FFmpeg4.3 系列 28

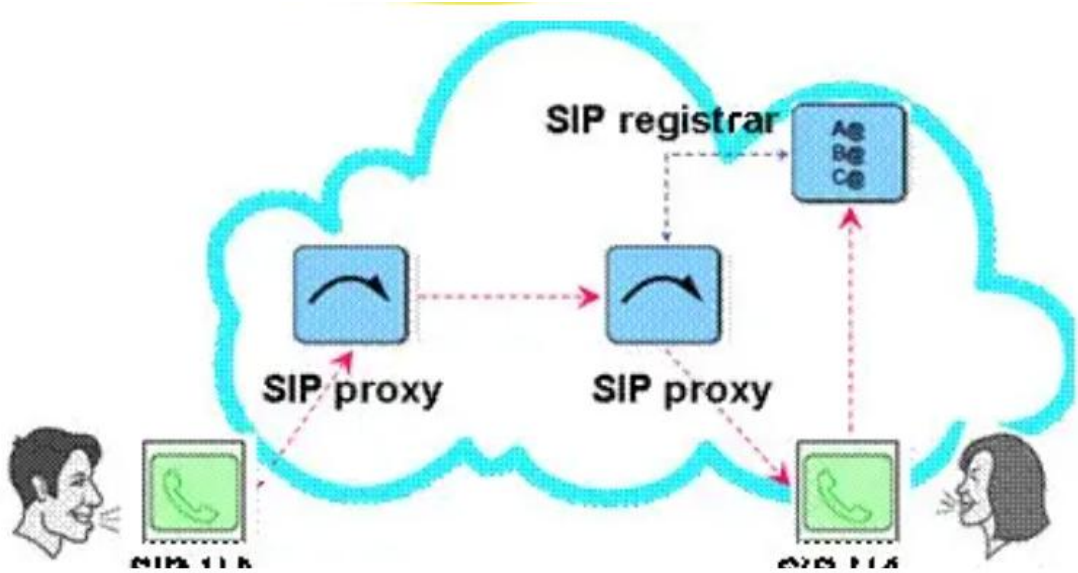
SIP+eXosip+pjsip 入门与实战

sip 协议讲解

RFC 即 Request For Comments（RFC），是一系列以编号排定的文件。文件收集了有关互联网相关信息，以及 UNIX 和互联网社区的软件文件。目前 RFC 文件是由 Internet Society（ISOC）赞助发行。基本的互联网通信协议都有在 RFC 文件内详细说明。RFC 文件还额外加入许多的论题在标准内，例如对于互联网新开发的协议及发展中所有的记录。因此几乎所有的互联网标准都有收录在 RFC 文件之中。

在这里提供给大家一个下载相关 RFC 文档的链接：https://www.rfc-editor.org/， 在这里你可以找到相关的原始文档。

SIP：RFC3261。



1、SIP 协议介绍

Internet 的许多应用都需要建立和管理一个会话，会话在这里的含义是在参与者之间的数据的交换。由于考虑到参与者的实际情况，这些应用的实现往往是很复杂的：参与者可能是在代理间移动，他们可能可以有多个名字，他们中间的通讯可能是基于不同的媒介（比如文本，多媒体，视频，音频等）－有时候是多种媒介一起交互。

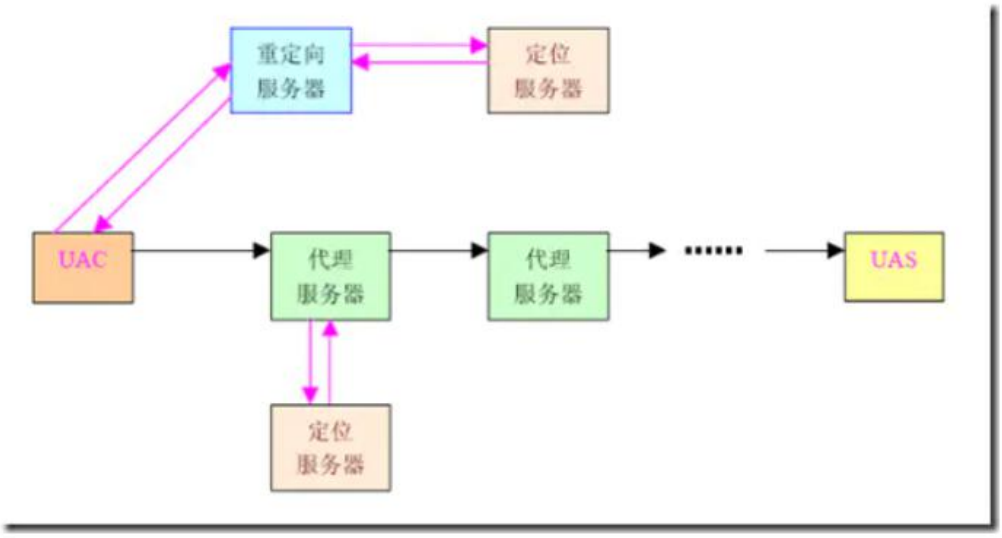
人们创造了无数种通讯协议应用于实时的多媒体会话数据比如声音，影像，或者文本。本 SIP（会话初始协议）和这些协议一样，同样允许使用 Internet 端点（用户代理）来寻找参与者并且允许建立一个可共享的会话描述。为了能够定位精确的会话参与者，并且也为了其他的目的，SIP 允许创建基础的 network hosts（叫做代理服务器），并且允许终端用户注册上去，发出会话邀请，或者发出其他请求。SIP 是一个轻形的，多用途的工具，可以用来创建，修改和终止会话，它独立运作于通讯协议之下，并且不依赖建立的会话类型。

SIP 协议是用于发起、控制和终结多媒体会话的信令协议。SIP 是 IETF 致力于将电话服务带入 IP 网络众多协议的一个组成部分（它与 SDP、RTP、RTCP、RTSP、RSVP、TRIP 等众多协议构成 SIP 系统协议栈）。其将要变成正在发展的 IP 电话——这个朝气蓬勃的电信工业——的标准之一。正如同电子邮件协议一样，SIP 将会变得越来越普及和大众化

2、SIP 协议功能概况

SIP 是一个应用层的控制协议，可以用来建立、修改、和终止多媒体会话（或者会议）例如 Internet 电话/视频会议系统。SIP 也可以邀请参与者参加已经存在的会话，比如多方会议。媒体可以在一个已经存在的会话中方便的增加（或者删除）。SIP 显示的支持**名字映射和重定向服务**，这个用于支持个人移动业务－用户可以使用一个唯一的外部标志而不用关系他们的实际网络地点。SIP 在建立和维持终止多媒体会话协议上，支持 5 个方面：

1） **用户定位**： 检查终端用户的位置，用于通讯。
2） **用户有效性**：检查用户参与会话的意愿程度。
3） **用户能力**：检查媒体和媒体的参数。
4） **建立会话**：“振铃”在被叫和主叫方建立会话参数。
5） **会话管理**：包括发送和终止会话，修改会话参数，激活服务等等。



SIP 不是一个垂直集成的通讯系统。SIP 可能叫做是一个部件更合适，它可以用作其他 IETF 协议的一个部分，用来构造完整的多媒体架构，必须搭配其他协议 rtp/sdp。

比如，这些架构将会包含实时数据传输协议（RTP）（RFC 1889）用来传输实时的数据并且提供 QoS 反馈，实时流协议（RSTP）(RFC 2326)用于控制流媒体的的传输，媒体网关控制协议（MEGACO）(RFC 3015)用来控制到公共电话交换网（PSTN）的网关，还有会话描述协议（SDP）（RFC 2327）用于描述多媒体会话。

因此，**SIP 应该和其他的协议一起工作，才能提供完整的对终端用户的服务**。虽然基本的 SIP 协议的功能组件并不依赖于这些协议。

SIP 本身并不提供服务。但是，SIP 提供了一个基础，可以用来实现不同的服务。比如，SIP 可以定位用户和传输一个封装好的对象到对方的当前位置。并且如果我们利用这点来通过 SDP 传输会话的描述，立刻，对方的用户代理可以得到这个会话的参数。如果我们用这个像传输会话描述（SESSION DESCRIPTION SD）一样呼叫方的照片，一个”呼叫 ID”服务很

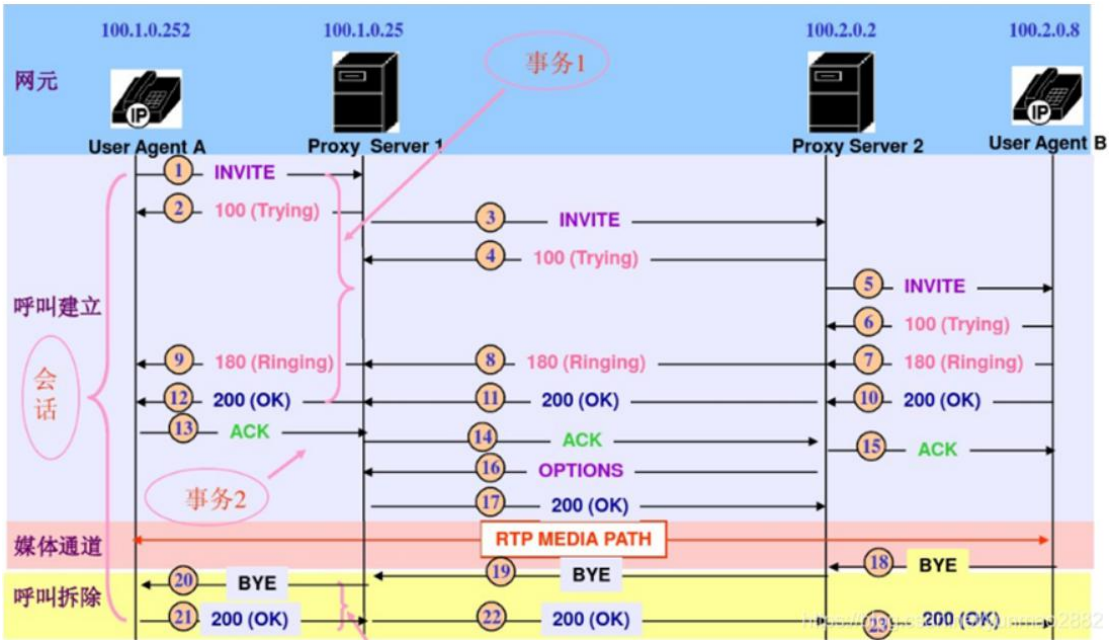
容易就建立了。这个简单的例子说明了，SIP 作为一个基础，可以在其上提供很多不同的服务。

SIP 并不提供会议控制服务（比如议席控制或者投票系统），并且并没有建议会议应该则那样管理。可以通过在 SIP 上建立其他的会议控制协议来发起一个会议。由于 SIP 可以管理参与会议的各方的会话，所以会议可以跨异构的网络，SIP 并不能，也不打算提供任何形式的网络资源预留管理。

安全对于提供的服务来说特别重要。要达到理想的安全程度，SIP 提供了一套安全服务，包括防止拒绝服务，认证服务（用户到用户，代理到用户），完整性保证，加密和隐私服务。

SIP 可以基于 IPV4 也可以基于 IPV6。

3、基于 SIP 协议会话建立的基本过程



上图为基于 SIP 的会话发起的基本过程。用 SIP 建立通讯通常需要六个步骤：

1）注册，发起和定位用户

2）进行媒体协商 - 通常采用 SDP 协议来携带媒体参数

3）由被叫方来决定是否接纳该呼叫

4）呼叫媒体流建立并交互{rtp 协议：参考系列 23 或 24}

5）呼叫更改或处理如呼叫转移等

6）呼叫终止

学习网站：http://www.hellotongtong.com/

福优学苑：【 QQ 咨询：3212001984】

加微信可以提供远程服务,微信服务二维码(13661137824)：

4、打电话过程分析 SIP

传统电话是电磁波的通信，当电话技术发展到 IP 技术时代，SIP 协议成为了电话通信标准协议，不仅可以通电话、还可以收发信息、视频、开会、放 PPT。

事实上，今天的通信业已全面采用 SIP 协议作为通信标准，无论是固定电话、还是移动电话，其后台都是以 SIP 协议完成通话、交换的。

一、从打电话的过程，理解 SIP 协议

打电话的过程分析

两个电话之间的一次通话称为一个**会话(Session)**,

首先，通话双方必须有一个电话号码， 通话步骤如下:

1, 电话 A 拨打电话 B 的号码, 邀请 B 通话 (Invite)
2, 电话 B 振铃(Ring)， 同时电话 A 可以听到电话 B 在振铃
3, 电话 B 提机表示确认应答， 双方通话开始
4, 双方通话
5, 通话过程中，如有任何一方挂电话，则通话结束。

上述会话过程 图示如下：

图中箭头表示传递信号的方向。 图中假设 B 先挂机
在传统电话网中，上述每个信号都是一个电磁波信号.

SIP 协议

SIP 协议，英文为 **Session Initiation Protocol**， 中文翻译为**会话发起协议**。顾名思义，就是在网络上发起会话。

协议(Protocol)是计算机与计算机之间的语言。SIP 协议的目的就是在 IP 网络中实现电话功能。在 IP 网络中，通话两端的不是电话机，而是运行在计算机上的软件电话（软电话）。

同传统电话，用 SIP 协议打一个电话，过程是一样的。两个软电话之间，也有电话号码，也需传递信号。这时的电话号码是 **SIP 帐号**。这时的信号不是一个电磁波信号，而是一个 IP 数据包（称为 SIP 消息）。

首先，通话双方都要有一个 **SIP 帐号**（也称为 URI， 是网络上的电话号码），不同于全数字的传统电话号码， SIP 帐号采用 URI 表示方法， 例如：

sip:peter@company.com:5060

其中：

(1) sip: 表示采用 sip 协议

(2) peter 是用户名，也称为帐号. 用字母和数字均可。

(3) company.com 是帐号所属的服务器域名(也可以用 IP 地址， 例如: sip:peter@192.168.1.100)

(4) 最后的 5060 是端口号。 **SIP 协议默认端口为 5060， 默认采用 UDP 传输**

:5060 的意思是，客户端在名为 company.com 的服务器的 5060 端口号上等待对方连接

如果端口号是 5060，也可以省略不写。

rtsp:554,rtmp:1935, ftp:21

则，上述 SIP 帐号写为: sip: peter@company.com

除了 sip:这几个字母， SIP 帐号就像一个邮件帐号

没错，SIP 协议设计者的意图就是让 SIP 帐号与邮件帐号一致，方便与邮箱服务整合。

对用户来说方便，你要打电话给我，我的电话号就是邮箱号。

SIP 消息

上面讲过，一个通话过程，两端要传递多种信号。在 SIP 协议中，这些信号是一种约定格式的 IP 数据包，称为 SIP 消息。

SIP 消息有好几种，让我们看一个通话过程理解一下

1，软电话 A 向 B 发送一个 SIP 消息 INVITE， 邀请 B 通话
2，软电话 B 振铃，向 A 回复一个 SIP 消息 RING， 通知 A 正在振铃中，请 A 等待
3，软电话 B 提机，向 A 发一个 SIP 消息 OK， 通知 A 可以通话了
4，软电话 A 向 B 回复一个回应消息 ACK，正式启动通话
5，接下来，双方通话
6，软电话 B 挂机，向 A 发一个 SIP 消息 BYE， 通知 A 通话结束
7，软电话 A 向 B 回复一个消息 OK，通话结束

可以看到，这个过程与人打电话的过程是一模一样的。只不过是采用 **IP 数据包(SIP 消息)**的形式传递信号而已。

通话过程中有多种 SIP 消息，每一种消息都是一个 IP 数据包。

这就是 SIP 协议，它约定了会话的发起过程 、结束过程。

5、SIP 流程简介及几个小问题

软电话的下方文本框中，记录了整个通话过程中发生的 SIP 消息（如上图）。软电话(主叫) 的记录内容取， 如下：

==========

2020-02-10 23:53:09,692 SENT to 192.168.31.131/50027
INVITE sip:some@192.168.31.131:50027 SIP/2.0 ... ...
==========
2035-02-10 23:53:09,719 RECEIVED from 192.168.31.131/50027
SIP/2.0 180 **Ringing** ... ...
==========
2035-02-11 00:00:14,040 RECEIVED from 192.168.31.131/50027
SIP/2.0 200 **OK** ... ...
==========
2035-02-11 00:00:14,226 SENT to 192.168.31.131/50027
ACK sip:192.168.31.131:50027;transport=**UDP** SIP/2.0 ... ...
==========
2035-02-11 00:04:34,727 RECEIVED from 192.168.31.131/50027
BYE sip:null@192.168.31.131:51971;transport=UDP SIP/2.0 ... ...
==========
2035-02-11 00:04:35,233 SENT to 192.168.31.131/50027
SIP/2.0 200 **OK** ... ...

