Osip 协议源代码框架详解

Prepared by	Mao minghua	Date	2009.09.25
Reviewed by		Date	
Approved by		Date	

Revision History

Version	Author	Reviewed By	Comments	Issued Date
0.1	Mao minghua		描述 osip 协议栈的源代码框架	

目录

Osip 源代码框架详解

1 符号及缩写

缩写	英文全称	中文名称
ICT	Invite Client Transaction	Invite 类型的客户端事务
IST	Invite Server Transaction	Invite 类型的服务端事务
NICT	Not Invite Client Transaction	非 Invite 类型的客户端事务
NIST	Not Invite Server Transaction	非 Invite 类型的服务端事务
IMS	IP Multimedia Subsystem	IP 多媒体子系统
PSVT	Packet service video telephony	分组域可视电话
SIP	Session Initiation Protocol	会话初始协议
UDP	User Datagram Protocol	用户数据报协议
URL	Uniform Resource Locator	统一资源定位器

2 整体描述

开源代码的 osip 协议栈分为两个源代码包，整个协议栈采用 lib 库的形式，在内部没有使用到任务，采取与 TCP/IP 协议栈一样的策略，所以在使用上需要上层管理任务直接调用 lib 库提供的接口。因为在 Lib 库内部没有使用到像定时器、发送队列等的任务，而同时需要使用到定时器，所以在 lib 库的内部采用轮训遍历的方式不停的检查是否有定时器超时，这在某种程度上会浪费 CPU 的允许时间。同时整个 lib 库实现了对 call, notify 等的管理，为了实现重入，在应用启用多线程的条件下，内部启用的信号量和锁的使用，在下面的分析中不涉及到信号量和锁机制。

Lib 库按照 sip 协议栈的层次关系分为两个 lib 包，底层的 osip lib 包实现对单个请求、应答、ACK 的处理，包括 message 的解析、拼装、内容 set 和 get，单个请求形成的 transaction 相关操作以及通信两端形成的一个 dialog 的操作。

Lib 库上层的 exosip lib 在底层 osip lib 库的实现基础上，实现对 sip 协议整理逻辑上的管理。Exosip 主要关注的是 sip 协议的业务流程，包括 call 的整体管理，notify 的整体管理，publish 的管理，register 的管理，option 的管理，refer 的管理和 subscription 的管理，其中最主要的为 call 和 register 的管理，这两个为 sip 协议栈必须实现的部分，另几个功能为 sip 协议栈扩展部分。从这几个业务的管理流程出发，在业务的底层它们会使用到相似的一些功能，如注册的认证，发送 message，接收 message，每个请求和应答形成的 transaction，多个

transaction 组合而成的 dialog。

在 message 的处理方面，可以分为两类，一类为发送的 message，因为是主动发送，所以上层管理层知道是什么类型的 message，lib 库直接提供各类接口供使用。一类为接收到的 message，因为不知道是哪种类型，所以需要根据解析出来的 message 的信息来进行处理，这部分的处理在 udp.c 文件中。

整个 lib 库的初始化在 exOsip 中介绍。

3 Osip 包的源代码框架解析

在 osip 源代码包中最主要的包括了 message 的相关操作，其中最重要的为 message 的解析，即从获取到的一个 message 中解析生成一个能够被代码直接处理的 message 数据结构——osip_message_t。与 message 结构相关的操作包括根据 message 数据结构的信息安装 sip 协议规范组装成一个 message 字符串；message 结构的初始化和释放；message 结构的拷贝操作；以及从 message 结构中获取各种已经解析的成员变量的值和设置各个成员变量的值。在 message 的解析部分，除了 message 的头之外，还包括了 body 的解析，涉及到 sdp 协议，包括对每个 sdp 字段的解析。

在 osip 源代码包中，设计了一个与同一个请求相关的所有 message 的集合——transaction，在发送或接收到一个新的请求的时候就会生成一个 transaction，其中 ACK 和 CANCEL 请求是比较特殊的，对于非 2xx 应答的 ACK 和初始 INVITE 请求是属于同一个 transaction 的，而对于 2xx 的请求是属于单独的 transaction 的，所以其重传操作由 UAC 来控制，而不在 INVITE 的 transaction 内部进行控制。CANCEL 的请求除了本身建立一个 transaction 外，根据协议它还会去匹配要 CANCEL 掉的请求的 transaction，如果匹配成功会 CANCEL 掉相应的 transaction。

在 osip 包中同样设计了 dialog 相关操作，包括 dialog 的建立，dialog 信息的保存，dialog 的匹配及删除等操作。

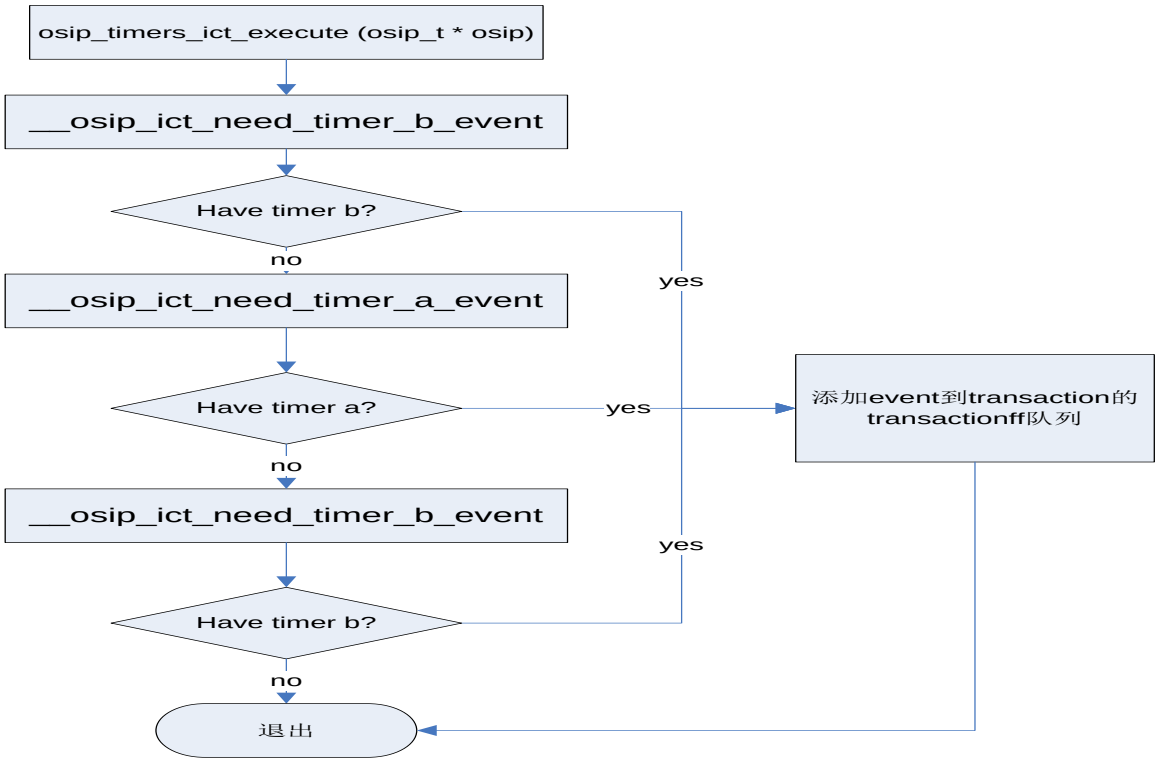
其它方面，包括多线程中使用到的锁和信号量及信号，内部使用到的链表，用于事件的队列(需要先进先出策略)，一些平台无关的封装，定时器以及常量等的定义。这部分比较简单，而且都是最底层函数，直接封装了系统调用层。

3.1 osip 的 transaction 的 event 的产生

transaction 的状态变化是由事件来驱动的，当 transaction 上有事件产生时，根据事件的类型和当前 transaction 的状态来处理该 event。

Transaction 上的事件分为两类：一为定时器事件，在设定的定时器超时时会产生相应的定时器事件；另一类为事件驱动事件，如发送一个请求、应答或接收到一个请求、应答，发送一个 ACK 和接收到一个 ACK，即是由报文产生的事件。

3.1.1 定时器事件的产生过程



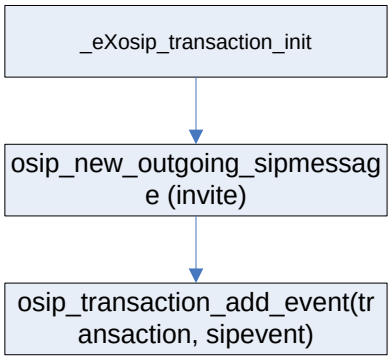
ICT、IST、NICT 和 NIST 的定时器的事件产生流程都一样，对于每一个 transaction，其定时器是有顺序的，ICT 流程中 TIMEOUT_B 的优先级最高，TIMEOUT_B 定时器触发后，会触发 kill transaction 的操作。当 transactionff 队列中有未处理的事件时，不处理定时器，直接返回，所以在 transactionff 队列中总的事件的数量是不多的。

所有的定时器函数调用底层同一个定时器检查函数__osip_transaction_need_timer_x_event。该函数会先检查该定时器是否启动，判断条件为(timer->tv_sec == -1)，如果启动，检查当前时间是否超过定时器中设定的时间，如果是，则产生新的定时器事件。

因为定时器没有一个单独的任务，所以是采样轮训的方式检查是否有新的定时器事件产生，而不是根据系统时钟中断进行检测，因此会比较占用系统资源。

定时器的启动和修改使用接口 osip_gettimeofday 和 add_gettimeofday。只需要设定定时器的超时时间，即设定了一个新的定时器。取消一个定时器，只需要修改定时器的 timer->tv_sev 为-1。

3.1.2 报文触发的事件



包括一个新的 invite、response、ack 的发送或接收，除了对非 2xx 的应答 ack 外，其他的请求和应答都会产生一个新的 transaction，并且产生一个新的 sipevent 事件。

