PJSIP 开发指南

关于 PJSIP:

PJSIP 是一个用 C 写的小巧而性能优异的 SIP 协议栈。

请访问:http://www.pjsip.org 获得更多信息。

请访问张文杰的博客:http://zhangwenjie.net 查看最新文档翻译结果。

关于本文档:

目录

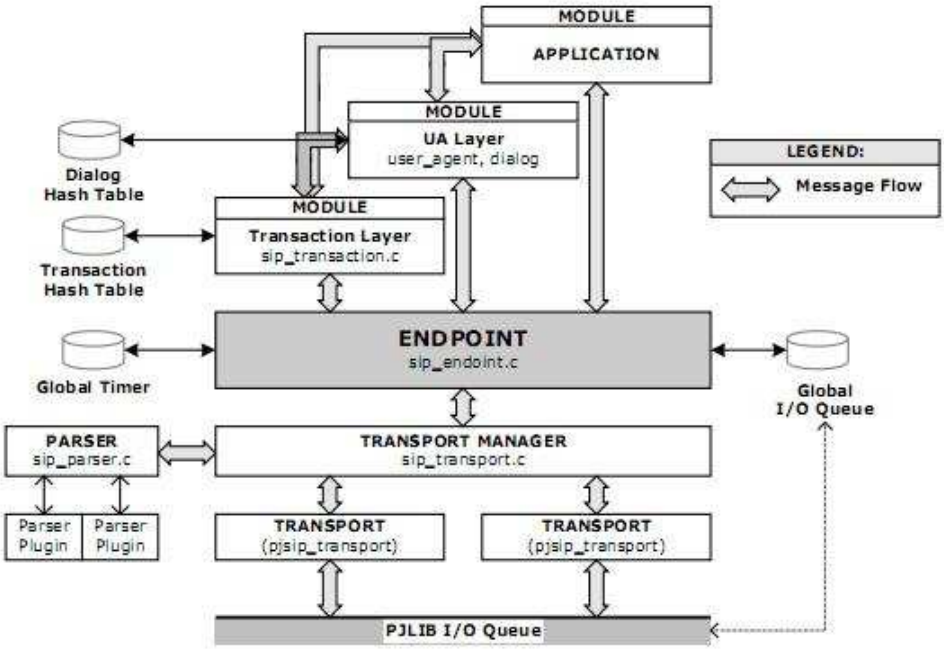
PJSIP 开发指南

第一章、通用设计

1.1 架构

1.1.1 通信图

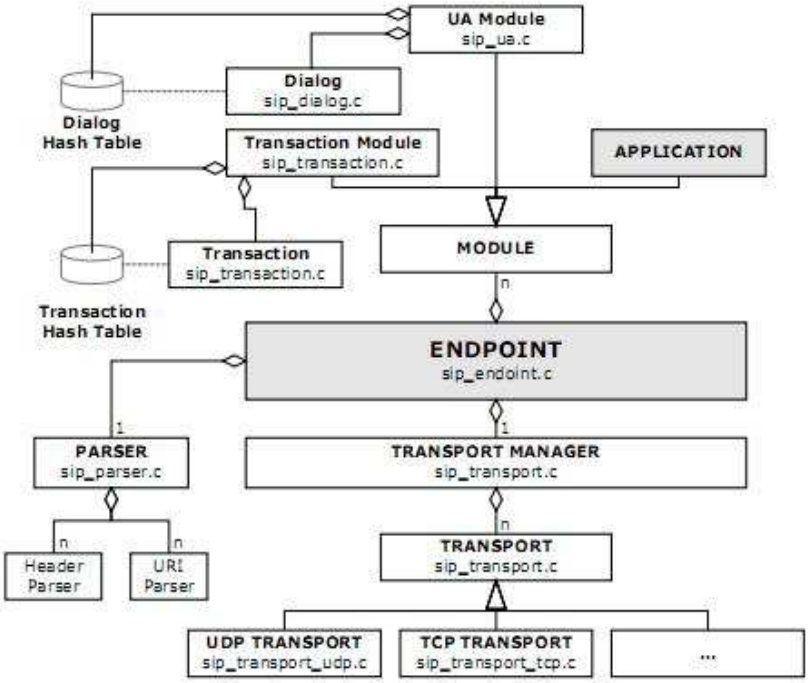
下面的示意图显示了(SIP)消息是如何在 PJSIP 组件之间来回传递的。



图表 1：协作表

1.1.2 类图

下面的示意图显示了“类图”:



图表 2：类图

1.2 Endpoint

SIP 协议栈的核心就是 SIP Endpoint，由不透明的类型 pjsip_endpoint 来表示。Endpoint 具体有以下的属性和职责：

- 它有内存池工厂，为所有 SIP 组件分配内存池
- 它有定时器堆实例，为所有 SIP 组件调度定时器
- 它有传输管理器实例。传输管理器有 SIP 传输商品，且控制消息解析和显示。
- 它拥有单实例的 PJSIP ioqueue 类型。Ioqueue 是用来分派网络事件的 proactor 模式。
- 它提供线程安全的轮询功能，这样应用程序中的线程可以轮询定时器和网络事件(PJSIP 本身不创建任何线程)。
- 它管理模块。PJSIP 模块是扩展协议栈的主要方法，而协议栈扩展并不局限于消息的解析和显示。
- 它从传输管理器接受到来的消息，并将这些消息分布到模块。

一些基本功能将会在下面的部分讲述，其余的会在后面的章节讲述。

1.2.1 内存池的分配和释放

SIP 组件的所有内存分配都是通过 endpoint 来完成的，以在整个应用程序中保证线程安全及强制策略的一致性。可应用策略的一个例子是内存池缓存，这里未使用的内存将保留以备将来使用，而不是释放。

Endpoint 提供以下方法来分配和释放内存池：

- pjsip_endpt_create_pool()
- pjsip_endpt_release_pool()

当创建 endpoint 时(使用 pjsip_endpt_create())，应用程序必须指定 endpoint 使用的内存

池工厂。Endpont 将在自己的生命期内保持此内存池工厂的指针，并由此来分配和释放内存池。

1.2.2 定时器管理

Endpoint 保留一个单实例的定时器堆来管理定时器。所有定时器的创建和所有 SIP 组件的定时器调度皆由 endpoint 来完成。

Endpoint 提供以下方法来管理定时器：

- pjsip_endpt_schedule_timer()
- pjsip_endpt_cancel_timer()

当 endpont 的轮询函数被调用时，endpoint 将检查定时器是否过期。

1.2.3 轮询协议栈

Endpoint 提供了一个单一的函数调用(pjsip_endpt_handle_events())来检查定时器和网络事件的出现。应用程序可以指定它将等待多长时间后去检查这些事件的出现。

PJSIP 协议栈从不创建线程。整个协议栈中的代码执行完全代表着应用程序创建的线程，无论是在一个 API 被调用时，或是应用程序调用轮询方法时。

轮询功能是可以基于定时器堆的内容来优化等待时间的(The polling function is also able to optimize the waiting time based on the timer heap's contents.)。例如，如果它知道一个定时器将在下一个 5 秒过期，它等待网络事件的时间将不会超过这 5 秒；在无网络事件出现时这样做将无必要地延长等待时间。当然定时器的精度在每个平台上都不同。

1.3 线程安全和线程复杂性

1.3.1 线程安全

线程安全的讨论是一个相当复杂的事情。但是，比较幸运的是，下面的设计原则在整个协议栈中的一致应用：

对象必须是线程安全的；而数据结构必须不是线程安全的。

具体到现在的话题，很自然，对象和简单数据结构的区别不是非常清楚。但是一些例子将会使你对此更明白一点。

数据结构的例子有：

- PJSIP 的数据结构，如链表(lists)、数组(arrays)、哈希表(hash tables)、字符串(strings)、以及内存池。
- SIP 的消息元素，如 URLs、header fields、以及 SIP 消息。

这些数据结构不是线程安全的，这些数据结构的线程安全由包含它们的对象来保证。如果使数据结构也线程安全，这将严重影响协议栈的性能，并消耗操作系统的资源。

相比之下，SIP 对象必须是线程安全的。我们称之为对象的例子有：

- PJLIB 对象，如 ioqueue
- PJSIP 对象，如 endpoint、transactions、dialogs、dialog usages,等等

1.3.2 线程复杂性

使事情变糟的是，一些对象在头文件中暴露了它们的声明(例如 pjsip_transaction 和 pjsip_dialog)。尽管这些对象暴露的 API 是保证线程安全的，应用程序代码在访问这些数据结构之前仍然必须在对象的互斥变量(mutex)上调用 pj_mutex_lock()来获取正确的锁。

使事情变得更糟的是，一个 dialog 提供不同的 API 来锁定 dialog。这样应用程序**应该**调用 pjsip_dlg_inc_lock()和 pjsip_dlg_dec_lock()，而不是 pj_mutex_lock()和 pj_mutex_unlock()。这两种方法的区别是，dialog 的 inc/dec 锁保证 dialog 不会在函数调用过程中被销毁；不然由于 gialog 已经被销毁，会使 pj_mutex_unlock()崩溃。

考虑下面的例子：

```
1. pj_mutex_lock(dlg->mutex);
2. psip_dlg_end_session(dlg,…);
3. pj_mutex_unlock(dlg->mutex);
```