每一段是一个 SIP 消息，第一行显示消息发生的时间，同时注明为 SENT 的是向外发送的消息，RECEIVED 为收到的消息。 后面的行显示 SIP 消息的简要内容

- **主叫方的几个 SIP 消息依次是：**

1，发送 INVITE
2，收到 Ringing
3，收到 OK
4， 发送 ACK
5， 收到 BYE (因为对方先挂机）
6，发送 OK

- **被叫方的几个 SIP 消息依次是：**

1，收到 INVITE
2，发送 Ringing
3，发送 OK (当按下 Pickup 时)
4，收到 ACK
5，发送 BYE (当按下 Hangup 时)
6，收到 OK

主叫和被叫的 SIP 消息画成相对应的下图：
（软电话 A 为主叫，B 为被叫）

这个实验完整地阐述了 SIP 协议发起会话、结束会话的典型过程。

- **问题探讨：**

1， 为什么 SIP 消息中没有传送语音数据的消息？

答： SIP 协议规定了会话的发起过程，但**没有规定会话的内容及格式**。会话内容可以是文本、语音、视频等。因此，SIP 协议要结合其它协议，如：用 SDP 协议描述要传递的内容格式，用 RTP，RTSP 流媒体协议传输媒体，才能完成整个通信过程。 SIP 协议这样做为了简化协议，留下扩展的灵活性。

对于语音，处理过程大体是这样：首先把语音录下来成为一组数据，把语音数据进行编码，再发送到对方。对方再解码。

2，SIP 消息数据包会不会被偷听？

答：如同 HTTP 协议可以叠加 SSL 保障传输安全。 SIP 协议可以叠加 TSL 安全传输协议。

3，上面过程为什么只是点对点的（P2P)

答：SIP 协议规定的是**点对点的协议(P2P)**。 通话内容的过程可以不需要服务器参与。

实际运用中，大多数情况都有一个代理服务器(Proxy)，每个软电话与服务器进行 SIP 通信即可。这个服务器就是电话交换机，所有的消息和话音都可以由这个服务器进行转发。

6、SIP 协议消息讲解

SIP 协议是一个基于文本的协议，使用 **UTF-8 字符集**（RFC2279[7]）。

一个 SIP 消息既可以是一个从客户端到服务器端的请求，也可以是一个从服务器端到客户端的一个应答。

这两种消息类型都由一个起始行，一个或者多个包头域，一个可选的消息中文组成。

一般消息＝
起始行
*消息包头
CRLF
[消息正文]

起始行＝请求行/状态行

起始行、每一个包头行，空行、都必须由**回车换行**组成（**CRLF**）。即使没有消息正文，也必须有一个空行跟随。

除了在字符集上的区别以外，很多 SIP 的消息和包头域的格式都同 HTTP/1.1 一样。但 SIP 并非一个 HTTP 的超集或者扩展。

SIP 是一个基于 utf8 文本编码格式的协议(这使其消息具有很好的可读性，并易于调试)。SIP 协议中描述了请求、地址（URL）、应答和个个头部字段的语法信息。整个语法信息以扩展巴克斯范式的形式描述，可以在 Columbia 获得。

这些语法定义参考了 Mail 和 HTTP 的定义方式。SIP 定义了 6 种请求的类型。最基础的方法有：

INVITE　ACK　CANCEL　BYE　INFO　OPTIONS

SIP 消息分类

SIP 协议是以层协议的形式组成的，就是说它的行为是以一套相对独立的处理阶段来描述的，每个阶段之间的关系不是很密切。

SIP 协议将 Server 和 User Agent 之间的通讯的消息分为两类：**请求消息和响应消息**

请求常用方法 [请求消息]

SIP 请求方法的列表

请求在两个 SIP 实体之间发起 SIP 事务，以建立，控制和终止会话。
常用的 SIP 请求消息如下：

- INVITE：用于与用户代理之间的媒体交换建立对话。
- ACK：客户端向服务器端证实它已经收到了对 INVITE 请求的最终响应。
- PRACK：表示对 1xx 响应消息的确认请求消息。
- BYE：表示终止一个已经建立的呼叫。
- CANCEL：表示在收到对请求的最终响应之前取消该请求，对于已完成的请求则无影响。
- **REGISTER**：该方法为用户代理实施位置服务，该位置服务向服务器指示其地址信息。
- OPTIONS：表示查询被叫的相关信息和功能。
- NOTIFY：通知用户有关新事件的通知。

请求常用响应代码 [响应消息]

SIP 响应代码列表
SIP 消息类型和消息列表

SIP 协议中的响应消息用于对请求消息进行响应，指示呼叫的成功或失败状态。

- 临时应答(1XX) ：对请求的临时响应表明请求有效且正在处理中。
- 会话成功(2XX) ：200 级响应表明成功完成请求。作为对 INVITE 的响应，它表示呼叫已建立。
- 重定向(3XX) ：该组指示完成请求需要重定向。该请求必须在新的目的地完成。
- 请求失败(4XX) ：请求包含错误的语法或无法在服务器上完成。
- 服务器失败(5XX) ：服务器无法完成一个明显有效的请求。
- 全局性错误(6XX) ：这是全球性的故障，因为请求无法在任何服务器上完成。

状态码

类型 状态码 状态说明

- 100 试呼叫（Trying）
- 180 振铃（Ringing）
- 181 呼叫正在前转（Call is Being Forwarded）
- 200 成功响应（OK）
- 302 临时迁移（Moved Temporarily）
- 400 错误请求（Bad Request）
- 401 未授权（Unauthorized）
- 403 禁止（Forbidden）

- 404 用户不存在（Not Found）
- 408 请求超时（Request Timeout）
- 480 暂时无人接听（Temporarily Unavailable）
- 486 线路忙（Busy Here）
- 504 服务器超时（Server Time-out）
- 600 全忙（Busy Everywhere）

最终响应与临时响应

最终响应（Final response）：用于结束 SIP 事务的响应，与临时响应相对。所有的 2XX，3XX，4XX，5XX 和 6XX 响应都是最终响应。

临时响应（Provisional response）：服务器用来表示工作进展，并不结束 SIP 事务的一种响应。编码为 1XX 的响应是临时响应，其他响应都是最终响应。

SIP 消息结构

SIP 请求由三部分组成：**请求行、请求头和请求体**。

SIP 是一个基于文本的协议，在这一点上与 HTTP 和 SMTP 相似，这里对比一个简单的 SIP 请求和 HTTP 请求：

```
GET /index.html HTTP/1.1
INVITE sip:seven@freeswitch.org.cn SIP/2.0
```

在 HTTP 中， GET 指明一个获取资源（文件）的动作，而 /index.html 则是资源的地址，最后是协议版本号。

而在 SIP 中，INVITE 表示发起一次请求，seven@freeswitch.org.cn 为请求的地址，称为 SIP URI，最后也是版本号。

其中，SIP URI 很类似一个电子邮件，其格式为“协议:名称@主机”。

与 HTTP 和 HTTPS 相对应，有 SIP 和 SIPS，后者是加密的；名称可以是一串数字的电话号码，也可以是字母表示的名称；而主机可以是一个域名，也可以是一个 IP 地址。

SIP 是一个对等的协议，类似 P2P。不像传统电话那样必须有一个中心的交换机，它可以在不需要服务器的情况下进行通信，只要通信双方都彼此知道对方地址（或者，只有一方知道另一方地 址），在不知道的情况下就需要服务器来中转。

INVITE 消息示例

- **1、Request-Line：请求行**

请求行包括三个部分：方法名、请求 URL、协议版本

请求的方法，定义了请求的性质（INVITE、NOTIFY、BYE 等），以及一个 Request-URI，指出请求应该发送的位置。

典型的 SIP URI 的格式为 sip:username@domainname 或 sip：username @ hostport。如下图示例（SIP 请求头示例）：

```
Request-Line: INVITE sip:2008@192.168.131.86:5062 SIP/2.0
    Method: INVITE
  Request-URI: sip:2008@192.168.131.86:5062
      Request-URI User Part: 2008
      Request-URI Host Part: 192.168.131.86
      Request-URI Host Port: 5062
```

- **2、Message Header：请求头**

```
Message Header
  Via: SIP/2.0/UDP 192.168.131.2:5060;rport;branch=z9hG4bKPjb24cc1f1-4a56-400c-a1f4-0eaf750d9d17
  From: "2006" <sip:2006@192.168.131.2>;tag=2be05dcf-d67e-4950-b164-62fb548755f6
  To: <sip:2008@192.168.131.86>
  Contact: "2006" <sip:2006@192.168.131.2:5060>
  Call-ID: 27e43c2f-0181-4940-9304-b6e4657ada41
  CSeq: 18081 INVITE
  Allow: OPTIONS, INFO, SUBSCRIBE, NOTIFY, PUBLISH, INVITE, ACK, BYE, CANCEL, UPDATE, PRACK, REGISTER, MESSAGE, REFER
  Supported: 100rel, timer, replaces, norefersub
  Session-Expires: 1800
  Min-SE: 90
  X-UCM-AudioRecord: *3
  X-UCM-CallPark: #72
  Max-Forwards: 70
  User-Agent: Grandstream UCM6510V1.4A 1.0.17.7
  Content-Type: application/sdp
  Content-Length:   661
```

- **via：SIP 版本号（2.0）、传输类型（UDP）、呼叫地址 、branch。**

branch 为分支，是一随机码，它被看作传输标识，标志会话**事务（详细的事务概念请参考 29 页）**。

<=Via 字段中地址是消息发送方或代理转发方设备地址，一般由主机地址和端口号组成

<=传输类型可以为 UDP、TCP、TLS、SCTP

- **From：表示请求消息的发送方和目标方**

<=如果里面有用户名标签，地址要求用尖括号包起来

<=对于 INVITE 消息，可以在 From 字段中包含 tag，它也是个随机码

- **To：请求消息的目标方**
- **Contact：是 INVITE 消息所必须的，用来告诉对方，回消息给谁。****

<=（注意区别：在 180RINGING 的时候，这里是目标方地址）**

- **Call-ID：用于标识一个特定邀请以及与这个邀请相关的所有后续事务（即标识一个会话）**

- **CSeq：字段是用来给同一个会话中的事务进行排序的，每发送一个新的请求，该值加 1，当排序到 65535（也就是 216 的最大数），会开始新一轮的排序。**
- **Allow：允许的请求消息类型**
- **Supported：**
- **Session-Expires：存活时间，用户的响应必须在这个时间范围内**
- **Min-SE：最小长度。**
- **X-UCM-AudioRecord：自定义字段**
- **X-UCM-CallPark：自定义字段**
- **Max-Forwards：最大转发数量限制了通讯中转发的数量。它是由一个整数组成，每转发一次，整数减一。如果在请求消息到达目的地之前该值变为零，那么请求将被拒绝并返回一个 483（跳数过多）错误响应消息。**
- **User-Agent：指明 UA 的用户类型**
- **Content-Type：消息实体类型**
- **Content-Length：消息实体长度，单位为字节**

本地将生成 From tag 和 Call-ID 全局唯一码，被叫方代理则生成 To tag 全局唯一码。这三个随机码做为整个对话中对话标识（dialog indentifier）在通话双方使用。

3、Message Body：请求体

请求体主要包含的就是 SDP 相关的东西，

```
Message Body
  Session Description Protocol
    Session Description Protocol Version (v): 0
    Owner/Creator, Session Id (o): - 1077372048 1077372048 IN IP4 192.168.131.2
    Session Name (s): Asterisk
    Connection Information (c): IN IP4 192.168.131.2
    Bandwidth Information (b): CT:384
    Time Description, active time (t): 0 0
    Media Description, name and address (m): audio 17786 RTP/AVP 9 0 8 3 2 18 97 101
    Media Attribute (a): rtpmap:9 G722/8000
    Media Attribute (a): rtpmap:0 PCMU/8000
    Media Attribute (a): rtpmap:8 PCMA/8000
    Media Attribute (a): rtpmap:3 GSM/8000
    Media Attribute (a): rtpmap:2 G726-32/8000
    Media Attribute (a): rtpmap:18 G729/8000
    Media Attribute (a): rtpmap:97 iLBC/8000
    Media Attribute (a): rtpmap:101 telephone-event/8000
    Media Attribute (a): fmtp:101 0-16
    Media Attribute (a): maxptime:30
    Media Attribute (a): rtcp:17787 IN IP4 192.168.131.2
    Media Attribute (a): sendrecv
    Media Description, name and address (m): video 13544 RTP/AVP 99
    Media Attribute (a): rtpmap:99 H264/90000
    Media Attribute (a): fmtp:99 packetization-mode=1;level-asymmetry-allowed=1;profile-level-id=42E01F
    Media Attribute (a): rtcp:13545 IN IP4 192.168.131.2
    Media Attribute (a): rtcp-fb:* nack
    Media Attribute (a): rtcp-fb:* nack pli
    Media Attribute (a): rtcp-fb:* ccm fir
    Media Attribute (a): sendrecv
```

SDP（会话描述协议），用于两个会话实体之间的媒体协商，并达成一致，属信令语言族，采用文本（字符）描述形式。

SDP（session description Protocal）会话描述协完全是一种会话描述格式 — 它不属于传输协议 — 它只使用不同的适当的传输协议，包括会话通知协议（SAP）、会话初始协议（SIP）、实时流协议（RTSP）、MIME 扩展协议的电子邮件以及超文本传输协议（HTTP）。SDP 协议

是也是基于文本的协议。SDP 不支持会话内容或媒体编码的协商，所以在流媒体中只用来描述媒体信息。媒体协商这一块要用 RTSP 来实现.

SDP 协议格式：SDP 描述由许多文本行组成，文本行的格式为<类型>=<值>，

<类型>是一个字母
<值>是结构化的文本串，其格式依<类型>而定。
注意：协议的等号前后不可有空格！！！type: 该字节为单字节（如： v，o, m 等）区分大小写。

＜type＞=< value > [CRLF]

其实可以将其简化一下，它就是一个在点对点连接中描述自己的字符串，我们可以将其封装在 JSON 中进行传输，在 PeerConnection 建立后将其通过服务器中转后，将自己的 SDP 描述符和对方的 SDP 描述符交给 PeerConnection 就行了。

- **v = 0 协议版本**

  协议版本，目前定义 0.版本，这是当前使用的唯一 SDP 版本。

o=<username>，<sess-id>，<sess-version>、<nettype>，<addrtype>，<unicast-address>

示例：o = - 6311806030512608073 2 IN IP4 127.0.0.1

<username>是用户在始发主机上的登录名，或者如果始发主机不支持用户标识的概念，则为“ - ” 。<username>不能包含空格。

<sess-id>是一个数字字符串，使得<username>，<sess-id>，<nettype>，<addrtype>和<unicast-address>的元组形成会话的全局唯一标识符。

<sess-version> 是此会话描述的版本号。

<nettype> 网络类型。是一个给出网络类型的文本字符串。最初“IN”被定义为具有“Internet”的含义，但其他值可以在将来注册。

<addrtype> 地址类型 IP4 和 IP6。

<unicast-address>是创建会话的机器的地址。

- **s = -（会话名称）**

“s =”字段是文本会话名称。
每个会话描述必须有且仅有一个“s =”字段。“s =”字段绝不能为空，并且应该包含 ISO 10646 字符（但也请参阅“a = charset”属性）。如果会话没有有意义的名称，则应使用值“s =”

（即一个空格作为会话名称）。

- **i = （会话描述）**

该字段提供有关会话的文本信息。有且最多只能有一个会话级的“i =”每个会话描述字段，并且最多一个“I =”每传媒领域。如果存在“a = charset”属性，则它指定在“i =”字段中使用的字符集。如果“a = charset”属性不存在，则“i =”字段必须包含采用 UTF-8 编码的 ISO 10646 字符。

- **u = （描述的 URI）**

URI 是由 WWW 客户端使用的统一资源标识符。URI 应该是一个指向 会话附加信息的指针。这个字段是可选的，但是如果它存在，它必须 在第一个媒体字段之前被指定。 每个会话描述不允许超过一个 URI 字段。

- **e = （电子邮件地址）和 p = （电话号码）**

可选。负责会议的人员的联系信息，但不一定是创建会议通知的同一个人。
如果有电子邮件地址或电话号码，则必须在第一个媒体字段之前指定。
RFC 2327 要求“e =”或“p =”是必需的。两者都是 现在可选，以反映实际使用情况

*c = （连接信息 - 如果包含，则不需要所有媒体）
c = <nettype> <addrtype> <connection-address>。第一个字段是网络类型；第二个字段是地址类型，它使得 SDP 可用于不基于 IP 的会话；第三个字段是链接地址。

- **b = <bwtype>：<带宽>**

这个可选字段表示会话或媒体使用的建议宽带。
bwtype 可以是 CT 或 AS，CT 方式是设置整个会议的带宽，AS 是设置单个会话的带宽。缺省带宽是千比特每秒。
学习网站：http://www.hellotongtong.com/
福优学苑：【 QQ 咨询：3212001984】
加微信可以提供远程服务,微信服务二维码(13661137824)：