3.2 osip 的 transaction 的 event 处理流程

在 sip 协议栈中为了更快更好的处理 transaction，根据协议栈的描述，划分为四种不同的 transaction，分别为 ICT、IST、NICT 和 NIST。四种不同的 transaction 会有不同的处理流程和状态转换表，以及使用到不同的定时器。

ICT、IST、NICT 和 NIST 的状态转换采样注册函数处理方式，为便于管理和使用注册函数，源码中使用了四个全局变量管理四种不同类型 transaction 的转换表：ict_fsm、ist_fsm、nist_fsm 和 nist_fsm。

osip 结构如下：

```
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 */
    ……
}
```

整体简单处理流程如下图：

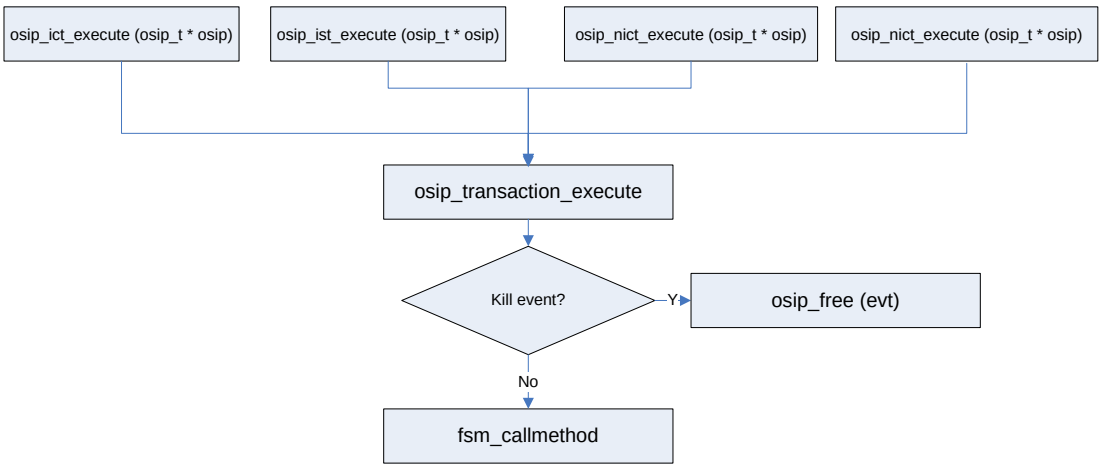


图 3-1：transaction 的 event 处理流程

3.2.1 ICT 的处理流程

如上图，ICT 事件处理时：

1) 检查 osip 管理结构中的 osip_ict_transactions 链表，如果没有链表元素，直接返回 OSIP_SUCCESS

2) 获取链表中元素个数，并保存 transaction 到临时局部数组

3) 遍历所有 transaction，osip_fifo_tryget 顺序获取每个 transaction 中的事件，调用 osip_transaction_execute 处理每个事件，直到所有 transaction 中的所有事件被处理完毕，然后返回。

Osip_transacton_execute 执行时，根据传入的参数 osip_transaction_t * transaction 中的 transactionde 类型获取到状态转移表的全局管理变量 ict_fsm，并且根据 event 的 type 和 transaction 的状态调用 fsm_callmethod——在文件 fsm_misc.c，是状态转移注册函数的总入口——查询找到 event 的处理函数，并调用处理函数进行 event 的处理。

ICT 的相关 event 的注册处理函数在 ict_fsm.c 文件和 ict.c 文件。ICT 使用到了 3 个定时器：

TIMEOUT_A、TIMEOUT_B 和 TIMEOUT_D。

在 client 端发送 Invite 而需要创建新的 ICT 的 transaction 时，TIMEOUT_B 被启动，时长为 64*DEFAULT_T1(DEFAULT_TI 为 500ms)，TIMEOUT_B 为整个 transaction 的生命周期时长，如果超过这个时间，transaction 会被结束。如同传输层使用的是没有传输保证的 UDP，则设置 TIMEOUT_A，TIMEOUT_D 的间隔时间为 DEFAULT_T1 和 64* DEFAULT_T1。如果传输层使用的是面向连接的 TCP 及相关协议，则直接使用 TCP 内部的重传机制，不在 SIP 协议层提供传输的保护机制，所以不启动 TIMEOUT_A 和 TIMEOUT_D。TIMEOUT_A 管理 Invite 的传送，在 Invite 被发送时，启动定时器 TIMEOUT_A，并且在超时时间内还没接收到 response 的时候，重发该 Invite。TIMEOUT_D 用于管理 ACK，当接收到的 response 不是>＝300 时，client 端发送 ACK，当重复接收到该 invite 的 response 时，重发该 ACK，确保 server 端在 kill tranction 前能接收到 ACK。

3.2.2 IST 的处理流程

同 ICT 的处理流程，处理 osip 中的 osip_ist_transaction 链表。IST 的相关 event 的注册处理函数在 ist_fsm.c 文件和 ist.c 文件。

IST 使用了定时器 TIMEOUT_G、TIMEOUT_H 和 TIMEOUT_I。使用方式与 ICTL 类似，详细见协议栈说明。

3.2.3 NICT 的处理流程

同 ICT 的处理流程，处理 osip 中的 osip_nict_transaction 链表。NICT 的相关 event 的注册处理函数在 nict_fsm.c 文件和 nict.c 文件。

NICT 使用了定时器 TIMEOUT_E、TIMEOUT_F 和 TIMEOUT_K。

3.2.4 NIST 的处理流程

同 ICT 的处理流程，处理 osip 中的 osip_nist_transaction 链表。NIST 的相关 event 的注册处

理函数在 nist_fsm.c 文件和 nist.c 文件。

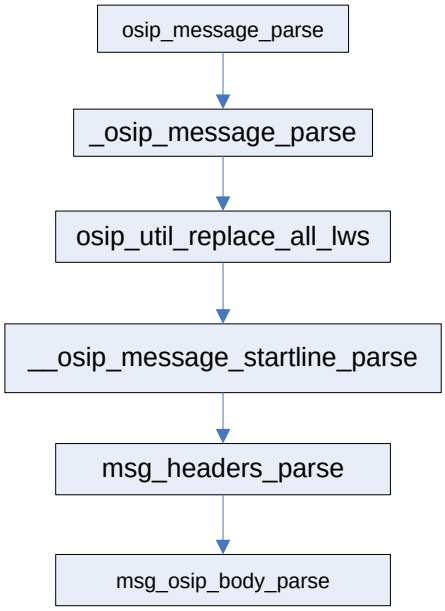
NIST 使用了定时器 TIMEOUT_J。

3.3 Osip 报文的解析

3.3.1 sip 协议报文的解析整理流程

当接收到一个 message 的时候，需要解析该 message，生成一个代码能够处理的数据结构，该结构定义为 struct osip_message，该结构定义的一个 message 的全部相关信息，这些信息主要是供 transaction 和 dialog 及 dialog 的更上一层如 call，notify 等的使用。

对一个 message 的解析流程如下图所示：



在接收到一个 message 时，调用函数 osip_message_parse 进行 message 的解析。首先调用函数 osip_util_replace_all_lws 替换掉 message 中的连续出现的‘\r\n\t’、‘\r\t’、‘\n\t’、‘\r\n ’、‘\r ’、‘\n ’为空格，message 是以‘\0’为结束标志的，message 的 headers 和 body 之间的分界是以’\r\n\r\n’为标志的，替换只替换到’\r\n\r\n’为止，即只替换 headers 部分出现的\t、\r、\n。由于 sip 协议栈规定，每个 headers 都是起新行，而且新行的头一个字符不为空格或\t，所以两个 header 之间的\r\n 不会被替换掉，替换的只是一个允许 multi 合并项的 header 的内部多个值之间的“\r\n\t”或“\r\n ”。

举例如下：有两个 header，其中 Subject 只允许单个值出现， Route 允许有多个值出现，而且允许分行，但是分行必须以空格或\t 开头，而 Subject 和 Route 行必需顶格开始，前面是没有空格或\t 的，osip_util_replace_all_lws 函数将 Route header value 中的两行间的\r\n\t 转化为空格，即在逻辑上就成为一行了。

Subject: Lunch
Route: <sip:alice@atlanta.com>, <sip:bob@biloxi.com>
<sip:carol@chicago.com>

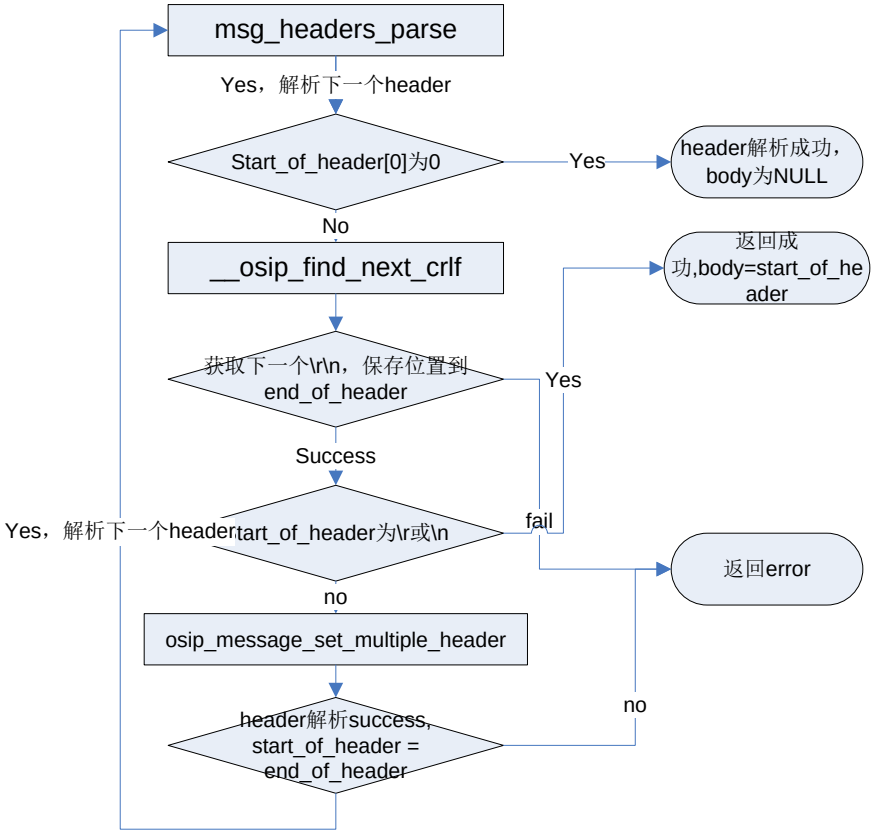
一个 message 由三部分组成，首先是 message 的 startline 部分，该行指明这是一个 sip 的 message，包括 sip 标志，请求或应答说明，状态值，然后以\r\n 做为和 headers 的分隔符。该\r\n 不会被 osip_util_replace_all_lws 替换为空格，如请求的 INVITE sip:bob@biloxi.com SIP/2.0 或应答的 SIP/2.0 200 OK，在三个属性之间有且仅有一个空格。起始行的解析由__osip_message_startline_parse 进行解析，解析得到 message 的类型，message 的 sipversion 以及 message 的 status_code，当 status_code 为初始化值 0 时，该 message 为一个请求，否则为应答。请求的 startline 由__osip_message_startline_parsereq 进行解析，得到请求的 request_uri;应答的 startline 由__osip_message_startline_parseresp 进行解析。Startline 部分的解析是严格安装出现的三个属性的顺序进行解析的，并将解析结果保存在 osip_message 的结构成员变量中。

