liblinphone 核心引擎实现了 linphone 所有的功能函数，而且能够方便的添加音频和视频的呼叫功能。Liblinphone 也提供高层的 API，用来初始化，接收或者终止呼叫。Liblinphone 依赖于下面三个组件：

1 Mediastreamer2

这是一个支持多种平台的轻量级的流技术引擎，主要适合于开发语音和视频电话应用程序。该引擎主要为 linphone 的多媒体流的收发，包括语音和视频的捕获、编码解码以及渲染。

2 ortp2

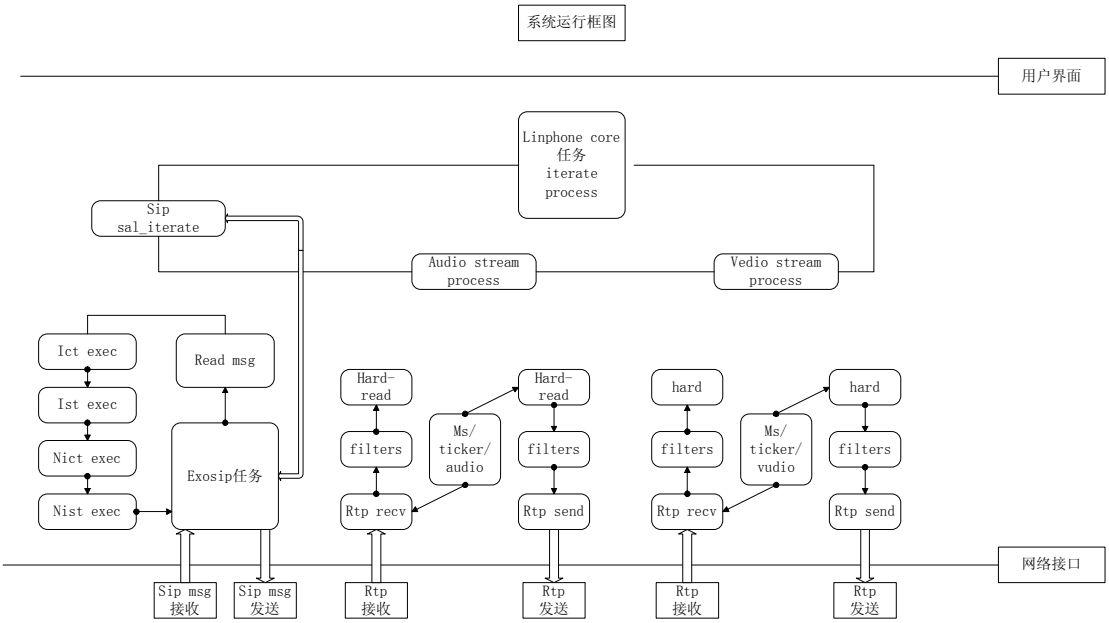
Ortp 是一个 RTP 库。为基于 RTP 协议的媒体流传输提供支持。通过 mediastream2 编码的数据就

是使用 ortp 库发送到网络的另一端。

3 eXosip2

Exosip2 为 sip 协议的实现。这部分实际上是由 exosip2 和 osip2 两个库共同完成的。使用 sip 协议完成路由、媒体协商以及会话的建立和管理，为直接的媒体流的传输提供基础。

二 linphone 系统框图



关于上面框图的一些说明：

通话双方在通信前使用 exosip 进行会话协商。上图左边部分展示这一部分的流程。Exosip 后台任务完成数据的接收和发送，并通过事件队列通知 linphone 底层的状态变化。

filter 的构建在会话协商成功建立后就顺带完成了，并且 ticker 任务也跑起来了。此时按照 filter graphics 构建的通道，音视频流不断的从硬件设备上读取，并经过编码压缩送给 RTP 会话，之后送到对端，对端到达的音视频流也经过 RTP 会话接收送到解码解压缩 filter，还原出原始的音视频流交给硬件设备播放。媒体数据在这两路流中源源不断的流动，完成了双方的可视通话。

上层 linphone 的 core 任务也不断的对底层进行迭代检查。所做的基本工作如下：

对于 sip 协议部分，core 一直等待从事件队列上拿事件。这些事件是 exosip 任务在处理 sip 消息过程中添加到事件队列上的。每当得到新的事件后，core 就从应用层的角度出发，进行处理。

对于视频流：基本上只处理 rtcp 数据包到达的事件。stream 上也有一个事件队列，用于保存该流上的相关事件。对于 rtcp 数据包事件，core 也只处理 sr 类型 rtcp 包，即发送端报告，得到 jitter 和包丢失率。如果设置了自适应比特率，则调用相关接口进行处理。此过程不断进行，直到当前事件上的包处理完。

对于音频流，检查流是否还是活动的。通过比较 RTP stats 中接收的数据包数目是否发生变化，如果在超时时间到达后，接收的数据量还没有发生变化，则认为音频没有响应。

三 linphone 中各个模块说明

1 Linphone coreapi 中子模块说明：

Coreapi 中的各个模块就是上层的处理模块，包括 configure 文件的处理接口，address 的处理接口，chat 的处理接口，sal 的处理接口，proxy 的处理接口，authorization 的处理接口，friends 的处理接口，callback 的处理接口，state 的处理接口，杂项处理接口等。这相当于高层的几个模块，提供给用户

的接口调用主要都在 linphonecore.c 中

1) Callback 模块：

该模块下的回调函数都是用于 sal 模块调用的。当 sal 处理完 sip 协议的处理后，就会调用相应的 callback 函数继续后续的处理，包括启动一个音视频传输流，启动响铃等。也就是说这里的 callback 完成了 media 媒体层的处理以及 linphone 上层的处理。

回调函数被保存在全局变量 linphone_sal_callbacks 中，在 linphone 初始化时调用 sal_set_callbacks 设置到 sal 的 callback 上去的。

2) Genera_stat 模块：

主要提供 linphone 全局状态的修改与设置的接口

3) Address 模块：

调用 sal 提供的接口，进行与地址相关的处理，这里的地址主要是 uri 相关的处理。包括获取地址以及地址中的部分信息或者设置这些信息。在上层地址是一个字符串指针，但是在内部处理时都会强制转换为 osip_from 结构体来处理。实际上就是对 linphone_address 结构体的处理。

4) Authorization 模块：

处理认证信息。各个认证用户的信息都被保存到 linphone_auto 结构体中兵串接在 linphone_core 结构体上。这里的接口就是处理这些数据结构，提供设置和获取相关信息的接口。

5) Chat 模块：

提供创建和销毁 chat room，向 chat room 发送消息和从 chat room 接收消息的接口，以及设置和获取用户数据的接口。类似于 authorization 模块，所有的 chat room 信息也是保存在 linphone_chat 结构体中并串接在 linphone_core 结构体上的。

6) Friends 模块：

提供处理 friends 相关信息的接口。所有的 friends 信息保存在 linphone_friend 结构体中并被串接在 linphone_core 结构体上，这样操作起来，包括设置，获取，添加以及移除都很方便。

7) Configure 模块：

提供配置文件处理的相关接口，包括配置文件的解析，配置文件中信息的获取，写入，同步等。配置文件解析后便于程序处理的信息主要都保存在 lpconfig 结构体中，这与文本文件中便于编写和阅读的配置文件本身不同。

配置文件中的各个配置模块本身也按照 section 的方式进行了划分，各个 section 也都是挂接在 lp_config 的 section 链表上的。这个模块可以单独提取出来进行测试。

8) Offer_answer 模块：

管理基于 sdp 的媒体协商。根据本地的支持能力和远端支持的能力，根据就低的原则，获得双方都可以支持的媒体信息。比如编解码格式等。

9) Presence 模块：

提供与在线状态相关的处理。

10) Proxy 模块：

处理代理相关的处理。代理相关的信息保存在 linphone_proxy 结构体上，但是该结构体只是代表了当前 linphone_core 使用的 proxy。代理可能不止一个，所有的代理其信息都被串接在链表上，并被挂接在 sip_conf 的 proxies 上。添加一个代理，取得一个代理以及其他相关的操作接口也都在该模块中提供处理接口。

11) Sal 模块：

Sal 模块其实应该是最重要的，最核心的模块了。该模块对 exosip 进行了简单的封装，间接的对 osip 模块进行了封装，使用该模块的接口可以完成 sip 协议的处理以及媒体描述的处理。

Sal.c 文件主要是对一些 sal 相关的结构体的操作，包括 SalMediaDescription 和 sal_op。处理包括创建这些结构体的实例，获取或者设置其中的一些操作域。

Sal_exosip2_sdp.c 基于 osip 库提供的 sdp 相关操作的接口，在 sal 层实现将其与 sal 相关的结构体关联起来操作。比如根据 SalMediaDescription 结构体信息将其转换为 sdp 结构体，或者反之。

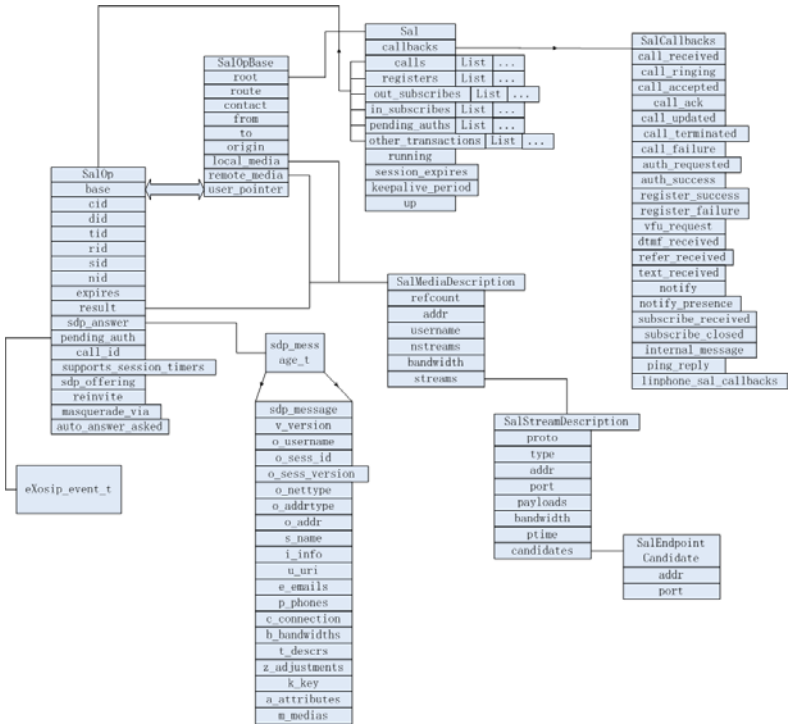
Sal_exosip2_presence.c 包括了对 in 和 out 的 subscribe 的操作。Text 数据的发送（基于 osip 和 exosip）。

Sal_exosip2.c sip 这块比较重要的封装。包含了对 sal_op 结构体的创建和基本操作。对 exosip 重要结构体的封装，包括初始化和释放。包含了对 sal 结构体的创建和基本操作的封装，更重要的是包含了对 sal 和 sal_op，sal_media_desc，sal_stream_desc 这些上层结构体与底层 osip_message，sip_message，sdp_message 等数据结构之间数据的转换和共享，以及对底层相关接口的调用。这种调用主要包括跟据上层结构体中包含的信息设置底层结构体，并调用底层接口完成具体功能，以及根据底层结构体得到的数据设置上层结构体的相关信息。

一个基本的描述就是：sal 作为 signal abstract layer 包含了上层所主要理解的交互信息，这些信息对于理解电话操作而言已经足够了，在底层，选择了 osip 和 exosip 来支持这项操作。所以实际上来说，可以用其他支持 sip 的库的接口来替代现有的，保留 sal 层接口的功能定义。在 linphone 中，虽然大部分使用了 sal 层的封装来完成 sip 交互过程，但是也调用 osip 和 exosip 库本身的其他接口，所以这层封装主要还是再次简化协议层的处理，使得功能更具体，而不是更单一。

几个关键数据结构之间的关系：

Sal 一个基本的结构体，通过这个结构体可以搜寻到上层所需的所有 sip 协议相关的信息。具体的 call，register 等信息保存在 sal_op 这个结构体中，多个实体通过链表串接起来，挂在 sal 上。Sal_op 包含了 sal_op_base 结构体，这个结构体保存了一些通用的不变的信息，对多个实体而言，比如路由信息，本地媒体信息，远端媒体信息等。其 root 指针由返回指向了 sal 这个基础，所以通过 sal_op 可以找到 sal。另外，在媒体信息中包含了所有流的信息，所有这些类似一个树的组织结构，sal 类似树根，通过它可以找到所有的枝叶及其上的信息。这些数据结构之间的关系如下图：



12) Core 模块：

上层 API 及其封装实现。通过这些 API 接口，可以快速构建基于 sip 的可视电话系统。

2 底层模块说明

1) Mediastream 实现的说明：

从代码上来看 mediastreamer 库，它的构成非常结构化。在 mediastream2 中实现了大量的 filter，包括声卡视频卡的 filter，编解码器的 filter，RTP 传输与接收的 filter 等等。每个.c 文件实现一个 filter，而且每个 filter 的实现也是非常结构化的。首先需要定义一个 filter 结构体实例，对于实例的各个部分进行赋值，主要是包括定义 filter 私有数据，filter 的 methods，init，preprocess，process，postprocess 和 uninit 这几个函数的实现等。而关键的一些实现，比如编解码器的处理，是基于 ffmpeg 库提供的接口来完成的，而声卡视频卡摄像头数据的捕获由其他库实现，但是也是基于标准的驱动接口来完成的。对于 RTP 的接收和传输则是基于 ortp 库来完成的。

另外提供了两个文件，audiostream 和 vediostream，用来处理音视频流的 filter 连接，linphonecore 主要就是调用这几个文件提供的接口来完成媒体流的启动的。

其他的就是辅助数据结构，包括 filter 的定义注册，queue 队列的操作，声卡和摄像头的 handler 的操作，ticker 的处理等。

考虑到设备描述符，实际上 mediastream 可以分为两部分来看，一部分就是之前主要的有关大量 filter 的操作定义部分，一部分就是后来的有关声卡设备和摄像头设备的设备描述符的定义和操作。关于摄像头部分，视频数据的捕获是通过创建一个新的任务来进行的，该任务源源不断的从设备上读取数据，当然除此之外也提供了配置设备的参数的接口，比如图形的大小格式等。而且对于 video 模块，设备处理和读取数据是与 V4lState 这个结构体挂钩的，这个结构体实例又挂到 filter 的私有数据上，这样数据读取任务不断的从驱动读取数据，放到该结构体实例上，filter 处理时从自己的私有数据结构指向的 queue 中取数据自然的就取到了摄像头捕获的数据，实际上这里包含了设备的操作（基于设备描述符）和 filter 的操作（基于 filter 的描述符）以及二者的结合。