在上面的例子中(假想的)，程序**可能**会在第三行代码处崩溃，因为 psip_dlg_end_session()有可能在一定情况下销毁 dialog。例如，出去的初始 INVITE 事务没有得到任何回应，因此事务会马上被销毁，造成 dialog 也被销毁。Dialog 的 inc/dec 锁通过临时增加 dialog 会话的计数器来避免这个问题，因而在 end_session()中 dialog 不会被销毁。Dialog 可能会在 dec_lock()方法中销毁。因此正确锁定 dialog 的顺序应该像这样：

```
1. pj_mutex_lock(dlg->mutex);
2. psip_dlg_end_session(dlg,…);
3. pj_mutex_unlock(dlg->mutex);
```

最后，真正使情事变糟的是，锁定的顺序必须正确，否则可能发现死锁。例如，应用程序在 dialog 中想既锁定 dialog，又锁定 transaction，应用程序必须在获取 transaction 锁之前获取 dialog 锁，否则当另一个线程正在以相反的顺序获取相同 dialog 和 transaction 的锁时，死锁将会发生。

1.3.3 解决方法(The Relief)

幸运的是，应用程序很少需要直接获取对象的锁。因此几乎不会出现以上所述的问题。如果可用，应用程序应该使用对象的 API 来存取对象。由于会对对象进行检查，对象

的 API 保证加锁的正确性及避免死锁和崩溃的出现。

当一个对象调用应用程序的回调函数时(如 dialog 和 transaction)，这此回调函数在对象的锁获取后正常调用，因此应用程序可以安全访问对象的数据结构而不用获取对象的锁。

第二章 模块

模块框架是在 PJSIP 程序中的各个软件组件中派发 SIP 消息的主要方法。PJSIP 中的所有软件组件，像事务层(transaction layer)和对话层(dialog layer)，都以模块的方式来实现。如果没有模块，核心协议栈(pjsip_endpoint 和 transport)根本不知道如何去处理 SIP 消息。

模块框架是基于简单但强大的接口抽象。对到来的消息，endpoint(pjsip_endpoint)从优先级高的模块开始向所有的模块派发消息，直到其中一个模块告诉框架它自己处理了此消息。对出去的消息，endpoint 在消息真正开始发送到网络之前派发这些消息到所有模块，允许所有模块有机会对消息做最后的修改。

2.1 模块特性

2.1.1 模块声明

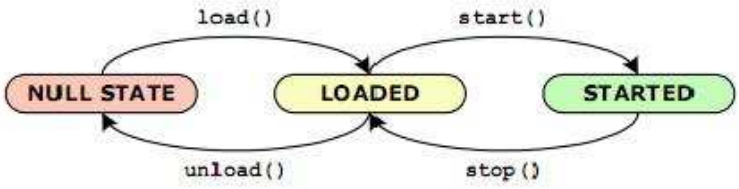
模块接口是在<pjsip/sip_module.h>中声明的：

```
struct pjsip_module
{
   PJ_DECL_LIST_MEMBER(struct pjsip_module);                    // For internal list mgmt.
   pj_str_t             name;                                   // Module name.
   int                  id;                                     // Module ID, set by endpt
   int                  priority;                               // Priority

   pj_status_t (*load)          (pjsip_endpoint *endpt);        // Called to load the mod.
   pj_status_t (*start)         (void);                         // Called to start.
   pj_status_t (*stop)          (void);                         // Called top stop.
   pj_status_t (*unload)        (void);                         // Called before unload
   pj_bool_t   (*on_rx_request) (pjsip_rx_data *rdata);         // Called on rx request
   pj_bool_t   (*on_rx_response)(pjsip_rx_data *rdata);         // Called on rx response
   pj_status_t (*on_tx_request) (pjsip_tx_data *tdata);         // Called on tx request
   pj_status_t (*on_tx_response)(pjsip_tx_data *tdata);         // Called on tx request
   void        (*on_tsx_state)  (pjsip_transaction *tsx,        // Called on transaction
                                 pjsip_event *event);           // state changed
};
```

在此声明中，所有的函数指针都是可选的；如果没有指定这些函数指针，那么他们将被认为是成功返回值的。

其中的四个函数指针，load、start、stop、unload 是由 endpoint 调用来控制模块的状态。下面的图显示了模块状态的生命期：



on_rx_request()和 on_rx_response()函数指针是模块从 endpoint(pjsip_endpt)或其它模块接受 SIP 消息的主要方法。这些回调函数的返回值是很重要的。如果回调函数返回非 0 值，从语义上意味着这个模块处理了这个消息；这个情况下，endpoint 将停止向其它模块派发此消息。在节 2.1.3 **模块对到来消息的处理**中会详细描述这个问题。

on_tx_request()和 on_tx_response()函数指针在消息传送之前由传输管理器来调用。这使一些类型的模块(如 sigcomp、message signaling)有机会对消息做最后的修改。所有的模块**必须**返回 PJ_SUCCESS(如 0 状态)，否则消息传输将会取消。在节 2.1.4 **模块对外出消息的处理**中会详细描述这个问题。

on_tsx_state()用来在每次事务状态改变时接受通知消息。事务状态改变可能由接受到消息、消息发了出去、定时器消息、传输错误事件等引起。更多关于这个回调函数将在节 2.1.5 “事务用户和状态回调”中描述。

2.1.2 模块优先级

模块优先级指定了哪个模块将被优先调用来处理回调函数的顺序。拥有高优先级的模块(优先级的数字较小)的 on_rx_request()和 on_rx_response()将先被调用，它的 on_tx_request()和 on_tx_response()将最后被调用。

下面表出了标准的可设置的模块优先级：

```
enum pjsip_module_priority
{
    PJSIP_MOD_PRIORITY_TRANSPORT_LAYER  = 8,  // Transport
    PJSIP_MOD_PRIORITY_TSX_LAYER        = 16, // Transaction layer.
    PJSIP_MOD_PRIORITY_UA_PROXY_LAYER   = 32, // UA or proxy layer
    PJSIP_MOD_PRIORITY_DIALOG_USAGE     = 48, // Invite usage, event subscr. framework.
    PJSIP_MOD_PRIORITY_APPLICATION      = 64, // Application has lowest priority.
};
```

记住：较低的优先级数字意味着较高的优先级。

优先级 PJSIP_MOD_PRIORITY_TRANSPORT_LAYER 是由传输管理器使用的。这个优先级当前只是用于控制消息的传输，比如比这个优先级低的模块(也就是优先级数字较高)，它的 on_tx_request()/on_tx_response()函数将在消息被传输层(transport layer)处理之前被调用；而有高优先级的模块的 on_tx_request()/on_tx_response()函数将在消息被传输层(transport layer)处理之后被调用。参见 2.1.4 **模块对外出消息的处理**以获得更多信息。

PJSIP_MOD_PRIORITY_TSX_LAYER 是被传输层(transport layer)模块使用的优先级。传输层(transport layer)将吸收所有属于同一个事务的到来消息。

PJSIP_MOD_PRIORITY_UA_PROXY_LAYER 是由 UA(如 dialog framework)或代理层(proxy layer 使用的优先级。UA 层吸收所有属于同一个对话集(dialog set)的到来消息(这也意味着有

分歧的回应)。(这里的翻译可能不准确：this means forked responses as well)。

PJSIP_MOD_PRIORITY_DIALOG_USAGE 由 dialog usages 使用。当前 PJSIP 实现了两种类型的 dialog usages：邀请会话(INVITE session)和事件订阅会话(event subscription session)(包括 REFER 订阅)。Dialog Usage 吸收所有在同一个对话中属于一个特定会话的消息。(The dialog usage absorbs messages inside a dialog that belong to particular session)。

PJSIP_MOD_PRIORITY_APPLICATION 是典型应用程序模块想使用 transactions、dialogs、及 dialog usages 时可使用的合适优先级值。

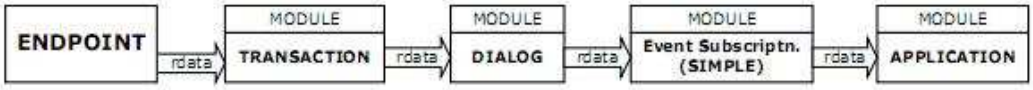
2.1.3 模块对到来消息的处理

当进来的消息到达后，它表现为接受消息的缓冲区(结构 pjsip_rx_data，参见节 5.1 “接受数据缓冲区”)。传输管理员解析消息，将解析后的数据结构放到接受消息缓冲区，然后传递消息到 endpoint。

Endpoint 通过调用每个注册模块的 on_rx_request() 或 on_rx_response()回调函数派发接受消息缓冲区，从高优先级的模块开始(低优先级数字)，直到有一个模块返回非 0 值。当有一个模块返回非 0 值时，endpoint 停止派发消息到其余的模块，因为它假设这个模块将关心这个消息的处理。

在回调函数中返回非 0 值的模块自己也可能派发消息到其它模块。例如，事务(transaction)模块，当收到匹配的消息后开始处理消息，然后派发此消息到它的事务用户(transaction user)，事务用户(transaction user)本身也必须是一个模块。事务(transaction)通过调用事务用户(transaction user)的回调函数 on_rx_request() 或 on_rx_response()来将消息传递给这个模块，然后设置接受消息缓冲区(receive message buffer)的事务字段(transaction field)，这样事务用户(transaction user)模块将可以区分事务外和事务内的消息。

下面的图显示了模块之间是如何层叠(cascadely)地调用其它模块的例子：



2.1.4 模块对外出消息的处理

一个外出的请求或响应消息表现为传输数据缓冲区(transmit data buffer)(pjsip_tx_data)，同时还有其它数据，包含消息结构本身、内存池、连续的缓冲区、以及传输信息。

当调用 pjsip_transport_send()来发送消息时，传输管理器从优先级低的模块(高优先级数字)开始，调用所有模块的 on_tx_request() 或 on_tx_response()回调函数。当这些回调函数被调用时，消息可能会也可能不会被传输层(transport layer)处理。传输层(transport layer)负责管理传输缓冲区中的这些信息：