然后解析 messge 的 headers 部分，调用函数 msg_headers_parse。说明见 osip 的 header 报文头解析。

如果 message 中在 headers 之后不是结束符'\0'，则继续解析 message 的负载部分，调用函数 msg_osip_body_parse 进行解析。Message 的 body 解析首先查询 headers 头解析中保存

的 content——即 body－－的属性：content_type，如果 content_type 中的 type 不为 multipart，即不支持多种 mime 方式的 content，说明 body 中就一个编码方式，直接将整个 body 解析为一个内容；如果 type 为 multitype，说明有多个编码方式的 body 组合在一起形成一个整体的 body，则以”--”为分隔符解析 body，将 body 分为多个 mime 编码方式的字符串，每个解析后的 body 内容保存在 osip_message 结构中的 bodies 结构成员中。

3.3.2 Osip 报文头的解析



在解析 message 的 header 的时候，因为前面的 osip_util_replace_all_lws 已经转化了单个 header 内部出现的\r、\n 和\t 为空格，所以每个 header 之间可以使用\r\n 做为分隔符进行分隔。如果字符串开头 start_of_header 已经到达结束符”\0”，则全部 header 解析完毕，返回成功；调用__osip_find_next_crlf 找到这个 header 的结束字符并保存在 end_of_header 中；如果 start_of_header 为\r 或\n，则已经解析到\r\n\r\n 即 headers 的结束字符串，则返回成功，并且保存 start_of_header 到 body 中，即 body 是从\r\n 字符串开始解析的，所以在 body 解析时，需要跳过\r\n 及之后的空格部分；根据 header 内部分隔符“：”，取出 header 的 hname 和 hvalue，其中 hvalue 在某些 hname 的情况下是允许为空的，然后调用

osip_message_set_multiple_header 来解析该 header 的 hvalue 字符串；解析成功后，置 start_of_header 为已经解析完的 header 的 end_of_header，开始解析下一个 header。

在 osip_message_set_multiple_header 中，将 headers 分为两类，一类如上面例子中的 Subject，只允许一个值，则直接调用 osip_message_set__header 进行解析；一类如上面例子中的 Router，允许多个值，根据 sip 协议，每个值之间以“,”进行分隔，所以需要查询整个 hvalue 字符串，根据”,”将 hvalue 分隔成多个值，每个值调用 osip_message_set__header 进行解析并保存解析结果到 osip_message 的数据成员变量中。因为 hvalue 允许使用引号将值引起来，所以需要特别处理“,”是否出现在引号内部的问题。只有在引号外部的“,”才是 header 值的分隔符，而内部的“,”只是一个 header 值的一部分。

osip 源码中 osip_message_set__header 对于 message headers 的解析采用注册函数的方式实现，采用这种方式能够在后继版本很方便的进行新的 header 的添加，并且不会影响到整个源代码的框架流程。

Osip_parser_cfg.c 文件中定义了 header 头解析所使用到的全局管理变量：static __osip_message_config_t pconfig[NUMBER_OF_HEADERS];

__osip_message_config_t 的结构定义如下：

```
typedef struct ___osip_message_config_t
{
   char *hname;
   int (*setheader) (osip_message_t *, const char *);
   int ignored_when_invalid;
}__osip_message_config_t;
```

hname 为 sip 协议定义的头字段的字符串，这些字符串定义在 osip_const.h 文件中；函数指针 setheader 为该协议 header 的对应的解析函数；ignored_when_invalid 为是否忽略该 header 解析错误的标志，该标志值为 1 时，在解析该协议 header 发送错误时，忽略该错误，除 sip 协议规定的几个必要 header 之外，其他头应该采用忽略方式。

为了更快的根据 header 的 hname，找到对应的 setheader 解析函数，采用了 hash 表的查询方式，根据 hname 生成一个 hash 值，并且需要保证没有两个不同的 hname 对应到同一个 hash 值中，以提高查询的速度。调用__osip_message_is_known_header (hname)获取到在数组中的 index，调用__osip_message_call_method (my_index, sip, hvalue)解析协议 header，并且解

析的结果保存在结构 osip_message_t * dest,中。

每一个 header 都包含几个通用的操作：header 字符串的解析函数，即上段讲到的 osip_message_set_xxx 解析函数；header 解析后的结构的获取函数，osip_message_get_xxx 函数；根据 header 解析后的结构生成字符串的函数：osip_xxx_str；header 解析后的结构的 copy 函数 osip_xxx_clone；header 解析后的结构的是否函数：osip_xxx_free；以及 header 解析结构的初始化函数：osip_xxx_init。

对每个 header 的几个相关操作最终目的是提供协议的整个 header 的整体操作，包括 osip_message_init，osip_message_free，osip_message_clone 和 osip_message_parse。

3.3.3 uri 的解析

绝大部分的 header 的解析都是相识的，只有其中有参数的部分的 header 的解析会比较复杂，最主要的有 from、to、contact 等，因为除了本身就有参数之外，其值中的 request_uri 本身也可以包含有参数，而这两种参数之间是有区别的。

Sip 协议栈规定 header 的表示分为 header’s name, header’s value 和 header’s parameter。其中 name 和 value 之间用“：”分隔，value 与 parameter 之间用“；”分隔，parameter 之间也使用“；”相分隔。

在结构定义中 header 的 value 根据具体 header 包含的信息进行结构变量的定义，而如果包含 parameter 则直接定义一个 gen_params 的链表，所有的 parameter 都保存在这个链表中。

如下面 from 的定义，包含有 from 的名称及一个 url，及相关的 parameter：

```
struct osip_from
{
   char *displayname;          /**< Display Name */
   osip_uri_t *url;            /**< url */
   osip_list_t gen_params;     /**< other From parameters */
```

};

对应 parameter 的解析直接调用__osip_generic_param_parseall，该函数解析 header 的单个 hvalue 字符串中包含的所有 parameter，在函数内部会根据“;”将字符串划分为几个 parameter，然后解析每个 parameter，将解析结果保存在 gen_params 链表中。Parameter 的格式为 pname=pvalue 类型，等号两边允许空格。

From、to、contact 以及 via 中间都可能出现 url。url 的解析接口为 osip_uri_parse，输入为 url 的字符串，解析的结构保存在结构 osip_uri_t 之中。url 包含有三部分内容：url 的基本信息，url 的 header 头部分和 url 的参数部分。开始部分与 header 头部分用“?”进行分隔，header 头之间用”&”进行分隔，header 头部分与参数部分用”;”进行分隔，参数之间也使用“;”进行分隔。Header 部分调用函数 osip_uri_parse_headers 进行解析，结果保存在 osip_uri_t 结构中的 url_headers 成员变量中；parameter 部分调用函数 osip_uri_parse_params 进行解析，其结果保存在 osip_uri_t 的 url_params 成员变量中。

在 from、to、contact 等包含 url 的 header 中，如果 url 中包含 parameter，则整个 url 必需使用“<” “>”括起来，以表示一个完整 url 部分。所以解析 from 等 header 时需要检查是否包含”<”字符。

3.3.4 添加一个新的协议 header 字段

1) 需要添加多个一个对该字段进行解析的文件，包含一个 header 常用到的几个基本通用操作，如果该 header 有特殊的地方需要处理，需要增加相关的处理函数，文件名一般定义为 osip_xxx.c 和 osip_xxx.h

2) 需要在 parser_init 中注册新的 header 的解析函数，需要修改 static __osip_message_config_t pconfig[NUMBER_OF_HEADERS] 中的 NUMBER_OF_HEADERS 宏值。

3) 在 osip_const.h 中添加新的 header 的宏定义，osip 的相关的常量宏定义都定义在该文件

4) 在 osip_message.c 文件额 osip_message_init 函数中添加对该 header 相关结构的初始化操作。在 osip_message_free 函数中同样添加对该 header 的相关释放操作，在 osip_message_clone 中添加对该 header 的 clone 相关操作。

5) 在 osip_message_to_str.c 文件中的_osip_message_to_str 函数中添加该 header 转化为 string 的函数注册。

6) 如果该 header 不允许重复多个出现，即不允许 multiple header，则在 osip_message_parse.c 文件的 osip_message_set_multiple_header 函数中添加对该 header 的处理。

7) 在 osip_message.h 的头文件中的 osip_message 结构中添加对该 header 字段的结构。

8) 在 osip_headers.h 文件中添加新的 header 的头文件引用。

3.4 osip 的 transaction 的管理

transaction 的操作主要包括 transaction 的初始化、transaction 的 free、transaction 的匹配、从 transaction 中获取信息和设置 transaction 信息。