在 filter 中 webcam 设备可能不止一个，有关这些设备的操作辅助接口就在 webcam.c 中实现。

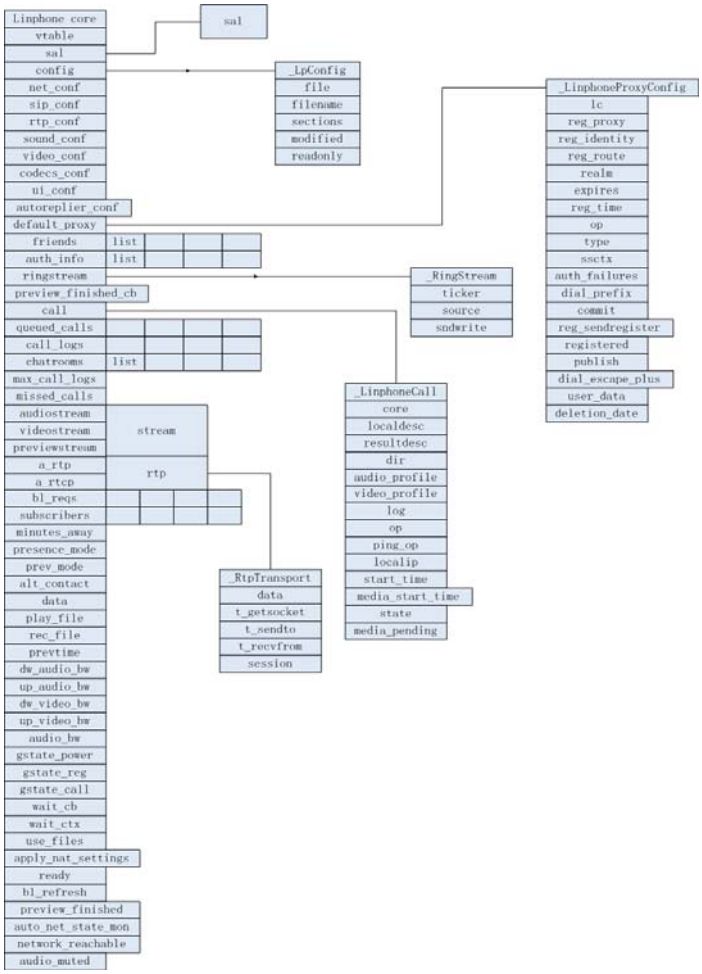
之前说了摄像头部分，对于声卡部分也是极其类似的，管理辅助函数在 mssndcard 文件中实现，另外还配以几个声卡设备本身的描述符文件来描述，其与 filter 的关系类同于视频部分。还有就是这些设备的描述符结构体汇总在 mscommon 中，而 filter 的汇总在 alldescs.h 头文件中。

2) ortp

关于 ortp 模块的说明参见参考资料 1

四 linphone 中数据结构说明

程序中定义了一个比较大的数据结构体——linphonecore，将其作为总的控制结构。通过该结构体，可以找到所需的大部分信息。这些信息要么是直接在该结构体中定义，要么是在其包含的模块相关的子结构体中。可以想象，内存中保存的该结构体，就像整体软件信息的总控点，通过该总控点，可以直接或者间接的得到系统相关的信息，这在一定程度上可以简化系统的架构和实现。（在许多开源软件中都可以看到这种代码架构方式。）下图展示了 linphonecore 结构体：



指针 sal 指向 sal 数据结构图，通过该指针，就可以找到 sal 模块用到的所有数据结构体。

Config 部分指向 linphone 的配置信息，包括网络，sip 协议，rtp 传输，音视频参数以及编解码器信息。

call 指针挂载了所有的通话，每一路通话都由 linphonecall 结构体表示。

Audiostream 和 videostream 保存了媒体信息，a_rtp 和 a_rtcp 保存了 RTP 相关的信息。

五 linphone 的初始化过程

首先初始化全局的状态：power 为 GSTATE_POWER_OFF，即关闭状态；reg 为 GSTATE_REG_NONE，没有注册；call 为 GSTATE_CALL_IDLE，空闲状态。
设置新的 power 状态为 startup。

1) Ortp 库初始化，调用 ortp_init

在 ortp_init 中主要创建了 payload type 链表，所有支持的 payload type 都被创建在一起了，这里注册的 payloadtype 就是底层 RTP 传输所能够支持的。

添加 payload type，先为音频，后为视频。

2) 调用 ms_init 初始化 mediastream 库，该库封装了媒体处理接口，使得多媒体数据的处理变得简单。

Ms_init 中形成了三个全局链表，一个为 filter 描述符链表，所有 media streamer 支持的 filter 的描述符在这里被串接在一起了，包括编解码，音视频读写以及 RTP 发送和接收处理相关的 filter。第二个为音频卡描述符链表，所有支持的音频卡相关的描述符也被串接在一起。第三个就是视频卡描述符链表，处理类似音频卡描述符。

3) 调用 sal_init 进行 sip 协议栈的初始化。该过程将返回一个 sal 结构体。

Exosip 全局结构体的创建以及初始化。

需要注意，在这里相当于有三层封装调用：一层为 sal 层的封装调用，一层为 exosip 层的封装调用，最底层为 osip 层的基本调用。

另外需要注意的是在这里没有创建 exosip 任务，而是在后面的读取并配置 sip 配置信息时才创建 exosip 任务，并监听特定端口。

将 lc->sal 的 up 指针指回 linphone core 全局结构体

设置 sal 上的回调函数，这些回调函数在对应的 sip 协议处理完后用于调用来处理外层有关 call 与 media 流的一些处理。

如果配置文件中没有设置 sip 会话的过期时间，则在这时将其设置为 200

将所有 sip setup 配置串联到 registered_sip_setups 全局链表上

4) 读取配置文件中有关音频的设置并对声卡进行设置

根据配置文件中描述的声卡设备 id 将声卡设备配置到 linphone core 结构体的 sound 配置结构体对应声卡的描述项上。

设置用于响铃的音频文件的路径。该文件路径会被保存到 sound configure 结构体的 local_ring 项目上。

类似上面，设置用于提示远端响铃的音频文件的路径，最终保存到 sound configure 结构体的 remote ring 项目上。

检查系统中的声卡设备

配置是否使能回声消除

配置回声限制

配置增益

配置回放增益

5) 读取配置文件中有关网络部分的配置并对网络参量进行配置

读取配置文件中有关带宽的设置，并对 linphone 中音视频使用的带宽进行配置。首先保存带宽值到 net config 结构体中，音频带宽配置为配置文件中读出的值与默认值的小者，视频带宽配置为读出值减去音频值减去 10 和 0 之间的较大者。这里 10 相当于是一个缓冲。

设置 stun server

设置 nat 防火墙地址

配置是否使用防火墙策略

配置是否只对 SDP 进行 nat 转换

配置 mtu 值

设置信息分包时间

6) 读取配置文件中有关 RTP 部分的配置并对该模块进行配置

首先设置音视频的 RTP 端口号

配置 RTP 音视频抖动补偿时间，默认为 60 毫秒

配置 nortp 超时时间，即没有 RTP 或者 rtcp 数据包时 linphone 认为对端 crash 或者网络中断的超时值

设置在静音时不进行 RTP 传送，默认为 FALSE

7) 读取配置文件中有关编解码器相关的信息并设置到 linphone 的编码器信息结构体上

读取配置文件中的音频编解码器信息，如果有，则查找之前初始化 ms 模块时建立的全局过滤器描述符信息表，如果找到，则说明相应的编解码器支持，否则不支持配置文件中添加的编解码器。对于视频也类似。

通过上述操作，所有的配置文件中可支持的编解码器信息都被创建到 audio_codecs 和 video_codecs 两个链表上了，之后将它们添加到 linphone 的 codecs_conf 结构体上。

另外，在这步操作时，也将系统支持的其他编解码器添加到了 codes_conf 结构体上，基本原理如下：对于 video，查找所有的 RTP 初始化时创建的 payloadtype 表，如果是 video 类型的 payload，并且被 mediastreamer 支持，但是不在配置文件描述中，就将其添加到链表上。也就是系统初始化时支持的但是在配置文件中没有说明的编解码器也会被添加到配置结构上。

更新已分配的音频带宽值。找到需要最大带宽的音频编解码器，将其带宽需求除以 1000 作为 linphone 的 audio_bw 值，同时重新设置 linphone 网络配置中的上下行带宽值。

Desc_list 上的编解码器信息与 RTP 初始化时设置的 payload type 信息的区别：

Desc_list 将 medie streamer 支持的所有 filter 的信息都串联在一个链表上，不仅包括了编码器解码器，还包括了 RTP 发送和接收 filter，以及音视频的读写 filter。这些都是站在 media streamer 这个中间层的角度来考虑。Filter 的构成也很规范，包括了一些必要的描述信息以及数据的读写处理接口。

Payload type 链表是所有 RTP 传输中支持的 payload 的链表，每一个 type 的描述信息主要描述了该类型的 type 的时钟速率，正常比特率，采样比特等，可以看出这些信息与传输也是紧密相关的。除此之外，也描述了编码信息，这部分信息与 desc_list 中的编码器信息部分会存在交集。

8) 读取配置文件中有关 sip 协议的相关信息，并以此来配置 linphone 的 sip 模块。

配置是否在发送数字时使用 sip info 信息。

配置是否在发送数字时使用 rfc2833 信息

配置是否使用 ipv6

配置 sip 的传输端口信息。指定是使用随机值还是知名 5060 端口

将端口信息设置到 linphone core 中，并启动 sip 监听。这样，当 sip 协议数据到达时即可被处理。

首先调用 sal_listen_port 启动监听端口。在这里，协议层被选择和设置，一般情况下都是 udp，这里为 eXtl_udp。之后创建并启动_eXosip_thread 任务，该任务处理 sip 协议数据的接收，协议的处理，状态机的处理，数据的发送等。即几乎所有与 sip 协议有关的处理都会在该任务中处理完。最后保存用户代理信息。

获取配置文件中的联系人信息，如果联系人为空，或者配置文件中联系人信息不为空，但在将其设置为主联系人信息时出错（比如格式错误），则基于环境变量中的 host 和 user 信息创建主联系人，否则将配置文件中的联系人信息设置到 sip_conf 结构体的联系人上

如果配置文件中设置了猜测主机名，则将该配置设置到 linphone core 的 sipconfigure 结构体上

配置 incoming call 的超时时间，如果超过超时时间没有 answer 则 terminate 该 call

读取并配置代理信息，所有的代理者信息都被添加到 sip configure 的代理者链表上

读取并配置默认代理者信息。默认代理者会从所有代理者中挑选，根据配置文件，然后放到

linphone core 结构体的 default_proxy 上

读取并配置认证信息。首先从配置文件中读取 usrname，userid，password，ha1，realm 等信息，并根据这些信息创建一个新的认证信息结构体，将其添加到 linphone core 的 auth_info 链表上。同时，查找所有处于 pending 状态的待认证事件，如果 linphone core 中能找到一致的认证信息结构体，则对其进行认证。

根据配置文件对 sip_conf 的其他变量进行设置

9) 读取配置文件中有关 video 模块的配置并将其设置到 linphone core 中

首先读取所有的 video 设备，将其添加到 video_conf 的 cams 项上

读取配置文件中 video 部分有关 device 的设置，在系统中查找是否已经有该设备。如果没有找到，则使用 default 设备配置。如果原来保存的设备不为空，并且与新设备不一样，则基于新设备重新触发 preview 预览。如果 linphone 已经准备好了，并且 video_conf 上的设备不为空，则将该设备的描述串写入配置文件，也即覆盖之前的配置。另外，如果如果描述串中有 static picture 则描述串在被重新写到配置文件之前会设置为空。

根据配置文件设置视频 size

根据配置文件设置其他配置项，包括是否进行 capture，是否 display，是否 self_view 等。

10) 设置 linphone 之前和当前的模式都为在线状态。设置最大 call logs 为 15

11) 读取并配置 ui

读取配置文件 friends section 部分的信息，创建 friends 结构体并保存配置。将所有的 friends 添加到 friends 链表上。并额外做一些其他处理。

最后读取 call logs 信息，并创建 call logs 结构体保存 logs 信息，将所有 logs 添加到 call logs 链表上。

12) 显示 ready。

全局状态配置为 power on

将 sip_conf 中的 auto_net_state_mon 设置到 linphone core 上

Linphone core 的 Ready 设置为 TRUE

至此，整个初始化过程完成。初始化后的内存数据结构体及状态：

首先需要创建 linphone 顶层最全局结构体 linphonecore 也即程序中多处用到的 lc。对于 phone 的很多相关操作，该对象是主要的 handler，在整个程序运行过程中在内存中只有一个实例存在。

初始化 ortp 库，加载支持的音视频 RTP payload 类型到全局结构体 av_profile 上。同时全局结构体 linphone_payload_types 也指向这些 payload 元素列表。

初始化 ms 库。Desc_list 全局指针挂载了所有支持的 filter 描述符信息。加载所支持的声卡设备的描述符到全局变量 scm 上，加载支持的所有视频捕获设备的描述符到视频全局变量 scm 上。

初始化 sal 模块，在此过程中初始化 exosip 库，在 exosip 初始化过程中初始化了 osip 库。在此过程中，从下到上，osip 全局状态机被加载，osip 全局结构体对象也被创建，其上的事务链表，callback 等子项也被设置，exosip 全局结构体对象也被创建，其上底层协议处理部分以及内存分配部分也部分的被初始化，osip 对象被挂载到了 exosip 对象上。Sal 结构体对象被分配内存。

Linphone core 被挂载到 sal 上了，sal 的回调处理函数被加载。

配置文件中的配置项被一步步的加载，同时 linphone 的配置部分也被不断的在内存中创建出。状态被更新，整个初始化过程也就此完成。

六 linphone 建立通话过程说明

1 拨号 call 过程

用户执行呼叫，调用 call 命令

Linphone 中调用该命令对应的执行函数 lpc_cmd_call

如果 lc-call 不为空，也就是说 linphone 全局结构体上当前的会话还存在，则输出打印，要求用户先关闭当前 call；否则调用 linphone_care_invite 处理 call 命令。（该版本只支持一个 call 存在）

在 linphone_core_invite 中：

首先调用 linphone_core_interpret_usl 解析 URL 地址。输入的地址可能是一串字符，通过该函数将其中的关键元素解析出来，主要是按照 osip_from 结构体的格式解析出来。

然后调用 linphone_core_invite_address 进一步的以 osip_from 格式的地址为参数进行处理，完了释放地址参数。

在 linphone_core_invite_address 中：

在该函数中对地址信息进行进一步的简单处理，获取到对外的本地信息，然后调用 linphone_call_new_outgoing 发起会话请求。

linphone_call_new_outgoing 中，

首先创建 linphone call 内存对象实例。方向设置为 call outgoing，创建一个 salop 对象实例，其 root 指针指向初始化时创建的全局 sal 结构体上，并对 op 的一些基本域进行设置。在会话过程中，与 sal 相关的操作主要还是基于 salop 的。另外，userpointer 指针指向 call 本身。

调用 linphone_core_get_local_ip 获取本地的 ip 地址。如果设置了 nat 防火墙策略，则就用 nat 防火墙地址为本地 ip 地址，否则，如果配置了 ipv6，则使用 ipv6 地址。如果仍然没有得到结果，则调用 sal_get_default_local_ip 接口来获取。该接口使用底层 exosip 提供的方法，即利用 socket 的 gethostbyname 接口来在与网关建立 connected 连接后从 socket 中获取本地 ip 地址信息。