7、SIP 主要概念模型

实体模型概述

SIP 协议模型定义了 User Agent 和 Server 等两类主要实体。

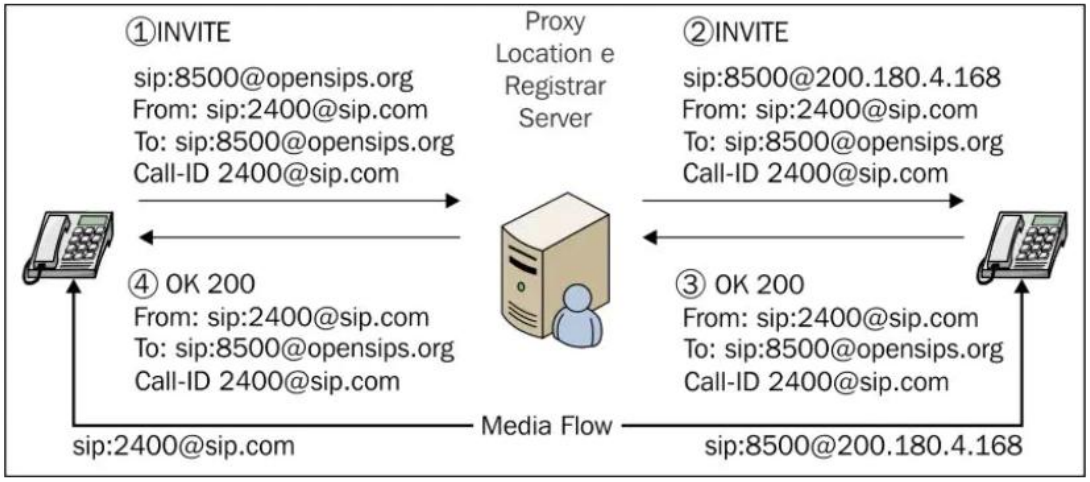
SIP 协议把 User Agent（即 UA）分为两个部分：User Agent Client 和 User Agent Server。

呼叫方（称 User Agent Client）发出邀请 （或呼叫），
被叫方（称 User Agent Server）接受或拒绝邀请 （或呼叫）。

SIP Server

Proxy Server 代理服务器

Proxy Server 作为 UAC 和 UAS 间的中间媒体，它转发 UAC 发来的的邀请，在转发之前，根据被叫标识请求位置服务器获得被叫的可能位置，然后分别向它们发出邀请；



Redirect Server 重定向服务器

Redirect Server 接受 UAC 来的邀请，根据被叫标识请求位置服务器获得被叫的可能位置，把这些信息返回给邀请的发起者（UAC），和 Proxy Server 的不同之处就在于它不转发邀请，邀请由主叫终端自己完成。

设想 bob 和 alice 是经人介绍认识的，而他们还不熟悉，bob 想请 alice 吃饭就需要一个中间人（M）传话，而这个中间人就叫代理服务器（Proxy Server）。还有另一种中间人叫做重定向服务器（Redirect Server），它类似于这样的方式工作——中间人 M 告诉 bob，我也不知道 alice 在哪里，但我老婆知道，要不然我告诉你我老婆的电话，你直接问她吧，我老婆叫 W。这样，M 就成了一个重定向服务器，而他老婆 W 则是真正的代理服务器。这两种服务器都是 UAS，它们主要是提供一对欲通话的 UA 之间的路由选择功能。具有这种功能的设备通常称为边界会话控制器（SBC，Service Border Controller）。

Register Server 注册服务器

主要用于登记分组终端的当前位置和位置服务的原始数据；

试想这样一种情况，alice 还是个学生，没有自己的手机，但它又希望 bob 能随时找到她，于是当她在学校时就告诉中间人 M 说她在学校，如果有事打她可以打宿舍的电话；而当她回家时也通知 M 说有事打家里电话。只要 alice 换一个新的位置，它就要向 M 重新

“注册”新位置的电话，以让 M 能随时找到她，这时候 M 就是一个注册服务器。

Back to Back 背靠背用户代理

RFC 3261 并没有定义 B2BUA 的功能，它只是一对 UAS 和 UAC 的串联。FreeSWITCH 就是一个典型的 B2BUA。

我们来看上述故 事的另一个版本：M 和 W 是一对恩爱夫妻。M 认识 bob 而 W 认识 alice。M 和 W 有意搓合两个年轻人，但见面时由于两人太腼腆而互相没留电话号码。

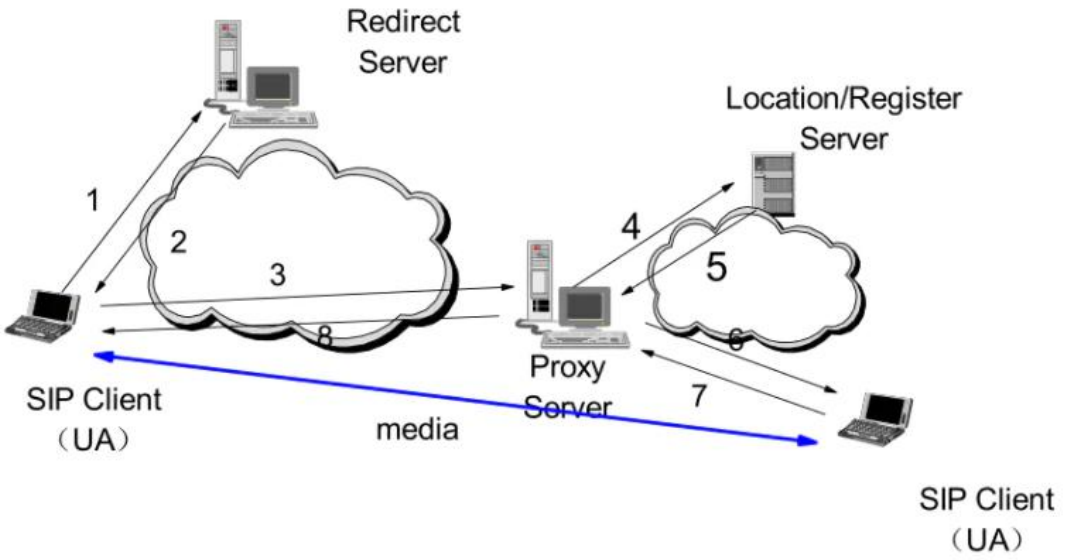
事后 bob 相知道 alice 对他感觉如何，于是打电话问 M，M 不认识 alice，就转身问老婆 W （注意这次 M 没有直接把 W 电话给 bob），W 接着打电话给 alice，alice 说印象还不错，W 就把这句话告诉 M， M 又转过身告诉 bob。

M 和 W 一个面向 bob，一个对着 alice，他们两个合在一起，称作 B2BUA。在这里，bob 是 UAC，因为他发起请求；M 是 UAS，因为他接受 bob 的请求并为他服务；我们把 M 和 W 看做一个整体，他们背靠着背（站着坐着躺着都行），W 是 UAC，因为她又向 alice 发起了请求，最后 alice 是 UAS。

其实这里 UAC 和 UAS 的概念也不是那么重要，重要的是要理解这个背靠背的用户代理。因为事情还没有完，bob 一听说 alice 对他印象还不错，心花怒放，便想请 alice 吃饭，他告诉 M， M 告诉 W， W 又告诉 alice，alice 问去哪吃，W 又只好问 M， M 再问 bob…… 在这对年轻人挂断电话这前， M 和 W 只能“背对背”的工作。

这几种代理服务器的描述文字比较多，但是说的很形象。

SIP 基本网络模型



8、SIP 协议的结构

SIP 是一个分层的协议，意思是说 SIP 协议由一组相当无关的处理层次组成，这些层次之间只有松散的关系。协议分成不同层次来描述是为了能够更清晰的表达，在同一个小节里有功能的公共要素的交叉描述。本协议并没有规定一个具体的实现。当我们说一个要素”包含”某一个层，我们的意思是这个要素复核这个层定义的规则。

不是 SIP 每一个要素都一定包含每一个层。此外，SIP 定义的要素是逻辑上的要素，不是物理要素。一个物理的实现可以实现不同的逻辑要素，或许甚至是基于串行事务处理原理。SIP 最底层的是它的语法和编码层。编码方式是采用扩展的 Backus-Naur Form grammar(BNF 范式)。

第二层是传输层。它定义了一个客户端如何发送请求和接收应答，以及一个服务器如何接收请求和发送应答。所有的 SIP 要素都包含一个通讯层。

第三层是事务层。事务是 SIP 的基本组成部分。一个事务是客户发送的一个请求事务（通过通讯层）发送到一个服务器事务，连同服务器事务的所有的该请求的应答发送回客户端事务。事务层处理应用服务层的重发，匹配请求的应答，以及应用服务层的超时。任何一个用户代理客户端（user agent client UAC）完成的事情都是由一组事务构成的。用户代理包含一

个事务层，来实现有状态的代理服务器。无状态的代理服务器并不包含事务层。事务层包含一个客户元素(可以认为是一个客户事务)和一个服务器元素(可以认为是一个服务器事务)，他们都可以用一个有限状态机来处理特定的请求。

在事务层之上是事务用户（TU）。每一个 SIP 实体，除了无状态代理，都是一个事务用户。当一个 TU 发出一个请求，它首先创建一个客户事务实例（client transaction instance）并且和请求一起发送，这包括了目标 IP 地址、端口号、以及发送请求的设备。TU 可以创建客户事务，也可以取消客户事务。当客户取消一个事务，它请求服务器终止正在处理的事务，并且回滚状态到该事务开始前的状态，并且产生指定的该事务的错误报告。这是由 CANCEL 请求完成的，这个请求有自己的事务，并且包含一个被取消的事务。

SIP 要素，包含，用户代理客户端和服务器，无状态和有状态代理服务器和注册服务器，包含一个可以互相区别的核心（Cores）。Cores，除了无状态代理服务器，都是事务用户。UAC(用户代理客户端)和 UAS(用户代理服务端)的 cores 的行为依赖于实现（method），对所有的实现来说，有几个公共的原则。对 UAC 来说，这些规则约束请求的建立；对 UAS 来说，这些规则约束请求的处理和应答。由于注册服务在 SIP 中是一个重要的角色，所以 UAS 处理 REGISTER 请求有一个特别的名字：登记员（registrar，登记服务器）。

在对话中，有其他的相关会被发送。一个对话是一个持续一定时间的两个用户之间的端到端的 SIP 关系。对话过程要求两个用户代理之间的信息是有序的而且请求被正确路由传输的。在这个规范中，只有 INVITE 请求可以用来建立会话。当一个 UAC 在一个对话中发出请求的时候，它不仅一般 UAC 规则而且也遵循对话中的请求规则。

SIP 中最重要的方法就是 INVITE 方法，它用来在不同的参与者中创建会话使用。一个会话由一组参与者，他们之间用于交流的媒体流组成。

osip 与 eXosip 开源库

osip 简介

oSIP 项目启动于 2000 年 7 月，第一个发布的版本是在 2001 年 5 月(0.5.0)。

oSIP 开发库是第一个自由软件项目。在第三代网络体系中，越来越多的电信运营商将要使用 IP 电话（Linux 也成为支撑平台的幸运儿）。发展的一个侧面是在不久的将来，Linux 将更多支持多媒体工具。oSIP，作为 SIP 开发库，将允许建造互操作的注册服务器、用户代理（软件电话）和代理服务器。所有的这些都平添了 Linux 将作为下一代电话产品的机会。

但 oSIP 的目标并非仅仅在 PC 应用。OSIP 具有足够的灵活和微小，以便在小的操作系统（例如手持设备）满足其特定要求。从 0.7.0 版本发布，线程的支持作为可选项。作为开发者的我们，可以在应用程序设计时进行选择。OSIP 在将来会完美的适用于蜂窝设备和嵌入式系统当中。oSIP 被众所周知应用于实时操作系统 VxWorks 当中，并且其他支持将是简单的事情。

oSIP 现在支持 utf8 和完整的 SIP 语法。它包含一个有很小例外、很好适应性的 **SIP 语法分析器**。OSIP 中包含一个语法分析器，其能够读写任何在 rfc 中描述的 SIP 消息。早期 oSIP 能够分析很小一部分头部，例如 Via、Call-ID、To、From、Contact、Cseq、Route、Record-Route、mime-version、Content-Type 和 Content-length。所有其他头部以字符串的格式存储。

要应用 Osip 到我们的程序中去，首先要看官方文档，文档中对 Osip 协议栈提供的各个功能部件如何使用都有比较详细的描述，但未进行整体性的分析，某些中文的指导文档也都停留在对其简单的翻译，不能为不熟悉该协议栈使用的用户快速参考使用，本文档不按照 Osip 的代码进行按功能分块说明，而是根据实际使用时的代码使用顺序来对主要逻辑流程进行分析，并适当对流程中使用到的功能部件进行说明，具体更详细的功能说明或疑问可直接查看官方文档对应部分的解释或直接查看功能函数源代码即可解决。

eXosip 协议栈的简介

eXosip 是 Osip2 协议栈的封装和调用。它实现了作为单个 sip 终端的大部分功能，如 register、call、subscription 等。

eXosip 是 Osip2 的一个扩展协议集，它部分封装了 Osip2 协议栈，使得它**更容易被使用**。使用 sip 协议建立多媒体会话是一个复杂的过程，exosip 库开发的目的在于**隐藏这种复杂性**。

正如它的名称所表示的，eXosip2 - the eXtended osip Library，它扩展了 osip 库，实现了一个**简单的高层 API**。通过使用 exosip，我们可以避免直接使用 osip 带来的困难。需要注意，**exosip 并不是对 osip 的简单封装包裹，而是扩展**。

Osip 专注于 sip 消息的解析，事务状态机的实现，而 exosip 则基于 osip 实现了 **call、options、register、publish** 等更倾向于**功能性的接口**。当然，这些实现都是依赖于底层 osip 库已有的功能的。

EXosip 使用 UDP socket 套接字实现底层 sip 协议的接收/发送。并且封装了 sip 消息的解释器。

EXosip 使用定时轮循的方式调用 Osip2 的 transaction 处理函数，这部分是协议栈运转的核心。透过添加/读取 transaction 消息管道的方式，驱动 transaction 的状态机，使得来自远

端的 sip 信令能汇报给调用程序，来自调用程序的反馈能通过 sip 信令回传给远端。

EXosip 增加了对各个类型 transaction 的超时处理，确保所有资源都能循环使用，不会被耗用殆尽。

EXosip 使用 jevent 消息管道来向上通知调用程序底层发生的事件，调用程序只要读取该消息管道，就能获得感兴趣的事件，进行相关的处理。

EXosip 里比较重要的应用有 j_calls、j_subscribes、j_notifies、j_reg、j_pub、osip_negotiation 和 authinfos。J_calls 对应呼叫链表，记录所有当前活动的呼叫。J_reg 对应注册链表，记录所有当前活动的注册信息。Osip_negotiation 记录本地的能力集，用于能力交换。Authinfos 记录需要的认证信息。

osip+eXosip 在 Windows 下的源码编译

编译源码

- libosip2-3.6.0**.tar.gz**
- libeXosip2-3.6.0.**tar.gz**

第一步，下载 osip 和 eXosip

osip: http://ftp.twaren.net/Unix/NonGNU//osip/libosip2-3.6.0.tar.gz
eXosip： http://download.savannah.gnu.org/releases/exosip/libeXosip2-3.6.0.tar.gz

第二步，解压，编译 osip：
1.进入 libosip2-3.6.0\platform\vsnet 目录，用 VS2010 直接打开 osip.sln 文件，项目自动转换
2.更改 libosip2-3.6.0\platform\vsnet\osip2.def 文件，在文件末尾追加
osip_transaction_set_naptr_record @138

3.更改 libosip2-3.6.0\platform\vsnet\osipparser2.def 文件，在文件末尾追加
osip_realloc @416
osip_strcasestr @417
__osip_uri_escape_userinfo @418
然后，重新编译 osip2，导出 lib 和 dll，即可

4.先编译 osipparser2，再编译 osip2，最后在 libosip2-3.6.0\platform\vsnet\Debug DLL 下生成

库文件：

osip2.lib

osip2.dll

osipparser2.lib

osipparser2.dll

IPHlpApi.Lib

DnsAPI.Lib

Crypt32.Lib

ws2_32.lib

同时，**需要 openssl 的开发环境(include+lib+dll)**

同时，还**需要 windows 下的 socket 库文件：winsock2.h + ws2_32.lib**

- **大坑 1：**
- **原因：osip2 这个库，没有到处对应的 函数**

1>eXosip.obj : error LNK2019: 无法解析的外部符号 _osip_transaction_set_naptr_record，该符号在函数 __eXosip_transaction_init 中被引用

1>eXtl_tcp.obj : error LNK2019: 无法解析的外部符号 _osip_realloc，该符号在函数 __tcp_tl_recv 中被引用

1>eXtl_tls.obj : error LNK2001: 无法解析的外部符号 _osip_realloc

1>eXutils.obj : error LNK2001: 无法解析的外部符号 _osip_realloc

1>eXtl_tcp.obj : error LNK2019: 无法解析的外部符号 _osip_strcasestr，该符号在函数 _handle_messages 中被引用

1>eXtl_tls.obj : error LNK2001: 无法解析的外部符号 _osip_strcasestr

1>eXtl_tls.obj : error LNK2019: 无法解析的外部符号 __imp__CertCloseStore@8，该符号在函数 __tls_add_certificates 中被引用

1>eXtl_tls.obj : error LNK2019: 无法解析的外部符号 __imp__CertEnumCertificatesInStore@8，该符号在函数 __tls_add_certificates 中被引用