根据 sip 协议描述一个 transaction 由 5 个必要部分组成：from、to、topvia、call-id 和 cseq，这 5 个部分一起识别某一个 transaction，如果缺少任何一部分，该 transaction 就会设置失败。

所以对每个部分的设置都会有一个设置函数：__osip_transaction_set_topvia 用于设置 topvia，对于发送端 topvia 为自己的 via，对于接收端 topvia 为将 message 转发到自己的最后一个 sip-proxy 服务器，__osip_transaction_set_from 用于设置 message 的发送端，__osip_transaction_set_to 用于设置 message 的接收端，__osip_transaction_set_call_id 用于设置一个 dialog 的标识值，该值是随机生成的，算法保证很长一段时间内生成的 cal_id 是不相同的，__osip_transaction_set_cseq 用于设置 cseq 值，该值在同一个 dialog 内部是一直保持增长的，即同一个 dialog 的后面的 transaction 的 cseq 会比前面的 transaction 的值大，按照 sip 协议其初始值可以是随机数，代码实现中如果是非 register 请求，从 1 开始，如果是 register 请求的 dialog，从 20 开始。

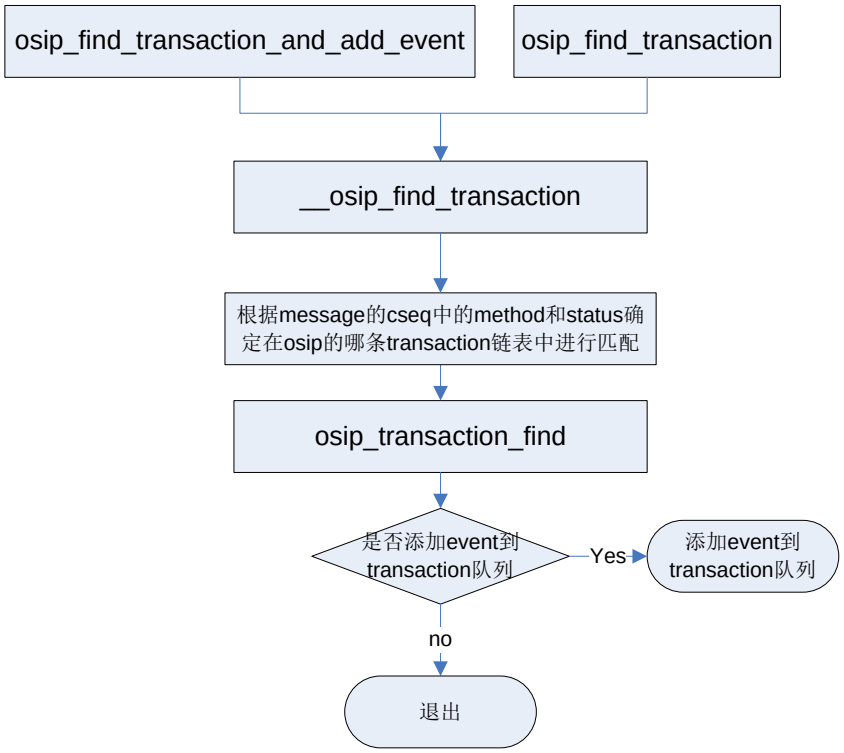
Transaction 的初始化发生在接收到一个新的请求或发送一个请求的时候，该请求以及经过解析成为一个可以直接使用请求信息的结构 osip_message_t。其初始化具体过程如上面所述，在设置完那 5 个部分后，还需要初始化 event 的队列，以及根据 osip_message_t 的 type 初始化使用到的定时器结构，如 ICT 的 ict_context。其它部分的初始化在 exosip 源代码中实现，相关的如 your_instance、in_socket、out_socket 和 record 都是未了方便 exosip 中对 transaction 的管理而设置的。

Transaction 中的 event 的相关操作在如前面所述。

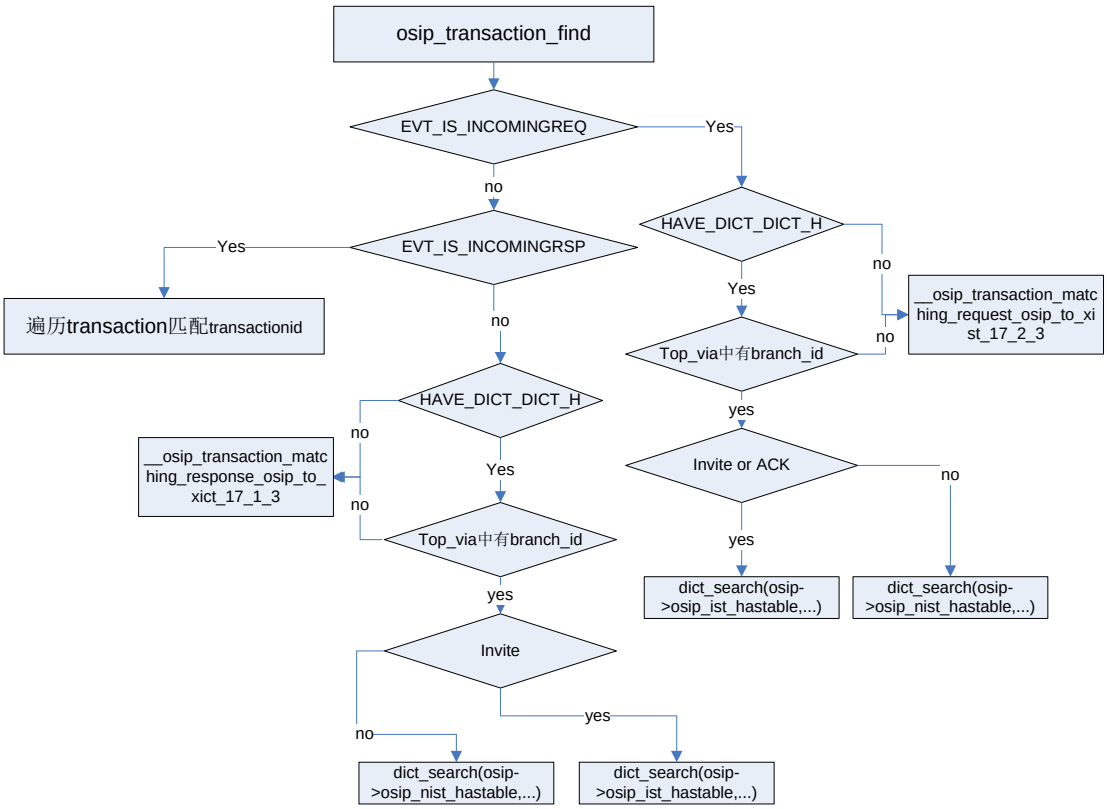
在 transaction 的匹配中，根据 RFC3261 的最新 sip 协议的描述，由 topvia 的 branch_id 是否相同来匹配，如果相同，就是同一个 transaction 的请求和应答及 ACK，为兼容旧版本的 transaction 的匹配规则，同时支持根据 call_id, cseq, from_tag, to_tag 来匹配 transaction。如下图，为便于管理的 transaction，所有的 transaction 保持在 osip_t 结构的四条链表中，按照处理流程的不同分为发送出去的 INVITE 类型请求和应答、其它类型请求和应答，接收到的 INVITE 请求和应答、其它请求和应答。

```
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 */
……
#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
  };
```



在 transaction 的匹配过程中，如果是发出的请求，因为本地的 transaction 都会分配一个不重复的 transaction_id，所以只需要比配 transaction_id 即可；对于 incoming 的 message,如果是 request，则匹配 branch_id 或者为兼容前面版本进行 transaction 的比配，按照协议 RFC3261 的 17-2-3 节的方式进行匹配；如果 incoming 的 message 是 response，则匹配 branch_id 或根据 RFC3261 的 17-1-3 节的规则进行匹配。



3.5 osip 中 dialog 的管理

dialog 的相关的管理操作包括 dialog 的初始化建立过程，dialog 的销毁 free 过程，以及 dialog 的匹配。此外 dialog 中保存了相关的路由信息和 cseq 信息。

Dialog 结构如下，由 call_id、local_tag 和 remote_tag 唯一确定一个 dialog：

```
struct osip_dialog
{
  char *call_id;                    /**< Call-ID*/
  char *local_tag;                  /**< local tag */
  char *remote_tag;                  /**< remote tag */
  osip_list_t route_set;           /**< route set */
```

```
    int local_cseq;                          /**< last local cseq */
    int remote_cseq;                         /**< last remote cseq*/
    osip_to_t *remote_uri;                   /**< remote_uri */
    osip_from_t *local_uri;                  /**< local_uri */
    osip_contact_t *remote_contact_uri;    /**< remote contact_uri */
    int secure;                              /**< use secure transport layer */
    osip_dialog_type_t type;                 /**< type of dialog (CALLEE or CALLER) */
    state_t state;          /**< DIALOG_EARLY || DIALOG_CONFIRMED || DIALOG_CLOSED */
    void *your_instance;                     /**< for application data reference */
  };
```

在 dialog 的初始化时，需要根据是 client 端或 server 端来确定 dialog 结构中的 call_id、local_tag 和 remote_tag 的值。根据是 client 端或 server 端来确定 dialog 的 type，并且设置 dialog 的状态。当做为 client 端，并且是在接收到发出的 request 的 response 时，调用 osip_dialog_init_as_uac 进行初始化 dialog；如果是接收到 server 端发送过来的 request，则调用 osip_dialog_init_as_uac_with_remote_request 进行 dialog 的初始化。如果是 server 端，调用 osip_dialog_init_as_uas 进行初始化 dialog。

在 dialog 的匹配时，当是 client 端时，调用 osip_dialog_match_as_uac 进行匹配。检查接收到的 response 和 dialog 中的 call_id，to_tag 和 from_tag 是否匹配，如果全部匹配，则匹配到了该 dialog。为兼容前面的版本，在 dialog 的 to 或者接收到的 message 的 to header 没有 tag 的情况下，比较 dialog 和 message 的 from_uri, to_uri。如果匹配，则同样匹配的 dialog。