- 传输信息,以及
- 输出消息结构到连续的缓冲区

优先级比 PJSIP_MOD_PRIORITY_TRANSPORT_LAYER 低的模块将在这些信息获取之前接受到消息。这意味着，目标地址还没有计算出来，消息还没有输出到连续的缓冲区。

如果模块想在消息被输出到缓冲区前修改消息结构，那么模块应该设置它的**优先级数字**比传输层(transport layer)大。如果模块想看到在网络中传输的真实字节包(如为了记录日志)，那么模块应该应该设置它的**优先级数字**比传输层(transport layer)小。

注：一个实际的情况是日志模块想要设置它的优先级比传输层(transport layer)高。这样在消息输出到相邻缓冲区及目标地址计算出来后打印日志。

在所有情况下，模块的回调函数必须返回 PJ_SUCCESS。如果一个模块返回了其它错误代码，消息发送将会取消，这个错误将返回给调用者 pjsip_transport_send()。

2.1.5 事务用户及状态回调

模块定义中的一个特殊回调函数(on_tsx_state)用来在事务状态变化时从这个特定的事务接受通知。这个回调函数是唯一的 ,因为事务状态的改变可能是由于非消息相关的事件(如定时器超时或传输错误)。

这个回调函数只在特定事务中模块注册为事务用户(transaction user for a particular transaction)时才会被调用。在一个事务中只允许有一个事务用户。事务用户可以以每事务的基础来设置到事务。

对于在对话中创建的事务来说，事务用户代表这个对话设置到 UA 层模块。当应用程序手动创建事务时，应用程序将自身设置为事务用户。

on_tsx_state 回调函数在收到**重传**的请求或响应时并不会被调用。注意：传输或接受**临时响应**(provisional response)不会被认为是重传,这意味着传输或接受**临时响应**仍然会引发这个回调函数被调用。

2.1.6 特定于模块的数据

一些 PJSIP 组件拥有容器，这样模块可以将模块特定的数据放到这个组件中。一般这个容器命名为 mod_data，它是一个 void 类型的指针数组，由模块 ID 进行索引。

例如，到来消息包缓冲区((pjsip_rx_data)有以下的模块特定数据容器声明：

```
struct pjsip_rx_data
{
    ...
    struct {
        void *mod_data[PJSIP_MAX_MODULE];
    } endpt_info;
};
```

mod_data 数据由模块 ID 进行索引。模块 ID 在向 endpoint 注册时确定下来。

当一个到来消息包缓冲区(pjsip_rx_data)向模块传递时，一个模块可以将此模块特定的

数据放到合适索引的 mod_data 中，这样以后这些数据可以由此模块或应用程序取出。例如，事务层(the transaction layer)会将匹配的事务实例放到 mod_data 中，用户代理层(user agent layer)也会将匹配的对话实例(dialog instance)放到 mod_data 中。应用程序可以调用 pjsip_rdata_get_tsx() 或 pjsip_rdata_get_dlg()来取得这些数据，这只是一个简单的数组查寻：

```
// This code can be found in sip_transaction.c

static pjsip_module mod_tsx_layer;

pjsip_transaction *pjsip_rdata_get_tsx(pjsip_rx_data *rdata)
{
   return rdata->endpt_info.mod_data[mod_tsx_layer.id];
}
```

2.1.7 回调函数总结

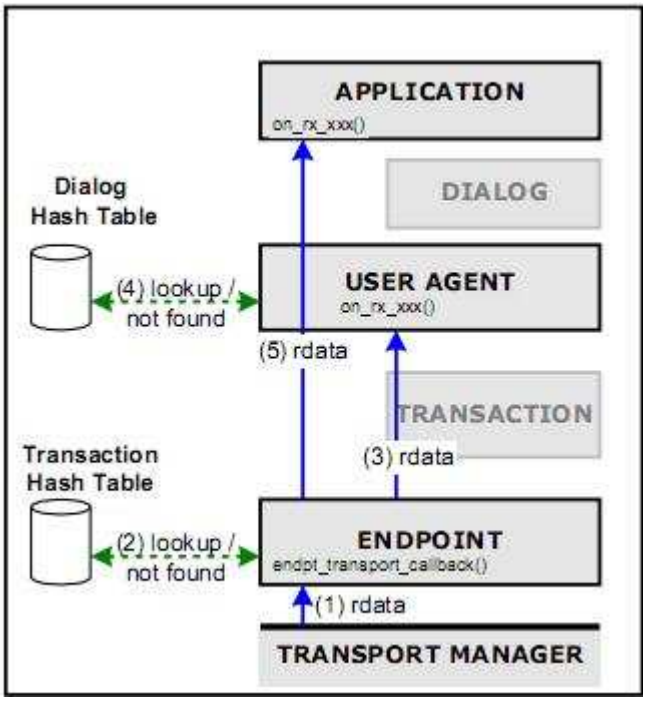
下面的表格总结了事件的发生及引发的回调函数。当然 on_tsx_state()回调函数只在应用程序选择要处理请求状态时才会引发。

Event	on_rx_request() or on_rx_response()	on_tsx_state()
Receipt of new requests or responses	Called	Called
Receipt retransmissions of requests or responses.	Called ONLY when priority number is lower than transaction layer[1]	Not Called
Transmission of new requests or responses.	Not Called	Called
Retransmissions of requests or responses.	Not Called	Not Called
Transaction timeout	Not Called	Called
Other transaction failure events (e.g. DNS query failure, transport failure)	Not Called	Called

注：[1] 这是因为匹配的事务会阻止消息进一步分派(通过返回 PJ_TRUE)，且它在收到重传消息时不会调用 TU 顺调函数。

2.1.8 回调函数样例表

事务外和对话外的到来消息(从外进来的消息不属于任何事务或对话)



流程如下：

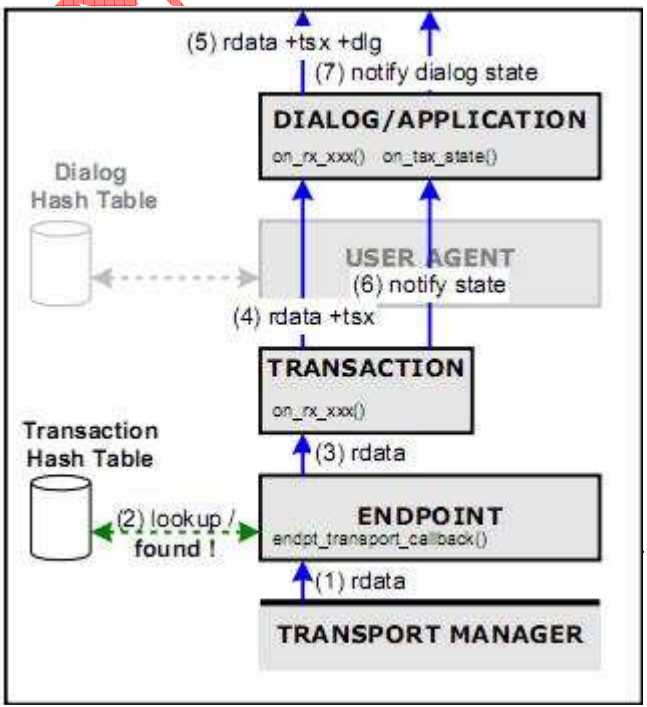
1) 传输管理器(pjsip_tpmgr)将所有收到的消息传递给 endpoint(在解析了以后)

2) Endpoint(pjsip_endpt)分派消息到所有的回调函数。回调函数列表的第一项是事务层(transaction layer)。事务层在事务表中查寻消息，没有找到匹配的事务。

3) Endpoint 继续分派消息到回调函数列表的下一个，这时是用户代理(user agent)。

4) 用户代理在对话表(dialog table)中查寻消息，没有找到匹配的对话集(dialog set) 。

5) Endpoint 继续分派消息到下一个注册的回调函数,直到应用程序。应用程序处理这个消息(如无状态地响应、创建 UAS 事务、proxy the request、建立对话等)



事务内的到来消息(Incoming Message Inside Transaction)

流程如下：

1) 传输管理器(pjsip_tpmgr)将所有收到的消息传递给 endpoint(在解析

了以后)

2) Endpoint(pjsip_endpt)分派消息到所有的回调函数。回调函数列表的第一项是事务层(transaction layer)。事务层在事务表中查寻消息，找到一个匹配的事务。

3) 因为事务的回调函数返回 PJ_TRUE，endpoint 不再继往下分派消息。

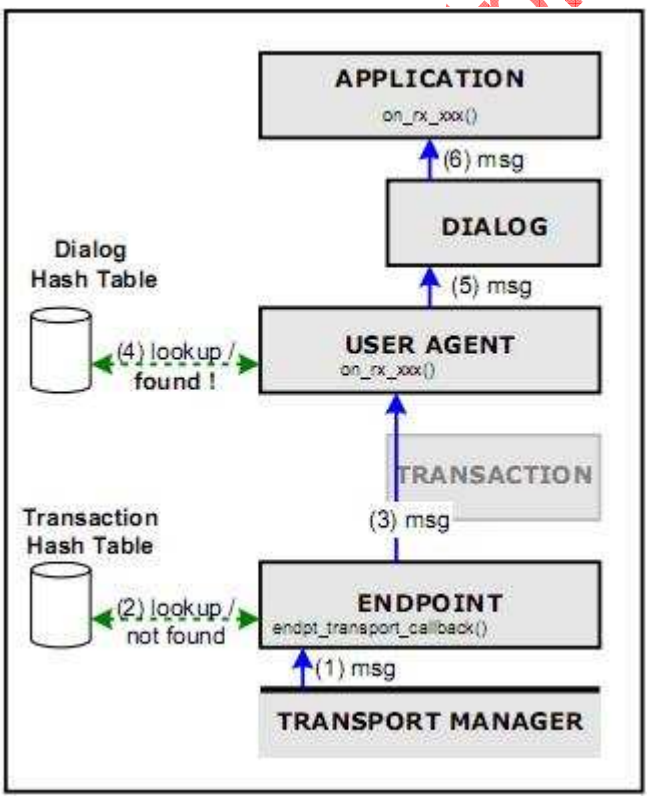
4) 事务开始处理响应(如更新 FSM)。如果这个水利中一个重传消息，处理就至此为止。否则事务将消息传递给事务用户(transaction user TU)，这可以是一个 dialog 或应用程序。