1>eXtl_tls.obj : error LNK2019: 无法解析的外部符号 __imp__CertOpenSystemStoreW@8，该符号在函数 __tls_add_certificates 中被引用

1>eXtl_tls.obj : error LNK2019: 无法解析的外部符号 __imp__CryptAcquireCertificatePrivateKey@24，该符号在函数 __tls_set_certificate 中被引用

1>eXtl_tls.obj : error LNK2019: 无法解析的外部符号 __imp__CertFreeCertificateContext@4，该符号在函数 __tls_set_certificate 中被引用

1>eXutils.obj : error LNK2019: 无法解析的外部符号 _GetIfEntry@4，该符号在函数 _eXosip_dns_get_local_fqdn 中被引用

1>eXutils.obj : error LNK2019: 无法解析的外部符号 _GetIpAddrTable@12，该符号在函数 _eXosip_dns_get_local_fqdn 中被引用

1>eXutils.obj : error LNK2019: 无法解析的外部符号 _DnsFree@8，该符号在函数 __eXosip_dnsutils_srv_lookup 中被引用

1>eXutils.obj : error LNK2019: 无法解析的外部符号 _DnsQuery_A@24，该符号在函数 __eXosip_dnsutils_srv_lookup 中被引用

1>eXutils.obj : error LNK2019: 无法解析的外部符号 _DnsQueryConfig@24，该符号在函数 _eXosip_dnsutils_naptr 中被引用

1>jrequest.obj : error LNK2019: 无法解析的外部符号 ___osip_uri_escape_userinfo，该符号在函数

__eXosip_dialog_add_contact 中被引用
1>jresponse.obj : error LNK2001: 无法解析的外部符号 ___osip_uri_escape_userinfo
1>Debug\eXosip.dll : fatal error LNK1120: 14 个无法解析的外部命令

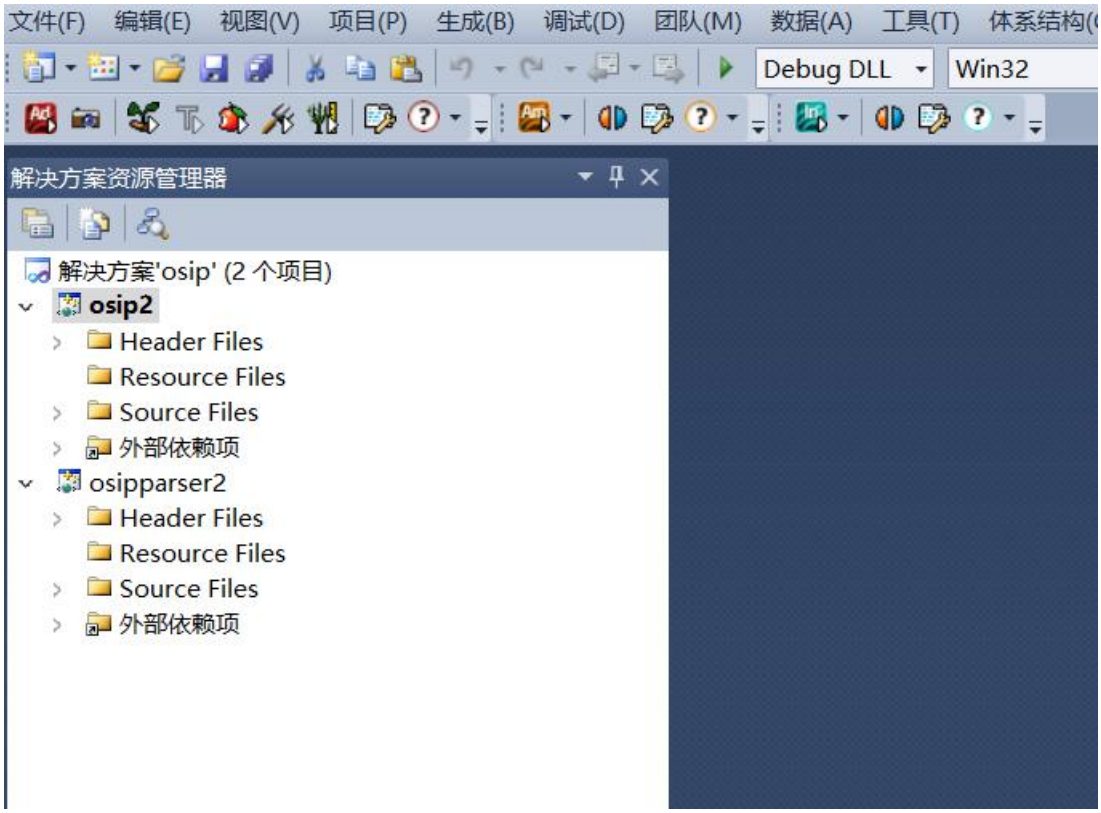
- **大坑 2：**

1>eXtl_tls.obj : error LNK2019: 无法解析的外部符号 __imp__CertCloseStore@8，该符号在函数 __tls_add_certificates 中被引用
1>eXtl_tls.obj : error LNK2019: 无法解析的外部符号 __imp__CertEnumCertificatesInStore@8，该符号在函数 __tls_add_certificates 中被引用
1>eXtl_tls.obj : error LNK2019: 无法解析的外部符号 __imp__CertOpenSystemStoreW@8，该符号在函数 __tls_add_certificates 中被引用
1>eXtl_tls.obj : error LNK2019: 无法解析的外部符号 __imp__CryptAcquireCertificatePrivateKey@24，该符号在函数 __tls_set_certificate 中被引用
1>eXtl_tls.obj : error LNK2019: 无法解析的外部符号 __imp__CertFreeCertificateContext@4，该符号在函数 __tls_set_certificate 中被引用
1>eXutils.obj : error LNK2019: 无法解析的外部符号 _GetIfEntry@4，该符号在函数 _eXosip_dns_get_local_fqdn 中被引用
1>eXutils.obj : error LNK2019: 无法解析的外部符号 _GetIpAddrTable@12，该符号在函数 _eXosip_dns_get_local_fqdn 中被引用
1>eXutils.obj : error LNK2019: 无法解析的外部符号 _DnsFree@8，该符号在函数 __eXosip_dnsutils_srv_lookup 中被引用
1>eXutils.obj : error LNK2019: 无法解析的外部符号 _DnsQuery_A@24，该符号在函数 __eXosip_dnsutils_srv_lookup 中被引用
1>eXutils.obj : error LNK2019: 无法解析的外部符号 _DnsQueryConfig@24，该符号在函数 _eXosip_dnsutils_naptr 中被引用
1>Debug\eXosip.dll : fatal error LNK1120: 10 个无法解析的外部命令

第三步，解压，编译 eXosip

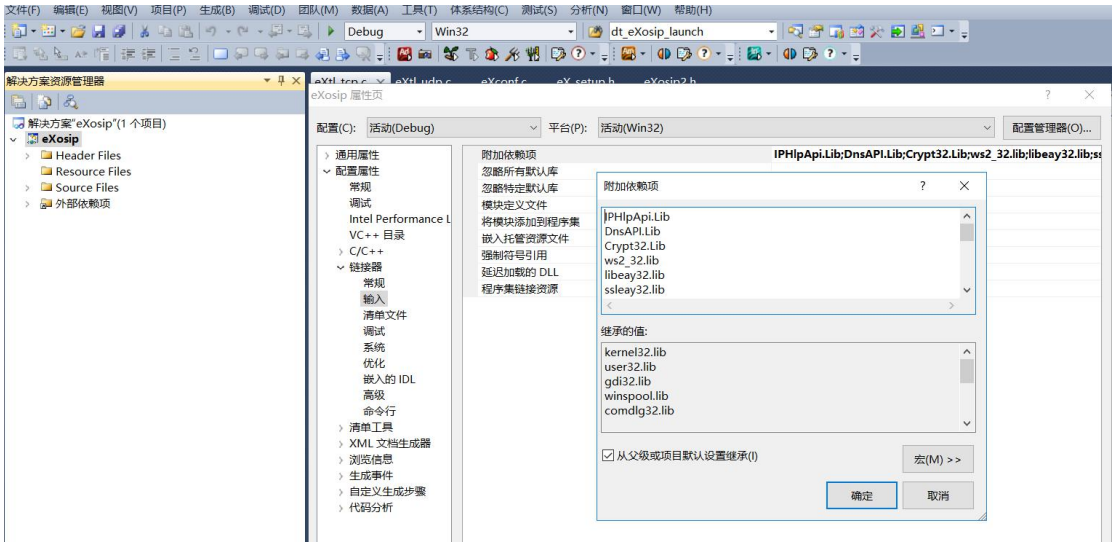
进入 libeXosip2-3.6.0\platform\vsnet 目录，用 VS2010 直接打开 eXosip.sln 文件，项目自动转换：

1.将 osip2.lib,osip2.dll,osipparser2.lib,osipparser2.dll 拷贝到 Debug 目录下
2.C/C++ --> 预处理器 --> 预处理器定义：删除 HAVE_OPENSSL_SSL_H 或者配置 openssl 环境
3.C/C++ --> 常规 --> 附加包含目录: 将 osip 的头文件 libosip2-3.6.0\include 包含进来
4.编译，生成 eXosip.lib



学习网站：http://www.hellotongtong.com/
福优学苑：【 QQ 咨询：3212001984】
加微信可以提供远程服务,微信服务二维码(13661137824)：

IPHlpApi.Lib
DnsAPI.Lib
Crypt32.Lib
ws2_32.lib
libeay32.lib
ssleay32.lib
osipparser2.lib
osip2.lib



开发环境的搭建

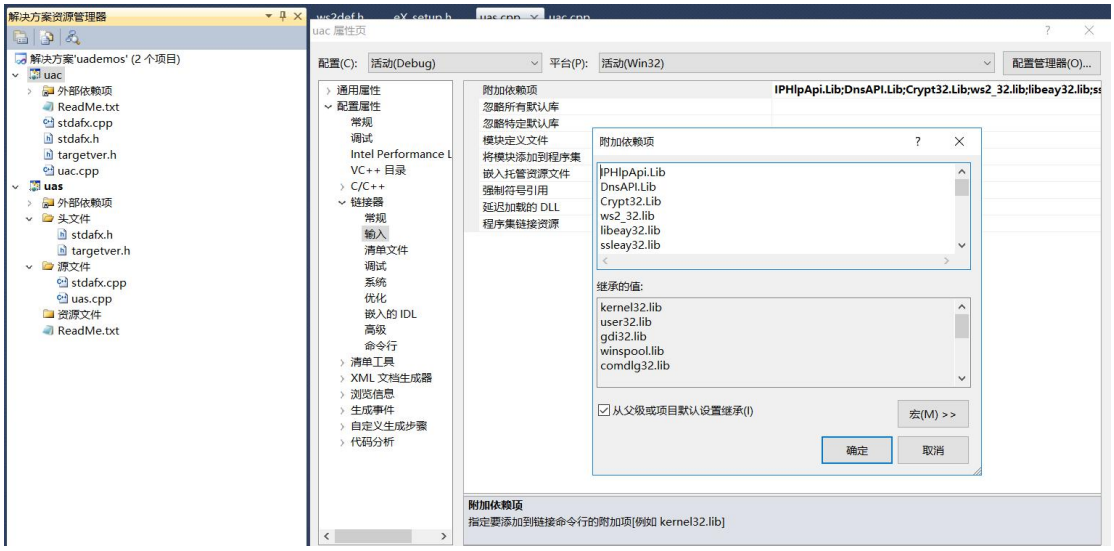
1. 链接器 --> 输入 --> 附加依赖项：增加静态库引用：Dnsapi.lib;Iphlpapi.lib;Ws2_32.lib;osip2.lib;osipparser2.lib;exosip.lib;

2.C/C++ -->常规 -->附加包含目录：将 osip 和 eXosip 的头文件 libosip2-3.6.0\include、libeXosip2-3.6.0\include 包含进来

3. 链接器 --> 常规 --> 附加库目录：将 osip 和 eXosip 的库包含进来，libeXosip2-3.6.0\platform\vsnet\Debug

4. 配置属性 --> 调试 --> 环境：将 osip2.dll 和 osipparser2.dll 包括进来 PARH=libeXosip2-3.6.0\platform\vsnet\Debug

- **头文件：**
- **库文件：**



osip+eXosip 在 Linux 下的编译

程序包准备:ubuntu18

libosip2-3.xx.tar.gz
libeXosip2-3.xx.tar.gz
注意安装： apt-get install libssl1.0-dev
如果上面配置的时候没有 --prefix 这个配置项，则默认安装到**/usr/local/lib**，
export LD_LIBRARY_PATH=/usr/local/lib/:$LD_LIBRARY_PATH

将两个程序包放在同一个目录下

- **先编译 osip2**

tar zxvf libosip2-3.6.0.tar.gz
cd libosip2-3.6.0
./configure
make && make install

- **解压 eXosip2**

tar xzxvf libeXosip2-3.6.0.tar.gz
拷贝头文件和库文件
然后将 osip 头文件和编译生成的库文件（libosip2.a 和 libosipparser2.a）拷贝到相应的目录

cp -rf libosip2-3.6.0/include/osip2 libeXosip2-3.6.0/include/osip2
cp -rf libosip2-3.6.0/include/osipparser2 libeXosip2-3.6.0/include/osipparser2
cp libosip2-3.6.0/src/osip2/.libs/libosip2.a /usr/lib
cp libosip2-3.6.0/src/osipparser2/.libs/libosipparser2.a /usr/lib

- **再编译 eXosip2**

cd libeXosip2-3.6.0
./configure
make

如果上面配置的时候没有 --prefix 这个配置项，则默认安装到**/usr/local/lib**，
export LD_LIBRARY_PATH=/usr/local/lib/:$LD_LIBRARY_PATH

Osip 开发详解

准备工作：核心数据结构

先认识几个结构体：**osip_t，osip_message_t，osip_dialog_t，osip_transaction_t**；

osip_t 是一个结构体，所有要使用 Osip 协议栈的事务处理能力的程序都要第一步就初始化它（相对应于只使用 osipparser 库进行 SIP 消息字段解析的应用来说，如果只使用 parser 库到自己的程序中，想必对 SIP 协议栈已经很熟悉了，不需再往下看了^_^），它内部主要

是定义了 Osip 协议栈的四个主要事务链表、消息实际发送函数及状态机各状态事件下的回调函数等；

```
typedef struct osip osip_t;

/**
 * Structure for osip handling.
 * @struct osip
 */
   struct osip {

        void *application_context;
                                /**< User defined Pointer */

        /* list of transactions for ict, ist, nict, nist */
        osip_list_t osip_ict_transactions;
                                        /**< list of ict transactions */
        osip_list_t osip_ist_transactions;
                                        /**< list of ist transactions */
        osip_list_t osip_nict_transactions;
                                        /**< list of nict transactions */
        osip_list_t osip_nist_transactions;
                                        /**< list of nist transactions */

        osip_list_t ixt_retransmissions;/**< list of ixt elements */

        osip_message_cb_t msg_callbacks[OSIP_MESSAGE_CALLBACK_COUNT];
/**@internal */
        osip_kill_transaction_cb_t kill_callbacks[OSIP_KILL_CALLBACK_COUNT];

/**@internal */
        osip_transport_error_cb_t
            tp_error_callbacks[OSIP_TRANSPORT_ERROR_CALLBACK_COUNT];

            /**@internal */

        int (*cb_send_message) (osip_transaction_t *, osip_message_t *, char *,
                                int, int);
                                        /**@internal */

#if defined(HAVE_DICT_DICT_H)
        dict *osip_ict_hastable;          /**< htable of ict transactions */
        dict *osip_ist_hastable;          /**< htable of ist transactions */
        dict *osip_nict_hastable;         /**< htable of nict transactions */
```

```
        dict *osip_nist_hastable;          /**< htable of nist transactions */
#endif
    };
```

osip_message_t 是 SIP 消息的 C 语言结构体存储空间，收到 SIP 消息解析后存在该结构中方便程序使用接收到的消息中的指定的字段，发送消息前为方便设置要发送的字段值，将要发送的内容存在该结构中等发送时转为字符串；

osip_dialog_t 则是 SIP RFC 中的 dialog 或叫 call leg 的定义，它标识了 uac 和 uas 的一对关系，并一直保持到会话(session)结束，一个完整的 dialog 主要包括 from,to,callid,fromtag,totag,state 等（可查看源码），其中 fromtag,totag,callid 在一个 dialog 成功建立后才完整，体现在 SIP 消息中，就是 From、To 的 tag，Call-id 字段的值相同时，这些消息是属于它们对应的一个 Dialog 的，例如将要发起 invite 时，只有 fromtag,callid 填充有值，在收到 to 远端的响应时，收到 totag 填充到 dialog 中，建立成功一个 dialog，后继的逻辑均是使用这个 dialog 进行处理（如 transaction 事务处理），state 表示本 dialog 的状态，与 transaction 的 state 有很大的关联，共用由 Enum 结构 state_t 定义；

osip_transaction_t 则是 RFC 中的**事务**的定义，它表示的是一个会话的某个 Dialog 之间的**某一次消息发送及其完整的响应**，例如 invite-100-180-200-ack 这是一个完整的事务，bye-200 这也是一个完整的事务，体现在 SIP 消息中，就是 Via 中的 branch 的值相同表示属于一个事务的消息（当然，事务是在 Dialog 中的，所以 From、To 的 tag，Call-id 值也是相同的），事务对于 UAC,UAS 的终端类型不同及消息的不同，分为四类，前面说的 invite 的事务，主叫 uac 中会关联一个 **ict** 事务，被叫 uas 会关联一个 **ist** 事务，而除了 invite 之外，都归类定义主叫 **nict**，被叫 **nist**，在 Osip 中，它是靠有限状态机来实现的上述四种事务（osip_fsm_type_t 中定义）的，它的主要属性值有 callid,transactionid，分别来标识 dialog 和 transaction，其中还有一个时间戳 birth_time 标识事务创建时间，可由超时处理函数用来判断和决定超时情况下的事务的进行和销毁，而它的 state 属性是非常重要的，根据上述的事务类型不同，其值也不同，它是前面提到的状态机的“状态”，在实际状态机的逻辑执行中是一个关键值；