当是 server 端时，调用 osip_dialog_match_as_uas 进行匹配。其匹配方法与 client 端的匹配方法相同。

对 from_tag 和 to_tag 的匹配处理，在 transaction 的匹配过程中同样使用到。

4 Exosip 包的源代码框架解析

在 exosip 源代码包中包含了提供给上层管理软件调用的关于 call、message、option、refer、register、subscription、publish 和 insubscription 的 API，这些 API 的实现都在 ex 开头的 c 文件中。

为这些接口进行服务的函数，包括和 osip lib 库进行通信的部分的实现在以 j 开头的源文件中如 jcall.c，其中 jrequest.c 和 jresponse.c 实现了 request 和 response 的 message 的通用构造实现。

同时 exosip 实现了传输层的四种不同的传输方式供上层的 sip 协议栈进行选择，分别为 extl_dtls.c、extl_tcp.c、extl_tls.c 和 extl_udp.c，它们以注册的方式在 Lib 库启动的时候注册到 lib 库的钩子中。

为了支持多线程间的通信，在两个线程间采用 pipe 的方式进行实现，如果没有使用多线程，这部分源代码在编译时会被屏蔽掉。

对接收到的 message 进行的逻辑处理在文件 udp.c 中，这部分是整个协议栈逻辑比较复杂的地方。需要根据 sip 协议栈的标识描述和代码实现框架进行整理的把握。

4.1 Lib 库的初始化和销毁

整个 sip 的 lib 库有一个总的管理结构 struct exosip_t eXosip，该全局变量在 lib 库被使用之前需要初始化，初始化函数为 exconf.c 文件的 eXosip_init 函数。Exosip_t 结构如下：

```
  struct eXosip_tt
  {
    struct eXtl_protocol *eXtl;
    char transport[10];
    char *user_agent;

    eXosip_call_t *j_calls;        /* my calls             */
#ifndef MINISIZE
    eXosip_subscribe_t *j_subscribes;    /* my friends          */
    eXosip_notify_t *j_notifies;         /* my susbscribers */
#endif
    osip_list_t   j_transactions;

    eXosip_reg_t *j_reg;           /* my registrations */
#ifndef MINISIZE
    eXosip_pub_t *j_pub;           /* my publications   */
#endif

#ifdef OSIP_MT
    void *j_cond;
    void *j_mutexlock;
#endif

    osip_t *j_osip;
    int j_stop_ua;
#ifdef OSIP_MT
```

```
    void *j_thread;
    jpipe_t *j_socketctl;
    jpipe_t *j_socketctl_event;
#endif

    osip_fifo_t *j_events;

    jauthinfo_t *authinfos;

    int keep_alive;
    int learn_port;
#ifndef MINISIZE
    int http_port;
    char http_proxy[256];
    char http_outbound_proxy[256];
    int dontsend_101;
#endif
    int use_rport;
    char ipv4_for_gateway[256];
    char ipv6_for_gateway[256];
#ifndef MINISIZE
    char event_package[256];
#endif
    struct eXosip_dns_cache dns_entries[MAX_EXOSIP_DNS_ENTRY];
    struct eXosip_account_info account_entries[MAX_EXOSIP_ACCOUNT_INFO];
    struct eXosip_http_auth http_auths[MAX_EXOSIP_HTTP_AUTH];

    CbSipCallback cbsipCallback;

    p_access_network_info *p_a_n_i;
    digest_cave_response *cav_v;
  };
```

在该结构中，最重要的几个成员变量如下：

1) int j_stop_ua; 协议栈启动和停止的控制参数，当 j_stop_ua 为 0 时，lib 库启动，为非 0 时，lib 库停止。
2) eXosip_call_t *j_calls; j_calls 用于管理全部的通话，所有的 call 在这个结构中形成一个链表结构。Call 结构如下：

```
  struct eXosip_call_t
  {

    int c_id;
    eXosip_dialog_t *c_dialogs;
```

```
    osip_transaction_t *c_inc_tr;
    osip_transaction_t *c_out_tr;
    int c_retry;                    /* avoid too many unsuccessfull retry */
    void *external_reference;

    eXosip_call_t *next;
    eXosip_call_t *parent;
  };
```

其中的 c_id 为分配的 call_id，c_dialogs 为同一个 call 中的 dialog 的集合。c_inc_tr 和 c_out_tr 为创建该 call 时的初始化的 transaction，一个终端对于一个 call 不是 client 端就是 server 端，所以 c_inc_tr 和 c_out_tr 只可能有一个是有 transaction 的，其中有一个为 NULL。

3) eXosip_reg_t *j_reg; j_reg 管理 sip 的注册服务，所有 register 相关的 transaction 在 j_reg 中形成一个链表。根据 sip 协议，新的一次的注册必需等到前一次主次完成后才能进行，所以同一个 register 不会像 call 一样，同时有多个 transaction 存在。

4) struct eXtl_protocol *eXtl; 为使用的传输层的协议，在 exosip 的初始化中需要设置，在传输层现在了 4 种不同的传输实现方式，它们的实现以一个结构的形式存在，在 exosip_t 初始化时注册到 exosip 全局变量的 extl 成员变量中。

5) osip_list_t j_transactions;用于管理全部的删除但是还没有系统回收的 transaction。这些 transaction 不属于 call 或 register 或者是 call、register 中删除的 transaction。

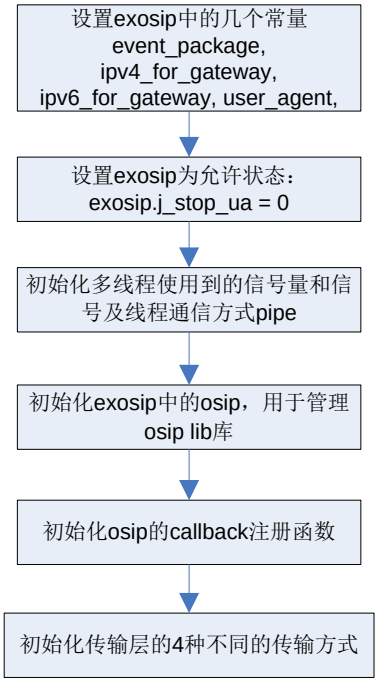
6) osip_t *j_osip; 为 osip lib 库的管理结构，osip Lib 的结构在一个任务中也只会有一个变量存在，用于管理全部的 sip 协议栈中出现的 transaction。根据上面的描述，同一个 transaction 同时被 j_osip 管理和 call/reg 管理，所以当一个 transaction 被删除时，需要从 j_osip 中先将该 transaction 从管理结构中删除，同时从 call 或者 reg 中删除，然后加入到 j_transaction 链表中。在 j_osip 初始化时，初始化了 4 条不同类型的 transaction 的管理链表，同时初始化了 message header 的解析函数的全局注册变量，在函数 increase_ref_count 中初始化。同时 osip lib 库为了实现重入，对被初始化的次数进行了计数。

7) 下面几个字段与协议的认证有关，在 register 时获取到认证信息，在 call 时需要将认证信息添加到 http_anth 或 proxy_anth 中以便在进行 call 时能通过认证。

```
p_access_network_info *p_a_n_i;
digest_cave_response *cav_v;
jauthinfo_t *authinfos;
struct eXosip_dns_cache dns_entries[MAX_EXOSIP_DNS_ENTRY];
struct eXosip_account_info account_entries[MAX_EXOSIP_ACCOUNT_INFO];
struct eXosip_http_auth http_auths[MAX_EXOSIP_HTTP_AUTH];
```

8) 在 OSIP_MT 内部的是为了支持多线程而需要的信号量、锁和线程间的通信方式的 pipe。其中 pipe 经过了封装，在类 unix 系统中直接采用 pipe，在 windows 中采用 socket 通信来模拟 pipe。
9) 在 MINISIZE 内部的为扩展版本，当需要 publish，notify 等功能时，可以启用。

初始化的流程如下，即初始化全局管理变量 exosip 的成员变量的值：



在初始化中，osip lib 库中的发送 message 操作、接收 message 操作、transaction 的 kill 操作、定时器超时操作等会影响到 exosip 中对 transaction、dialog、call、register 等的影响，这些操作由上次的 exosip 决定，所以在进行这些操作时会回调 exosip 注册在 osip 变量中的回调函数。在初始化 exosip 全局变量时会注册这些回调函数，在 eXosip_set_callbacks 函数中实现，这些注册函数的实现在 jcallback.c 文件中实现。
exosip lib 库的销毁调用函数 eXosip_quit，其操作与 exosip_init 的操作相反，其先置位 j_stop_ua 为飞 0，使处理线程停止处理 sip 协议上的 rejister 和 call。然后释放所有申请的内存，释放 exosip 上保存的所有的 call、register 和里面使用到的 dialog 及 transaction。

在 exosip 全局变量的初始化中，部分和业务相关的字段在 exosip 的外部进行初始化。如部分字段的初始化在函数 eXosip_listen_addr，eXosip_masquerade_contact，eXosip_set_user_agent，eXosip_add_authentication_info 中实现，参见 sip_reg.c 文件的 winmain，该函数是 register 的启动测试函数，里面会初始化 register 使用到的几个值，Sip_reg.c 是对 lib 库应用的一个简单实例。

4.2 Lib 库的主处理线程

在 exosip lib 库初始化成功之后，如果启用多线程，则在新的线程中一直执行 exosip_execute，在主线程中执行 eXosip_automatic_action；如果没有启用多线程，则在单线程中每次获取一个 exosip lib 库上报的事件，然后执行 exosip_execute 和

eXosip_automatic_action。在主线程中每次都会去获取 exosip Lib 包上报的 exEvent，在处理 exEvent 时会处理 call 中 200 应答的重发机制。