调用 create_local_media_description 创建本地媒体描述结构体。在该接口中，创建一个 SalMediaDescription 对象实例，并用之前获取的信息对其进行设置，包括流数量为 1，本地 ip 地址，用户名，带宽。针对其携带的流，流的地址为本地 ip 地址，流的端口为配置的 RTP 音频端口，proto 为 SalProtoRtpAvp，类型为 audio，ptime 为 netconfig 中的 down_ptime。对于 payload 的配置逻辑如下：从编解码器配置项中读取所有的音频编解码器 payload，如果是 enabled，并且 linphone 当前支持该编解码器，带宽也允许，则将该 payload 添加到 streamer 的 payload 链表上。目前 payload 的带宽是通过 payload 的 Bitrate 计算出来的。计算方法：包大小为 ip4 头+UDP 头+RTP 头+Bitrate 除以 50*8，即除以 400。因为 Bitrate 为基于比特单位，除以 8 变为字节，但是这里除以 50 是为什么？貌似是包的数量，也就是一秒钟采样的数据用 50 个包来传送。此时计算出来的为包的大小，将其再乘以 8 乘以 50 变为包含 ip、UDP 及 RTP 包头的情况下的 Bitrate，此值除以 1000 作为 linphonecore 中的音频带宽值。基于上述计算得出来的音频带宽值和网络配置中配置的上下行带宽值更新 linphonecore 的带宽配置的方法如下：如果给的带宽值为 0，即表示无穷，音视频的下载带宽设置为 -1，否则，计算出来的音频带宽值和给定带宽值中较小者作为音频下载带宽，给定带宽值与音频下载带宽的差值减去 10 与 0 的较大者作为视频下载带宽值，也就是视频带宽或者为 0，或者为可用带宽减去音频所用的带宽（优先保证音频）再减去 10 作为缓冲界限后的值。上传带宽的配置方法也一致。如果当前配置的带宽值中的较小者（上行和下行选择）大于当前编解码器 Bitrate 所占用的带宽，则将其 clone 一份放到 streamer 的 payload 链表上，否则不添加。在处理完配置的音频编解码器 payload 后，从全局 av_profile 中找到 telephone-event 也将其添加到 streamer 的 payload 链表上。Streamer 的带宽值被设置为音频下载带宽值，应该小于等于 media 的带宽配置。如果视频也被允许，那么 streamer1 将作为视频流的描述符，同样从 RTP 配置信息中得到视频端口赋给该描述符，proto 同音频部分，类型此时为 SalVideo。类似音频部分，将 codec 配置中的符合带宽限制的视频编解码器 payload 添加到该 streamer 的 payload 链表上。如果视频下载带宽不为 0，则该 streamer 的带宽值被设置为视频下载带宽值。至此，media 描述符就创建完成。Media 描述符当前是被挂载到 call 的 localdesc 上。

调用 linphone_call_init_common 对 call 的其他域进行设置，状态为 LCStateInit，start_time 为当前时间，media_start_time 为 0，创建 call_log 实例记录拨号记录，通知所有的 friends 我们当前的 onthephone 状态。如果是设置了 stun 服务器，则调用 linphone_core_run_stun_tests 测试 stun 服务器，并配置 streamer 的 endpoint candidate 的端口和地址。

调用 discover_mtu 获取当前的 mtu 值，这只在当前 netconfig 中的 mtu 设置为 0 时才进行。发现 mtu 的过程也是通过向对端发送数据，然后根据 socket 的 options 操作来查看，比如根据收到的 ICMP 包信息，根据获取到的 mtu 值重新设置 mediastream 的 mtu。

综上，在 new outgoing 的过程中，我们创建了 call 实例，salop 实例，media 实例同时包含 streamer 实例。基本关系为 call-->salop，call-->media-->stream。本质上来讲还是在初始化 call，但在此过程中也初始化了需要的 salop，以及 media。

上一步通过 linphone_call_new_outgoing 为发起一个新的会话做好了准备，包括创建了需要的 call 实例，salop 实例等，这些相关信息保存到 call 对象中，该对象在此时被挂载到 linphonecore 上。

如果目的代理不为空，或者 sipconfi 的 ping_with_options 为 FALSE，则掉用 linphone_core_start_invite 发起会话请求，否则，call 的状态被设置为 LCStatePreEstablishing，该状态指示稍后，即 ping 完后继续发起 invite 请求。为了完成 ping 操作，先创建一个用于 ping 的 salop，调用 sal_ping 基于该 op 以及 from 和 real_url 参数发送 sip 的 ping 数据，重新将 call 的 start_time 设置为当前时间。

对于 linphone_core_start_invite 调用：

在调用该接口前，我们已经创建了 linphone core 实例，并在发起 invite 请求的准备过程中创建了 call 实例，以及得到了 dest_proxy 地址信息。在这些准备工作完成的前提下，系统进一步处理 invite 相关的后续操作：

首先调用 get_fixed_contact 来获取联系人信息。

如果当前设置了防火墙，并配置了 nat 地址，则从 linphone core 结构体实例中获取首要的 contact 信息。这些信息基本上是从 sip_conf 中拿取的。

如果上一步失败，并且 call 上的 salop 结构体实例已经被创建了，并且其上的 contact 信息不为空，则返回空，表明不需要修改 contact 信息。

如果上一步失败，则判断 ping_op 操作是否成功完成，如果是，则使用 ping_op 上的 contact 信息。

如果上一步失败，并使用了代理，则使用 register 时的 contact 信息

如果还失败，则使用本地 ip 地址和配置到 linphone core 上的端口信息组合出 contact 信息返回

如果在上一步获取 sip_conf 中的 contact 信息时返回失败，则该接口此时返回空

通过上面调用，如果获取到了 contact 信息，则将其设置到 call 的 op 的 contact 域上。

Call 的 state 设置为 LCStateInit

调用 linphone_core_init_media_streams 初始化媒体流。参数为 linphone core 实例和 call 实例

首先从 call 实例的 localdesc 上拿到 media 描述符

基于 media 描述符上的 stream[0]也就是音频流的端口调用 audio_stream_new 创建 audiostream。

在 audio_stream_new 中，创建了一个 audiostream 结构体实例，stream 的 session 域被初始化为一个 RTP session，这是通过调用 create_duplex_rtpsession 创建的。在该接口中，创建了一个全双工的 RTP session 结构体实例。首先通过调用 rtp_session_new 为 RTP session 实例分配内存，并调用 rtp_session_init 对这个 session 进行初始化。这里的初始化包括设置 session 的 mode（send or recv or send_and_recv）。并根据 mode 设置 session 的 flags。如果可以发送，发送 ssrc 初始化为一个随机值。并设置默认的源描述信息 for rtcp，挂在 session 的 sd 上。设置 session 的 rcv 和 snd 的 profile，此处都设置为 av_profile 全局变量了。初始化 rtp 和 rtcp 的 socket 都为-1，配置默认的接收和发送 socket buffer 大小。

从 rtp_session_init 中出来接着配置 RTP session 的一些类似全局的参数。Recv_buff_size 配置为 MAX_RTP_SIZE，调度模式为关闭状态，阻塞模式为不使用该模式，自适应平衡抖动补偿，对称 RTP，设置本地地址，此时会创建 rtp 和 rtcpsocket，并对 socket 的参数进行配置。比如是否 reuse address，设置 socket buffer size 等。注册 timestamp 和 ssrc_changed 事件的回调函数，以及 ssrc changed 的触发阈。最后返回创建的 RTP session 结构体实例。

返回的 RTP session 实例被挂到了 stream 的 session 域上。接着调用 ms_filter_new 创建了 RTP send filter 结构体实例。这被挂载到了 stream 的 RTP send 域上了。最后对流的相关参数进行了初始化。在 ms_filter_new 中，会遍历系统最初初始化时创建的 filter 描述符链表 desc_list，从其上找到 id 与当前要创建的 id 一致的描述符，然后调用 ms_filter_new_from_desc 基于该描述符创建 filter。在这个接口中，首先初始化了一个 msfilter 的结构体实例，然后把当前找到的描述符挂到该 filter 的 desc 域上，调用描述符的 init 接口对 filter 进行初始化，最后返回这个 filter。

Audiostream 创建成功后被挂载到了 linphone core 结构体实例的 audiostream 域上。

基于初始化 linphone core config 时，从配置文件及系统中获取的对 sound 部分的配置信息对 audiostream 进行实际使用上的配置。也就是针对具体使用实例的配置。之前可能就是全局的系统参数级别的配置。这包括增益的配置，回音消除的配置，echo limiter 的配置，自动增益的配置，噪音的相关配置等。

如果在 linphone core 初始化时已经初始化了 rtp_transport，则将 audiostream 上的 RTP session 上的 rtp 和 rtcp 上的 tr 指向 linphone core 的 a_rtp 和 a_rtcp 上。相应的，这两个结构体的 session 指针指向这里的 session。

至此，音频流的初始化工作基本完成。

如果系统定义了视频流的支持，则开始进行视频流的初始化。这是通过 video_stream_new 接口完成的，参数类似音频部分，只不过这里是用的 media 的 stream[1]描述符上保存的视频 RTP 端口。

在 video_stream_new 中，创建了一个 videostream 结构体实例。类似与音频部分，通过调用 create_duplex_rtpsession 接口创建一个全双工的视频 RTP 会话，并将其挂载到 videostream 的 session 域上。

在 create_duplex_rtpsession 接口中，创建全双工的 RTP session 结构体实例。并作初始化工作，这步同音频部分。

调用 ortp_ev_queue_new 创建一个 ortp event queue，事件队列。

调用 ms_filter_new 创建一个 RTP send filter，在该接口中会创建一个 msfilter 实例。初始化工作同音频部分。

调用 rtp_session_register_event_queue 同时将流的事件队列注册到 RTP 会话上。也就是 session 的 eventqs 也指向 stream 的 evs。

设置视频的高度和宽度，返回视频流 videostream 实例。

从上面的初始化工作中可以看出，音频流和视频流的 RTP session 和 filter 的创建是一致的，调用相同的接口完成，因此可以看出是通用的，只是相应的挂载到了音频流和视频流上。至此基于 call 的音视频流的初始化工作就完成了。

如果在 sip_conf 中没有设置 sdp_200_ack，则将 call 的 media_pending 设置为 TRUE。另外，将 call 的 localdesc 描述符设置给 call 的 op，这是通过 sal_call_set_local_media_description 接口来完成的。在该接口中，首先增加 localdesc 的引用计数，同时递减 op 上的 localdesc 的引用计数，如果减一后为零，则释放其资源，同时将参数中给出的 localdesc 给该 op。

从 call 的 log 实例上获取 from 和 to 地址，分别作为发起 call 会话的 from 和正式 url 即 real_url。至此，基本的初始化的工作做得差不多了，调用 sal_call 发起 sip 协议请求。Sip 的 Invite 消息报文在

此时才真正的发给对端。

在 sal_call 中：

会话操作基于 call 的 op 实例发起。从参数中获取 from 和 to 地址，将其设置到 op 的 from 和 to 域中，并调用 sal_exosip_fix_route 来检查路由。

在 sal_exosip_fix_route 接口中，首先判断 op 的 route 是否为空，如果是空，就不做任何处理，说明没有配置路由，否则，继续。

调用 osip_route_init 分配并初始化一个 osip_from 结构体实例。然后调用 osip_route_parse 将 op 中的字符串形式的 route 解析为这里 osip_from 格式的结构体，如果失败，说明 route 配置有误，将 op 中的 route 重新该为 null，否则从 uri 列表中查找是否有名为 lr 的节点，如果没有则添加一个，值为 null，并将此修改更新到 op 的 route 项中。最后释放操作过程中为临时变量分配的内存。

Route 检查了，现在调用 eXosip_call_build_initial_invite 接口创建一个 invite 消息。在该接口中，我们首先将 to 参数指定的字符串形式的目的地址解析到 osip_from 格式的结构体中，如果解析失败，返回。接着调用 generating_request_out_out_dialog 创建一个最小的 out_of_dialog 的 request 请求消息体。

在 generating_request_out_out_dialog 中，此时我们已经操作到 sip 底层协议部分了，主要是调用 exosip 和 osip 库来进行相关操作。这里我们首先检查 exosip 全局变量下的 extl 是否为空，也就是底层网络协议的支持，默认是 UDP。如果这块为空，说明底层协议支持没有完成，后续将不能收发数据包，直接返回错误。接着根据 extl 中 proto_family 指定的协议簇通过 socket 接口来猜测主机的 ip 地址，如果失败则返回错误。接着调用 osip_message_init 分配并初始化一个 osip_message 结构体实例。到时候 sip 数据包中按照协议规定的格式的各个位置的数据会被解析到 osip_message 结构体中，进而在程序中传递和处理。接着准备 invite 类型 sip 包的请求行，包括 method，invite，协议 2.0，status_code 为 0，reason_phrase 为 null。如果 method 为 register，则调用 osip_uri_init 为 request 分配并初始化 uri 结构体实例，并将 proxy 参数中的信息解析到该结构体中。基本按照同样的方法将 from 参数的数据解析到 request 的 to 的 uri 中。

对于非 register 类型的请求，操作则如下：此时将 to 参数解析并设置到 request 的 to 的 uri 中，如果成功，并且其 uri 不为空，则将 uri 上 headersl 链上的元素一个一个取出来。如果这个原始不是 from，to，call-id，creq，via，contact 则将其拷贝给 request。这步首先调用 osip_header_init 分配并初始化一个 osip_header 的结构体实例，然后将之前取出来的 header 的 name 和 value 拷贝给这个结构体，最后将其挂载到 request 的 headers 的链表上，也就是 osip 结构体的 headers 的链表上。无论是不是上述拷贝的 header，最后都从原来的链表上移除并释放内存。也就是从最初的从 to 参数解析到 osip_from 结构体实例上的。对于非 register 请求，如果存在代理，调用 osip_route_init 分配并初始化一个 osip_from 结构体实例，将 proxy 参数数据解析到该结构体中，类似之前，从该结构体中查找名称为 lr 的 url_params，将 osip 上的 to 上的 url 拷贝到 req_uri 中，并将解析后的 proxy 添加到 osip 的 routes 上。否则，就将 proxy 上的 url 给 osip 的 req_uri。释放 proxy 本身的内存。此时调用 osip_message_set_route 为 osip 的 router 分配内存并将 to 参数的数据解析赋值给它。如果不存在代理，则直接将 osip 的 to 的 url 给 osip 的 req_uri。

至此，register 和非 register 类型的请求的不同处理完成了。调用 osip_message_set_from 将参数 from 解析并配置到 osip 结构体中。调用 osip_from_set_tag 将新生产的一个随机数作为 from 的 tag。分配并初始化 osip_call_id 类型的结构体实例，生成随机数配置其 number，并将其挂到 request 的 call_id 上。分配并初始化 osip_creq 类型的结构体实例，如果是 register 请求，则其 number 设置为 1 否则为 20，然后也将其挂载到 osip 的 cseq 上。

调用_eXosip_request_add_via 配置 request 的 via。

设置 max-forward 为 70。如果 method 为 options，则配置 request 的 accept 为“application/sdp”。用 eXosip 全局变量上的 user_agent 配置 request 请求中的 user_agent 域。最后返回 osip 的 request 消