Osip 初始化

提到 osip 的初始化，可能大家都看过官方文档里第一页的代码，首先就是 **osip_init**(&osip) 初始化了全局的 osip_t 结构体，然后对它的回调函数进行设置，很多人估计就是一看到这密密麻麻的一页多的 **call_back** 设置被吓到了，但结合前面分析的三个结构体的含义，这里的含义就很清晰了：

osip_t 中有一个 cb_send_message 函数指针，它是 Osip 最终与外界网络交互的接口，它的参数有(osip_transaction_t * trn,　　/*本消息所属的事务*/

```
osip_message_t * sipmsg, /*待发送的消息结构体*/
char *dest_socket_str,        /*目标地址*/
int32_t dest_port,                  /*目标端口*/
int32_t send_sock)            /*用来发送消息的 socket*/
```

其中 trn 传入主要是为了方便获取事务的上下文数据，它有一个 void 指针 your_instance，可以用来传入更多数据方便发送消息时参考，例如将该事务所属的 dialog 指针传入；

而 sipmsg 则是我们要发送的 SIP 消息的 C 结构体，使用 osip_message_to_str 将其按 RFC 文档格式转换为一个字符串（osip 中的 parser 模块的主要功能），再通过任意你自己的网络数据发送函数使用 send_sock 发送给 dest_socket_str 和 dest_port 指定的目标，当然，要记得使用 osip_free 释放刚才发送出去的字符串占用的内存，Osip 中很多 osipparser 提供的消息解析处理函数都是动态内存分配的，使用完毕后需要及时释放；

使用 osip_set_cb_send_message 成功设置回调函数，我们的 SIP 消息就有了出口了，下面继续分析（当然，了解到了上面的流程，也可以手工指定了）。

下面的回调函数分为三类，分别是**普通事务消息**（osip_message_callback_type_t 中定义）的处理回调函数、**事务销毁事件**（osip_kill_callback_type_t 中定义）的清理回调函数以及事务执行过程中的**错误事件**（osip_transport_error_callback_type_t 中定义）处理回调函数：

先说简单的，事务销毁事件，事务正常结束（成功完成状态机流程）或由超时处理函数强制终结等情况下均调用了这些回调函数，一般就是释放事务结构体，为 ICT,NICT,IST,NIST 各设置或共用一个回调函数均可，只要正确释放不再使用的内存即可；

错误处理函数则是在整个状态机执行过程中发生的任何错误的出口，一般用来安插 log 函数方便调试，也可以直接设为空函数；

而最关键的就是**正常消息的处理回调函数**了，其量是非常大的，但仔细分下类，也和上面的回调函数一样，也是分为四类，我们可有根据实际程序的需要来进行设置，例如，SIP 电话机就不需要处理 OSIP_NIST_REGISTER_RECEIVED 这个 SIP 注册服务器才需要处理的 Register 消息事件了，精简一下，如果只是要做一个只需要实现主叫功能且不考虑错误情况的 UAC 的 Demo 软电话程序，**则只需要设置如下几个事件的回调函数**：

Ict,ist,nict, nist

- OSIP_ICT_INVITE_SENT　发出 Invite 开始呼叫
- OSIP_ICT_STATUS_1XX_RECEIVED　收到 180
- OSIP_ICT_STATUS_2XX_RECEIVED　收到 200
- OSIP_ICT_ACK_SENT　　发出 ack 确定呼叫
- OSIP_NICT_BYE_SENT　　发出 bye 结束呼叫
- OSIP_NICT_STATUS_2XX_RECEIVED　收到 200 确认结束呼叫
- OSIP_NIST_BYE_RECEIVED　收到 bye 结束呼叫
- OSIP_NIST_STATUS_2XX_SENT　发出 200 确定结束呼叫

而要增加接受呼叫的被叫 UAS 功能，则只需要增加如下事件：

- OSIP_IST_INVITE_RECEIVED　收到 invite 开始呼叫
- OSIP_IST_STATUS_1XX_SENT　发出 180
- OSIP_IST_STATUS_2XX_SENT　发出 200
- OSIP_IST_ACK_RECEIVED　　收到 ack 确认呼叫

具体的函数定义，则直接参考 osip_message_cb_t，osip_kill_transaction_cb_t，osip_transport_error_cb_t 即可，回调函数的设置同上可以手工设置，也可以使用 Osip 提供的对应的 osip_set_xxx_callback 函数；

发出 SIP 消息

要发送 SIP 消息，从上面的分析可知有几个必要的条件，osip_messag_t 结构的待发送消息，osip_dialog_t 结构体的 dialog 以及 osip_transaction_t 的事务；

首先 **osip_malloc** 新分配一个 dialog，使用 osip_to_init，osip_to_parse，osip_to_free 这类 parser 函数功能函数按 RFC 设置 call-id,from,to,local_cseq 等必要字段（原则是：后面生成实际 SIP 消息结构体要用到的字段就需要设置），使用 osip_message_init 初始化一个 sipmsg，根据 dialog 来填充该结构体（不同的消息填充的数据是不同的，没有捷径可走，只能看 RFC 根据需要填充字段），如果要给 SIP 消息添加 Body 例如 SDP 段，需要使用 osip_message_set_body，osip_message_set_content_type 函数，设置的值是纯文本，如果是 SDP，Osip 有提供简单的解析和生成便捷函数例如

sdp_message_to_str,sdp_message_a_attribute_add，但只是简单的字符操作，要填充合法的字段需要自己参考 SDP 的 RFC 文档，同样没捷径可走。

现在我们有了两个必要条件了，还有最后一个也是最关键的部件，就是事务的创建和触发，

int **osip_transaction_init**(
osip_transaction_t ** transaction, /*返回的事务结构体指针*/
osip_fsm_type_t ctx_type, /*事务类型 ICT/NICT/IST/NIST*/
osip_t * osip, /*前文说的全局变量*/
osip_message_t * request) /*前面生成的 sipmsg*/

创建了一个新的事务，并自动根据事务类型、dialog 和 sipmsg 进行了初始化，最重要的是它使用了__osip_add_ict 等函数，将本事务插入到全局的 osip_t 结构体的全局 FIFO 链表中去了，不同的事务类型对应不同的 FIFO，由前文可知，本类函数有四个，FIFO 也有四个，对应 ICT,NICT,IST,NIST，注意这个这里使用 osip_transaction_set_out_socket 把发送 sip 消息的 socket 接口配给该事务，方便自动调用前面设置的发送消息回调函数使用它自动发送消息；

前文提到了 transaction 里的 state 作为状态机的“状态”，要执行状态机，就需要有“事件”来触发，事件结构体 osip_event_t 需要使用 osip_new_outgoing_sipmessage 来对 sipmsg 进行探测生成，设置正确的事件值，省却了我们手工设置的工作，它调用 evt_set_type_outgoing_sipmessage 来设置“事件”type_t，并将 sipmsg 挂到事件结构体的 sip 属性值上，有了根据消息分析出的事件后，使用 osip_fifo_add(trn->transactionff, ev)将事件插入到事务的事件 FIFO 中，即 transactionff 属性；

有了上面的发送消息的必要条件了，**消息是如何实际出发的呢**？上面提到了，SIP 消息的发送和响应是一个事务，不能隔离开来，即消息的发送需要**事务状态机**来控制，我们上面设置了状态机的状态和事件，要触发它，就是要执行状态机了：

- **osip_ict_execute**
- **osip_nict_execute**
- **osip_ist_execute**
- **osip_nist_execute**

分别用来遍历前面提到的四个事务 FIFO，取出事务，再依次取出事务内的事件 FIFO 上的事件，使用 **osip_transaction_execute** 依次执行（有兴趣的可以更深一步去查看，可以看到它最终就是调用了我们前面设置的消息回调函数，至于具体调用哪个，这就是 OSIP 协议栈内部帮我们做的大量的工作了^_^）；

如果某个事务不能正常终结怎么办呢？例如发出了 Invite 没有收到任何响应，按 RFC 定义，不同的事务有不同的超时时间，osip_timers_ict[nict|ist|nist]_execute 这些函数就是来根据取出的事务的时间戳与当前时间取差后与规定的超时时间比对，如果超时，就自动设置了超时“事件”并将事务“状态”设为终结，使用前面设定的消息超时事件回调函数处理即可

（如果设置了）；

如果网络质量不稳定，经常丢失消息，需要使用 osip_retransmissions_execute 函数来自动重发消息而不是等待超时；

为了即时响应 SIP 消息的处理推动状态机，上述的九个函数需要不停执行，可以将它放入单独线程中。

收到 SIP 消息

有了前面的发送 SIP 消息的理解，接收消息的处理就方便理解了，收到 SIP 消息，使用 osip_parse 进行解析，得到一个 osip_message_t 的 sipmsg，使用 evt_set_type_incoming_sipmessage 得到事务的“事件”，并同上将 sipmsg 挂到事件结构体的 sip 字段，随后立即使用 osip_find_transaction_and_add_event 来根据“事件”查找事务（有兴趣可以深入看一下，事务的查找是通过 SIP 消息 Via 中的 branch 来匹配的），否则新建事务，然后推动状态机执行。

状态机内部逻辑

弄清了上面的状态机的大概逻辑，设置正确完备的回调函数，就可以正确使用 Osip 来进行工作了，如果要进一步深入 Osip，比如要扩展 Osip 的状态机处理自定义的消息字段和实现新的事务逻辑来生成新业务时，就需要对状态机的内部逻辑有一定的了解；

前面一再强调，Osip 内部的几个重要的数据结构 **osip_message_t,osip_dialog_t,osip_transaction_t**，其中面向用户的主要是前后两个，而中间的 dialog 则很多时候是在状态机内部使用的，例如：收到消息，解析到 sipmsg 中，查找 transaction 并进行驱动，随后找到它关联的 dialog（或者新生成）解析填充要发送的消息结构体 sipmsg，再次根据 dialog 和 sipmsg 查找或生成 transaction。

如果要扩展 Osip，要做工作主要有：

- 扩展 osip_message_t，增加要解析的字段或消息头，并参考原 Osip 函数生成对应的 SIP 字符串生成和解析函数；
- 扩展 osip_dialog_t，增加新的属性，对应 osip_message_t 的新增内容；
- 扩展状态机的事件和状态类型，设置对应的回调函数，并关联新增事件和状态类型到 osip_message_t 的解析函数或 osip_dialog_t 的初始化函数中，而 osip_transaction_t 大多

数时候不需要扩展，只要在对应的事务类型（大多数时候是 NICT、NIST）处理逻辑中，增加对新增事件和状态类型的判断和调用回调函数的逻辑即可。

Osip2 协议栈简介

Osip2 是一个开放源代码的 sip 协议栈，是开源代码中不多使用 **C 语言**写的协议栈之一，它具有短小简洁的特点，专注于 sip 底层解析使得它的效率比较高。但缺点也很明显，首先就是**可用性差**，**没有很好的 api 封装**，使得上层应用在调用协议栈时很破碎；其次，只做到了 **transaction 层次的协议过程解析**，缺少 **call**、**session**、**dialog** 等过程的解析，这也增加了使用的难度；再次，**缺少线程并发处理**的机制，使得它的处理能力有限。

Osip2 协议栈大致可以分为三部分：

- **sip 协议的语法分析、**
- **sip 协议的过程分析**
- **协议栈框架。**

1、Sip 协议的语法分析

主要是 osipparser2 部分，目前支持 RFC3261 和 RFC3265 定义的 sip 协议消息，包括 **INVITE**、**ACK**、OPTIONS、**CANCEL**、**BYE**、**SUBSCRIBE**、**NOTIFY**、**MESSAGE**、REFER 和 INFO。不支持 RFC3262 定义的 PRACK。

遵循 RFC3264 关于 SDP 的 offer/answer 模式。带有 SDP 的语法分析。
支持 MD5 加解密算法。支持 Authorization、www_authenticate 和 proxy_authenticate。

2、Sip 协议的过程分析

主要是 osip2 部分，基于 RFC3261、RFC3264 和 RFC3265 的 sip 协议描述过程，围绕 **transaction** 这一层来实现 sip 的解析。

Transaction 是指一个发送方和接收方的交互过程，由请求和应答组成。
请求分为 Invite 类型和 Non-Invite 类型。
应答分为响应型的应答和确认型的应答。

响应型的应答是指这个应答仅代表对方收到请求。请求经过处理后都必须返回确认型的应答。响应型的应答有 1xx，确认型的应答包括 2xx、3xx、4xx、5xx 和 6xx。

一个 transaction 由一个请求和一个或多个响应型应答、一个确认型应答组成。

Transaction 根据请求的不同和发送/接收的不同可以分为四类：**ict、nict、ist 和 nist**。

- Ict 是指 Invite client transaction，就是会话邀请的发起方。
- Nict 是指 Non-Invite client transaction，是指非邀请会话的发起方。
- Ist 是指 Invite server tranaction，是指会话邀请的接收方。
- Nist 是指 Non-Invite server transaction，是指非邀请会话的接收方。

每种类型的 transaction 都有自己相应的**状态机**，Osip2 协议栈根据状态机来处理所有的 **sip 事件**，所以这部分就是整个协议栈的核心。但是因为 **Osip2 只做到 transaction 这一层**，所以它可以忽略掉 call、registration 等应用的复杂性，显得相当简单，这就使得需要使用它的应用必须要自己处理应用的逻辑。必须注意的一点是，transaction 的资源在 Osip 里是由协议栈负责释放的，但是在 Osip2 里改成由使用的应用负责释放。

3、协议栈框架

框架并不是指代码的某一部分，而是指它的构成形式。
主要有三部分：

- **底层套接字接收/发送[win/linux: socket api,, libevent,libuv,libcurl,boost,...]，**
- **模块间通信管道，**
- **上层调用 api 接口。**

Osip2 并不实现底层套接字的接收/发送，由 **eXosip 实现，现在只支持 UDP** 的链路连接。

模块间的通信管道包括：transaction 的消息管道、jevent 的消息管道。

【仔细体会这段话：】

Transaction 的消息管道是驱动其状态机的部件，通过不断的接收来自底层套接字的远端信令(select,poll,iocompletionport)，或者来自上层调用的指令(api)，根据上述的状态机制来驱动这个 transaction 的运转。

Jevent 的消息管道是 eXosip 实现的，用于汇报底层事件，使得调用程序能处理感兴趣的事件。

上层调用的 api 接口大致有两类：sip 协议的调用接口和 sdp 协议的调用接口。

EXosip 封装了大部分的 sip 协议调用接口，一般来说都不需要直接调用 osip2 的接口函数。

exosip 的模块构成

底层连接管理

extl.c、extl_udp.c、extl_tcp.c、extl_dtls.c、extl_tls.c 是与网络连接有关的文件。

实现了连接的建立，数据的接收以及发送等相关的接口。

其中，extl_udp.c 为使用 UDP 连接的实现，extl_tcp.c 为使用 TCP 连接的实现。Extl_dtls.c 以及 extl_tls.c 都是使用安全 socket 连接的实现。

内部功能模块实现

Jauth.c、jcall.c、jdialog.c、jevents.c、jnotify.c、jpublish.c、jreg.c、jrequest.c、jresponse.c、jsubscribe.c 等文件实现了内部对一些模块的管理，这些模块正如其文件名所表示的，jauth.c 主要是认证，jcall.c 则是通话等等。

上层 API 封装实现

Excall_api.c、exinsubsription_api.c、exmessag_api.c、exoptions_api.c、expublish_api.c、exrefer_api.c、exregister_api.c、exsubsribtion_api.c 这几个以 api 为后缀的文件，实现各个子模块的管理。应用程序可以调用这里提供的接口，方便的构造或者发送 sip 消息。