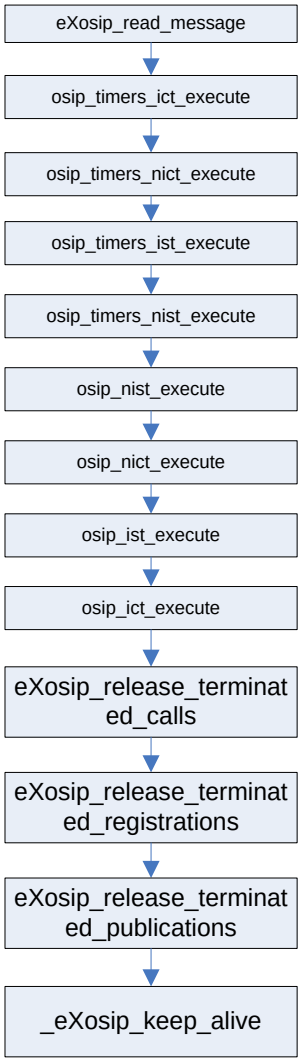
4.2.1 2xx 应答的重发处理机制

在 eXosip_retransmit_lost200ok 中对 2xx 的应答进行重发控制，根据 sip 协议标准，因为 2xx 的重传是需要 sip 协议栈进行控制的，当第一次发送 2xx 应答后，在规定时间内没有接收到 ACK 应答，则重传 2xx 应答。其重传的时间间隔第一次为 1s，以后每次翻倍，增长到 4 秒时每次都间隔为 4s。

2xx 的重发的停止发生在两种情况：一为 2xx 应答发送超时，在发送 9 次之后仍然没有接收到对端发送的 ACK 应答，则停止重发，说明 call 的通话出现的了通信错误，则结束该 call；二为在发送 2xx 应答之后，接收到了对端的 ACK 应答，该 ACK 应答匹配了这个 dialog，则只需要停止 2xx 的发送，同时释放 dialog 中保存的 2xx 消息，并置 dialog 中的 2xx 重复参数为停止。

4.2.2 Exosip_execute 执行流程

exosip_execute 的执行流程比较简单，因为在线程没有被 teminate 的情况下，线程会一直循环执行 exosip_execute，所以在 exosip_execute 内部只需要顺序执行相关的操作即可。其执行流程如下：



在每次执行 eXosip_execute 时，先会去读取 message，所以运行线程一直在监听是否有消息发送到本客户端，对于要发送的 message，都是管理程序主动调用接口进行发送的。在处理接收到的 message 时可能会创建新的 call、新的 transaction，生成新的 transaction 的 event，还有 exosip 的 event。其中 exosip 的 event 是上报给管理程序的，在管理程序中根据具体的实际情况进行处理。其能够参数的各种 event 的定义在文件 exosip.h 的枚举 typedef enum eXosip_event_type 中。

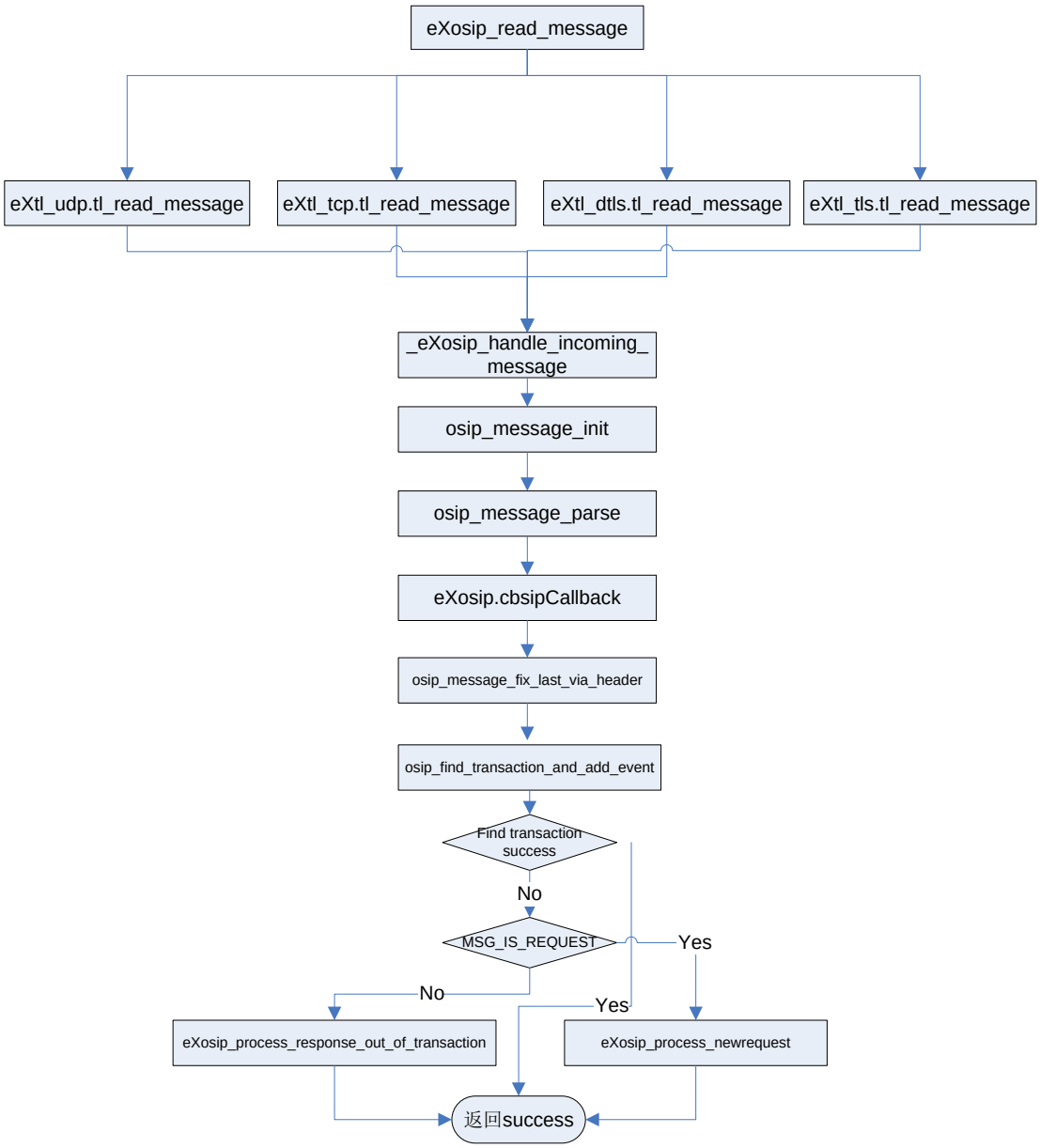
在读取完一个 message 并做了预处理之后(也可能没有新的 message 需要处理)，exosip_execute 开始处理 osip 中 4 条 transaction 链表中的定时器，如果定时器超时，则产生新的定时器事件并放入 transaction 的事件队列中，然后开始执行 4 条 transaction 链表中的每个 transaction 的事件，其中包括对 message 部分的后半部处理。这两部分的代码分析在 osip lib 包分析部分已经有分析。

在处理完上面部分后，开始释放已经结束的 call、registration 和 publication。其中在 registration 和 publication 的释放的时候，只是将其中的 transation 从 registration 和 publication 的结构中删除，然后放在 exosip 全局变量的 j_transaction 链表中。该链表中的 transaction 最终会在 eXosip_release_terminated_calls 中释放。

同时，如果传输层使用的是 UDP 协议，则需要调用_eXosip_keep_alive 为 registration 发送报文保存 UDP 的通信，防止 UDP 超时该端口被系统关闭。

4.2.2.1 Exosip_read_message 的处理

Exosip_read_message 根据使用的传输层协议，调用传输层的 tl_read_message 函数从 TCP/IP 协议栈底层读取 message。如果从传输层读取 message 成功，则交给 _eXosip_handle_incoming_message 进行处理。在_eXosip_handle_incoming_message 中，首先解析该 message，如果解析成功，在解析完之后，检查 message 中的必要字段 call_id number。如果管理程序在 exosip 中注册了 message 的消息处理函数，则回调注册该函数。然后根据 message 的类型，检查合法性并确定该 message 的产生的 transaction 的 event 的类型，因为是接收到 message，所以类型全部为 RCV_XXX 类型，在发送 message 时，产生 SND_XXX 的 event。因为是新接收到的 message，有三种可能，一是能匹配已经存在的 transaction，即是某个请求的应答或 ACK；如果不能匹配，根据 message 中的状态码，如果是 0,说明是一个请求，而且这个请求不能匹配已经存在的 transaction，所以是一个新请求，对新请求的处理在函数 eXosip_process_newrequest 中；如果 status 不为 0，则是一个 response，因为 response 是对一个 request 的回应，而发送 request 的时候在本端肯定已经建立了新的 transaction，如果逻辑处理正确，该 response 应该匹配到某一个存在的 transaction，现在没有匹配到，说明该 response 是一个错误发送的 response，对 response 的处理在函数 eXosip_process_response_out_of_transaction 中。Exosip_read_message 的处理流程如下：



eXosip_process_newrequest 的处理流程如下：

a) 根据 message 的类型，获取到 ctx_type 的类型，因为是接收端，所以本端为 server，如果 message 既不是 INVITE，也不是 ACK，同时不是其它 REQUEST 的情况，则直接释放这个 message。

b) 如果是 ACK，则肯定是对 200 的 response 的一个回应。不需要建立新的 transaction。

c) 如果是 CANCEL，则直接转 eXosip_process_cancel 进行处理。

d) 查看该 message 是否属于某一个 dialog，因为匹配 dialog 会比匹配 transaction 的条件简答，多个 transaction 可以属于同一个 dialog。如果匹配到某一个 dialog，则检查该新的 message 的 cseq_number 和 dialog 中保存的 cseq_number 的大小，如果没有大于 dialog 中保存的 remote cseq number，说明接收到的 message 是一个错误的 request，则释放该 message 并返回。

e) 如果是 INVITE 并且没有定义最小化该协议栈操作的情况下，则先发送一个 100 的临时应答。

f) 如果这个 message 匹配到某一个 dialog

i. 并且不是 ACK 和 BYE，则检查这个 dialog 是否已经结束，既该 dialog 已经发送或接收到过 BYE 请求，则根据 sip 协议标准发送一个 481 的错误提示应答。

ii. 如果不是 ACK，因为已经通过协议的合法性检查，同时匹配到一个 dialog，所以需要根据该 message 更新 dialog 的 remote cseq。

iii. 如果 message 是 INVITE

1. 检查这个 dialog 中最近的一个接收到的 INVITE 是否已经到结束状态，如果没有，则根据 sip 协议标准将这个 new INVITE 删除，并发送一个 500 的提示错误应答。