息结构体。

调用_eXosip_dialog_add_contact 添加联系人信息，最后挂载到 osip 的 contact 域上。

调用 osip_message_set_subject 添加 subject 信息，这个是 sip 头的 subject 信息。

最后调用 osip_message_set_expires 设置 sip 头的 expires 为 120 秒。意思就是说如果经过 120 秒对端还不响应就取消当前的 invite 请求的发送。

在通过上述操作创建 osip 基本消息结构体实例后，在外层继续对其进行一些初始化。这包括设置 allow，即允许的 options 方法。如果 op 的 base 中包含了 contact 信息，则将初始化 osip 结构体时添加的 contact 信息清除，重新将这里的 contact 信息设置进去，因为初始化时这些都是根据系统信息猜测的，在没有进行任何配置的情况下使用。如果 op 中 session_expires 不为零，则将 invite 的头部添加 session_expires，值为 200。并添加 timer 的 support。如果 op 中本地的媒体信息也已经被配置了，则将 op 的 sdp_offering 设置为 TRUE，将本地媒体信息的配置通过调用接口 set_sdp_from_desc 设置到请求 osip 结构体中。在上述接口中首先将媒体描述信息转换为 sdp_message 格式的结构体，在 sdp_message 中间格式的 sdp 数据转化为串行的字符串，最后将其挂载到 osip 结构体的 bodies 域上。

至此完成了基本的设置，调用 eXosip_lock()和 eXosip_unlock()保护 eXosip 底层的发送数据函数，这里为 eXosip_call_send_initial_invite。最后保存 call 的 id 到 op 的 cid 上，并调用 sal_add_call 将当前 op 挂载到根 sal 的 calls 链表上。所有创建成功的 op 都会被挂载到 sal 的 calls 上，程序通过 sal 实例就可以遍历所有的与 sip 协议处理有关的 op。

现在看看在接口 eXosip_call_send_initial_invite 中做了哪些操作：首先调用 eXosip_call_init 初始化一个 exosip_call 结构体实例，并为其分配内存。接着调用 exosip_transaction_init 初始化一个 osip_transaction 结构体实例。事务的初始化信息大部分都是来自之前初始化的 osip_message 类型的 request 结构体的。初始化完成之后就将 transaction 挂载到全局 osip 结构体的对应类型事务的链表上，同时，也将其挂载到 exosip_call 的 out 事务域上。根据 request 信息，new 一个 out 的 sipevent。这时 osip_event 类型的，不同于 exosip_event。为了关联具体的事物，events 的事务 ID 设置为之前创建的事务。设置事务的 your_instance 域，这个域用来保存一些有用的东西，这里使用 jinfo_t 类型的结构体将 exosip_dialog，exosip_call，exosip_subscribe 以及 exosip_notify 关联起来，交给事务的 your_instance。不过此时还只有 exosip_call 的信息，其他的实例还没有创建。下一步将之前创建的 sip_event 挂载到事务上，并将创建的 exosip_call 实例挂载到 exosip 的 j_calls 域上。调用 exosip_update 更新 exosip 实例，调用 exosip_wakeup 唤醒 exosip 任务的处理。最后返回刚创建的 call 的 id。因为我们在 exosip 的 osip 的事务队列上添加了一个事件，之后 exosip 任务会处理该事件，并根据事件描述发送 sip 的 invite 消息。

此时再判断 sip 的配置中 sdp_200_ack 是否设置了。和之前不同，如果设置了，则将 call 的 media_pending 设置为 TRUE，并同意将 call 中的 local media 描述设置到 call 的 op 上。感觉在 sal_call 中并没有改变 call 的 media description。调用 linphonecore 的 vtable 函数显示正在连接远端客户端。

如果之前调用 sal_call 时返回失败，则显示无法建立 call 提示，并 stop 媒体流，这是 linphonecore 上的媒体流，同时清除 call 上有关 RTP profile 的描述。最后调用 linphone_call_destroy 销毁之前创建的 call。其实这里可以看出通过顶层的 call 可以找到底层创建的所有实例。

如果之前调用成功了，则修改 linphonecore 上的状态为 call_out_invite。最后返回。至此，顶层的 call 命令处理完了。

从上面的发起会话请求的初始化过程来看，系统基本上是沿着从外层到内层，从上层到下层的次序来初始化的，外到内是说，先是 call，在到 call 上的 op，再到 op 上的 media desc，再到其上的 stream desc。这几步初始化了对媒体流的控制结构体，因为主要都是 desc 即描述信息。接着道 stream 本身，

到其上的 RTP session，到 filter，基本上就与流本身的传输相关了。从上到下是说，先初始化上层协议相关的结构体，比如 sip 协议相关的数据结构的初始化，接着是媒体和 RTP 等下层相关协议的初始化。

2 等待响应

此时 invite 请求包发送给对端了，客户端的相关状态也设置好了，就等待对端的响应并进行处理。

之前已经说明，call 调用是在 main 里面处理的，处理完成后调用相关接口将请求和数据交给 exosip 任务去实际的发送数据。之后 main 仍然每间隔一秒进行迭代操作，而 exosip 任务则处理数据包的收发和相关底层的超时操作。因为此时 call 的状态为 init，这样在 linphone core 迭代中不会去处理 call 相关的操作。

当对端有响应之后，该消息最先被 exosip 任务捕获到。Exosip 在调用 exosip_read_message 时会从 UDP 套接字接口上读取到对端的响应数据包。接着调用_exosip_handle_incoming_message 对缓冲在缓冲区的数据进行处理。

在该接口中，首先调用 osip_parse 解析缓冲区中的 sip 信息。在 osip_parse 中，创建一个 osip_event 类型的结构体实例，将 buf 中的 sip 数据解析到该结构体的 sip 域中，根据消息中的类型设置 events 的类型。对于输入的 message，有如下类型的定义，invite 请求，ack 请求，除此之外的请求，以及 1 开头、2 开头以及 3456 开头的状态响应。之后将该 events 返回。

调用 osip_message_fix_last_via_header 根据 host 和 port 信息检查 sip 中的 via 域。

调用 osip_find_transaction_and_add_event 查找该响应对应的事务，并将由此生出的 events 添加到事务的事件队列上。在该接口及子接口的调用中，首先根据响应消息的类型找到 osip 全局变量上其所属的 transactons 队列，再到队列中查找所有的事务，找到与当前给的 events 匹配的事务。这有点类似于二级过滤。如果找到了，就将刚才创建的 events 添加到该事务的事件队列上。

如果事件添加成功，则返回，否则，说明事件没有对应的事务，继续处理。如果消息是 request 消息，调用 exosip_process_newrequest 处理，否则调用 exosip_process_response_out_of_transaction 处理。在 newrequest 中，会创建一个新的 transaction，将事件挂到该事务上，并将事务挂接到 osip 全局变量对应的事务链表上。之后针对事件的类型，进行一些对应的处理，比如发送响应等。否则，将事务加载到 exosip 全局变量的事务队列上，返回。对于 response，说明收到了 out_of_transactions 的响应，在 exosip 上进行查找做一些处理。

对响应消息处理完后，或者是对消息直接做了响应，或者将需要处理的放到了事务队列的某个事务的事件队列上。接着，exosip 迭代检查处理事务队列上的所有事务。在处理过程中，如果需要传给上层处理，就会构造一个 exosip_event 结构体实例，添加到 exosip 的事件队列上，由 linphone_core 任务也就是 main 主线程去处理。

假设此时底层进行了协议的协商处理，基本顺序为 send(invite)-->recv(180ring)-->recv(200 ok)-->send(ack)会话建立。当处理到第三步，也就是主机准备走第四步时，对端的 200 ack 我们收到了，此时我们再发送自己的 ack 之前需要处理对端的媒体类型，并为媒体流做好进一步的准备。此时 callback 的处理顺序为首先进入 cb_rcv2xx 回调，在该回调处理中调用 cb_rcv2xxx_4invite 回调接口。在该接口中最终会调用 report_call_event 将 exosip_call_answered 类型的事件交给 linphone_core 任务。

Linphone_core 任务在 sal_iterate 处理中处理底层协议交上来的事件时，如果判断为上述事件就调用 call_accepted 接口。在 call_accepted 中：

首先调用 find_op 从 sal 全局结构体的 call 上查找 call_id 等于 events 的 cid 的 op，如果找到就返回。然后将 op 的 did 赋值为 events 的 did。调用 exosip_get_sdp_info 从对端最后一次的 with session

description 的 200ack 中取出有关 sdp 的信息。实际上在该接口中会将字符串形式的从对端收到的 sdp 信息解析到 sdp_message 结构体中并返回。如果获取成功，则根据该 sdp 创建主机端对远端客户的媒体描述，基本步骤如下：

调用 sal_media_description_new 创建远端的 media_desc 结构体实例，将其添加到 op 的 base 的 remote_media 上。调用 sdp_to_media_description 将之前解析出来的 sdp 信息配置到 media_desc 上。如果配置成功，则调用 sdp_process 处理音视频流处理。

在 sdp_processs 中，创建一个新的 media_desc 结构体实例，放到 op 的 results 上，如果 op 的 sdp_offering 被设置，则调用 offer_answer_intiate_outgoing 接口创建一个基于本地的 offers 和远端响应的能力的流。否则，调用 offer_answer_initiate_incoming 接口创建一个流基于本地的能力和远端的 offers。这个流会作为一个 answer 发送给远端的 offers。对于 outgoing，会遍历本地 offers 的所有 streamer，取出其 protocol 和 type，然后在远端 answer 中查找，如果找到，就使用这两个 stream_desc 调用 initiate_outgoing 设置 results 的流描述。在该接口中，取出二者的 payloads 的交集，端口，addr，带宽和 ptime 使用 remote 的 answer，最后 media 描述的 addr 也设置为 remote 的 answer 的值。对于 incoming，遍历远端的 offers 的所有 streamer，取出其 proto 和 type，在本地能力中找到匹配项。类似的，在 incoming_initiate 中，取出二者 payloads 的交集，而此时端口，adr，带宽，ptime 等则使用本地的。这几步实际上是找出共同支持的媒体描述，交给 results 保存。另外，如果是基于远端 offers 来适配的话，将根据 results 构造转换出 sdp_message 结构体的 sdp_answer 给 op，这在后续会被用来反馈给对端，表明我们支持的能力。

Sdp_process 处理完后，调用 exosip_call_build_ack 将协议规范中需要的 ack 发送给对端。在该接口中首先调用 exosip_call_dialog_find 在 exosip 的 jcall 上查找同样 dialog_id 的 call 及其上的 dialog。如果没有找到，则返回错误，没有 call 对应，否则继续调用 exosip_find_last_invite 基于之前找到的 exosip_dialog 在其上再查找携带 invite 消息类型的事务。这里会将 incoming 和 outgoing 类型的事务都找一下，如果只存在一个，则取其之，否则，比较两个的 birth_time，取离当前时间点更接近的一个。找到会话后，调用 exosip_build_request_within_dialog 构造需要回复的 osip_message 类型的 ack 结构体实例及其部分内容。初始化的部分信息直接从 dialog 上获取，因为它保存了部分有关会话的关键信息。另一部分信息是从事务上来获取的，调用 exosip_call_reuse_contact，从事务的 orig_request 上确定 osip_message 类型的信息，将其拷贝到这里创建的 osip_message 类型的 ack 上。之后再对其进行一些其他方面的配置。这样，build ack 的工作就完成了。

在 call_accepted 里对 ack 还有一些其他设置，包括将 op 上的 contact 信息设置到这里的 ack 上，另外，如果 op 的 sdp_answer 存在，调用 set_sdp 将其配置到 ack 上，实际就是配置到其 sdp message body 上。最后调用 exosip_call_send_ack 将该 ack 发送给对端。与之前有所不同，这里是直接调用底层发送接口 cb_snd_message 将响应送给对端，而不是作为一个事务添加到事务队列上再交给 exosip 任务去处理。

之后调用 sal 上注册的回调函数 call_accepted 来基于 op 进行处理。

在回调函数 call_accepted 中，首先判断 call 的状态，如果是 lcstateavrunning，则说明是已经 accepted 了，直接返回就可以了，否则继续处理。

如果 linphone_core 上 audiostream 的 ticker 不为空，说明在其他地方之前已经启动媒体传输了，先调用 linphone_core_stop_media_streams 停掉媒体流，再调用 linphone_core_init_media_streams 重新创建一个媒体流。

如果 call 的 resultdesc 不为空，则先释放它，通过递减引用计数来做到。之后将之前协商的存在于 op 上的 resultdesc 拷贝给这里的 call。之前是通过 localmedia 和 remotemedia 来协商最终二者都能兼容的媒体类型的。如果拷贝成功，则增加该 resultdesc 的引用计数，并将 call 的 media_pending 设置为 FALSE。

如果 resultdesc 存在了，并且其上的流描述信息不为空，则将 linphonecore 的 state 设置为

gstate_call_out_connected，然后调用 linphone_connect_incoming 继续处理来打开媒体流的传输。

在 linphone_connect_incoming 接口中，首先显示 connected 状态，然后将 call 的状态设置为 lcstateavrunning，如果 linphonecore 的 ringstream 不为空，则先调用 ring_stop 停止流，然后调用 linphone_core_start_media_streams 开启流媒体的传输。

在 start_media_streams 接口中，首先将 rtp 的 jitter 补偿设置为声卡延迟和 rtp 中有关 audio 的 jitter 配置二者大的一方。设置 call 的 media_start_time。

首先处理音频部分，调用 sal_media_description_find_stream 从 call 的 resultdesc 中找 proto 是 salprotortpavp 类型是 salaudio 的 stream，实际上是 stream 描述符。如找到了并且端口不为 0，则继续处理，否则表明没有响应媒体的定义，出错返回。有说明在本地支持前提下，在和对端协商结果下，可以进行音频流的处理和传输。此时，调用 make_profile 基于媒体描述和流的描述创建会话需要的 rtpprofile。在该接口中先调用 rtp_profile_new 创建一个 rtpprofile 类型的结构体，名称为 call profile，遍历之前找到的 straem 流的描述符，取出其上支持的所有的 payloadtype，将他们添加到刚创建的 call profile 中。另外，还有其他一些操作，包括保存了第一个 payload 的序号，根据带宽的调整，配置了相应的 payloadtype。通过该接口创建的 rtpprofile 被赋值给 call 供当前会话使用。之前在初始化时已经创建了全部 profile，av_profile，这是在 RTP 初始化过程中进行的，而这里创建的应该是该 profile 的一个子集，针对其上的 payloadtype 来说。如果 linphonecore 的 use_files 没有被设置，则取出音频播放设备和捕获设备的描述信息，调用接口 audio_stream_start_now 将这两个描述符作为参数传递进去启动音频传输。

audio_stream_start_now 接口进一步调用 audio_stream_start_full 进行处理。这几个接口是 mediastream 库提供的。在 start_full 中，第一步调用 rtp_session_set_profile 将 profile 设置到流的 rtpsession 上。如果参数远程端口大于零，则调用 rtp_session_set_remote_addr_full 将远程 ip 地址，远程端口以及远程 rtcp 端口号设置到 session 上。这里不仅将这些参数设置到 rtpsession 上的相关域保存起来了，而且也调用了 socket 的 connect 与远端同时建立连接了。之后会将已建立连接的标识 rtp_socket_connected 设置到 session 的 flags 上。接着将 payload 号和 jitter 补偿分别设置到 rtpsession 上。