其他

- Inet_ntop.c 实现 ip 地址的点分十进制与十六进制表示之间的转换。
- Jcallback.c 实现一堆回调函数，这些回调函数就是用来注册到 osip 库的。我们使用 exosip 库，就是避免直接使用 osip 库，因为一些工作 exosip 已经帮我们做了，所以这样一来，可以简化上层的实现。
- Udp.c 文件主要用来对通过 UDP 连接接收到的消息进行分类处理。
- Exutilis.c 文件实现一些杂项的函数。有 ip 地址到字符串之间的转换，域名的解析等一些辅助的功能函数。
- Exconf.c 文件实现了 exosip 的初始化相关的接口，包括后台任务的实现。实际上是“configuration api”的实现。
- Exosip.c 文件实现了与 exconf.c 文件相似的功能。比如管道的使用，exosip 上事务的创建和查找，register 和 subscribe 的更新，认证信息的处理等。

exosip 的初始化

Exosip 的初始化有两部分组成，这主要是从使用 exosip 的角度看。

对 exosip 全局结构体变量的配置

这步通过调用接口 eXosip_init 完成。主要完成工作如下：

- 初始化条件变量和互斥信号量。
- 调用 osip_init 初始化 osip 库，并将生成 osip 结构体给 exosip，同时也让 osip 的 application_contexgt 指针指向 exosip，也就是二者相互指向。
- 调用 eXosip_set_callbacks 设置 osip 的回调函数，所以回调函数都是 exosip 自己实现。
- 调用 jpipe 创建通信用的 pipe，之前已经说了，对于 windows 平台，是通过 socket 接口模拟实现的。
- 初始化其上的事务和事件队列。主要，这不同于 osip 的事务和事件队列。
- 调用 extl 指向的结构体的 init 函数指针，初始化网络接口。

在 socket 接口上进行监听

这步通过调用 **eXosip_listen_addr** 接口完成。

主要完成工作如下：

- 将 eXosip 全局变量的 eXtl 指针指向 eXtl_udp 全局变量。
- 根据参数，配置 extl_protocol 和 exosip 上有关 ip 端口地址等信息。另外，调用 extl_udp 的 tl_open 函数指针，完成在本机指定的端口上监听连接的工作。需要注意的是，虽然是监听，但是使用的 UDP 来建立连接的，所以消息的 recv 和发送在同一个 socket 上完成。在 osip 中设置的 out_socket 并不会起作用。
- 调用 osip_thread_create 创建 exosip 后台任务，**用于驱动 osip 的状态机**。（在 osip 中，在发送 sip 消息部分，提到将 9 个函数放到一个线程中执行，exosip 就是这样做的）

下面展示了初始化的示例代码：

```
int i;
TRACE_INITIALIZE (6, stdout);
i=eXosip_init();
if (i!=0)
return -1;

i = eXosip_listen_addr (IPPROTO_UDP, NULL, port, AF_INET, 0);

if (i!=0)
{
eXosip_quit();
```

```
fprintf (stderr, “could not initialize transport layer\n”);
return -1;
}

… then you have to send messages and wait for eXosip events…
```

这样，在初始化完成后，我们基本上完成了对内存中所用数据结构的配置，同时启动了一个后台任务负责 osip 状态机的驱动。

exosip 数据收发整体框架

接收过程

在初始化过程中我们创建了一个**后台任务**，现在可以看看这个后台任务都做了哪些操作。任务的执行函数为**_eXosip_thread**，在该接口中，循环不断的调用 **eXosip_execute**。在每一次的 eXosip_execute 执行中，完成如下的工作：

a. 首先计算出底层 osip 离当前时间最近的超时时间。也就是查看底层所有的超时事件，找出其中的最小值，然后将其与当前时间做差，结果就是最小的超时间隔了。这步是通过调用接口 **osip_timers_gettimeout** 完成的。主要检查 osip 全局结构体上的 ict、ist、nict、nist 以及 ixt 上所有事务的事件的超时时间。如果 ict 事务队列上没有事件，则说明没有有效的数据交互存在，返回值为默认的一年，实际上就是让后面的接收接口死等。如果有事务队列上的事件的超时时间小于当前值，则说明已经超时了，需要马上处理，此时将超时时间清为零，并返回。

b. 调用 **eXosip_read_message** 接口从底层接收消息并处理。如果返回-2，则任务退出。

c. **执行 osip 的状态机**。具体为执行 **osip_timers_ict**（ist|nict|nist）**_execute** 和 osip_ict（ist|nict|nist）_execute 这几个函数。最后还检查释放已经终结的 call、registrations 以及 publications。

d.如果 keep_alive 设置了，则调用**_eXosip_keep_alive** 检查发送 keep_alive 消息。

这样，当远端的终端代理发送 sip 消息过来时，会被之前创建的监听端口捕获（sip 协议默认的端口为 5060）。在调用 **eXosip_read_message** 接口时会将其接收上来。接收上来的数据存放在 buffer 中交给接口**_eXosip_handle_incoming_message** 来处理。

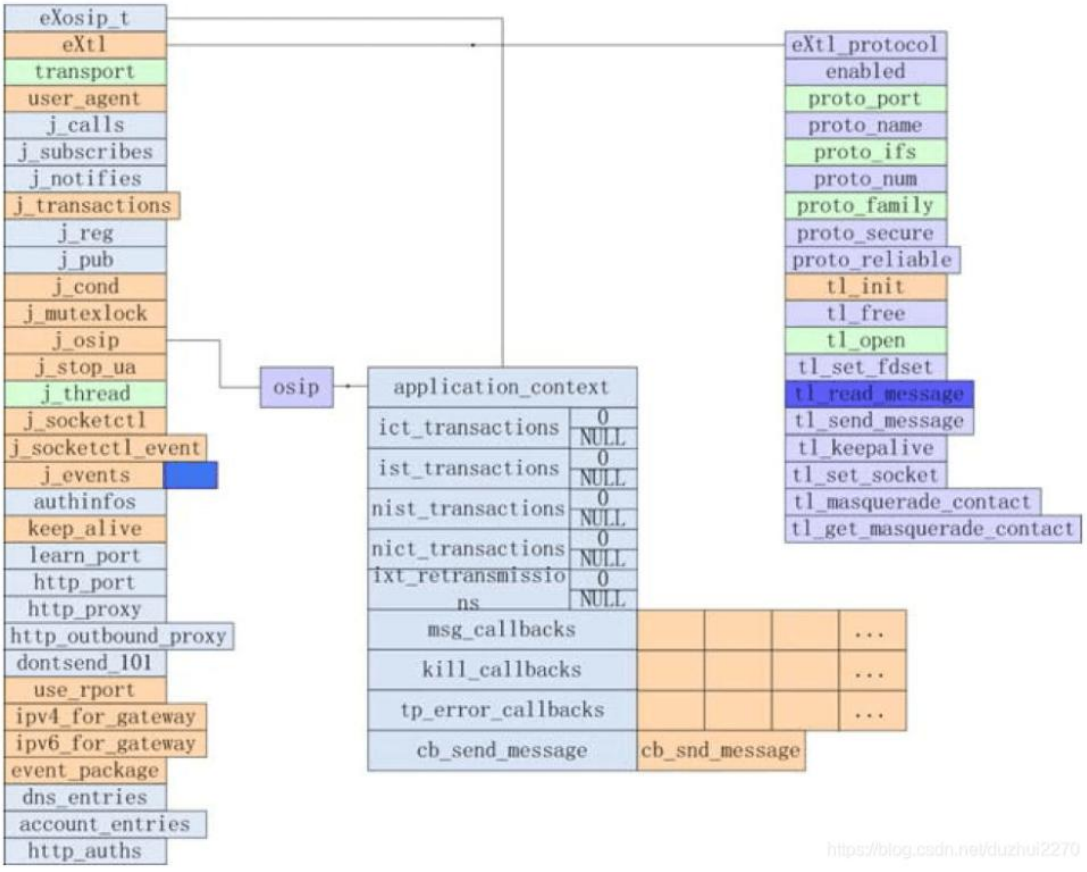
在其中首先调用 osip_parse 进行消息的解析，这是 osip 的核心功能之一。

数据解析后，会生成一个 osip_event 类型的事件。

接着调用 osip_message_fix_last_via_header 将接收到该消息的 ip 地址和端口根据需要设置到数据头的 via 域中。这在消息返回时有可能发挥作用。

为了能够让消息正确的被处理，调用 osip_find_transaction_and_add_event 接口将其添加到 osip 的事务队列上。处理在这之后发生了分叉，如果 osip 接纳了该事件，接口直接返回，因为这说明该事件在 osip 上已经有匹配的事务了，或者说该事件是某一个事务过程的一部分。这样在后面执行状态机的接口时，该事件会被正确的处理。如果 osip 没有拿走该事件，则说明针对该事件还没有事务与之对应。此时，我们首先检查其类型，如果是 request，则说明很可能是一个新的事件到来（这将触发服务端的状态机的建立），调用 eXosip_process_newrequest 接口进行处理。如果是 response，则调用接口 eXosip_process_response_out_of_transaction 处理。在 eXosip_process_newrequest 接口中，如果是合法的事件，则会为其创建一个新的事务。

也就是说这是新事务的第一个事件。经过一大堆的处理后，该事件可能就被 osip 消化了，或者被 exosip 消化了。如果需要上报给应用，由应用拿来对一些信息进行存储或者进行图形显示之类，则会将该事件添加到 exosip 的事件队列上。如下图所示：



应用程序在 exosip 初始化完之后需要调用如下类似的代码，不断从事件队列上读取事件，并进行处理。

```
eXosip_event_t *je;
```

```
for (;;)
{
je = eXosip_event_wait (0, 50);
eXosip_lock();
eXosip_automatic_action ();
eXosip_unlock();
if (je == NULL)
      break;

if (je->type == EXOSIP_CALL_NEW)
{
…
…
}
else if (je->type == EXOSIP_CALL_ACK)
{
…
…
}
else if (je->type == EXOSIP_CALL_ANSWERED)
{
…
…
}
else …
…
…
eXosip_event_free(je);
}
```

读到事件后，判断其类型进行对应的处理。这样整个接收流程就完成了。

发送过程

要发送数据时，需要根据消息类型，调用 exosip 对应模块的 api 接口函数来完成。

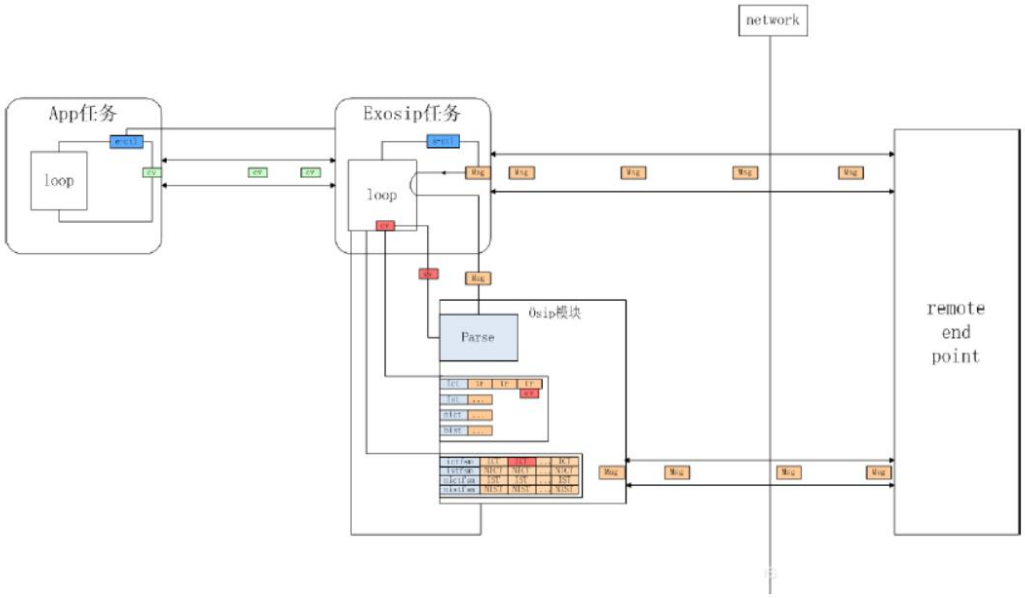
如果要发送的 sip 消息不属于当前已有的任何事务，则类似接收过程，调用 osip 的相关接口创建一个新的事务，同时根据消息生成一个事件，加到事务的事件队列上。

最后，唤醒 exosip 后台进程，使其驱动 osip 状态机，执行刚添加的事件，从而完成数据的状态处理和发送。

当然，也有一些消息并不通过 osip 状态机，而是由 exosip 直接调用回调函数 cb_snd_message 完成发送。

exosip 与上层应用以及 osip 之间的流程关系

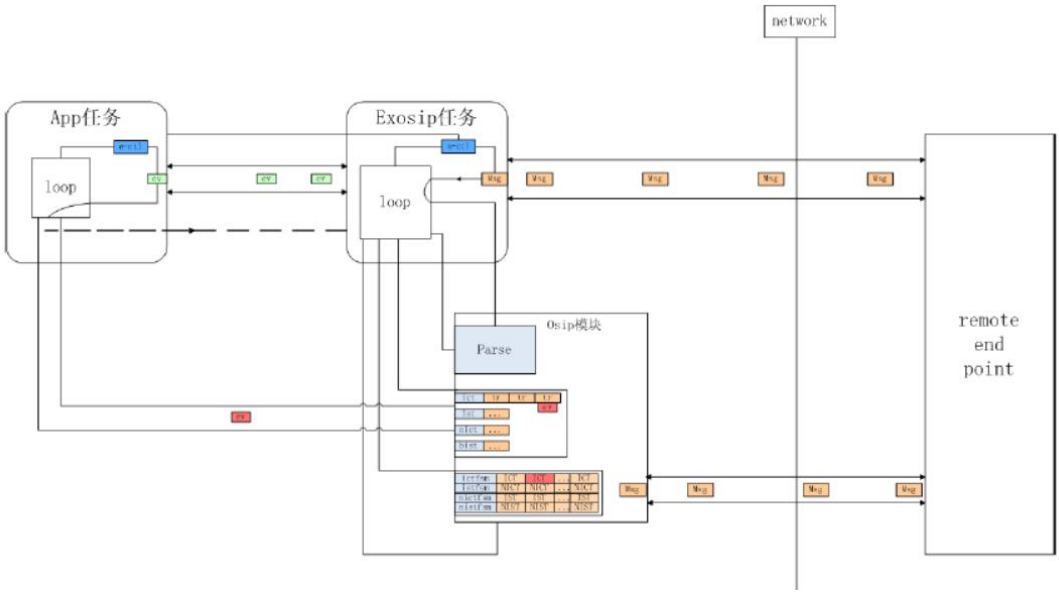
exosip 是对 osip 库的扩展，那么它与 osip 之间是什么样的关系呢，这可参看下图：



上图为接收过程的示意图。Exosip 后台任务不断从网络另一端读取 sip 消息，交给 osip 的 parser 模块解析，并将其转换为 events，添加到事务队列上。

同时，exosip 后台任务在不断的驱动 osip 的状态机，这样，事务队列上的事件就会被处理。如果需要响应对端，状态机会根据回调函数的设置，直接完成数据的发送。

同样，如果要将当前处理反馈给应用，则将其发送到事件队列上（这里是 exosip 的事件队列），并通过 e-ctl 管道通知应用进行处理。应用需要发送数据时，流程如下图所示：



此时，应用调用 exosip 提供的辅助函数（上图中虚线示意此关系），构造 osip 事件，将其添加到 osip 的事务队列上。同时，应用通过 s-ctl 管道通知 exosip 后台任务执行状态机。在 exosip 执行状态机的过程中，sip 消息会被发送到网络另一端的终端。