5) 如果 TU 是一个 dialog，这个 dialog 处理响应，然后传递响应到它的 Dialog User(DU,如应用程序)。

6) 如果消息的到来引起了事务状态的改变，事务会向 TU 通知它的新状态。

7) 如果 TU 是一个 dialog，它会进一步向应用程序通知 dialog 的状态改变。

对话内但事务外的到来消息(Incoming Message Inside Dialog but Outside Transaction)



流程如下：

1) 传输管理器(pjsip_tpmgr)将所有收到的消息传递给 endpoint(在解析了以后)

2) Endpoint(pjsip_endpt)分派消息到所有的回调函数。回调函数列表的第一项是事务层(transaction layer)。事务层在事务表中查寻消息，没有找到匹配的事务。

3) Endpoint 继续分派消息到回调函数列表的下一个，直到遇到用户代理模块(user agent module)。

4) 用户代理模块在 dialog hash table 中查寻消息的拥有者，并找到一个匹配的 dialog。

5) 用户代理将消息传递给这个找到的 dialog。

6) Dialog 总是为到来的请求创建事务，然后调用它的 dialog usages 的函数 on_rx_request() 和 on_tsx_state()来分派请求。

2.2 模块管理

模块由 PJSIP 的 endpoint(pjsip_endpoint)来管理。应用程序必须手动向 endpoint 注册每一个模块，这样模块才能被协议栈所识别。

2.2.1 模块管理 API

模块管理 API 在<pjsip/sip_endpt.h>中声明：

pj_status_t pjsip_endpt_register_module(pjsip_endpoint *endpt,
pjsip_module *module);

向 endpoint 注册一个模块。Endpoint 然后会调用模块的 load 和 start 函数来正确的初始化此模块，并为此模块分配一个唯一 ID。

pj_status_t pjsip_endpt_unregister_module(pjsip_endpoint *endpt, pjsip_module *module);

从 endpoint 卸载一个模块。Endpoint 会调用此模块的 stop 和 unload 函数来正确的关闭这个模块。

2.2.2 模块能力

模块可以向 endpoint 声明自己的新能力。当前 endpoint 管理着下面的能力：

- 允许的 SIP 方法(出现在 Allow 头域中)
- 支持的 SIP 扩展(出现在 Supported 头域中)

- 支持的内容类型(出现在 Accept 头域中)

这些头域将在合适的时候自动添加到外出的请求或响应消息中。

一个模块通过调用 pjsip_endpt_add_capability()方法声明自己的新能力：

```
pj_status_t pjsip_endpt_add_capability( pjsip_endpoint *endpt,
                                        pjsip_module *mod,
                                        int htype,
                                        const pj_str_t *hname,
                                        unsigned count,
                                        const pj_str_t tags[]);
```

向 endpoint 注册新能力。*Type* 参数指定向哪个头中添加这个新能力，可用的头有：PJSIP_H_ACCEPT, PJSIP_H_ALLOW, and PJSIP_H_SUPPORTED。*hname* 参数是可选的；它只用来指出头域中哪个能力是无法被核心协议栈所识别的。*count* 和 *tags* 参数包含要添加到头域中的标签字串符数组。

```
const pjsip_hdr* pjsip_endpt_get_capability( pjsip_endpoint *endpt,
                                             int htype,
                                             const pj_str_t *hname);
```

获取能力头域列表，其中包含了为指定头域已经向 endpoint 注册的所有能力。

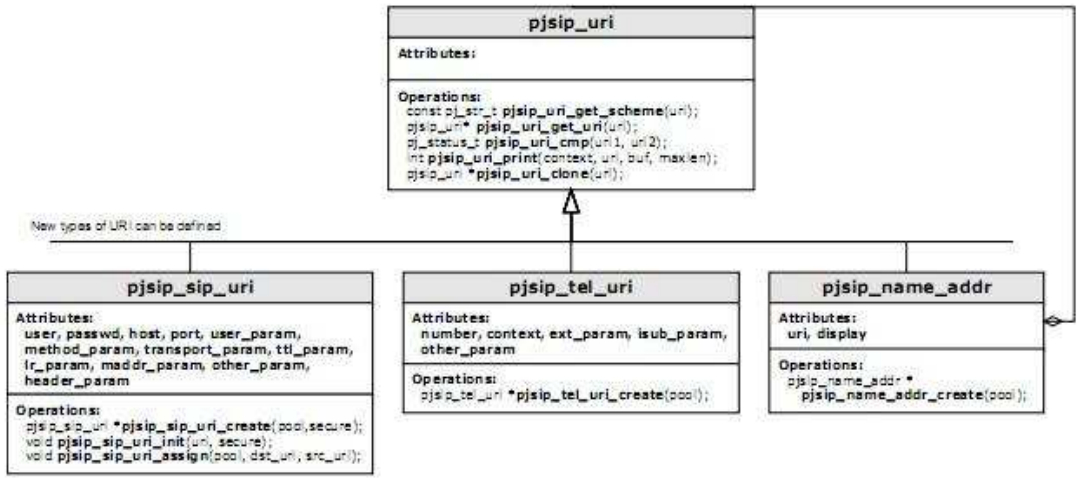
第三章 消息元素

3.1 统一资源指示器(Uniform Resource Indicator ,URI)

PJSIP 中的统一资源指示器(URI)几乎是仿照面向对象构造的(或有人说是基于对象的，不是面向对象的)。因此，URI 可以被协议栈统一对待，且新的 URI 类型也很容易引入进来。

3.1.1 URI“类图”

下图显示了 URI 对象是如何设计的：



每一个对象的更多信息将会在下面详细讲述。

3.1.2 URI 的上下文

URI 的上下文指定此 URI 在什么地方(如在请求行，在 From/To 头域，等等)使用。上下文指定在此时什么 URI 元素是允许出现的。例如，传输参数(transport parameter)是不允许出现在 From/To 头域中的。

在 PJSIP 中，上下文必须在要输出 URI 到缓冲区及比较两个 URI 时指定。这些情况下，在指定上下文中不允许出现的 URL 部分在处理输出 URI 到缓冲区及比较两个 URI 过程中被忽略。

```
enum pjsip_uri_context_e
{
    PJSIP_URI_IN_REQ_URI,       // The URI is in Request URI.
    PJSIP_URI_IN_FROMTO_HDR,    // The URI is in From/To header.
    PJSIP_URI_IN_CONTACT_HDR,   // The URI is in Contact header.
    PJSIP_URI_IN_ROUTING_HDR,   // The URI is in Route/Record-Route header.
    PJSIP_URI_IN_OTHER,         // Other context (web page, business card, etc.)
};
```

URI 上下文代码

3.1.3 基本的 URI

pjsip_uri 结构包含所有类型的 URI 必须有的属性。因为这样，所有类型的 URI 都可以类

型转换为 pjsip_uri，且操作方式统一。

```
struct pjsip_uri
{
   pjsip_uri_vptr *vptr;
};
```

URI 通用声明

pjsip_uri_vptr 指定了“虚拟”函数表，其中的成员由每一种类型的 URI 定义。不推荐应用程序直接访问这些函数指针；应用程序应该使用 URI 的 API，因为它们可读性更好(同时也可减少输入)。

```
struct pjsip_uri_vptr
{
   const pj_str_t* (*p_get_scheme)  ( const pjsip_uri *uri);
   pjsip_uri*      (*p_get_uri)     ( pjsip_uri *uri);
   int             (*p_print)       ( pjsip_uri_context_e context,
                                      const pjsip_uri *uri,
                                      char *buf, pj_size_t size);
   pj_status_t     (*p_compare)     ( pjsip_uri_context_e context,
                                      const pjsip_uri *uri1, const pjsip_uri *uri2);
   pjsip_uri *     (*p_clone)       ( pj_pool_t *pool, const pjsip_uri *uri);
};
```

URI 虚拟函数表

下面的函数可以应用到所有类型的 URI 对象。这些函数通常实现为内联函数，内联函数调用 URI 虚拟函数表中对应的函数指针。

- **const pj_str_t* pjsip_uri_get_scheme(const pjsip_uri *uri);**

  获取 URI 模式字符串(如 sip,sips,tel 等等)

- **pjsip_uri* pjsip_uri_get_uri(pjsip_uri *uri);**

  获取 URI 对象。通常所有 URI 对象会返回它自身，例外是命名地址(name address)会返回命名地址对象内的 URI。

- **pj_status_t pjsip_uri_cmp(pjsip_uri_context_e context, const pjsip_uri *uri1, const pjsip_uri *uri2);**

  依据指定的上下文比较 uri1 和 uri2。在指定上下文中不允许的参数在比较过程中会被忽略。如果两个 URI 相等，此函数会返回 PJ_SUCCESS。

- **int pjsip_uri_print(pjsip_uri_context_e context, const pjsip_uri *uri,char *buffer,**

pj_size_t max_size);