如果远端端口号大于零，调用 ms_filter_call_method 调用 filter 的 methods，id 为 ms_rtp_send_set_session。在这里，filter 下面有 filter-desc,filter_desc 上面有 methods，可以用来对 filter 进行配置。Ms_rtp_send_set_session 这个 id 将调用函数 sender_set_session 这个接口，在该接口中根据 rtpsession 中的 send 上的 profile 和 payload number 得到 payload type。如果 payload type 的 mime_type 为 g722，senderdata 数据结构体上的 rate 设置为 8000，否则根据 payload type 的 clock_rate 来设置。最后将 senderdata 的 session 指向 rtpsession。Senderdata 结构体实例是 send filter 的 data 指针指向其私有的数据，包含了与 RTP 发送有关的信息。对此，其他 filter 也类似，也就是说每个 filter 都有其私有的数据，用于其在 process 中使用，比如 recordfile 对应的 filter 的私有数据为文件指针，也就是保存数据的文件指针，这样就很好理解了。

调用 ms_filter_new 创建一个新的 filter，根据 id ms_rtp_recv_id 从 desc_list 上取得 id 一致的 filter 描述符，然后基于该描述符创建新的 filter，并将 filter 的 desc 指针指向该描述符。接着类似发送，调用 filter 的 methods，id 为 ms_rtp_recv_set_session 配置 rtpsesson 的 recvsession。该 id 会调用接口 receiver_set_session。在该接口中的操作同样类似于 sender 中的。

调用 ms_filter_new 创建 dtmf 的 filter 放到 stream 的 dtmfgen 上。

调用 rtp_session_signal_connect 接口注册回调函数，如果有电话音，telephone events 调用 on_dtmf_received，payload type changed 调用 payload_type_changed 接口。

接着配置音频读取和播放卡设备。如果 captcard 不为空，则基于该描述符调用 ms_snd_card_create_reader 创建音频数据读取的 filter。这是通过该描述符的 create_reader 函数完成的。如果为空，则调用 ms_filter_new 创建一个 ms_file_player_id 的 filter，交给 audio_stream 的 soundread。

另外，创建一个 ms_resample_id 的 filter 给 audio_stream 的 read_resampler。

如果 infile 不为空，调用 audio_stream_play 基于 stream 播放该文件。

如果 playcard 不空，则调用 ms_snd_card_create_writer 基于该 card 创建 soundwrite，及音频数据的发送 filter。否则，则调用 ms_filter_new 创建 ms_file_rec_id 的 filter，并在 outfile 参数不为空的情况下调用 audio_stream_record 记录接收的音频数据保存到 outfile 中。

到这里声卡设备的 filter 都已经创建了。接着创建编码和解码器的 filter。首先调用 rtp_profile_get_payload 从 profile 和 payload 参数中取出 payload type。接着调用 ms_filter_create_encoder 和 ms_filter_create_decoder 接口基于 payload type 的 mime_type 创建解码器和编码器的 filter，并保存到 stream 的 encoder 和 decoder 两个 filter 域上。实际上在这个接口中，都是根据 mime_type 查找 desc_list 全局链表，找到匹配的 desc，再创建新的 filter 结构体实例并把 desc 放到 filter 上。

如果支持回身消除，则创建 ms_speex_ec_id 的 filter 放到 streame 上，调用 ms_filter_set_sample_rate 对应的 methods 设置采样码率。如果 stream 上 ec_tail_len，ec_delay，ec_framesize 等不为空，就调用 filter 的对应的方法将这些配置值设置到 filter 上去。这些值可能是在初始化时或者协商媒体时得到的，当时只有转而配置到当前用的 filter 上才能具体的在本次通话过程中其作用。

如果 stream 的 el_type 不等于 elinactive 或者 use_gc 或者 use_ng 被置位了，则分别为 volsend 和 vlorecv 创建两个 filter，这可能是用来远程传输音量信息的。如果 el_type 不等于 ELInactive，则说明？调用 filter 的 MS_VOLUME_SET_PEER 对应的 methods 将 volsend 和 volrecv 联系起来。接着判断如果等于 ELControlFull 则调用 filter methods 配置 volrecv 的 MS_VOLUME_ENABLE_NOISE_GATE。如果 use_ng 被设置了，调用 methods 将 volsend 的 filter 的 MS_VOLUME_ENABLE_NOISE_GATE 使能。

如果 use_agc 被配置了，调用 volsend 该 filter 的 MS_VOLUME_ENABLE_AGC 对应的 methods 使能 agc

根据之前获得的 payload type 的 clockrate 信息调用 soundread 和 soundwrite filter 的 MS_FILTER_SET_SAMPLE_RATE 方法将这些配置信息设置到 filter 中。另外，如果 read_resampler 和 write_resampler 如果为空，则调用 ms_filter_new 基于 MS_RESAMPLE_ID 创建两个 filter。

基于 MS_FILTER_SET_NCHANNELS 方法设置 soundwirte 该 filter。

基于 payload type 中 clockrate 的配置来设置 encoder 编码器 filter 的采样率，通过 MS_FILTER_SET_SAMPLE_RATE 对应的 methods。用 normal_bitrate 配置通过 MS_FILTER_SET_BITRATE 配置编码器 filter 的 Bitrate。

基于 clock_rate 配置解码器 filter 的采样率，id 为 MS_FILTER_SET_SAMPLE_RATE 的方法完成。

如果 send_fmtp 不为空，将其添加到编码器 filter 上，如果 recv_fmtp 不为空，将其添加到解码器 filter 上。

调用 ms_filter_new 基于 MS_EQUALIZER_ID 创建平衡 filter，放到 stream 的 equalizer 上。并通过 MS_EQUALIZER_SET_ACTIVE 方法将 eq_active 设置到该 filter 上。

如果 read_resampler 和 write_resampler 不为空，调用 audio_stream_configure_resampler 用声卡 filter 和 RTPfilter 来配置均衡 filter。该接口将声卡或者 RTP 的采样率取出来，配置到 resample 中。对于 read_resample，使用 soundread 配置 from 的 rate，rtpsend 配置 to 的 rate，对于 writer_resample，rtprecv 和 soundwrite 分别为 from 和 to 的 rate。

下一步将所有的 filter 连接起来。通过调用 ms_connection_helper_start 和 ms_connection_helper_link 两个接口完成。

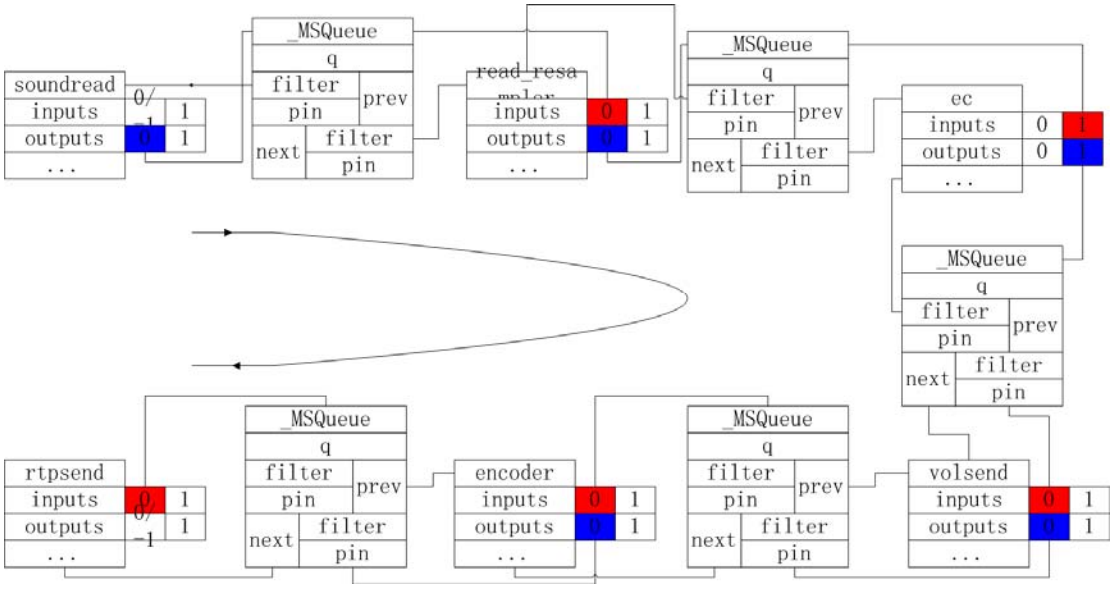
对于 sending graph，为：

Soundread-->read_resampler-->ec-->volsend-->encoder-->rtpsend

对于 recv graph，为：

Rtprecv-->decoder-->dtmfgen-->equalizer-->volrecv-->ec-->write_resampler-->soundwrite

连接后的数据结构体间的关系如下图所示，以 sending 为例



除了 filter 连接在一起外，还需要一个发动机来推动数据在所有 filter 间流动，之前是建好了通路，现在需要一个泵来推数据在这个通路中流动。在代码中就是需要创建一个 ticker。这时通过 ms_ticker_new 接口来完成的，挂载在 stream 上。通过该接口也同时创建了 ticker 需要的 thread，并启动了该 thread。接着调用 ms_ticker_attach 将 soundread 和 rtprecv 两个 filter 绑定到 ticker 上。在该接口中，我们首先遍历 filter 链上的所有 filter，调用 preprocess 接口，即将所有预先要处理的操作通过 preprocess 接口来完成，这在构造 filter 时需要注意。接着将要绑定的 filter 连到 ticker 的 execution_list 上。实际上后续 ticker 任务在跑动的时候，就是遍历 execution_list 链表，从各个 graph 的 filter 头开始将链上的各个 filter 的 process 接口调用执行一遍，这样反复循环，数据就从一个 filter 传输到下一个 filter 了。

至此 audio_stream_start_full 接口处理就完成了。

Ticker 任务运行的函数为 ms_ticker_run，在该接口中就是调用 run_graphs 接口将 graph 的通路走一遍的。在 run_graphs 中，遍历 ticker 的 execution_list 列表，对其上的 filter 头执行 run_graph。在 run_graph 接口中，如果 filter 的 last_ticks 不等于 ticker 的 ticks 才执行 filter 的处理。该 ticker 值可以用来表示该 filter 是否是刚处理过的 filter，避免接连的处理。如果需要处理，首先调用 filter_can_process 查看该 filter 之前是否有一个已经在运行了。这一步是通过如下方式判断的：遍历 filter 的所有 input 对应的 msqueue，如果该 queue 的 prev 指针对应的 filter 的 last_tick 不等于 ticker 上的 ticks，则返回失败，说明该 filter 之前的 filter 还在处理中，所以当前 filter 不可处理，也就是说该 filter 的所有 input 还没有完全被其之前的 filter 填充完，还有 input 没有接收完其前面 filter 的数据，因此将其放到未调用 filter 链表上。否则返回成功，处理该 filter。

此时将 ticker 的 ticks 赋值给 filter 的 last_tick，后续判断二者相等即可表明该 filter 已经被处理了。接着调用 call_process 处理。在该接口中，首先判断如果该 filter 的 desc 指示的 filter 的 ninputs 为零，表明该 filter 只有一个输入，或者 desc 指示的标识 flags 为 MS_FILTER_IS_PUMP，则调用 ms_filter_process 直接处理。否则，检查 filter 的所有 input 端口对应的 queue，如果有数据就调用 ms_filter_process 处理。至于将多个 input 端口的处理分开来，主要是考虑到在一次调用中需要将 filter 的所有 input 处理完，因此如果有多个 input，接口将多次调用 ms_filter_process 直到每个 input 都被处理了。

在 ms_filter_process 中，调用 filter 的 desc 的 process 接口函数指针进行处理。实际上每个 filter

的核心处理在创建该 filter 时，其 desc 已经全都准备好处理接口了。

退出 call_process 退回到 run_graph 中，处理完当前 filter 后，对于 filter 的所有 output 对应的 queue，对其 next 指针指向的 filter 继续调用 run_graph 接口进行处理。实际上就是对当前 filter 之后的所有 filter 调用 run_graph 继续处理。可以看出 run_graph 在这里是递归调用，所以 graph 上的所有 filter，当然前提是 graph 创建的按照规则创建的，那么所有的 filter 的 desc 的 process 接口都将被调用执行。等推出 run_graph，在 run_graphs 中会检查 unschedulable 变量，我们之前将所有没有处理的 filter 都放到该变量的链表上了，对于 unschedulable 继续调用 run_graphs，可以看出这里 run_graphs 也是递归调用。等退出 run_graphs，那么所有第一次可以处理的和第一次没有处理后续可以处理的 filter 都被处理过了，至此沿着 queue 建立的链接 graph 对应的 filter 通路就被走了一遍。

回到 ms_ticker_run，ticker 的 time 被增加 ticker 的 interval 变量指示的值。之后判断 time 和 ticker 运行的实际时间的差值，如果大于零，说明 ticker 在本次 interval 时间值到达之前就已经处理完所有 filter 了，那么就调用 sleepMs 打发掉这个差值。如果差值不大于零，说明休息时间到了，需要注意的是这里是在 while(1)中处理的，肯定会出现小于等于零的情况，如果差值的绝对值过大，则说明休息过头了，会打印警告信息，跳出 while(1)的处理接着处理下次的 run_graphs。

对于 RTP 部分的传送，就需要看看 filter 的 process 怎么处理，对其他 filter 也一样。

数据是在 filter 的 process 中传递的，因为 filter 基本上都是靠 msqueue 连接在一起的，所以数据的传递也基本是通过 queue 的 m_blk 来完成的。关于 m_blk 的构造，参考 ortp 分析。

之前是在 linphonecore 没有配置 use_files 的情况下直接使用声卡设备来获取和播放数据的，如果配置了上述变量，linphonecore 将用文件来代替声卡设备，此时调用 audio_stream_start_with_files 接口，将文件指针传递进去以取代声卡设备。内部处理逻辑基本上同使用声卡设备的处理，这里就不再进行说明了。

调用接口 post_configure_audio_streams 对 audio stream 进行后期的一些处理。在该接口中主要读取配置文件或者 sound.conf 结构体上的许多配置信息，调用 filter 的 call methods 接口，通过 id 选择相应的接口对 filter 进行配置。这里面配置项比较多繁杂，就不在详细说明。但是这里的配置可能比较影响最终的音频传输效果。

调用接口 audio_stream_set_rtcp_information 配置 rtcp 的版本信息。

至此音频处理完了，接着处理视频部分。视频部分流程基本同音频部分。

最后将 call 的 state 设置为 LCStateAVRunning。返回。

所有上述这些处理都是在 sal_iterate 的 process_event 中完成的。是当从底层取上来一个 CALL_ANSWER 事件触发上层任务来处理。接着调用 authentication_ok 后就跳出了 process_envent。也就跳出了 sal_iterate，到 linphone_core_iterate 中继续执行。

此时 linphonecore 上的 call 不再为空，而且状态也为 LCStateAVRunning 了，在该状态下，linphonecore 会在调试状态下显示当前的带宽使用情况，对于视频调用 video_stream_iterate 进行流量控制，对于音频调用 audio_stream_alive 保持连接畅通。

至此，协议的处理和音视频的处理以及二者的关联也就都完成了。

3 Answer 过程分析

事件的触发：对端发送 invite 请求给客户端，请求建立 sip 协商与连接，呼叫客户端，建立视频通话。

数据到达底层硬件，接收到后缓存给上层应用。

系统此时已经启动了 exosip_thread 线程，该线程循环运行，每次都会调用接收接口检查底层是否有数据到达。因此，当 invite 请求数据到来时，exosip 会调用 eXosip_read_message 接口通过底层

socket 接口接收数据，并解析数据，将解析的结果根据软件架构和 sip 状态机及事务处理机制存放到各个 osip_transaction 上。这块基本处理逻辑同上面的响应处理部分。

底层的 sip 处理模块，包括 osip 和 exosip，处理 invite 请求，通过状态机处理各种需要处理的状态，并发送响应给请求端。当底层 SIP 模块对 sip 消息处理差不多时，会将进一步的处理请求交给上层 linphone core 任务，因为它还要急着处理接收和发送 sip 消息的任务，至于上层的一些处理它就不管了。这种角色转换是通过发送一个事件到事务链表上完成的。

上层 linphone core 任务每隔一秒就会迭代处理 sal 模块和音视频模块。在 sal 模块，此时会接收到一个事务上的事件，并进入事件处理中。

从代码中的关联来看，exosip 任务处理 osip 事务状态机上的事件，完成后调用回调函数，回调函数将剩余的处理交给 exosip 的事件队列等待 main 任务主线程去处理。回调函数在文件 jcallback.c 中，udp.c 也有一些。上层事件处理在 sal_exosip2.c 中，事件定义在 exosip.h 中，因此可以通过关联这三个文件和打印来跟踪底层消息和上层消息之间的流动和传递。

关于上传给上层的事件：

注册成功，上传 1

Keepalive 过程，上传 27

底层协商 invite 通路打通，进行媒体处理，上传 5，建立一个新的 call.