学习网站：http://www.hellotongtong.com/
福优学苑：【 QQ 咨询：3212001984】
加微信可以提供远程服务,微信服务二维码(13661137824)：

osip+eXosip 案例代码

一个基于 osip 库的 SIP 协议的简单的 UAC 代理客户端和 UAS 代理服务器端，并进行了编译连接，代码整理后如下：

UAC 代理客户端

---------- UAC 代理客户端的代码整理 ----------
linux 编译命令：
gcc -o uac uaclinux.c -L /usr/local/lib -losip2 -leXosip2 -losipparser2

```
#include "stdafx.h"///linux to comment it

#include <eXosip2/eXosip.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
//#include <netinet/in.h> ///linux
```

```
#include <winsock2.h> ///windows

int main(int argc,char *argv[])
{

    struct eXosip_t *context_eXosip;

    eXosip_event_t *je;
    osip_message_t *reg=NULL;
    osip_message_t *invite=NULL;
    osip_message_t *ack=NULL;
    osip_message_t *info=NULL;
    osip_message_t *message=NULL;

    int call_id,dialog_id;
    int i,flag;
    int flag1=1;

    char *identity="sip:140@127.0.0.1";    //UAC1，端口是 15060
    char *registar="sip:133@127.0.0.1:15061"; //UAS,端口是 15061
    char *source_call="sip:140@127.0.0.1";
    char *dest_call="sip:133@127.0.0.1:15061";
    //identify 和 register 这一组地址是和 source 和 destination 地址相同的
    //在这个例子中，uac 和 uas 通信，则 source 就是自己的地址，而目的地址就是 uac1
的地址
    char command;
    char tmp[4096];

    printf("r   register to server\n");
    printf("c   cancel \n");
    printf("i   calling \n");
    printf("h   hang up\n");
    printf("q   quit\n");
    printf("s   run:INFO\n");
    printf("m   run:MESSAGE\nn");

    //初始化
    i=eXosip_init();

    if(i!=0)
    {
        printf("Couldn't initialize eXosip!\n");
        return -1;
    }
```

```
    else
    {
        printf("eXosip_init successfully!\n");
    }

    //绑定 uac 自己的端口 15060，并进行端口监听
    i=eXosip_listen_addr(IPPROTO_UDP,NULL,15060,AF_INET,0);
    if(i!=0)
    {
        eXosip_quit();
        fprintf(stderr,"Couldn't initialize transport layer!\n");
        return -1;
    }
    flag=1;

    while(flag)
    {
        //输入命令
        printf("Please input the command:\n");
        scanf("%c",&command);
        getchar();

        switch(command)
        {
        case 'r':
            printf("This modal is not completed!\n");
            break;
        case 'i'://INVITE，发起呼叫请求

i=eXosip_call_build_initial_invite(&invite,dest_call,source_call,NULL,"This is
a call for conversation");
            if(i!=0)
            {
                printf("Initial INVITE failed!\n");
                break;
            }
            //符合 SDP 格式，其中属性 a 是自定义格式，也就是说可以存放自己的信息，
            //但是只能有两列，比如帐户信息
            //但是经过测试，格式 vot 必不可少，原因未知，估计是协议栈在传输时需要
检查的
            snprintf(tmp,4096,
                     "v=0\r\n"
                     "o=anonymous 0 0 IN IP4 0.0.0.0\r\n"
                     "t=1 10\r\n"
```

```
                    "a=username:zhangsan\r\n"
                    "a=password:123456\r\n");

            osip_message_set_body(invite,tmp,strlen(tmp));
            osip_message_set_content_type(invite,"application/sdp");

            eXosip_lock();
            i=eXosip_call_send_initial_invite(invite); //invite SIP INVITE
message to send
            eXosip_unlock();

            //发送了 INVITE 消息，等待应答
            flag1=1;
            while(flag1)
            {
                je=eXosip_event_wait(0,200); //Wait for an eXosip event
                //(超时时间秒，超时时间毫秒)
                if(je==NULL)
                {
                    printf("No response or the time is over!\n");
                    break;
                }
                switch(je->type)   //可能会到来的事件类型
                {
                case EXOSIP_CALL_INVITE:   //收到一个 INVITE 请求
                    printf("a new invite received!\n");
                    break;
                case EXOSIP_CALL_PROCEEDING: //收到 100 trying 消息，表示请求正在
处理中
                    printf("proceeding!\n");
                    break;
                case EXOSIP_CALL_RINGING:   //收到 180 Ringing 应答，表示接收到
INVITE 请求的 UAS 正在向被叫用户振铃
                    printf("ringing!\n");
                    printf("call_id is %d,dialog_id is %d \n",je->cid,je->did);
                    break;
                case EXOSIP_CALL_ANSWERED: //收到 200 OK，表示请求已经被成功接受，
用户应答
                    printf("ok!connected!\n");
                    call_id=je->cid;
                    dialog_id=je->did;
                    printf("call_id is %d,dialog_id is %d \n",je->cid,je->did);

                    //回送 ack 应答消息
```

```
                    eXosip_call_build_ack(je->did,&ack);
                    eXosip_call_send_ack(je->did,ack);
                    flag1=0; //推出 While 循环
                    break;
                case EXOSIP_CALL_CLOSED: //a BYE was received for this call
                    printf("the other sid closed!\n");
                    break;
                case EXOSIP_CALL_ACK: //ACK received for 200ok to INVITE
                    printf("ACK received!\n");
                    break;
                default: //收到其他应答
                    printf("other response!\n");
                    break;
                }
                eXosip_event_free(je); //Free ressource in an eXosip event
            }
            break;

        case 'h':   //挂断
            printf("Holded!\n");

            eXosip_lock();
            eXosip_call_terminate(call_id,dialog_id);
            eXosip_unlock();
            break;

        case 'c':
            printf("This modal is not commpleted!\n");
            break;

        case 's': //传输 INFO 方法
            eXosip_call_build_info(dialog_id,&info);
            snprintf(tmp,4096,"\nThis is a sip message(Method:INFO)");
            osip_message_set_body(info,tmp,strlen(tmp));
            //格式可以任意设定，text/plain 代表文本信息;
            osip_message_set_content_type(info,"text/plain");
            eXosip_call_send_request(dialog_id,info);
            break;

        case 'm':
            //传输 MESSAGE 方法，也就是即时消息，和 INFO 方法相比，我认为主要区别
是：
            //MESSAGE 不用建立连接，直接传输信息，而 INFO 消息必须在建立 INVITE 的
基础上传输
```

```
            printf("the method : MESSAGE\n");

eXosip_message_build_request(&message,"MESSAGE",dest_call,source_call,NULL);
            //内容，方法，       to       ，from      ，route
            snprintf(tmp,4096,"This is a sip message(Method:MESSAGE)");
            osip_message_set_body(message,tmp,strlen(tmp));
            //假设格式是 xml
            osip_message_set_content_type(message,"text/xml");
            eXosip_message_send_request(message);
            break;

        case 'q':
            eXosip_quit();
            printf("Exit the setup!\n");
            flag=0;
            break;
        }
    }

    return(0);
}
```

UAS 代理服务器端

---------- UAS 代理服务器端的代码整理 ----------

```
// uas.cpp : 定义控制台应用程序的入口点。
//

#include "stdafx.h"

# include <eXosip2/eXosip.h>
# include <stdio.h>
# include <stdlib.h>
# include <stdarg.h>
// # include <netinet/in.h>

#include <Winsock2.h>
```

```
int main (int argc, char *argv[])
{
    eXosip_event_t *je = NULL;
    osip_message_t *ack = NULL;
    osip_message_t *invite = NULL;
    osip_message_t *answer = NULL;
    sdp_message_t *remote_sdp = NULL;
    int call_id, dialog_id;
    int i,j;
    int id;
    char *sour_call = "sip:140@127.0.0.1";
    char *dest_call = "sip:133@127.0.0.1:15060";//client ip
    char command;
    char tmp[4096];
    char localip[128];
    int pos = 0;
    //初始化 sip
    i = eXosip_init ();
    if (i != 0)
    {
        printf ("Can't initialize eXosip!\n");
        return -1;
    }
    else
    {
        printf ("eXosip_init successfully!\n");
    }
    i = eXosip_listen_addr (IPPROTO_UDP, NULL, 15061, AF_INET, 0);
    if (i != 0)
    {
        eXosip_quit ();
        fprintf (stderr, "eXosip_listen_addr error!\nCouldn't initialize
transport layer!\n");
    }
    for(;;)
    {
        //侦听是否有消息到来
        je = eXosip_event_wait (0,50);
        //协议栈带有此语句,具体作用未知
        eXosip_lock ();
        eXosip_default_action (je);
        eXosip_automatic_refresh ();
        eXosip_unlock ();
        if (je == NULL)//没有接收到消息
```

```
            continue;
        // printf ("the cid is %s, did is %s/n", je->did, je->cid);
        switch (je->type)
        {
        case EXOSIP_MESSAGE_NEW://新的消息到来
            printf (" EXOSIP_MESSAGE_NEW!\n");
            if (MSG_IS_MESSAGE (je->request))//如果接受到的消息类型是 MESSAGE
            {
                {
                    osip_body_t *body;
                    osip_message_get_body (je->request, 0, &body);
                    printf ("I get the msg is: %s\n", body->body);
                    //printf ("the cid is %s, did is %s/n", je->did, je->cid);
                }
                //按照规则，需要回复 OK 信息
                eXosip_message_build_answer (je->tid, 200,&answer);
                eXosip_message_send_answer (je->tid, 200,answer);
            }
            break;


        case EXOSIP_CALL_INVITE:
            //得到接收到消息的具体信息
            printf ("Received a INVITE msg from %s:%s, UserName is %s, password
is %s\n",je->request->req_uri->host,
                    je->request->req_uri->port, je->request->req_uri->username,
je->request->req_uri->password);
            //得到消息体,认为该消息就是 SDP 格式.
            remote_sdp = eXosip_get_remote_sdp (je->did);
            call_id = je->cid;
            dialog_id = je->did;

            eXosip_lock ();
            eXosip_call_send_answer (je->tid, 180, NULL);//ringing
            i = eXosip_call_build_answer (je->tid, 200, &answer);
            if (i != 0)
            {
                printf ("This request msg is invalid!Cann't response!\n");
                eXosip_call_send_answer (je->tid, 400, NULL);
            }
            else
            {
                snprintf (tmp, 4096,
                          "v=0\r\n"
```

```
                    "o=anonymous 0 0 IN IP4 0.0.0.0\r\n"
                    "t=1 10\r\n"
                    "a=username:rainfish\r\n"
                    "a=password:123\r\n");

            //设置回复的 SDP 消息体,下一步计划分析消息体
            //没有分析消息体，直接回复原来的消息，这一块做的不好。
            osip_message_set_body (answer, tmp, strlen(tmp));
            osip_message_set_content_type (answer, "application/sdp");

            eXosip_call_send_answer (je->tid, 200, answer);
            printf ("send 200 over!\n");
        }
        eXosip_unlock ();

        //显示出在 sdp 消息体中的 attribute 的内容,里面计划存放我们的信息
        printf ("the INFO is :\n");
        while (!osip_list_eol ( &(remote_sdp->a_attributes), pos))
        {
            sdp_attribute_t *at;

            at = (sdp_attribute_t *) osip_list_get (&remote_sdp->a_attributes,
pos);
            printf ("%s : %s\n", at->a_att_field, at->a_att_value);//这里解
释了为什么在 SDP 消息体中属性 a 里面存放必须是两列

            pos ++;
        }
        break;
    case EXOSIP_CALL_ACK:
        printf ("ACK recieved!\n");
        // printf ("the cid is %s, did is %s/n", je->did, je->cid);
        break;
    case EXOSIP_CALL_CLOSED:
        printf ("the remote hold the session!\n");
        // eXosip_call_build_ack(dialog_id, &ack);
        //eXosip_call_send_ack(dialog_id, ack);
        i = eXosip_call_build_answer (je->tid, 200, &answer);
        if (i != 0)
        {
            printf ("This request msg is invalid!Cann't response!\n");
            eXosip_call_send_answer (je->tid, 400, NULL);

        }
```

```
            else
            {
                eXosip_call_send_answer (je->tid, 200, answer);
                printf ("bye send 200 over!\n");
            }
            break;
        case EXOSIP_CALL_MESSAGE_NEW://至于该类型和 EXOSIP_MESSAGE_NEW 的区别，源代码这么解释的
            /*
            // request related events within calls (except INVITE)
                  EXOSIP_CALL_MESSAGE_NEW,          < announce new incoming request.
            // response received for request outside calls
                    EXOSIP_MESSAGE_NEW,          < announce new incoming request.
                    理解是：EXOSIP_CALL_MESSAGE_NEW 是一个呼叫中的新的消息到来，比如 ring trying 都算，所以在接受到后必须判断
                    该消息类型，EXOSIP_MESSAGE_NEW 而是表示不是呼叫内的消息到来。
                    该解释有不妥地方，仅供参考。
            */
            printf(" EXOSIP_CALL_MESSAGE_NEW\n");
            if (MSG_IS_INFO(je->request) ) //如果传输的是 INFO 方法
            {
                eXosip_lock ();
                i = eXosip_call_build_answer (je->tid, 200, &answer);
                if (i == 0)
                {
                    eXosip_call_send_answer (je->tid, 200, answer);
                }
                eXosip_unlock ();
                {
                    osip_body_t *body;
                    osip_message_get_body (je->request, 0, &body);
                    printf ("the body is %s\n", body->body);
                }
            }
            break;
        default:
            printf ("Could not parse the msg!\n");
        }
    }
}
```

pjsip 开源库

pjsip 简介

PJSIP 是一个开放源代码的 SIP 协议栈，它支持多种 SIP 的扩展功能 。它的实现是为了能在**嵌入式设备上高效实现 SIP/VOIP**。

PJSIP 是一个包含了 SIP、SDP、RTP、RTCP、STUN、ICE 等协议实现的开源库。它把基于信令协议 SIP 的多媒体框架和 NAT 穿透功能整合成高层次、抽象的多媒体通信 API，这套 API 能够很容易的一直到各种构架中，不管是桌面计算机，还是嵌入式设备等。

PJSIP 是一个开源的 SIP 协议库，它实现了 SIP、SDP、RTP、STUN、TURN 和 ICE。PJSIP 作为基于 SIP 的一个多媒体通信框架提供了非常清晰的 API，以及 NAT 穿越的功能。PJSIP 具有非常好的移植性，几乎支持现今所有系统：从桌面系统、嵌入式系统到智能手机。

PJSIP 同时**支持语音、视频、状态呈现和即时通讯**。PJSIP 具有非常完善的文档，对开发者非常友好。

PJSIP 由 Benny Prijono、Perry Ismangil 在 2005 年创建，之后不久，Nanang Izzuddin、Sauw Ming 加入开发团队。2006 年成立 Teluu Ltd.，成为开发和维护 PJSIP 的公司。PJSIP 采用双 License：GPLv2 以及商业许可证，开发者可以根据需要选择不同的 License。

PJSIP 包括：

- PJSIP - Open Source SIP Stack[开源的 SIP 协议栈]
- PJMEDIA - Open Source Media Stack[开源的媒体栈]
- PJNATH - Open Source NAT Traversal Helper Library[开源的 NAT-T 辅助库]
- PJLIB-UTIL - Auxiliary Library[辅助工具库]
- PJLIB - Ultra Portable Base Framework Library[基础框架库]

PJSIP 优点

- 高度的可移殖性

只需简单的编译一次，它能够在多种平台上运行（所有 Windows 系统列, Windows Mobile, Linux, 所有 Unix 系列, MacOS X, RTEMS, Symbian OS, 等等）。

- 极小的内存需求

官方宣称编译后的库，完全实现 SIP 的功能**只需要 150K 的内存空间**，这使得 PJISPi 不仅仅是嵌入开发的理想平台，并且实用于那些内存运行于极小内存平台的应用，这也意味着极小的用户下载时间。

- 高效的性能

这意味着极小的 CPU 运算需求下能同时实现更多的通话。

支持多种 SIP 功能及扩展功能

多种 SIP 功能和扩展功能，例如多人会话，事件驱动框架，会话控制（presence），即时信息，电话传输，等等在库文件里得以实现。

- 丰富的文档资料

PJSIP 开发人员提供了大量的极有价值的文档资料供大家使用。

免费的开源多媒体通信库，实现了基于标准的协议
主要实现的协议 SIP，SDP，RTP，STUN，TURN 和 ICE
信令协议（SIP）与多媒体框架和 NAT 穿越功能集成到高级多媒体通信 API 中
STUN：（Simple Traversal of UDP over NATs，NAT 的 UDP 简单穿越）是一种网络协议
TURN：(使用中继穿越 NAT)的协议，它允许一台主机使用中继服务与对端进行报文传输
ICE：Interactive Connectivity Establishment,即交互式连接建立

PJSIP 开源库详解

PJSIP 是一个包含了 SIP、SDP、RTP、RTCP、STUN、ICE 等协议实现的开源库。它把基于信令协议 SIP 的多媒体框架和 NAT 穿透功能整合成高层次、抽象的多媒体通信 API，这套 API 能够很容易的一直到各种构架中，不管是桌面计算机，还是嵌入式设备等。