依据指定的上下文将 URI 格式化输出到指定的缓冲区。在指定上下文中不允许出现的参数将不会包含在格式化输出过程。

- **pjsip_uri* pjsip_uri_clone(pj_pool_t *pool, const pjsip_uri *uri);**

  使用指定的内存池对 uri 进行深度克隆。

3.1.4 SIP 和 SIPS URI

结构 pjsip_sip_uri 表现了 SIP 和 SIPS URI 模式(scheme)，它在<pjsip/sip_uri.h>中声明：

```
struct pjsip_sip_uri
{
   pjsip_uri_vptr *vptr;            // Pointer to virtual function table.
   pj_str_t        user;            // Optional user part.
   pj_str_t        passwd;          // Optional password part.
   pj_str_t        host;            // Host part, always exists.
   int             port;            // Optional port number, or zero.
   pj_str_t        user_param;      // Optional user parameter
   pj_str_t        method_param;    // Optional method parameter.
   pj_str_t        transport_param; // Optional transport parameter.
   int             ttl_param;       // Optional TTL param, or -1.
   int             lr_param;        // Optional loose routing param, or 0
   pj_str_t        maddr_param;     // Optional maddr param
   pjsip_param     other_param;     // Other parameters as list.
   pjsip_param     header_param;    // Optional header parameters as list.
};
```

SIP URI 声明

下面列出的函数是特定于 SIP/SIPS URI 对象的。除了这些函数外，应用程序还可以使用由上一节描述的基本 URI 函数来操作 SIP/SIPS URI。

- **pjsip_sip_uri* pjsip_sip_uri_create(pj_pool_t *pool, pj_bool_t secure);**

  使用指定的内存池创建一个新的 SIP URL。如果 *secure* 标志设置为非 0，那么创建的是 SIPS URL。这个函数会设置 SIP 或 SIPS URL 虚拟表成员 vptr,并设置其它成员为空值。

- **void pjsip_sip_uri_init(pjsip_sip_uri *url, pj_bool_t secure);**

  初始化一个 SIP URL 结构。

- **void pjsip_sip_uri_assign(pj_pool_t *pool, pjsip_sip_uri *url, const pjsip_sip_uri *rhs);**

  执行深度拷贝。将 rhs 拷贝到 url。

3.1.5 电话 URI(Tel URI)

结构 pjsip_tel_uri 表现了 tel: URL，它声明在<pjsip/sip_tel_uri.h>中：

```
struct pjsip_tel_uri
{
   pjsip_uri_vptr *vptr;            // Pointer to virtual function table.
   pj_str_t        number;          // Global or local phone number
   pj_str_t        context;         // Phone context (for local number).
   pj_str_t        ext_param;       // Extension param.
   pj_str_t        isub_param;      // ISDN sub-address param.
   pjsip_param     other_param;     // Other parameters.
};
```

Tel URI 声明

下面列出的函数是特定于 Tel URI 对象的。除了这些函数外，应用程序还可以使用由上一节描述的基本 URI 函数来操作 Tel URI。

- **pjsip_tel_uri* pjsip_tel_uri_create(pj_pool_t *pool);**

  创建一个新的 Tel : URI。

- **int pjsip_tel_nb_cmp(const pj_str_t *nb1, const pj_str_t *nb2);**

  这个工具函数对两个电话号码(telephone numbers)依据 RFC 3966 中定义的规则进行相等性比较。在比较过程中，它可以识别全球和本地号码，并忽略视觉分隔符(visual separators)。

3.1.6 命名地址(Name Address)

一个命名地址(pjsip_name_addr)并不是真正创建一个新类型的 URI，只是封装了已有的 URI(例如 SIP URI)并添加了显示名字。

```
struct pjsip_name_addr
{
   pjsip_uri_vptr  *vptr;           // Pointer to virtual function table.
   pj_str_t         display;        // Display name.
   pjsip_uri       *uri;            // The URI.
};
```

命名地址声明

下面列出的函数是特定于命名地址对象的。除了这些函数外，应用程序还可以使用由上一节描述的基本 URI 函数来操作命名地址。

- **pjsip_name_addr* pjsip_name_addr_create(pj_pool_t *pool);**

  创建一个新的命名地址。这将会初始化虚拟函数表指针，将显示名字设置这空，设置 uri 成员为 NULL。

- **void pjsip_name_addr_assign(pj_pool_t *pool, pjsip_name_addr *name_addr, const pjsip_name_addr *rhs);**

  将 rhs 拷贝到 name_addr。

3.1.7 URI 操作程序样例

```
#include <pjlib.h>
#include <pjsip_core.h>
#include <stdlib.h>           // exit()

static pj_caching_pool cp;

static void my_error_exit(const char *title, pj_status_t errcode)
{
   char errbuf[80];
   pjsip_strerror(errcode, errbuf, sizeof(errbuf));
   PJ_LOG(3,("main", "%s: %s", title, errbuf));
   exit(1);
}

static void my_init_pjlib(void)
{
   pj_status_t status;
   // Init PJLIB
   status = pj_init();
   if (status != PJ_SUCCESS) my_error_exit("pj_init() error", status);
   // Init caching pool factory.
   pj_caching_pool_init( &cp, &pj_pool_factory_default_policy, 0);
}

static void my_print_uri( const char *title, pjsip_uri *uri )
{
   char buf[80];
   int len;

   len = pjsip_uri_print( PJSIP_URI_IN_OTHER, uri, buf, sizeof(buf)-1);
   if (len < 0)
      my_error_exit("Not enough buffer to print URI", -1);

   buf[len] = '\0';
   PJ_LOG(3, ("main", "%s: %s", title, buf));
}

int main()
{
   pj_pool_t *pool;
   pjsip_name_addr *name_addr;
   pjsip_sip_uri *sip_uri;

   // Init PJLIB
   my_init_pjlib();

   // Create pool to allocate memory
   pool = pj_pool_create(&cp.factory, "mypool", 4000, 4000, NULL);
   if (!pool) my_error_exit("Unable to create pool", PJ_ENOMEM);

   // Create and initialize a SIP URI instance
   sip_uri = pjsip_sip_uri_create(pool, PJ_FALSE);
   sip_uri->user = pj_str("alice");
   sip_uri->host = pj_str("sip.example.com");

   my_print_uri("The SIP URI is", (pjsip_uri*)sip_uri);

   // Create a name address to put the SIP URI
   name_addr = pjsip_name_addr_create(pool);
   name_addr->uri = (pjsip_uri*) sip_uri;
   name_addr->display = "Alice Cooper";

   my_print_uri("The name address is", (pjsip_uri*)name_addr);
   // Done
}
```

3.2 SIP 方法

3.2.1 SIP 方法表现(pjsip_method)

在 PJSIP 中 SIP 方法的表现也是可以扩展的。它在不需要重新编译库的情况下就可以支持新的方法。

```
struct pjsip_method
{
   pjsip_method_e id;      // Method ID, from pjsip_method_e.
   pj_str_t       name;    // Method name, which will always contain the method string.
};
```

SIP 方法声明

PJSIP 核心库只声明了 SIP 核心标准(RFC 3261)中指明的方法。对于这些核心的方法，pjsip_method 结构中的 id 字段包含下列枚举的合适的值：

```
enum pjsip_method_e
{
   PJSIP_INVITE_METHOD,
   PJSIP_CANCEL_METHOD,
   PJSIP_ACK_METHOD,
   PJSIP_BYE_METHOD,
   PJSIP_REGISTER_METHOD,
   PJSIP_OPTIONS_METHOD,

   PJSIP_OTHER_METHOD,
};
```

SIP 方法 ID 枚举

对于 ID 枚举中没有指定的方法，pjsip_method 结构的 id 字段将包 PJSIP_OTHER_METHOD 值。这种情况下，应用程序必须检查 pjsip_method 结构的 name 字段以确定实际的方法。

3.2.2 SIP 方法 API

下面列出的方法可以用来操作 PJSIP 的 SIP 方法对象。

- **void pjsip_method_init(pjsip_method *method, pj_pool_t *pool,const pj_str_t**

*method_name);

从一个字符串初始一个 SIP 方法。这将初始化方法的 ID 字段为正确的值。

- void pjsip_method_init_np(pjsip_method *method,pj_str_t *method_name);

  在不复制字符串的情况下从 *method_name* 参数初始化一个 SIP 方法。np 代表着 no pool。Id 字段将相应地进行初始化。

- void pjsip_method_set(pjsip_method *method, pjsip_method_id_e method_id);

  依据方法 ID 枚举初始化一个 SIP 方法。*Namep* 字段相应地进行初始化。

- void pjsip_method_copy(pj_pool_t *pool, pjsip_method *method, const pjsip_method *rhs);

  拷贝 *rhs* 到 *method*。

- int pjsip_method_cmp(const pjsip_method *method1,const pjsip_method *method2);