对于 invite，关键的一个事件处理是 EXOSIP_CALL_INVITE 类型的事件。也正如上面的介绍，当会话请求是由对端发起时，底层 exosip 任务接收到数据包后查找事件队列此时并不能找到针对该响应的事件队列，此时会调用 exosip_process_newrequest 来处理，在该接口中如果判断到是 invite 消息时会调用 exosip_process_new_invite 接口处理。在 new_invite 中处理完必要的初始化工作后，会调用 report_call_event 将 exosip_call_invite 消息上报。对于该事件，linphone core 调用 inc_new_call 来处理。

offer_answer_initiate_outgoing 接口创建一个 salmediea 媒体描述符。在这里我们会获取我们本地支持的音视频编解码类型，来和对端 sdp 中给出的编解码类型进行比较，如果有匹配的，说明相互音视频编解码的类型协商成功，后续进行音视频处理时就用协商好的类型。如果音视频类型没有在本地找到，也就是说对端发起 invite 请求时给定的类型本地都不支持，对于 linphone 来讲，不会对这种错误进行其他措施来努力使协商成功，而是认为当前 sip 协商不成功，无法建立 sal media 媒体描述符。此时一个空的媒体描述符会被返回，后续处理中除了给对端发送 415 错误状态外，linphone core 还会将 call，就是刚才建立的 call 销毁，取消本次会话。（这在调试客户端时遇到了，当添加上对之前不支持媒体类型的支持后，即使支持是错误的，也能够使 sip 会话成功，只是后续媒体流就会有问题。另外，sip 客户端的等待 answer 的时间有点短，这个需要调整一下。）

4 关于 RTP 及音视频流的网络传输

在 linphone 中，RTP 传输是通过 ortp 库完成的。Ortp 库的初始化在 linphone 初始化时就调用了，为 ortp_init。要基于 RTP 传输音视频，初始化完成后最主要的一步就是创建 RTP 会话结构体，也就是 rtpsession，后续所有的有关 RTP 传输方面的配置等都是基于该 session 来完成的。

这部分先从 RTPfilter 看起，进一步的从 rtpsender 的 filter 来看。

在 sender_process 接口中，从参数 filter 上可以得到私有数据 SenderData，从 SenderData 上可以得到 rtpsession，这个 session 就是之前初始化时创建的。

如果 session 为空，就是没有 session，则直接调用 ms_queue_flush 扔掉到该 filter 的队列的数据直接返回。否则继续。

在 while 循环中处理 filter 上的所有数据。这里是调用 ms_queue_get 从 filter 的 inputs 队列上取 mblk 类型的数据块。如果不为空就一直取，作为 while 循环的条件，所以在该循环中会取光队列上

的所有数据块并对其进行处理。

根据 filter 和数据块调用 get_cur_timestamp 得到当前的时间戳。如果私有数据部分定义了 skip，调用 send_dtmf 发送 dtmf 信息。

如果私有数据的 skip 和 mute_mic 有一个不为 FALSE 就简单的调用 freemsg 释放掉该数据块，否则取得该数据块中承载的 payloadtype 的 number 号，调用 rtp_session_create_packet 创建一个 RTP 数据包。如果 payload type 的 number 号大于 0，那么调用 rtp_set_payload_type 将 RTP header 的 payloadtype 设置为该值。这里实际上是将 RTP 的 header 封装到 mblk 中的，该结构体的 b_cont 指针指向当前从队列中取得的 mblk 上。接着调用 rtp_session_sendm_with_ts 发送数据。

有关 rtp_session_sendm_with_ts 发送的处理，参见 ortp 分析文档。

数据发送完成后，对 RTP 的 senderdata 私有数据进行一些配置，接着继续下次的 while 循环处理。

其实基本的流程还是挺简单的，就是从 filter 上不断的取 mblk，然后基于当前会话构造 RTP 数据包，并将其发送出去。外部的其他数据就包括 filter 上的私有数据 senderdata。而 session 早在 sip 协商过程中就在媒体初始化中完成了。

看完了发送部分，接着再看接收部分。RTP 数据流的接收在 filter 的 receiver_process 中处理。首先计算一个时间戳值，接着在 while 中掉用 rtp_session_recvm_with_ts 接收数据包，这作为一个循环条件，只要 session 上的数据没有接收完就一直接收。数据接收上来后，对于一个 mblk，从 RTP 头中取得时间戳保存到 mblk 的 reserved1 中，取得 markbit 和 payload type 放到 reserved2 中。接着调用 rtp_get_payload 得到真正的 payload 数据，实际上就是让数据开始指针跳过 RTP 头，指向真正的数据区开始部分，并重新设置数据的长度。这些都是在 mblk 自身上进行的，没有创建新的 mblk。最后将调整后的 mblk 放到 filter 的 output0 中，交给下一个 filter 去处理。这是通过调用 ms_queue_put 完成的，filter 的 output 是个 queue。

同样，对于取数据的接口 rtp_session_recvm_with_ts，处理步骤参考 ortp 分析文档。

至此，RTP 的接收和发送就基本上理通了。

5 总结

创建或者初始化一个新的会话主要并且必须进行的操作步骤包括：

linphone_call_new_incoming：创建新的 call 结构体

创建新的 sal 结构体

sal_call_set_local_media_description：添加描述信息

linphone_core_init_media_streams：创建 audio 和 video stream 结构体，并设置参数

Sal 开头的后续操作解决协议协商问题

Videostreamstart 等相关接口处理流的 filter 创建和 link 以及 atacher 到 ticker 上，启动 RTP 数据传输。

初始化基本完成后，会有响铃提示音，不管是拨出着还是拨入，都会响铃。对于拨入，响铃提示音提示输入 answer，linphone 据此会进行下一步的处理，如果超时时间达到，还没有输入 answer，就会自动 terminate 当前的 call。

用户输入 answer 后，调用 lpc_cmd_answer 进行该命令的处理，内部调用 linphone_core_accept_call 处理会话相关逻辑。此时首先停止 ring stream，也就是响铃流。其次调用 sal_call_accept 发送 acknowledge 给对端，最后调用 linphone_core_start_media_streams 启动流的传输。

对于主动拨出的会话，sal 在处理 events 的时候会处理到 ring 事件，然后调用 ring_start 启动 ring stream，之后客户端会提示响铃。当对端输入 answer 后，本地在 sal 迭代处理时处理 EXOSIP_CALL_ANSWERED 消息，该消息表明对端输入了 answer 接受了 invite。在该消息的处理中，调用 call_accepted，在 call_accepted 中检查时间链上的 response，处理流，构建并发送 acknowledge

响应，之后调用 call_accepted 回调函数。在回调函数中调用 linphone_connect_incoming 处理输入流请求。在该接口处理中，显示 connected 状态，设置 call 的状态为 LCStateAVRunning，如果 ringstreamer 还存在，则调用 ring_stop 停止响铃。之后调用 linphone_core_start_media_streams 启动流媒体传输处理。

对于 linphone，当 call 创建后，只有三个状态来处理，一个是超时的 invite 重发，一个是响铃处理，一个是视频流的监控。Call 的 state 主要在每次的迭代处理中进行检查。主要有 init 状态，preestablishing 状态，ringing 状态和 avrunning 状态。第一个为初始化，在 init 一个 call 时，或者发起 invite 时用，第二个为 sip 会话协商，在 options ping 中用，第三个为响铃状态，第四个为音视频流处理状态。

Linphone core 全局状态中有关 call 的状态主要指明当前的 linphone core 上的 call 处于 idle、输出 invite，输出 connected、输入 invite、输入 connected、end、error、invalid、输出 ringing 等状态。其实表明了当前会话协商的状态如何。这些状态，严格来说，针对每个 call 都会有一套类似的状态处理，只是当前版本还不支持多 call 会话而已。

对于当前这种只支持一个 call 实例的情况，idle，invite 状态会与 linphone 的 init prestablishing 状态重叠，而输出 ringing 状态和 linphoen 的 ringing 状态重叠，connected 状态后 linphone 也就到了 avrunning 状态了。

最后，在看代码过程中，需要区分 linphone core、call 以及 sal 三个实例。

七 linphone 会话执行过程 log 分析

开机初始化部分

```
ortp-error-oRTP-0.16.3 initialized.
ortp-error-Registering all filters...
ortp-error-Registering all soundcard handlers
ALSA lib pcm_hw.c:1433:(_snd_pcm_hw_open) Invalid value for card
ALSA lib pcm_hw.c:1433:(_snd_pcm_hw_open) Invalid value for card
ortp-error-Card ALSA: default device added
ALSA lib conf.c:4600:(snd_config_expand) Unknown parameters 0
ALSA lib control.c:902:(snd_ctl_open_noupdate) Invalid CTL default:0
ortp-error-Card ALSA: STx7105 audio subsystem added
ortp-error-Card OSS: /dev/dsp added
ortp-error-Registering all webcam handlers
ortp-error-Webcam StaticImage: Static picture added
ortp-error-Loading plugins
ortp-error-Cannot open directory /home/ywg/st7162_gw/webs_base/src/linphone/sh-target/lib/mediastreamer/plugins: No such file or directory
ortp-error-ms_init() done
ortp-error-Cannot open directory /home/ywg/st7162_gw/webs_base/src/linphone/sh-target/lib/liblinphone/plugins: No such file or directory
| INFO2 | <eXutils.c: 811> DNS resolution with 0.0.0.0:5060
| INFO2 | <eXutils.c: 827> getaddrinfo returned the following addresses:
| INFO2 | <eXutils.c: 832> 0.0.0.0 port 5060
*****entry eXosip_listen_addr start exosip thread
```

*****entry _eXosip_thread ...
*****entry eXosip_execute -- timeout is 31536000--0
| INFO2 | <eXconf.c: 747> eXosip: Reseting timer to 15s before waking up!
Ready
Warning: video is disabled in linphonec, use -V or -C or -D to enable.
linphonec> ortp-error-New local ip address is 192.168.4.175
ortp-error-Network state is now [UP]

接收到连接请求包 options 包
| INFO1 | <eXtl_udp.c: 285> Received message:
OPTIONS sip:root@192.168.4.175 SIP/2.0
Via: SIP/2.0/UDP 192.168.4.177:5060;rport;branch=z9hG4bK12788
From: <sip:toto@192.168.4.177>;tag=25462
To: <sip:root@192.168.4.175>
Call-ID: 6778
CSeq: 20 OPTIONS
Accept: application/sdp
Max-Forwards: 70
User-Agent: Linphone/3.3.2 (eXosip2/3.3.0)
Expires: 120
Content-Length: 0

| INFO1 | <eXtl_udp.c: 332> Message received from: 192.168.4.177:5060
| INFO1 | <eXtl_udp.c: 339> Message received from: 192.168.4.177:5060
| INFO3 | <osip_event.c: 89> MESSAGE REC. CALLID:6778
| INFO1 | <udp.c: 1554> Message received from: 192.168.4.177:5060
| INFO1 | <udp.c: 1564> This is a request
| INFO2 | <osip_transaction.c: 138> allocating transaction ressource 1 6778
| INFO2 | <nist.c: 32> allocating NIST context
| INFO2 | <osip.c: 1677> execute nist transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 1
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 15
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 12
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 43be68
*****entry fsm_callmethod --- type12 state 15
| INFO3 | <jcallback.c: 772> cb_rcvunkrequest (id=1)
*****entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1677> execute nist transaction events !
*****entry _eXosip_thread ...
*****entry eXosip_execute -- timeout is 31536000--0
| INFO2 | <eXconf.c: 747> eXosip: Reseting timer to 15s before waking up!

| INFO2 | <osip.c: 1677> execute nist transaction events !

第一次解析没有完成，消息包又被重新发送了一次，再次接收到刚才的消息
*****entry _eXosip_thread ...
*****entry eXosip_execute -- timeout is 31536000--0
| INFO2 | <eXconf.c: 747> eXosip: Reseting timer to 15s before waking up!
| INFO1 | <eXtl_udp.c: 285> Received message:
OPTIONS sip:root@192.168.4.175 SIP/2.0
Via: SIP/2.0/UDP 192.168.4.177:5060;rport;branch=z9hG4bK12788
From: <sip:toto@192.168.4.177>;tag=25462
To: <sip:root@192.168.4.175>
Call-ID: 6778
CSeq: 20 OPTIONS
Accept: application/sdp
Max-Forwards: 70
User-Agent: Linphone/3.3.2 (eXosip2/3.3.0)
Expires: 120
Content-Length: 0

| INFO1 | <eXtl_udp.c: 332> Message received from: 192.168.4.177:5060
| INFO1 | <eXtl_udp.c: 339> Message received from: 192.168.4.177:5060
| INFO3 | <osip_event.c: 89> MESSAGE REC. CALLID:6778
| INFO1 | <udp.c: 1554> Message received from: 192.168.4.177:5060
| INFO2 | <osip.c: 1677> execute nist transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 1
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 16
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 12
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 43b550
| INFO3 | <osip_transaction.c: 398> USELESS event!
| INFO2 | <osip.c: 1677> execute nist transaction events !