2. 接着检查这个 dialog 中最近的一个发送出去的 INVITE 是否已经到结束状态，如果没有，则根据 sip 协议标准将这个 new INVITE 删除，并发送一个 491 的提示错误应答。

3. 否则该 INVITE 是个合法的请求，则更新 dialog 的 route set，因为该 INVITE 并不是创建该 dialog 的第一个请求，所以调用 eXosip_process_reinvite 处理该 INVITE 请求。

iv. 如果 message 是 BYE 请求

1. 先检查该 BYE 请求的参数合法性，是否包含 to tag，因为本端发送的 response 里面肯定包含有 to tag，所以对端发送的 BYE 应该是一个包含 to tag 的合法的 message。
2. 检查 dialog 中是否已经接收到 BYE，如果已经接收到 BYE，说明对方重发了 BYE 请求，直接回复 500 错误提示应答
3. 否则调用 eXosip_process_bye 处理 BYE 请求。

v. 如果是 ACK，直接调用 eXosip_process_ack 处理该 ACK 请求

vi. 如果是其它请求，则调用 eXosip_process_message_within_dialog 处理该请求。

g) 否则，没有匹到某一个 dialog，说明是全新的一个请求

i. 如果是 ACK，说明该 200 的 ACK 没有匹配到任何 dialog，所以是一个错误的 ACK，直接释放 message 即可，因为是 ACK，所以并没有建立 transaction，不需要对 transaction 进行操作。

ii. 如果是 INFO，直接回复 481 应答。

iii. 如果是 INVITE，调用 eXosip_process_new_invite 处理这个新请求，如果该请求合法，则会生成一个新的 call，并且在 call 上生成一个 dialog，该 dialog 是服务端的 dialog，因为本端是 UAS。

iv. 如果是 BYE 请求，则和 ACK 一样，该 BYE 请求没有匹配到 dialog，回复一个 481 的错误提示应答通知对端需要结束的 dialog 不存在。

v. 如果是其它类型请求，则因为是不符合 sip 标准的请求方式，所以将创建的 transaction 添加到待删除的队列中即可。

4.2.2.2 eXosip_process_response_out_of_transaction 的处理流程：

a) 因为 message 为应答，而且没有匹配到 transaction，所以肯定是一个错误的应答。先检查 message 本身的合法性，如果不合法，直接释放并返回

b) 查询所以的 call 的 dialog，查看该 response 是否匹配到 dialog 或者是还没有建立 dialog 的 call

c) 如果没有匹配到某一个 call，说明该 response 与 call 无关，直接释放。

d) 如果#ifndef MINISIZE，且匹配到某一个 dialog，说明重复接收了 200 应答，可能原因是对方还没有接收到本端发送的 ACK，但是本端的 transaction 在接收到 200 应答时已经被 kill 了，所以没有匹配到 transaction，但是匹配到了 dialog。如果接收到 200 应答的 cseq 和本端发送的 cseq 的 number 相等，则重新发送 ACK 应答。处理完上述情况后，释放该 message 并返回。

e) 如果只是匹配到了 call，说明 dialog 还没有建立，但是给 200 应答是一个错误的应答，否则会匹配到 call 的 c_out_tr transaction。则为该 200 应答临时建立一个 dialog 并发送 ACK 回应，然后发送一个 BYE 请求结束该 call，因为该 call 已经发送错误了。

4.2.3 eXosip_automatic_action 处理流程

该函数用于处理哪些认证失败的 call、register、notify、publish 等，在接收到认证服务器回应为 401 或 407 或需要转发的情况下，进行重新尝试。

a) 遍历所有的 call:

i. 如果 c_id < 1 则不用处理，call 的 id 小于 1 的都是被删除的但是还没有被清理出 j_calls 链表的 call。

ii. 如果该 call 的 dialog 还没有建立起来，说明是本端发送第一个 INVITE 请求建立的新 call，如果是对端发送的第一个 INVITE，则本端要回应一个 response，在回应 response 的时候，dialog 就被建立起来了。

1. 检查建立该 call 的第一个 INVITE 请求建立起来的 transaction 的状态，如果已经终结，并且该 call 还没有到结束超时时间 120 秒，且接收到的回复的状态码为 401 或 407，则重发送请求，并且从回复中提取认

证信息。重发次数最多为 3 次，如果 3 次都失败，则等待直到该 call 超时结束。

2. 同上，如果回复应答的 status 为[300, 399]，则重新发送请求，并且转换发送目的地，重复发送次数也限制为 3 次。这两种情况全部调用 _eXosip_call_retry_request 进行处理。

iii. 遍历 call 中所有的 dialog，对于已经 dialog 信息存在的 dialog 进行处理，处理方式同上，也是检查两类情况，一为应答 status 为 401 或 407 的认证失败错误提示，一为[300, 399]server 端地址需要更改的转发提示。

b) 遍历所有的 register，只处理 r_id >=1 且有 transaction 的 register，r_id < 1 或者没有 last transaction 的 register 是已经被删除的 register，不需要处理。

i. 如果重发时间不为 0，既该 register 需要一直重新发送注册，且注册时间已经超时，该 register 从注册完到现在已经超过 900 秒，则调用函数 eXosip_register_send_register 进行重注册。

ii. 如果注册时间到现在为止 大于规定的重注册时间间隔-60 秒，也发起重注册，既如果设置了重注册时间间隔，必需在重注册时间间隔到达之前的 60 秒就开始发起重注册。

iii. 或者如果需要重注册，而且离上次注册时间已经超过 120 秒，并且没有接收到注册服务器的应答或者应答不是成功注册的应答，则也发起重注册。

iv. 如果还未设置需要重注册，即第一次注册失败，且是因为认证失败而引起的注册失败，则检查注册类型。如果是 WPDIF 注册方式，则检查回复中的认证码 nonce 是否和上一次注册失败时保存的 nonce 值相同，如果相同则发起重注册；如果注册方式不是 WPDIF，则直接发起重注册。为了保证 WPDIF 注册方式的成功，在第一次注册失败时，需要提取当次服务器端的回复中的 nonce 值，以便在确定是否发起重注册的时候判断条件为真。

c) Notify、subscription、pub 的处理同 call。

4.3 Call 的处理

在 exosip lib 库中除了一直在运行的处理对端发送过来的 message 的线程外，还提供了本端做为发送端发送各种 request、ACK。

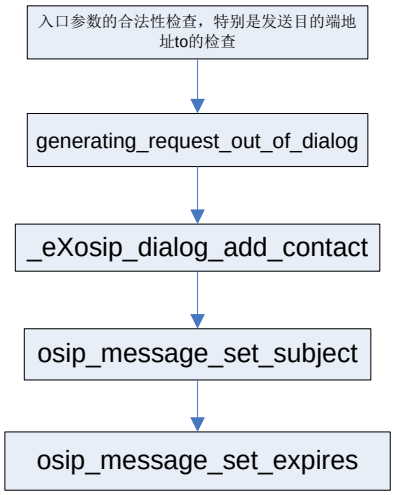
因为所有的 message 最终都属于某个 transaction，而且对 message 的处理最终都放在 transaction 的 event 队列中进行处理，所以所有提供的包括 call、notify、publish、register、subscription、refer 等功能都是通过生成一个 transaction 上的 event 事件与一直运行的处理线程进行联系，当生成 event 之后，处理线程在轮询所有 transaction 时会处理到这些 event，当某个 event 需要立即处理的时候，则可以手工启动 event 的处理线程，而不需要等到该处理线程处理到该 transaction。

Exosip lib 库提供了 call 的 4 类接口：第一次建立一个 call 的 initial invite 创建及发送接口；在 dialog 中创建及发送的其它 request；在 dialog 中创建及发送对从对端发送过来的 request 的 response；在 invite 请求中回应对端 response 的 ACK 的创建和发送。

4.3.1 创建 Call 的第一个 INVITE

在 excall_api.c 文件中的 eXosip_call_build_initial_invite 和 eXosip_call_send_initial_invite 提供了本端发起一个新的 call 时的接口。

eXosip_call_build_initial_invite 的流程如下：

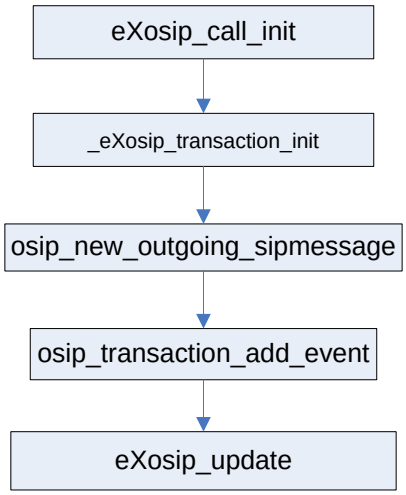


在创建一个新的 INVITE 时，而且该 INVITE 是 call 的第一个 INVITE，则需要检查必要参数目的端地址 to 的合法性。生成一个 INVITE message 需要使用到的信息大部分在 exosip lib 库启动的时候就设定了，包括 sip 协议的版本，from tag 的产生机制等。需要特别指定的只有发送目的端 to。

在检查合法性之后，调用通用的请求构造函数 generating_request_out_of_dialog 构造生成一个 request message，在构造参数中指定要构建的是一个 INVITE。因为是在 dialog 创建之前构建 INVITE 请求，所以调用的接口为 out_of_dialog 的，即不需要从 dialog 中获取信息；如果是 dialog 内构造的新的请求，则根据 sip 协议，其新的 request 的 local cseq number 必须大于 dialog 中的 local cseq number，并且因为属于一个 dialog，所以其 call_id， from tag, to tag 必须同 dialog 相同。同时如果不是 dialog 中的新的 INVITE 请求用于更改 route set，则还要使用 dialog 中的 route set 用于该请求中。