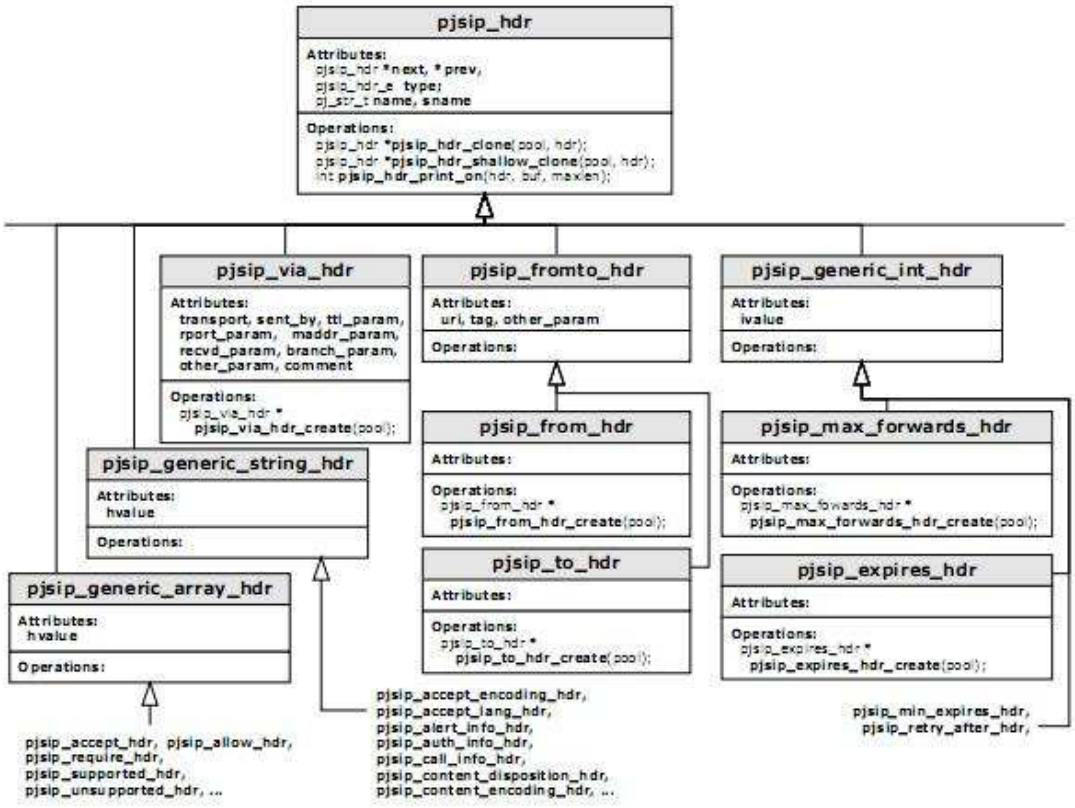
  比较方法 *method1* 与方法 *method2* 的相等性。如果两个方法相等，函数将返回 0 值 ;如果方法 *method1* 比 *method2* 小 ,则返回-1 值 ;如果方法 *method1* 比 *method2* 大，则返回 1 值。

3.3 头域(Header Fields)

PJSIP 中的所有头域都有相同的头属性(header properties)，如头类型(header type)、名字(name)、短名字(short name)以及虚拟方法表。因此，在协议栈中所有的头域都可以统一处理。

3.3.1 头的“类图”

下图显示了 PJSIP 头域的类图片段(the snippet of PJSIP header “class diagram”)。在图中显示的域是只是其中一部分。PJSIP 库实现了核心 SIP 规范(RFC 3261)中指定的所有头域。其它的头域将会在相应的 PJSIP 扩展模块中实现。



头域“类图”

可以从类图中看出，每一个特定的头域通常只提供特定于此头域的方法。例如，创建头域实例的方法。

3.3.2 头域结构(Header Structure)

为了保证头域包含共同的头属性，且这些属性的内存布局是正确并相同的，头的声明必须将 PJSIP_DECL_HDR_MEMBER 宏作为第一个头字段成员来调用，并指定头名字为此宏的参数。

```
#define PJSIP_DECL_HDR_MEMBER(hdr)              \
            /** List members. */                \
            PJ_DECL_LIST_MEMBER(hdr);           \
            /** Header type */                  \
            pjsip_hdr_e type;                   \
            /** Header name. */                 \
            pj_str_t name;                      \
            /** Header short name version. */   \
            pj_str_t sname;                     \
            /** Virtual function table. */      \
            pjsip_hdr_vptr *vptr
```

通用头声明

PJSIP 定将了 pjsip_hdr 结构，此结构包含所有头域都有的共有属性。因此，所有头域都可类型转换为 pjsip_hdr，所有它们的操作方式是一致的。

```
struct pjsip_hdr
{
PJSIP_DECL_HDR_MEMBER(struct pjsip_hdr);
};
```

3.3.3 通用的头方法

结构 pjsip_hdr_vptr 指定了"虚拟"方法表，由每一个头类型来实现。"虚拟"方法表包含的方法如下：

```
struct pjsip_hdr_vptr
{
    pjsip_hdr *(*clone)         ( pj_pool_t *pool, const pjsip_hdr *hdr );
    pjsip_hdr *(*shallow_clone)( pj_pool_t *pool, const pjsip_hdr *hdr );
    int        (*print_on)      ( pjsip_hdr *hdr, char *buf, pj_size_t len );
};
```

头虚拟方法表(Header Virtual Function Table)

尽管应用程序可以自由且直接地调用 pjsip_hdr_vptr 中的函数指针，但推荐的方法是调用如下的头 API,因为这样程序可读性更好。

- **pjsip_hdr *pjsip_hdr_clone(pj_pool_t *pool, const pjsip_hdr *hdr);**

  对 hdr 头进行深度拷贝。

- **pjsip_hdr *pjsip_hdr_shallow_clone(pj_pool_t *pool, const pjsip_hdr *hdr);**

  对 hdr 头进行浅拷贝。浅拷贝创建一个新的头域对象，其值是从源对象拷贝过来的。

然而，大多数值仍指向源头域的值。通常浅拷贝只是使用 memcpy()函数从源对象拷贝数据到新对象，因此此操作比深拷贝要快。

但是要留心浅拷贝。必须理解新的头域和源头域的指针都指向相同的值。因此，当包含源头域的内存池销毁后，尽管使用不同的内存池进行了浅拷贝，但新对象仍然变得无效。或如果源头域的某些值被修改了，那么相应的新头域的值也修改了。

尽管如此，浅拷贝在库中仍然使用得比较广泛。例如，在会话中一个 dialog 拥有的头字段值或多或少是持久的（如 From,To,Call-Id,Route,Contact），当建立一个请求消息时，dialog 可以浅拷贝这些头域并放到请求消息中(而不是进行完深拷贝)。

- int pjsip_hdr_print_on(pjsip_hdr *hdr, char *buf,pj_size_t max_size);
  格式输出头域到缓冲区中(如在传输前)。这个函数返回格式输出到缓冲区后的字节数，或如果缓冲区溢出时，返回-1。

3.3.4 Supported 头域(Supported Header Fields)

“标准”的 PJSIP 头域声明在<pjsip/sip_msg.h>。其它的头域可能放在与实现特定功能或扩展相关的头文件中（如 SIMPLE extension 使用的头文件）。

每一个头域通常只定义一个操作此头域的特定 API，例如创建特定头域的函数。其它 API 是由虚拟函数表导出的。

创建各自头域 的 API 在规范上的命名是在头域名称后面加上_create 后缀。例如，调用函数 pjsip_via_hdr_create()将创建一个 pjsip_via_hdr 实例。

请参考<pjsip/sip_msg.h>以获得 PJSIP 核心定义的头域完全列表。

3.3.5 头数组元素(Header Array Elements)

很多 SIP 头（例如 Reuqire、Contact、Via 等 ）可以分组成一个单一的头域和由逗号分

隔的内容，例如：

```
Contact: <sip:alice@sip.example.com>;q=1.0, <tel:+442081234567>;q=0.5
Via: SIP/2.0/UDP proxy1.example.com;branch=z9hG4bK87asdks7, SIP/2.0/UDP
     proxy2.example.com;branch=z9hG4bK77asjd
```

当解析器在头中遇到这样的数组时，它会将数组分解为多个单独的头，并维护他们出现的顺序。因此对于上面的例子，解析器会修改消息成如下的样子：

```
Contact: <sip:alice@sip.example.com>;q=1.0
Contact: <tel:+442081234567>;q=0.5
Via: SIP/2.0/UDP proxy1.example.com;branch=z9hG4bK87asdks7
Via: SIP/2.0/UDP proxy2.example.com;branch=z9hG4bK77asjd
```

SIP 标准规范指出在处理消息时，对于上面的两种表现形式不存在任何差异。因此我们相信对头数组支持的移除(removal of header array support)完全不会限制 PJSIP 的功能。

我们强加这种限制和理由是，基于我们的经验，对头数组支持的移除会大大简化对头域的处理。如果对头数组进行支持，程序不仅必须检查所有的头域，而且不得不检查某些头域以查看它们是否包含数组。移除头数组支持后，程序只需检查消息中的主要头域。

3.4 消息体(pjsip_msg_body)

SIP 消息体在 PJSIP 表现为 pjsip_msg_body 结构。这个结构定义在<pjsip/sip_msg.h>中。

```
struct pjsip_msg_body
{
    /** MIME content type.
     *  For incoming messages, the parser will fill in this member with the
     *  content type found in Content-Type header.
     *
     *  For outgoing messages, application must fill in this member with
     *  appropriate value, because the stack will generate Content-Type header
     *  based on the value specified here.
     */
    pjsip_media_type content_type;

    /** Pointer to buffer which holds the message body data.
     *  For incoming messages, the parser will fill in this member with the
     *  pointer to the body string.
     *
     *  When sending outgoing message, this member doesn't need to point to the
     *  actual message body string. It can be assigned with arbitrary pointer,
     *  because the value will only need to be understood by the print_body()
     *  function. The stack itself will not try to interpret this value, but
     *  instead will always call the print_body() whenever it needs to get the
     *  actual body string.
     */
    void *data;

    /** The length of the data.
     *  For incoming messages, the parser will fill in this member with the
     *  actual length of message body.
     *
     *  When sending outgoing message, again just like the "data" member, the
     *  "len" member doesn't need to point to the actual length of the body
     *  string.
     */
    unsigned len;

    /** Pointer to function to print this message body.
     *  Application must set a proper function here when sending outgoing
     *  message.
     *
     *  @param msg_body       This structure itself.
     *  @param buf            The buffer.
     *  @param size           The buffer size.
     *
     *  @return               The length of the string printed, or -1 if there is
     *                        not enough space in the buffer to print the whole
     *                        message body.
     */
    int (*print_body)         ( struct pjsip_msg_body *msg_body,
                                char *buf, pj_size_t size );

    /** Pointer to function to clone the data in this message body.
     */
    void* (*clone_data)       ( pj_pool_t *pool, const void *data, unsigned len );
};
```