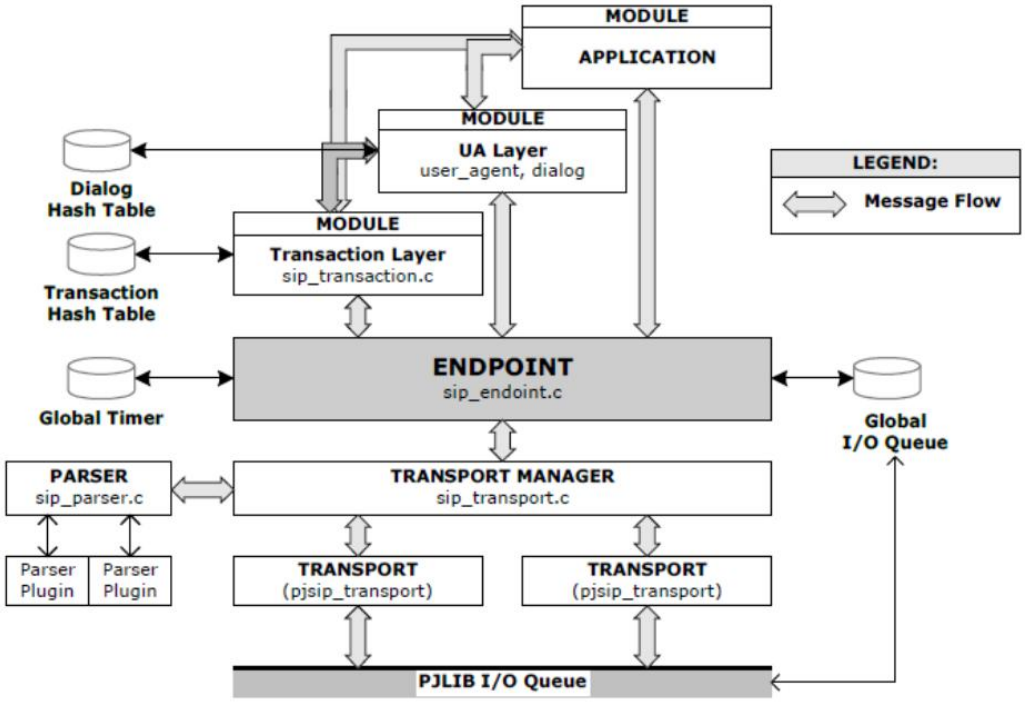
PJSIP 的组织架构介绍

PJSIP 开源库中主要包含两部分，一部分是 SIP 协议栈（SIP 消息处理），另一部分媒体流处理模块（RTP 包的处理）。

SIP 协议栈模块

SIP 协议栈这个模块，开源库由底层往上层做了各个层次的封装。

- **pjlib 是最底层的，最基础的库**，它实现的是平台抽象与框架（数据结构、内存分配、文件 I/O、线程、线程同步等），是 SIP 协议栈的基石。其他所有与 SIP 相关的模块都是基于 PJLIB 来实现的。
- **pjlib-util** 则是封装了一些常用的算法，例如 MD5、CRC32 等，除此之外封装了一些涉及到字符串、文件格式解析操作的 API，例如 XML 格式解析。
- **pjsip-core** 则是 SIP 协议栈的核心，在该库中，包含了三个非常重要的模块，分别是 SIP endpoint、SIP **transaction** module、SIP **dialog** module、**transport** layer。后续会着重介绍前三个模块。
- pjsip-simple 则是 SIP 事件与出席框架，如果你程序中要实现出席制定，则该库是必备的。
- **pjsip-ua** 是 INVITE 会话的高层抽象，使用该套 API 比较容易创建一个 SIP 会话。此外该库还实现了 SIP client 的注册 API。
- **pjsua** 是 PJSIP 开源库中能够使用到的最高层次抽象 API，该库是基于 pjsip-ua 及以下库做了高层次的分装。



基于上图，SIP **endpoint** 是整个协议栈模块的核心，一般来说，一个进程内只能创建一个 SIP endpoint。因此 SIP endpoint 是基于 PJSIP 开发的应用程序内所有 SIP objects 的管理与拥有者。根据官方文档的描述，SIP endpoint 主要承担了一下角色：

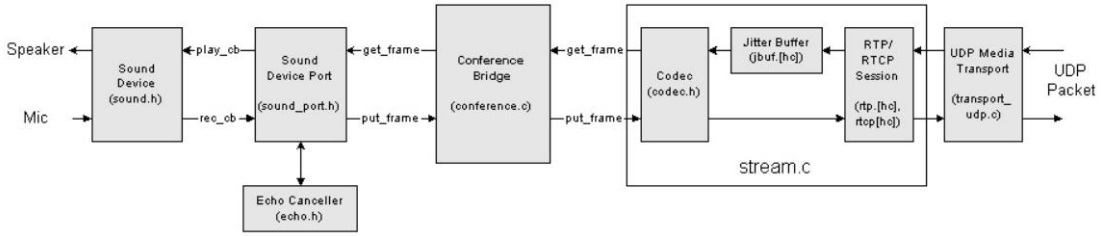
- 为所有的 SIP 对象管理内存池的分配与释放（在基于 pjsip 的应用程序中，动态内存的分配是在内存池基础上进行的）
- 接收来自于传输层的消息，并将消息分配到上层，这里的上层指的是图中的 SIP transaction module、SIP dialog module、application module。优先级顺序是 SIP transaction module > SIP dialog module > application module。如果消息被上层接收处理，则消息由接收的那层继续往上传递处理。例如，SIP endpoint 收到的 SIP 消息，会先较低给 SIP transaction module，如果 SIP transaction module 在 transaction hash table 中找到消息所对应的 transaction，则 SIP transaction module 在完成相应的处理后，会将消息尝试继续往上传递；如果 SIP transaction module 在 transaction hash table 中没有找到消息所对应的 transaction，则该 SIP 消息由 SIP endpoint 继续往上传递。当 SIP 消息不能被上层所有 module 处理，则该消息由 SIP endpoint 来做默认处理。
- SIP endpoint 负责管理模块（module），module 在这里是对该库进行扩展的一种方法，在代码里代表的是一个结构体数据，上面会定义 module 名字、优先级、以及一些函数指针。开发者可以自己定义一些优先级高于 SIP transaction module 的 module 来截获对 SIP 消息的处理。
- 它为所有对象和分发事件提供单个轮询函数。
  transport layer 是 sip 消息的接收与发送模块，目前支持 TCP、UDP、TLS 三种方式。

媒体流处理模块

该模块主要包含两部分，一是 media transport，负责接收媒体流；二是媒体端口（media port）框架，该框架实现了各种媒体端口，每一个 media port 上定义各种操作（创建、销毁、get/put 等），常用媒体端口有：File writer（记录媒体文件），File player（播放媒体文件）、stream port 、conference port（可以实现多方通话）、master port 等。

media transport 目前支持 **RTP（UDP）**、SRTP（加密）、ICE（NAT 穿透）

当 SIP 会话建立后，底层的媒体流处理流程可参考下图：



在上图，从左往右，分别是 media transport、stream port、conference port、sound device port、sound device 。前四个需要自己在程序里创建，最后一个 sound device 是与 sound device port 相关联的，创建 sound device port 的时候便会关联到 sound device。媒体流数据是通过各个 media port 操作进行传递的，在上图中驱动媒体流由左往右流动的“驱动器是”sound device port，该端口是通过 sound device 的硬件驱动不停向与它连接的 media port 实施/get or put fram 操作，从而媒体流得以流动。

在媒体流处理模块中，像 sound device port 的端口，我们成为主动型端口或者驱动型端口。媒体流处理模块中另外一个主动型端口就是 master port。

在上图中最重要的是 stream port ，如果你使用了 pjmedia 库，则必少不了 stream port 。在 stream port 中，从接收 RTP 包的角度讲，RTP 包会被做一下处理：

decode RTP into frame ---> put each frame into jitter buffer ---> decode frame into samples with codec

从发送 RTP 包的角度讲，除了包含媒体流数据的 RTP 包外，还会存在 keep alive UDP pakcet。

stream port 与 media transport 之间的连接是通过 attach 和 detach 操作完成的，该操作是在创建 stream port 执行。除此之外，为了能正常接收 RTP 流，我们需要为 media transport 提供轮训机制，通常我们使用 SIP endpoint 的 I\O queue 即可，这个是在创建 media transport 时通过参数设置的。

注：* jitter buffer 是一种缓冲技术，主要用于音频视频流的缓冲处理。
学习网站：http://www.hellotongtong.com/
福优学苑：【 QQ 咨询：3212001984】
加微信可以提供远程服务,微信服务二维码(13661137824)：

下载编译 PJSIP windows10 平台

平台 windows10+VS2015
pjsip 下载地址下载最近压缩包，解压
官方介绍文档 介绍需要依赖的文件配置
我下载的是最新版本 pjproject-2.10
https://www.pjsip.org/docs/book-latest/html/index.html
https://www.pjsip.org/download.htm

pjproject-2.7.1 目录介绍

- lib: [PJPROJECT 的 lib 库]
- pjlib:[基础框架库]
- pjlib-util:[辅助工具库]
- pjmedia:[开源的媒体栈]
- pjnath:[开源的 NAT-T 辅助库]
- pjsip:[开源的 SIP 协议栈]
- pjsip-apps[demo]

PJSIP 编译前准备

安装 VS2015 官网其他版本可能会有问题
安装 SDK： DirectX SDK、Platform SDK
安装视频支持 SDK：DirectShow SDK，包含在 Windows SDK、SDL、libyuv、OpenH264

- **创建 pjlib/include/pj/config_site.h，配置 config_site.h**

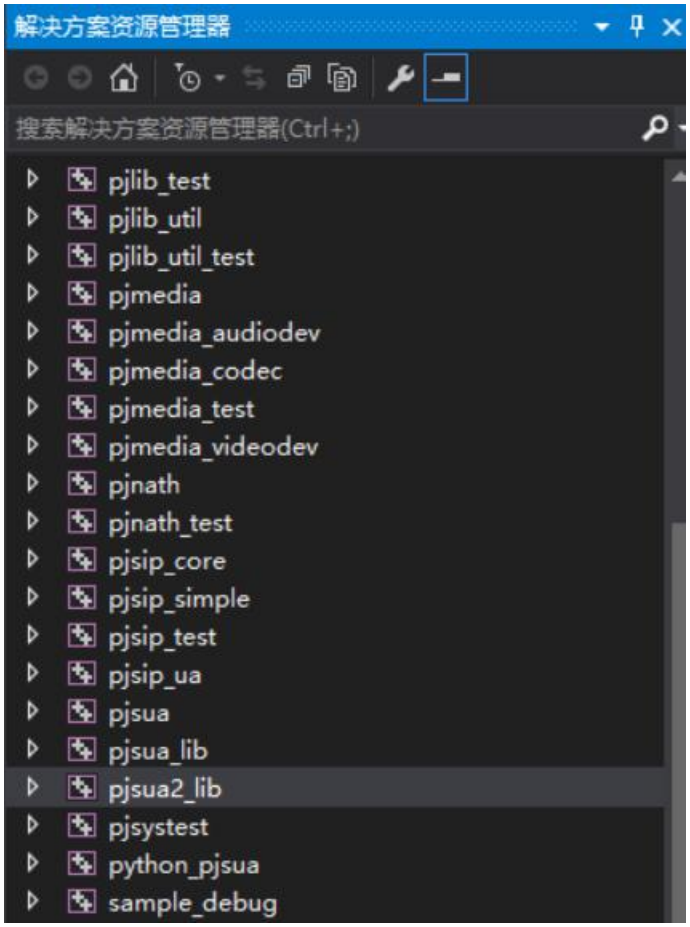
```
#include <pj/config_site_sample.h> //配置了参数
/*支持视频的参数*/
#define PJMEDIA_HAS_VIDEO 0
#define PJMEDIA_HAS_OPENH264_CODEC 0
#define PJMEDIA_HAS_LIBYUV 1
#define PJMEDIA_VIDEO_DEV_HAS_SDL 0
```

#define PJMEDIA_VIDEO_DEV_HAS_DSHOW 0

/*ffmpeg 支持*/
#define PJMEDIA_HAS_FFMPEG 0

Windows 编译

打开 pjproject-2.10 项目中的 pjproject-vs14.sln



设置 pjsua 设置为启动项目 设置为 debug 和 win32 平台
编译：可能会报错，添加依赖库
编译成功后，在 pjsip-apps/bin 生成 pjsua 引用 lib 库生成在 lib 目录下面
按照同样的方法 编译 sample-debug 可以用来调试案例

Linux 编译

sudo apt-get install libsdl2-dev
sudo apt-get install libsdl2-image-dev

sudo apt-get install libsdl2-mixer-dev
sudo apt-get install libsdl2-ttf-dev

步骤：
chmod -R 777 pjproject-2.10
cd pjproject-2.10
./configure
make dep
make
make install

编译自己的应用

引入这些库 **pjlib、pjlib-UTIL、pjnath、pjmedia、PJSIP、lib**
使用头文件 (官方介绍的需要引入的头文件)

```
#include <pjlib.h>
#include <pjlib-util.h>
#include <pjnath.h>
#include <pjsip.h>
#include <pjsip_ua.h>
#include <pjsip_simple.h>
#include <pjsua-lib / pjsua.h>
#include <pjmedia.h>
#include <pjmedia-codec.h>
```

在项目设置中声明 PJ_WIN32 = 1 宏
链接系统特定的库，如：wsock32.lib，ws2_32.lib，ole32.lib

- 头文件：
- 库文件：

- **代码：pjsua_demo.c**

Windows 需要引入 socket 库： ws2_32.lib

```
#include <pjsua-lib/pjsua.h>

#define THIS_FILE   "APP"

#define SIP_DOMAIN      "example.com"
```

```
#define SIP_USER    "alice"
#define SIP_PASSWD      "secret"


/* Callback called by the library upon receiving incoming call */
static void on_incoming_call(pjsua_acc_id acc_id, pjsua_call_id call_id,
                             pjsip_rx_data *rdata)
{
    pjsua_call_info ci;

    PJ_UNUSED_ARG(acc_id);
    PJ_UNUSED_ARG(rdata);

    pjsua_call_get_info(call_id, &ci);

    PJ_LOG(3,(THIS_FILE, "Incoming call from %.*s!!",
                  (int)ci.remote_info.slen,
                  ci.remote_info.ptr));

    /* Automatically answer incoming calls with 200/OK */
    pjsua_call_answer(call_id, 200, NULL, NULL);
}

/* Callback called by the library when call's state has changed */
static void on_call_state(pjsua_call_id call_id, pjsip_event *e)
{
    pjsua_call_info ci;

    PJ_UNUSED_ARG(e);

    pjsua_call_get_info(call_id, &ci);
    PJ_LOG(3,(THIS_FILE, "Call %d state=%.*s", call_id,
                  (int)ci.state_text.slen,
                  ci.state_text.ptr));
}

/* Callback called by the library when call's media state has changed */
static void on_call_media_state(pjsua_call_id call_id)
{
    pjsua_call_info ci;

    pjsua_call_get_info(call_id, &ci);

    if (ci.media_status == PJSUA_CALL_MEDIA_ACTIVE) {
```

```
        // When media is active, connect call to sound device.
        pjsua_conf_connect(ci.conf_slot, 0);
        pjsua_conf_connect(0, ci.conf_slot);
        }
}

/* Display error and exit application */
static void error_exit(const char *title, pj_status_t status)
{
        pjsua_perror(THIS_FILE, title, status);
        pjsua_destroy();
        exit(1);
}

/*
 * main()
 *
 * argv[1] may contain URL to call.
 */
int main(int argc, char *argv[])
{
        pjsua_acc_id acc_id;
        pj_status_t status;

        /* Create pjsua first! */
        status = pjsua_create();
        if (status != PJ_SUCCESS) error_exit("Error in pjsua_create()", status);

        /* If argument is specified, it's got to be a valid SIP URL */
        if (argc > 1) {
        status = pjsua_verify_url(argv[1]);
        if (status != PJ_SUCCESS) error_exit("Invalid URL in argv", status);
        }

        /* Init pjsua */
        {
        pjsua_config cfg;
        pjsua_logging_config log_cfg;

        pjsua_config_default(&cfg);
        cfg.cb.on_incoming_call = &on_incoming_call;
        cfg.cb.on_call_media_state = &on_call_media_state;
        cfg.cb.on_call_state = &on_call_state;
```

```
pjsua_logging_config_default(&log_cfg);
log_cfg.console_level = 4;

status = pjsua_init(&cfg, &log_cfg, NULL);
if (status != PJ_SUCCESS) error_exit("Error in pjsua_init()", status);
}

/* Add UDP transport. */
{
pjsua_transport_config cfg;

pjsua_transport_config_default(&cfg);
cfg.port = 5060;
status = pjsua_transport_create(PJSIP_TRANSPORT_UDP, &cfg, NULL);
if (status != PJ_SUCCESS) error_exit("Error creating transport", status);
}

/* Initialization is done, now start pjsua */
status = pjsua_start();
if (status != PJ_SUCCESS) error_exit("Error starting pjsua", status);

/* Register to SIP server by creating SIP account. */
{
pjsua_acc_config cfg;

pjsua_acc_config_default(&cfg);
cfg.id = pj_str("sip:" SIP_USER "@" SIP_DOMAIN);
cfg.reg_uri = pj_str("sip:" SIP_DOMAIN);
cfg.cred_count = 1;
cfg.cred_info[0].realm = pj_str(SIP_DOMAIN);
cfg.cred_info[0].scheme = pj_str("digest");
cfg.cred_info[0].username = pj_str(SIP_USER);
cfg.cred_info[0].data_type = PJSIP_CRED_DATA_PLAIN_PASSWD;
cfg.cred_info[0].data = pj_str(SIP_PASSWD);

status = pjsua_acc_add(&cfg, PJ_TRUE, &acc_id);
if (status != PJ_SUCCESS) error_exit("Error adding account", status);
}

/* If URL is specified, make call to the URL. */
if (argc > 1) {
pj_str_t uri = pj_str(argv[1]);
status = pjsua_call_make_call(acc_id, &uri, 0, NULL, NULL, NULL);
if (status != PJ_SUCCESS) error_exit("Error making call", status);
```

```
    }

    /* Wait until user press "q" to quit. */
    for (;;) {
    char option[10];

    puts("Press 'h' to hangup all calls, 'q' to quit");
    if (fgets(option, sizeof(option), stdin) == NULL) {
        puts("EOF while reading stdin, will quit now..");
        break;
    }

    if (option[0] == 'q')
        break;

    if (option[0] == 'h')
        pjsua_call_hangup_all();
    }

    /* Destroy pjsua */
    pjsua_destroy();

    return 0;
}
```

如何调试

最好两台机器：
或者修改端口号：

5060.exe:直接启动
5070.exe:需要带 url: alice@192.168.1.9:5060