在创建成功通用的 dialog 外的请求后，添加 INVITE 相关部分字段，包括 contact，subject 和超时时间 expires。其中 contact 之间使用本端内部设置的 IP 地址。

eXosip_call_send_initial_invite 的流程：



首先 initial invite 是用于创建一个新的 call 的，所以在发送一个 initial invite 时，创建一个 new call，并且为该 invite 创建一个新的 transaction，该 transaction 类型为 ICT，同时将该 transaction 做为这个 new call 的 c_out_tr，因为整个 call 是有该 transaction 创建的，当该 transaction 到达 complete 状态的时候该 call 就建立起来了。但是此时并不建立 dialog，因为 dialog 是两端通信协商后的结果，只有收到了对端的非 100 的 1xx 应答或>200 的应答才会建立一个 dialog。

在创建完 call 和 transaction 之后，根据要发送的 INVITE 生成一个 transaction 上的 event，将该 event 添加到该 transaction 的 event 队列中；并将 call 添加到 exosip 管理的 call 链表中，然后给 call 分配一个 call_id，最后唤醒处理线程对 transaction 上的 event 进行处理。

在两个线程的交互方面是通过 transaction 的 event 队列来完成的，eXosip_call_send_initial_invite 只是将要发送的 invite 生成一个 event 添加到 event 队列，真正的发送在处理线程处理该 event 时才会进行。

4.3.2 INVITE 的 ACK 应答的创建和发送

对端发送过来的 INVITE 等请求的处理在 exosip_read_message 中已经进行了处理，对 request 的应答是 sip 协议栈自动完成的，同时对非 2xx 的应答也自动完成。当接收到的是 2xx 的应答，在 transaction 层的处理会上报一个 EXOSIP_CALL_ANSWERED 事件给 UAC 层，此时需要管理程序处理该事件，创建一个 ACK 应答并发送该 ACK 应答，同时如果在超时时间内继续接收到该 2xx 应答，只需要重新发送该 ACK 应答即可。

客户端接收到 2xx 应答的处理参见 Jcallbakck.c 文件的 cb_rcv2xx 函数。

eXosip_call_build_ack 的处理流程：

a) 根据所属的 dialog，查询得到所属的 call 和 dialog 的结构。
b) 获取该 dialog 中还未处理完的 transaction，如果该 transaction 不是 INVITE 的 transaction 则返回错误，因为 ACK 只会出现在 INVITE 的请求 transaction 中。
c) 调用函数_eXosip_build_request_within_dialog 构建 ACK message 并且设置 ACK 的联系地址 contact 和 INVITE 相同。
d) 设置 ACK 的 cseq number 同 INVITE 的 cseq number，ACK 是一个特殊的请求，其 method 与对应的 INVITE 不同，但是使用相同的 cseq number，用于对端确定该 ACK 是哪个 INVITE 的 ACK。
e) 同时设置 ACK 的认证信息 anthorization 为 INVITE 的 anthorization。

eXosip_call_send_ack 的处理流程：

a) 参数检查，需要确定要发送的 ACK 所属的 dialog 是否正确。并查询得到 call

和 dialog 的结构地址。

b) 检查第一个路由项，如果没有 ”lr” 标识，说明是前一版本的路由设置规则，根据协议的向后兼容性，需要调整发送目的 request_uri 为第一个路由的 request_uri，并且保存原发送目的 request_uri 为路由集合中的最后一条路由。

c) 直接发送该 ACK，不做为一个 event 添加到 transaction 是防止中间的消耗时间过长，导致对端的 2xx 应答重发。

d) 保存该 ACK，在对端重发 2xx 应答时重发该 ACK。

4.3.3 dialog 内的请求的创建和发送

eXosip_call_build_request 处理流程：

因为是 dialog 已经建立完成，所以创建的新的 dialog 内的请求只需要知道 request 的类型即可以。创建一个新的 request 所需要的信息在 dialog 内部已经保存，包括 call_id、from、from tag、to、to tag、cseq number、request_uri 以及认证信息 anthentication information。

在创建一个新的请求的时候，需要检查是否有未处理完的请求存在，因为按照 sip 协议标准，一个 call 内的请求是要按照顺序进行处理的，即上一个请求没有处理完，下一个请求不应该被发送出去。其中 INVITE 请求比较特殊，只要没有 INVITE 请求没有处理完，就可以发送下一个 INVITE 请求，而不需要等待像 notify、option 等的请求。

eXosip_call_send_request 处理流程：

a) 检查要发送的 request 的合法性，并且检查所属的 dialog 的合法性。

b) 检查该 dialog 上是否有 transaction 没有处理完毕，如果有，则返回错误，不允许多个请求同时在一个 call 上处理。这个检查和 build 时是一样的。

c) 为新请求创建一个 transaction，并且将该 transaction 加入到 dialog 的 d_out_tr 链表中，因为是请求发送方，所以本端为 client 端，所以创建的 transaction 为 NICT 或 ICT，并且是属于本端发送出去的 transaction。

d) 根据该请求生成一个 event，加入到该 transaction 的 event 队列中。

e) 最后唤醒处理线程对该 event 进行处理，即将该 request 发送出去。

4.3.4 Dialog 内 answer 的创建和发送

在接收到 dialog 内部的 request 时，需要发送 response。

eXosip_call_build_answer 的处理流程：

a) 根据 transaction id 查询得到 call、dialog、transaction 的结构。因为是接收到一个请求，所以在处理请求的时候已经创建了新的 transaction，所以查询在正常

情况下不会失败。

b) 如果是 INVITE 的请求，则调用_eXosip_answer_invite_123456xx 进行 response 的 message 的构建，根据传入的最后一个参数 0 标识只构建 message 不发送该 message。

c) 如果不是 INVITE，则直接调用 eXosip_build_response_default 生成一个通用的应答 message，如果回复的应答状态为(100, 399]，则最终本端和对端会建立 dialog，则调用函数 complete_answer_that_establish_a_dialog 从 request 中获取部分信息。

eXosip_call_send_answer 的流程：

a) 参数的合法性检查，如果回应的 status 不在[100,699]之间，或者 transaction id <0，则返回错误提示。

b) 根据 transaction id 查询得到 call、dialog、transaction 的结构，如果没有找到，则返回错误。

c) 检查该 transaction 的合法性，如果其状态已经结束，则返回错误。

d) 如果 answer 还没有创建，并且是 INVITE 的应答，且应答 status 为 2xx 应答，则返回错误。

e) 如果要发送的 answer 还没有创建，且是 INVITE 的应答，则调用 _eXosip_answer_invite_123456xx 创建并发送应答。如果不是 INVITE 的应答，则返回错误。

f) 如果是 INVITE 的应答，且应答的 status 为 2xx，并且 dialog 已经创建，则保存该 2xx 的应答到 dialog 中，并且置 dialog 的状态为 confirmed，在未收到 ACK 的情况下，该 2xx 应答会被重新发送。

g) 所有合法性检查通过，则生成一个 transaction 上的 event 事件，并且添加到 transaction 的 event 链表上。

4.4 Register 的处理

同 call 一样，exosip lib 库通用提供了创建一个新的 register 和发送 rejister 的接口，管理程序只要调用接口创建一个新的 register 并且调用发送接口进行发送即可。

Register 注册包括初始的注册，改变超时时间和取消该注册。其区别主要为发送给注册服务器的参数 expires 即超时时间的不同。

如果 expires 为 0，则为终止该注册；如果为 expires 大于 0，则为修改或注册该 register。为避免太频繁的重注册行为，规定注册的无效时间最小为 100s，而且服务器可以自己配置

该最小值比 100 大。

4.4.1 向一个服务器第一次注册

向一个服务器第一次注册时，调用接口 eXosip_register_build_initial_register 生成一个新的 register message。该函数会进行一些合法性检查，其处理流程如下：

a) 查询所有的 register，检查其注册服务器的地址与现在要注册的服务器的地址是否相同，如果有相同的服务器地址存在，则删除原有的注册的 transaction，进行重新注册。

b) 调用 eXosip_reg_init 生成一个新的 register 管理结构并且添加到 exosip 的 j_reg 管理链表中。

c) 调整这个新的 register 的重注册时间，如果输入的超时时间 expires <= 0，说明是一个注销行为，则设置重注册时间为 0，即不需要重注册。如果设置的重注册时间小于 100，则调整为最小值 100。

d) 调用_eXosip_register_build_register 创建一个标准的 register message，并返回新生成的 register 管理结构的 id。

4.4.2 调整一个注册的注册超时时间

接口 eXosip_register_build_register 用于创建一个注册 message，用于调整已经注册成功的注册的超时时间。

a) 接口根据传入的 register 的 id 查询得到 register 的管理结构。如果查询失败，则返回错误。

b) 重置该 register 的超时时间为传入的参数 expires，并且根据协议的规范调整其范围到[100, 3600]。

c) 检查被调整 expires 的注册的前一个请求处理是否已经结束，如果没有，则返回错误提示。

d) 调用_eXosip_register_build_register 创建一个新的 register message 并返回 register 的 id。

4.4.3 发送一个 register 注册

前两个的创建新的 register message 之后，都需要调用接口 eXosip_register_send_register 发送新创建的 message。

Register 的发送流程如下：

a) 检查要发送注册 message 的 register 的上一个注册请求的状态码是否已经到结束状态，如果不是，则返回错误。

b) 为要发送的 register message 创建一个新的 transaction，每个新的请求都对应一个新的 transaction。并且将该 transaction 挂接在 register 的管理结构 jr->r_last_tr

中。每个主次只保存最近的一个 transaction，因为注册必须是串行的，在上一个注册还没有处理完毕的情况下，不允许在同一个注册服务器上发送新的注册请求。

c) 根据发送的 message 生成一个 transaction 上面的 event 并挂接在 transaction 的 event 队列中。

d) 唤醒处理线程，处理 transaction 的 event。

Exosip lib 包中提供的 notify、publish、subscribe、message 和 options 等的功能和 call、register 的功能是相近的，不做详细解释。