消息体定义

下面的函数用来操作 SIP 消息体对象：

- **pj_status_t pjsip_msg_body_clone(pj_pool_t *pool,pjsip_msg_body *dst_body,const pjsip_msg_body *src_body);**

  将消息体从 *src_body* 克隆到 *dst_body*。这将复制(duplicate)源消息体成员 *clone_data* 中的消息内容。

3.5 消息(pjsip_msg)

PJSIP 中的请求(request)和响应(response)消息均表现为<pjsip/sip_msg.h>中的 pjsip_msg 结构。下面的代码片段显示了结构 pjsip_mg 的声明及其它的支持结构体。

```
enum pjsip_msg_type_e
{
   PJSIP_REQUEST_MSG,          // Indicates request message.
   PJSIP_RESPONSE_MSG,         // Indicates response message.
};

struct pjsip_request_line
{
   pjsip_method  method;       // Method for this request line.
   pjsip_uri    *uri;          // URI for this request line.
};

struct pjsip_status_line
{
   int           code;         // Status code.
   pj_str_t      reason;       // Reason string.
};

struct pjsip_msg
{
   /** Message type (ie request or response). */
   pjsip_msg_type_e  type;

   /** The first line of the message can be either request line for request
    *  messages, or status line for response messages. It is represented here
    *  as a union.
    */
   union
   {
      /** Request Line. */
      struct pjsip_request_line   req;

      /** Status Line. */
      struct pjsip_status_line    status;

   } line;

   /** List of message headers. */
   pjsip_hdr hdr;

   /** Pointer to message body, or NULL if no message body is attached to
    *  this message.
    */
   pjsip_msg_body *body;
};
```

SIP 消息声明

下面的函数用来操作 SIP 消息对象：

- **pjsip_msg* pjsip_msg_create(pj_pool_t *pool, pjsip_msg_type_e type);**

  依据类型创建一个请求或响应消息。

- **pjsip_hdr* pjsip_msg_find_hdr(pjsip_msg *msg,pjsip_hdr_e hdr_type,pjsip_hdr**

***start);**

从指定的头列表位置 *start* 开始，在消息中寻找指定类型的头。如果 *start* 是 NULL，函数从消息的第一个头开始查找。如果在指定位置及其后没有找到相应的头，则返回 NULL。

- **pjsip_hdr* pjsip_msg_find_hdr_by_name(pjsip_msg *msg,const pj_str_t *name,pjsip_hdr *start);**

  从指定的头列表位置 *start* 开始，在消息中寻找指定名称的头，以长名称和短名称两种方式寻找。如果 *start* 是 NULL,函数从消息的第一个头开始查找。如果在指定位置及其后没有找到相应的头，则返回 NULL。

- **void pjsip_msg_add_hdr(pjsip_msg *msg,pjsip_hdr *hdr);**

  将 *hdr* 作为最后一个头添加到 *msg* 中。

- **void pjsip_msg_insert_first_hdr(pjsip_msg *msg,pjsip_hdr *hdr);**

  将 *hdr* 作为第一个头添加到 *msg* 中。

- **pj_ssize_t pjsip_msg_print(pjsip_msg *msg,char *buf,pj_size_t size);**

  格式输出整个消息内容到缓冲区中。这个函数返回格式输出到缓冲区后的字节数，或如果缓冲区溢出时，返回-1。

3.6 SIP 状态码(SIP Status Codes)

由 SIP 核心规范(RFC 3261)定义的 SIP 状态码在 PJSIP 中表现为<pjsip/sip_msg.h>中的枚举值 pjsip_status_code。另外，缺省的状态文字(reason text)可以由调用 pjsip_get_status_text()函数来获取。

下面的代码片段显示了 PJSIP 中的状态码声明。

PJSIP 也定义了新的状态码(如 7xx)，用于消息处理(如传输错误，DNS 错误)过程中的致命错误。这类状态码只是在内部使用，不会用于消息传输。

```
enum pjsip_status_code
{
    PJSIP_SC_TRYING = 100,
    PJSIP_SC_RINGING = 180,
    PJSIP_SC_CALL_BEING_FORWARDED = 181,
    PJSIP_SC_QUEUED = 182,
    PJSIP_SC_PROGRESS = 183,

    PJSIP_SC_OK = 200,

    PJSIP_SC_MULTIPLE_CHOICES = 300,
    PJSIP_SC_MOVED_PERMANENTLY = 301,
    PJSIP_SC_MOVED_TEMPORARILY = 302,
    PJSIP_SC_USE_PROXY = 305,
    PJSIP_SC_ALTERNATIVE_SERVICE = 380,

    PJSIP_SC_BAD_REQUEST = 400,
    PJSIP_SC_UNAUTHORIZED = 401,
    PJSIP_SC_PAYMENT_REQUIRED = 402,
    PJSIP_SC_FORBIDDEN = 403,
    PJSIP_SC_NOT_FOUND = 404,
    PJSIP_SC_METHOD_NOT_ALLOWED = 405,
    PJSIP_SC_NOT_ACCEPTABLE = 406,
    PJSIP_SC_PROXY_AUTHENTICATION_REQUIRED = 407,
    PJSIP_SC_REQUEST_TIMEOUT = 408,
    PJSIP_SC_GONE = 410,
    PJSIP_SC_REQUEST_ENTITY_TOO_LARGE = 413,
    PJSIP_SC_REQUEST_URI_TOO_LONG = 414,
    PJSIP_SC_UNSUPPORTED_MEDIA_TYPE = 415,
    PJSIP_SC_UNSUPPORTED_URI_SCHEME = 416,
    PJSIP_SC_BAD_EXTENSION = 420,
    PJSIP_SC_EXTENSION_REQUIRED = 421,
    PJSIP_SC_INTERVAL_TOO_BRIEF = 423,
    PJSIP_SC_TEMPORARILY_UNAVAILABLE = 480,
    PJSIP_SC_CALL_TSX_DOES_NOT_EXIST = 481,
    PJSIP_SC_LOOP_DETECTED = 482,
    PJSIP_SC_TOO_MANY_HOPS = 483,
    PJSIP_SC_ADDRESS_INCOMPLETE = 484,
    PJSIP_AC_AMBIGUOUS = 485,
    PJSIP_SC_BUSY_HERE = 486,
    PJSIP_SC_REQUEST_TERMINATED = 487,
    PJSIP_SC_NOT_ACCEPTABLE_HERE = 488,
    PJSIP_SC_REQUEST_PENDING = 491,
    PJSIP_SC_UNDECIPHERABLE = 493,

    PJSIP_SC_INTERNAL_SERVER_ERROR = 500,
    PJSIP_SC_NOT_IMPLEMENTED = 501,
    PJSIP_SC_BAD_GATEWAY = 502,
    PJSIP_SC_SERVICE_UNAVAILABLE = 503,
    PJSIP_SC_SERVER_TIMEOUT = 504,
    PJSIP_SC_VERSION_NOT_SUPPORTED = 505,
    PJSIP_SC_MESSAGE_TOO_LARGE = 513,

    PJSIP_SC_BUSY_EVERYWHERE = 600,
    PJSIP_SC_DECLINE = 603,
    PJSIP_SC_DOES_NOT_EXIST_ANYWHERE = 604,
    PJSIP_SC_NOT_ACCEPTABLE_ANYWHERE = 606,

    PJSIP_SC_TSX_TIMEOUT = 701,
    PJSIP_SC_TSX_RESOLVE_ERROR = 702,
    PJSIP_SC_TSX_TRANSPORT_ERROR = 703,
};

/* Get the default status text for the status code.  */
const pj_str_t* pjsip_get_status_text(int status_code);
```

SIP 状态码常量

3.7 非标准参数元素(Non-Standard Parameter Elements)

在 PJSIP 中，已知的或"标准的"参数(例如,URI 参数头域参数)通过表现为相应结构的单独属性/字段。非标准的参数将放到一个参数列表中，每一个表现为 pjsip_param 结构。非标准参数通常在父结构(owning structure)中声明为 other_param 字段。

3.7.1 数据结构表现(Data Structure Representation)

结构 pjsip_param 描述了列表中的每一个单独的参数：

```
struct pjsip_param
{
   PJ_DECL_LIST_MEMBER(struct pjsip_param);  // Generic list member.
   pj_str_t     name;                        // Param/header name.
   pj_str_t     value;                       // Param/header value.
};
```

非标准参数声明(Non-Standard Parameter Declaration)

对此结构用法的例子，请参见 pjsip_sip_uri 声明(3.1.4 "SIP and SIPS URI")中的 other_param 和 header_param 字段或 pjsip_tel_uri 声明(3.1.5 "Tel URI")中 other_param 字段。

3.7.2 非标准参数操作

存在一些函数来辅助操作参数列表中的非标准参数。

- **pjsip_param* pjsip_param_find(const pjsip_param *param_list, const pj_str_t *name);**

  依据指定的参数名称(parameter name)执行大小写不敏感的查寻。

- **void pjsip_param_clone(pj_pool_t *pool, pjsip_param *dst_list, const pjsip_param *src_list);**