上层接收到底层处理完的事件，接着处理刚才的消息
ortp-error-linphone process event get a message 27
ortp-error-in other_request
将请求响应作为一个事件交给底层 exosip 任务

底层 exosip 任务读取到上层事件，发送响应
| INFO2 | <osip.c: 1677> execute nist transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 1
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 16
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 20

| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 43b550
*****entry fsm_callmethod --- type20 state 16
*****entry cb_snd_message to send a msg
| INFO2 | <eXutils.c: 811> DNS resolution with 192.168.4.177:5060
| INFO2 | <eXutils.c: 827> getaddrinfo returned the following addresses:
| INFO2 | <eXutils.c: 832> 192.168.4.177 port 5060
| INFO1 | <eXtl_udp.c: 574> Message sent: (to dest=192.168.4.177:5060)
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.4.177:5060;rport=5060;branch=z9hG4bK12788
From: <sip:toto@192.168.4.177>;tag=25462
To: <sip:root@192.168.4.175>;tag=1919264482
Call-ID: 6778
CSeq: 20 OPTIONS
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, SUBSCRIBE, NOTIFY, INFO
Accept: application/sdp
User-Agent: Linphone/3.3.2 (eXosip2/3.1.0)
Content-Length: 0

| INFO3 | <jcallback.c: 2145> cb_snd123456xx (id=1)

*****entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1677> execute nist transaction events !

服务器发送 invite 包过来，由 exosip 任务处理
*****entry _eXosip_thread ...
*****entry eXosip_execute -- timeout is 31--999774
| INFO2 | <eXconf.c: 747> eXosip: Reseting timer to 15s before waking up!
| INFO1 | <eXtl_udp.c: 285> Received message:
INVITE sip:root@192.168.4.175 SIP/2.0
Via: SIP/2.0/UDP 192.168.4.177:5060;rport;branch=z9hG4bK20585
From: <sip:toto@192.168.4.177>;tag=22326
To: <sip:root@192.168.4.175>
Call-ID: 21074
CSeq: 20 INVITE
Contact: <sip:toto@192.168.4.177>
Content-Type: application/sdp
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO
Max-Forwards: 70
User-Agent: Linphone/3.3.2 (eXosip2/3.3.0)
Subject: Phone call
Content-Length: 406

```
v=0
o=toto 123456 654321 IN IP4 192.168.4.177
s=A conversation
c=IN IP4 192.168.4.177
t=0 0
m=audio 7078 RTP/AVP 112 111 110 3 0 8 101
a=rtpmap:112 speex/32000/1
a=fmtp:112 vbr=on
a=rtpmap:111 speex/16000/1
a=fmtp:111 vbr=on
a=rtpmap:110 speex/8000/1
a=fmtp:110 vbr=on
a=rtpmap:3 GSM/8000/1
a=rtpmap:0 PCMU/8000/1
a=rtpmap:8 PCMA/8000/1
a=rtpmap:101 telephone-event/8000/1
a=fmtp:101 0-11

| INFO1 | <eXtl_udp.c: 332> Message received from: 192.168.4.177:5060
| INFO1 | <eXtl_udp.c: 339> Message received from: 192.168.4.177:5060
| INFO3 | <osip_event.c: 89> MESSAGE REC. CALLID:21074
| INFO1 | <udp.c: 1554> Message received from: 192.168.4.177:5060
| INFO1 | <udp.c: 1564> This is a request
| INFO2 | <osip_transaction.c: 138> allocating transaction ressource 2 21074
| INFO2 | <ist.c: 31> allocating IST context
| INFO2 | <osip.c: 1544> execute ist transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 2
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 5
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 10
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 43b6a0
*****entry fsm_callmethod --- type10 state 5
| INFO3 | <jcallback.c: 712> cb_rcvinvite (id=2)    //osip 状态机处理第一个事件，并调用回调
*****entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1544> execute ist transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 2
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 6
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 19
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 43afc0
*****entry fsm_callmethod --- type19 state 6      //osip 状态机处理第二个事件，并调用回调
*****entry cb_snd_message to send a msg
| INFO2 | <eXutils.c: 811> DNS resolution with 192.168.4.177:5060
| INFO2 | <eXutils.c: 827> getaddrinfo returned the following addresses:
```

| INFO2 | <eXutils.c: 832> 192.168.4.177 port 5060
| INFO1 | <eXtl_udp.c: 574> Message sent: (to dest=192.168.4.177:5060)
SIP/2.0 100 Trying
Via: SIP/2.0/UDP 192.168.4.177:5060;rport=5060;branch=z9hG4bK20585
From: <sip:toto@192.168.4.177>;tag=22326
To: <sip:root@192.168.4.175>
Call-ID: 21074
CSeq: 20 INVITE
User-Agent: Linphone/3.3.2 (eXosip2/3.1.0)
Content-Length: 0

| INFO3 | <jcallback.c: 2145> cb_snd123456xx (id=2)
*****entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1544> execute ist transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 2
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 6
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 19
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 440cb8
*****entry fsm_callmethod --- type19 state 6
*****entry cb_snd_message to send a msg　　//osip 状态机处理第二个事件，并调用回调
| INFO2 | <eXutils.c: 811> DNS resolution with 192.168.4.177:5060
| INFO2 | <eXutils.c: 827> getaddrinfo returned the following addresses:
| INFO2 | <eXutils.c: 832> 192.168.4.177 port 5060
| INFO1 | <eXtl_udp.c: 574> Message sent: (to dest=192.168.4.177:5060)
SIP/2.0 101 Dialog Establishement
Via: SIP/2.0/UDP 192.168.4.177:5060;rport=5060;branch=z9hG4bK20585
From: <sip:toto@192.168.4.177>;tag=22326
To: <sip:root@192.168.4.175>;tag=1263961124
Call-ID: 21074
CSeq: 20 INVITE
Contact: <sip:root@192.168.4.175:5060>
User-Agent: Linphone/3.3.2 (eXosip2/3.1.0)
Content-Length: 0

| INFO3 | <jcallback.c: 2145> cb_snd123456xx (id=2)
*****entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1544> execute ist transaction events ! //处理其他 transactions，之前是一个 tr 上的 events
| INFO2 | <osip.c: 1544> execute ist transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !

底层在处理完 SIP 协议响应以后，发送一个 sip 会话事件给上层 linphone 应用，这里为新的 call
ortp-error-linphone process event get a message 5
ortp-error-CALL_NEW
在 call-new 的处理中，sal 根据之前保留的有关该会话的请求信息，解析出其中的 SDP 数据。
在 sdp_to_media_description 中
ortp-error-Found payload speex/32000 fmtp=vbr=on
ortp-error-Found payload speex/16000 fmtp=vbr=on
ortp-error-Found payload speex/8000 fmtp=vbr=on
ortp-error-Found payload GSM/8000 fmtp=
ortp-error-Found payload PCMU/8000 fmtp=
ortp-error-Found payload PCMA/8000 fmtp=
ortp-error-Found payload telephone-event/8000 fmtp=0-11
释放 sdp 信息结构，相关信息已被拷贝出来，到 sdl_op 中
调用 sal_add_call 添加当前的 call
调用 call_received 回调函数处理当前 call
调用 sal_ping 发送一个 options 的 ping 给对端，这样在发送 answer 之前有一个获取我们自己网络地址的机会。在 sal_ping 中构建 options 请求信息，创建一个 nict 事务，将请求事件放到该事务上。
| INFO2 | <osip_transaction.c: 138> allocating transaction ressource 3 771801749
| INFO2 | <nict.c: 34> allocating NICT context
底层任务会检查到该事件并进行处理

底层任务将该 options 发给对端，实际上是由 exosip 任务来处理的
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 3
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 10
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 18
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 43ec50
*****entry fsm_callmethod --- type18 state 10
*****entry cb_snd_message to send a msg
| INFO2 | <eXutils.c: 811> DNS resolution with 192.168.4.177:5060
| INFO2 | <eXutils.c: 827> getaddrinfo returned the following addresses:
| INFO2 | <eXutils.c: 832> 192.168.4.177 port 5060
| INFO1 | <eXtl_udp.c: 574> Message sent: (to dest=192.168.4.177:5060)
OPTIONS sip:192.168.4.177:5060 SIP/2.0
Via: SIP/2.0/UDP 192.168.4.175:5060;rport;branch=z9hG4bK2129816964
Route: <sip:toto@192.168.4.177>;tag=22326
From: <sip:root@192.168.4.175>;tag=1798315166
To: <sip:toto@192.168.4.177>;tag=22326
Call-ID: 771801749
CSeq: 20 OPTIONS
Accept: application/sdp
Max-Forwards: 70

User-Agent: Linphone/3.3.2 (eXosip2/3.1.0)
Expires: 120
Content-Length: 0

| INFO3 | <jcallback.c: 904> cb_sndoptions (id=3)
*****entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO2 | <osip.c: 1544> execute ist transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !

上层任务继续接着初始化当前创建的新 call。
ortp-error-Notifying all friends that we are in status 5

底层任务随后又接收到了对端对 options 的 200OK 响应
*****entry _eXosip_thread ...
*****entry eXosip_execute -- timeout is 0--499660
| INFO2 | <eXconf.c: 762> eXosip: timer sec:0 usec:100000!
ortp-error-Notifying all friends that we are in status 5
| INFO1 | <eXtl_udp.c: 285> Received message:
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.4.175:5060;rport=5060;branch=z9hG4bK2129816964
From: <sip:root@192.168.4.175>;tag=1798315166
To: <sip:toto@192.168.4.177>;tag=22326
Call-ID: 771801749
CSeq: 20 OPTIONS
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, SUBSCRIBE, NOTIFY, INFO
Accept: application/sdp
User-Agent: Linphone/3.3.2 (eXosip2/3.3.0)
Content-Length: 0

| INFO1 | <eXtl_udp.c: 332> Message received from: 192.168.4.177:5060
| INFO1 | <eXtl_udp.c: 339> Message received from: 192.168.4.177:5060
| INFO3 | <osip_event.c: 89> MESSAGE REC. CALLID:771801749
| INFO1 | <udp.c: 1554> Message received from: 192.168.4.177:5060
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 3
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 11
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 14
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 444730

*****entry fsm_callmethod --- type14 state 11
| INFO3 | <jcallback.c: 1441> cb_rcv2xx (id=3)
*****entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO2 | <osip.c: 1544> execute ist transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !

上层任务继续处理会话相关信息
ortp-error-Partial MTU discovered : 1500
ortp-error-send(): Message too long
ortp-error-Doing SDP offer/answer process
ortp-error-Processing for stream 0
ortp-error-No match for GSM/8000
<sip:toto@192.168.4.177> is contacting you.
linphonec> ortp-error-Starting local ring...
启动响铃后通知对端这边启动响铃了。这作为一个事件添加到 ist 的事务链上

Exosip 任务发现有事件需要处理，遍历事件链表处理事务，这里发送 180 响铃数据包给对端
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO2 | <osip.c: 1544> execute ist transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 2
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 6
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 19
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 446c08
*****entry fsm_callmethod --- type19 state 6
*****entry cb_snd_message to send a msg
| INFO2 | <eXutils.c: 811> DNS resolution with 192.168.4.177:5060
| INFO2 | <eXutils.c: 827> getaddrinfo returned the following addresses:
| INFO2 | <eXutils.c: 832> 192.168.4.177 port 5060
| INFO1 | <eXtl_udp.c: 574> Message sent: (to dest=192.168.4.177:5060)
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP 192.168.4.177:5060;rport=5060;branch=z9hG4bK20585
From: <sip:toto@192.168.4.177>;tag=22326
To: <sip:root@192.168.4.175>;tag=1263961124
Call-ID: 21074
CSeq: 20 INVITE
Contact: <sip:root@192.168.4.175:5060>
User-Agent: Linphone/3.3.2 (eXosip2/3.1.0)
Content-Length: 0

| INFO3 | <jcallback.c: 2145> cb_snd123456xx (id=2)
******entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1544> execute ist transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !

上层调用 vtable 中注册的回调函数 inv_recv == linphonec_call_received
Receiving a call from <sip:toto@192.168.4.177>
并将该地址设置到 caller 上

之前有另一个事件应该是被加到上层的处理队列上了，也就是当前事务上有事件需要处理。在 other_request_reply 中
ortp-error-linphone process event get a message 29
ortp-error-Contact address updated to <sip:root@192.168.4.175> for this dialog
ortp-error-ping reply !
之后退出 sal_iterator 迭代处理。该迭代处理主要处理 exosip 事务相关的事件
退出后处理 call 会话相关处理，主要与用户交互相关。

ortp-error-incoming call ringing for 0 seconds 通知用户有电话
间隙处理 plugin 任务，并查看 configure 文件是否修改了，如果有修改就同步

上层任务不断循环调用 linphone_core_iterate 处理核心事件，由于没有新的消息到来，剩余的处理主要就是根据当前会话的状态跟用户的交互
ortp-error-incoming call ringing for 2 seconds
ortp-error-incoming call ringing for 3 seconds
aortp-error-incoming call ringing for 3 seconds
nortp-error-incoming call ringing for 3 seconds
sortp-error-incoming call ringing for 3 seconds
wortp-error-incoming call ringing for 4 seconds
eortp-error-incoming call ringing for 4 seconds
rortp-error-incoming call ringing for 4 seconds
******entry linphonec_main_loop　当用于输入 answer 后，解析处理用户命令：
Parse the command.　调用 lpc_cmd_answer 出来 answer 命令

调用 get_fixed_contact 确认 contact 联系人
ortp-error-Contact has been fixed using OPTIONS to <sip:root@192.168.4.175>
调用 sal_call_accept 发送 200 ok 响应包给对端
这步上层任务主要根据当前事务链表构造出响应消息，并产生一个新的事件放到当前事务的事件队列上

Exosip 任务发现有新的事件，执行事件，完成实际的数据包发送

```
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO2 | <osip.c: 1544> execute ist transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 2
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 6
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 20
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 446b18
******entry fsm_callmethod --- type20 state 6
******entry cb_snd_message to send a msg
| INFO2 | <eXutils.c: 811> DNS resolution with 192.168.4.177:5060
| INFO2 | <eXutils.c: 827> getaddrinfo returned the following addresses:
| INFO2 | <eXutils.c: 832> 192.168.4.177 port 5060
| INFO1 | <eXtl_udp.c: 574> Message sent: (to dest=192.168.4.177:5060)
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.4.177:5060;rport=5060;branch=z9hG4bK20585
From: <sip:toto@192.168.4.177>;tag=22326
To: <sip:root@192.168.4.175>;tag=1263961124
Call-ID: 21074
CSeq: 20 INVITE
Contact: <sip:root@192.168.4.175>
Content-Type: application/sdp
User-Agent: Linphone/3.3.2 (eXosip2/3.1.0)
Content-Length:      381

v=0
o=root 123456 654321 IN IP4 192.168.4.175
s=A conversation
c=IN IP4 192.168.4.175
t=0 0
m=audio 7078 RTP/AVP 112 111 110 0 8 101
a=rtpmap:112 speex/32000/1
a=fmtp:112 vbr=on
a=rtpmap:111 speex/16000/1
a=fmtp:111 vbr=on
a=rtpmap:110 speex/8000/1
a=fmtp:110 vbr=on
a=rtpmap:0 PCMU/8000/1
a=rtpmap:8 PCMA/8000/1
a=rtpmap:101 telephone-event/8000/1
a=fmtp:101 0-11

| INFO3 | <jcallback.c: 2145> cb_snd123456xx (id=2)
| INFO1 | <jcallback.c: 565> cb_nict_kill_transaction (id=2)
```

*****entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1544> execute ist transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !

上层任务打印一个 connectd
Connected.
构建 filter 并连接用来处理音视频流
linphonec> ortp-error-Payload's bitrate is -1
ortp-error-ms_filter_link: MSOssRead:0x446a60,0-->MSSpeexEC:0x43b1b0,1
ortp-error-ms_filter_link: MSSpeexEC:0x43b1b0,1-->MSVolume:0x440bd0,0
ortp-error-ms_filter_link: MSVolume:0x440bd0,0-->MSSpeexEnc:0x446ee0,0
ortp-error-ms_filter_link: MSSpeexEnc:0x446ee0,0-->MSRtpSend:0x447598,0
ortp-error-ms_filter_link: MSRtpRecv:0x4448f0,0-->MSSpeexDec:0x43b150,0
ortp-error-ms_filter_link: MSSpeexDec:0x43b150,0-->MSDtmfGen:0x444930,0
ortp-error-ms_filter_link: MSDtmfGen:0x444930,0-->MSEqualizer:0x441860,0
ortp-error-ms_filter_link: MSEqualizer:0x441860,0-->MSVolume:0x440c30,0
ortp-error-ms_filter_link: MSVolume:0x440c30,0-->MSSpeexEC:0x43b1b0,0
ortp-error-ms_filter_link: MSSpeexEC:0x43b1b0,0-->MSOssWrite:0x446aa0,0
ortp-error-MS ticker priority set to max
ortp-error-/dev/dsp opened: rate=32000,bits=16,stereo=0 blocksize=2048.
ortp-error-Initializing speex echo canceler with framesize=128, filterlength=8000, delay_samples=0

Exosip 任务接收到对端的 acknowledge
| INFO1 | <eXtl_udp.c: 285> Received message:
ACK sip:root@192.168.4.175 SIP/2.0
Via: SIP/2.0/UDP 192.168.4.177:5060;rport;branch=z9hG4bK29010
From: <sip:toto@192.168.4.177>;tag=22326
To: <sip:root@192.168.4.175>;tag=1263961124
Call-ID: 21074
CSeq: 20 ACK
Contact: <sip:toto@192.168.4.177>
Max-Forwards: 70
User-Agent: Linphone/3.3.2 (eXosip2/3.3.0)
Content-Length: 0

| INFO1 | <eXtl_udp.c: 332> Message received from: 192.168.4.177:5060
| INFO1 | <eXtl_udp.c: 339> Message received from: 192.168.4.177:5060
| INFO3 | <osip_event.c: 89> MESSAGE REC. CALLID:21074

| INFO1 | <udp.c: 1554> Message received from: 192.168.4.177:5060
| INFO1 | <udp.c: 1564> This is a request
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !

上层任务也处理完 answer 命令的处理
ortp-error-Using bitrate 29600 for speex encoder.
ortp-error-Filter MSRtpRecv is already being scheduled; nothing to do.
ortp-error-call answered.
Finished the command.

底层任务空转运行
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !
*****entry _eXosip_thread ...
*****entry eXosip_execute -- timeout is 0--546140
| INFO2 | <eXconf.c: 762> eXosip: timer sec:0 usec:100000!
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !
*****entry _eXosip_thread ...
*****entry eXosip_execute -- timeout is 0--446170
| INFO2 | <eXconf.c: 762> eXosip: timer sec:0 usec:100000!
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !
*****entry _eXosip_thread ...
*****entry eXosip_execute -- timeout is 0--346166
| INFO2 | <eXconf.c: 762> eXosip: timer sec:0 usec:100000!

上层任务处理之前 ack 的事件
linphonec> ortp-error-linphone process event get a message 15
ortp-error-CALL_ACK
ortp-error-bandwidth usage: audio=[d=661.2,u=0.0] video=[d=0.0,u=0.0] kbit/sec
ortp-error-bandwidth usage: audio=[d=40.9,u=0.0] video=[d=0.0,u=0.0] kbit/sec
ortp-error-bandwidth usage: audio=[d=40.9,u=0.0] video=[d=0.0,u=0.0] kbit/sec
。。。
此时，call 的 state 已经被设置为 avrunning 了，每次 core 迭代的时候会显示当前的带宽使用情况

另外，在构建 filter graphics 成功后，我们创建了一个新的任务 msticker 来驱动 graph 运行。在 run_graph 过程中会调用 filter 的 process 回调处理 filter
一个 ticker 一般对应一个音频或者视频流，针对这个流我们一般又有输入流和输出流处理，输出流主要来源于物理声卡或者视频卡，输入流主要来源于 RTP 会话。
这里是音频流打印示例，表示从对端收到了音频数据流。
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...
receiver_process have recvd a blk data and will put it to output streamer ...

在 exosip 的超时事件中我们删除了 nict 事务链
*****entry _eXosip_thread ...
*****entry eXosip_execute -- timeout is 0--246101
| INFO2 | <eXconf.c: 762> eXosip: timer sec:0 usec:100000!
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 3
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 13
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 5
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 0
*****entry fsm_callmethod --- type5 state 13
| INFO1 | <jcallback.c: 565> cb_nict_kill_transaction (id=3)
*****entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1611> execute nict transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !
| INFO3 | <udp.c: 2268> Release a terminated transaction
| INFO4 | <osip_transaction.c: 269> transaction already removed from list 3!
| INFO2 | <osip_transaction.c: 292> free transaction ressource 3 771801749
| INFO2 | <nict.c: 131> free nict ressource

当对端终止当前通话时，发送 Bye 消息过来
| INFO1 | <eXtl_udp.c: 285> Received message:

BYE sip:root@192.168.4.175 SIP/2.0
Via: SIP/2.0/UDP 192.168.4.177:5060;rport;branch=z9hG4bK16575
From: <sip:toto@192.168.4.177>;tag=22326
To: <sip:root@192.168.4.175>;tag=1263961124
Call-ID: 21074
CSeq: 21 BYE
Contact: <sip:toto@192.168.4.177:5060>
Max-Forwards: 70
User-Agent: Linphone/3.3.2 (eXosip2/3.3.0)
Content-Length: 0

| INFO1 | <eXtl_udp.c: 332> Message received from: 192.168.4.177:5060
| INFO1 | <eXtl_udp.c: 339> Message received from: 192.168.4.177:5060
| INFO3 | <osip_event.c: 89> MESSAGE REC. CALLID:21074
| INFO1 | <udp.c: 1554> Message received from: 192.168.4.177:5060
| INFO1 | <udp.c: 1564> This is a request
| INFO2 | <osip_transaction.c: 138> allocating transaction ressource 4 21074
| INFO2 | <nist.c: 32> allocating NIST context
| INFO2 | <osip.c: 1677> execute nist transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 4
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 15
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 12
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 444730
*****entry fsm_callmethod --- type12 state 15
| INFO3 | <jcallback.c: 772> cb_rcvunkrequest (id=4)
| INFO3 | <jcallback.c: 810> cb_rcv? (id=4)
*****entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1677> execute nist transaction events !
| INFO4 | <osip_transaction.c: 363> sipevent tr->transactionid: 4
| INFO4 | <osip_transaction.c: 366> sipevent tr->state: 16
| INFO4 | <osip_transaction.c: 369> sipevent evt->type: 20
| INFO4 | <osip_transaction.c: 372> sipevent evt->sip: 44b488
*****entry fsm_callmethod --- type20 state 16
*****entry cb_snd_message to send a msg　　回送 200ok 给对端
| INFO2 | <eXutils.c: 811> DNS resolution with 192.168.4.177:5060
| INFO2 | <eXutils.c: 827> getaddrinfo returned the following addresses:
| INFO2 | <eXutils.c: 832> 192.168.4.177 port 5060
| INFO1 | <eXtl_udp.c: 574> Message sent: (to dest=192.168.4.177:5060)
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.4.177:5060;rport=5060;branch=z9hG4bK16575
From: <sip:toto@192.168.4.177>;tag=22326
To: <sip:root@192.168.4.175>;tag=1263961124

Call-ID: 21074
CSeq: 21 BYE
User-Agent: Linphone/3.3.2 (eXosip2/3.1.0)
Content-Length: 0

| INFO3 | <jcallback.c: 2145> cb_snd123456xx (id=4)
*****entry osip_transaction_execute --
| INFO4 | <osip_transaction.c: 412> sipevent evt: method called!
| INFO2 | <osip.c: 1677> execute nist transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !

上层处理 close 事务。关闭 call，unlink filter，释放资源，退出 audio ticker 任务
ortp-error-linphone process event get a message 18
ortp-error-EXOSIP_CALL_MESSAGE_NEW
ortp-error-linphone process event get a message 25
ortp-error-CALL_CLOSED or CANCELLED

ortp-error-Current call terminated...
ortp-error-Filter MSRtpRecv is not scheduled; nothing to do.
ortp-error-ms_filter_unlink: MSOssRead:0x446a60,0-->MSSpeexEC:0x43b1b0,1
ortp-error-ms_filter_unlink: MSSpeexEC:0x43b1b0,1-->MSVolume:0x440bd0,0
ortp-error-ms_filter_unlink: MSVolume:0x440bd0,0-->MSSpeexEnc:0x446ee0,0
ortp-error-ms_filter_unlink: MSSpeexEnc:0x446ee0,0-->MSRtpSend:0x447598,0
ortp-error-ms_filter_unlink: MSRtpRecv:0x4448f0,0-->MSSpeexDec:0x43b150,0
ortp-error-ms_filter_unlink: MSSpeexDec:0x43b150,0-->MSDtmfGen:0x444930,0
ortp-error-ms_filter_unlink: MSDtmfGen:0x444930,0-->MSEqualizer:0x441860,0
ortp-error-ms_filter_unlink: MSEqualizer:0x441860,0-->MSVolume:0x440c30,0
ortp-error-ms_filter_unlink: MSVolume:0x440c30,0-->MSSpeexEC:0x43b1b0,0
ortp-error-ms_filter_unlink: MSSpeexEC:0x43b1b0,0-->MSOssWrite:0x446aa0,0
ortp-error-Audio MSTicker thread exiting
Call terminated.
linphonec> Receiving a bye from <sip:toto@192.168.4.177>
ortp-error-Notifying all friends that we are in status 1

执行 quite 命令，上层任务退出循环状态，执行 terminate 命令，释放 osip 资源，退出 exosip 任务，释放上层资源，退出。
linphonec> quit
*****entry linphonec_main_loop
Parse the command.

Finished the command.
Terminating...
No active call.
| INFO2 | <osip.c: 1677> execute nist transaction events !
| INFO2 | <osip.c: 1677> execute nist transaction events !
*****out _eXosip_thread --j_stop_ua=1
| INFO1 | <eXconf.c: 224> Release a terminated transaction
| INFO4 | <osip_transaction.c: 269> transaction already removed from list 2!
| INFO2 | <osip_transaction.c: 292> free transaction ressource 2 21074
| INFO2 | <ist.c: 83> free ist ressource
| INFO2 | <osip_transaction.c: 292> free transaction ressource 4 21074
| INFO2 | <nist.c: 76> free nist ressource
| INFO2 | <osip_transaction.c: 292> free transaction ressource 1 6778
| INFO2 | <nist.c: 76> free nist ressource

八 linphone 使用参考

1 启动 linphone

Linphone 编译可能会比较繁琐一点，作为一款视频通话软件，依赖的其他库比较多，需要将这些依赖库都先编译出来。

Linphone 运行命令支持如下选项：

待补充

2 使用 linphone 命令

Linphone 使用 readline 库完成界面数据的接收与响应。启动 linphone 后 readline 就等待用户的输入，同时 readline 每隔一秒会去处理 linphone core 中的所有待处理事件。当输入 linphone 命令后 readline 会读取到该事件，并交由 linphone 核心去处理。

Linphone 支持的命令有：

待补充

除了基于 readline 库，linphone 还支持通过管道的方式来向核心发送处理命令。这种方式下，会将 linphone 以 daemon 的方式运行起来，用户操作时向其发送命令（实际上是发送到管道中），linphone 从管道中读取命令执行。基本的使用方式如下：（基于这种方式可以更好的构建用户层的应用程序，比如用户界面的实现）

应用设置相关接口时使用 system 系统调用方式来配置 linphone
系统调用方式如下
system("linphonecsh init");
其他命令调用类似，传递给 system 系统调用的参数为一个命令字符串

接口说明如下：
1 创建 linphone 后台进程
linphonecsh init
调用后续接口时必须先执行这一步，可在应用启动时调用。

2 销毁创建的 linphone 后台进程

linphonecsh exit

退出应用时调用该接口，销毁启动时创建的 linphone 后台进程。

3 拨号

linphonecsh dial <sip uri or number>

比如

linphone dial sip:102@192.168.4.203

4 挂断当前的 call

linphonecsh generic terminate

5 注册到服务器

有如下两种方式，目前先用第二种方式来测试，这样与界面元素对应

linphonecsh register --hostname 192.168.4.203 --usrname 103 --password 1234

linphonecsh generic 'register <identify> <proxy> <password>'

注销注册

linphonecsh generic 'unregister'

调用该接口时注销默认的注册信息

6 获取注册状态

linphonecsh status register

用于获取是否注册成功的信息，对于反馈的信息如何传给应用还需考虑，目前可先放空

7 接通电话

linphonecsh generic answer

有电话接入时如何通知应用还需考虑实现，目前先放空。后续所有获取信息的接口都可以先放空，在命令行下，这些信息是输出到标准输出，对于我们盒子一般而言是串口，所以如何将这些导到应用中可能还需要对程序进行一些改动。

8 好友管理

查看好友列表

linphonecsh generic 'friend list'

call 一个好友

linphonecsh generic 'friend call <index>'

添加好友

linphonecsh generic 'friend add <name> <addr>'

删除好友

linphonecsh generic 'friend delete <index>'

9 查看历史记录
读取本地保存的文件来实现
linphonecsh generic 'call-logs'

10 配置 nat
linphonecsh generic 'nat <addr>'

11 配置 stun 服务器
linphonecsh generic 'stun <addr>'

12 配置防火墙策略
选择直连方式时
linphonecsh generic 'firewall none'

选择 nat 防火墙策略时
linphonecsh generic 'firewall nat'

选择 stun 服务器策略时
linphonecsh generic 'firewall stun'

查看当前的防火墙策略
linphonecsh generic 'firewall'

13 配置 sip 端口
linphonecsh generic 'ports sip <number>'
其他端口的配置?

14 配置回音消除
设置回音在延迟多长时间后开启
linphonecsh generic 'ec on <delay> '

关闭回音消除功能
linphonecsh generic 'ec off'

查看当前的配置
linphonecsh generic 'ec show'

15 静音配置
打开静音
linphonecsh generic mute

取消静音

linphonecsh generic unmute

16 静音时关闭 rtp 传输

linphonecsh generic 'nortp-on-audio-mute 1'

其他值为打开状态

3 使用 linphone 配置文件

关于配置文件的相关配置选项说明：

待补充