  对参数列表(parameter list)执行浅克隆。

- **pj_ssize_t pjsip_param_print_on(const pjsip_param *param_list, char *buf, pj_size_t max_size, const pj_cis_t *pname_unres, const pj_cis_t *pvalue_unres, int sep);**

格式化输出参数列表到指定的缓冲区。*pname_unres* 和 *pvalue_unres* 分别指定 *pname* 和 *pvalue* 中哪些字符允许不被转义地出现。没有指定的字符将被此方法转义。此方法 的参数 *sep* 指定 paramters 之间的分隔符(通常对于普通参数是分号";" 或对于头域参数是逗号",")。

3.7.3 转义规则

PJSIP 提供在解析过程中的自动非转义实现(automatic un-escapement)和格式化过程中仅对以下消息元素的转义实现：

- 所有类型的 URI 及它们的元素依据它们单独的转义规则自动进行转义或非转义。
- 出现在消息元素中的参数(如 URL、头域等等) 自动进行转义或非转义。

其它消息元素将由协议栈原样地、不加解释地(un-interpreted)传递。

第四章 解析器(Parser)

4.1 特性(Features)

PJSIP 解析器的一些特性：

- 它是自顶而下(top-down)，全新手写(handwritten)的解析器。它使用了 PJLIB 的扫描器(scanner)，此扫描器非常快，并减轻了解析器的复杂性。因此使解析器可读性更好。
- 如上所说，它非常快。
- 它是可重入的。这使它在多处理器的机器上有规模扩展性(scalable)。
- 它具有功能扩展性(extensible)。可以使用模块来向解析器插入新的头域或 URI 类型。

解析器采用了许多可以想到的技巧来获取最高的性能(The parser features almost a lot of tricks thinkable to achieve the highest performance),例如：

- 它对所有的消息元素使用零拷贝。即当一个元素(例如 *pvalue*)解析后，解析器并不不会拷贝 *pvalue* 的所有内容到消息中的合适字段中，而是只将指针和长度放到消息中的合适字段中。因为 PJSIP 在整个库中使用 pj_str_t 类型,不需要字符串以 NULL 结尾。
- 它使用使用 PJSIP 的内存池(pj_pool_t)为消息结构分配内存。这比传统的 malloc()函数快了数倍。
- 它使用零同步(zero synchronization)。解析器是完全可重入的,因此不需要同步功能。
- 它使用 PJLIB 的 try/catch 异常框架，这不仅大大简化了解析器及增强了可读性，而且减少了解析器的的错误检查步骤。使用异常框架,只需在解析器的最顶部安装一个异常处理器即可。

PJSIP 没有实现的一个特性是迟解析(lazy parsing)。很多人都在吹嘘它的使用。在早期的设计阶段，我们决定不实现迟解析，因为以下几个原因：

- 它将使事情变得复杂，特别是错误处理。使用迟解析后,基本上程序的各个部分在后期阶段解析器解析失败且程序需要访问特定消息元素时都要准备处理错误情况。
- 后来,我们相信 PJSIP 解析器无论如何都很快而不再需要迟解析。尽管这样说,PJSIP 解析器也有一些开关可以打开，这样对一些应用可以忽略对一些头(ignore parsing of some headers for some type of applications)的解析。例如，代理，它不需要检查一些头类型。

4.2 函数

PJSIP 的主要解析器声明在<pjsip/sip_parser.h>，定义在<pjsip/sip_parser.c>。库的其它部分可能提供另外的解析函数及解析器的扩展()例如<pjsip/sip_tel_uri.c>提供了解析 TEL URI 的函数并注册此函数到主解析器)。

4.2.1 消息解析

- **pj_status_t pjsip_find_msg(const char *buf, pj_size_t size, pj_bool_t is_datagram, pj_size_t *msg_size);**

  检查 *buf* 中的到来包中是否包含的效的 SIP 消息。如果检测到是有效的 SIP 消息，*msg_size* 中将是消息大小。如果指定了参数 *is_datagram*，这个函数将返回 PJ_SUCCESS。

  注意：此函数期望 *buf* 缓冲区是以 NULL 结尾的。

- **pjsip_msg* pjsip_parse_msg(pj_pool_t *pool, char *buf, pj_size_t size,pjsip_parser_err_report *err_list);**

  将缓冲区 *buf* 解析成 SIP 消息。如果至少 SIP 请求/响应行(request/status line)已经解析成功,那么解析器将返回此消息。如果参数 *err_list* 不为空，那么将包含解析中遇到的错误。

  注意：此函数期望 *buf* 缓冲区是以 NULL 结尾的。

- **pjsip_msg* pjsip_parse_rdata(char *buf, pj_size_t size,pjsip_rx_data *rdata);**

  将缓冲区 *buf* 解析成 SIP 消息。如果至少 SIP 请求/响应行(request/status line)已经解析成功,那么解析器将返回此消息。另外，此函数将更新 *rdata* 的 *msg_info* 部分中的 *headers* 指针。

  注意：此函数期望 *buf* 缓冲区是以 NULL 结尾的。

4.2.2 URI 解析

- **pjsip_uri* pjsip_parse_uri(pj_pool_t *pool, char *buf, pj_size_t size,unsigned option);**

  将缓冲区 *buf* 解析成 SIP URI。如果在 *option* 中指定了 PJSIP_PARSE_URI_AS_NAMEADDR，函数总是将 URI 包装成命名地址；如果在 *option* 中指定了 PJSIP_PARSE_URI_IN_FROM_TO_HDR，且 URI 没有包含在括号中，函数将不会解析 URI 之后 的参数(它将认为是头参数 header parameters ,而不是 URI 参数)。此函数可以解析任何 PISIP 库可以识别的 URI 类型，并依据模式(scheme)可以返回正确的 URI 实例。

  注意：此函数期望 *buf* 缓冲区是以 NULL 结尾的。

4.2.3 头解析(Header Parsing)

- **void* pjsip_parse_hdr(pj_pool_t *pool, const pj_str_t *hname,char *line, pj_size_t size,int *parsed_len);**

  依据头类型 *hname*，以行的方式解析头的内容(即冒号后的头的部分)。它返回合适的头实例。

  注意：此函数期望 *buf* 缓冲区是以 NULL 结尾的。

- **pj_status_t pjsip_parse_headers(pj_pool_t *pool,char *input, pj_size_t size,pj_list *hdr_list);**

  解析在 *input* 中发现的多个头，并将结果放到 *hdr_list* 中。此函数期望头或以换行符分隔(如在 SIP 消息中)或以&分隔(如在 URI 中)。最后一个 header 的分隔符是可选的。

  注意：此函数期望 *input* 缓冲区是以 NULL 结尾的。

4.3 扩展解析器(Extending Parser)

解析器可以以注册新的函数指针(function pointers)来解析新类型的头域或新类型的 URI。

- **typedef pjsip_hdr* (pjsip_parse_hdr_func)(pjsip_parse_ctx *context);**
- **pj_status_t pjsip_register_hdr_parser(const char *hname,const char *hshortname,pjsip_parse_hdr_func *fptr);**

  注册新的函数来解析新类型的 SIP 头域。

- **typedef void* (pjsip_parse_uri_func)(pj_scanner *scanner, pj_pool_t *pool,pj_bool_t parse_params);**
- **pj_status_t pjsip_register_uri_parser(char *scheme,pjsip_parse_uri_func *func);**

  注册新的函数来解析新类型的 SIP URI 模式(scheme)。

第五章 消息缓冲区(Message Buffers)

5.1 接受数据缓冲区(Receive Data Buffer)

一个 PJSIP 接受到的 SIP 消息，将以 pjsip_rx_data 类型而不是简单的消息本身，传递到 PJSIP 的不同软件组件。这个结构(pjsip_rx_data)包含所有描述接受到的消息的所有信息。

接受和传输(transmit)数据缓冲区声明在<pjsip/sip_transport.h>。

5.1.1 接受数据缓冲区结构(Receive Data Buffer Structure)

```
struct pjsip_rx_data
{
   // This part contains static info about the buffer.
   struct
   {
      pj_pool_t               *pool;         // Pool owned by this buffer
      pjsip_transport         *transport;    // The transport that received the msg.
      pjsip_rx_data_op_key     op_key;       // Ioqueue's operation key
   } tp_info;

   // This part contains information about the packet
   struct
   {
      pj_time_val              timestamp;                  // Packet arrival time
      char                     packet[PJSIP_MAX_PKT_LEN];  // The packet buffer
      pj_uint32_t              zero;                       // Zero padding.
      int                      len;                        // Packet length
      pj_sockaddr              addr;                       // Source address
      int                      addr_len;                   // Address length.
   } pkt_info;

   // This part describes the message and message elements after parsing.
   struct
   {
      char                    *msg_buf;      // Pointer to start of msg in the buf.
      int                      len;          // Message length.
      pjsip_msg               *msg;          // The parsed message.

      // Shortcut to important headers:

      pj_str_t                 call_id;      // Call-ID string.
      pjsip_from_hdr          *from;         // From header.
      pjsip_to_hdr            *to;           // To header.
      pjsip_via_hdr           *via;          // First Via header.
      pjsip_cseq_hdr          *cseq;         // CSeq header.
      pjsip_max_forwards_hdr  *max_fwd;      // Max-Forwards header.
      pjsip_route_hdr         *route;        // First Route header.
      pjsip_rr_hdr            *record_route; // First Record-Route header.
      pjsip_ctype_hdr         *ctype;        // Content-Type header.
      pjsip_clen_hdr          *clen;         // Content-Length header.
      pjsip_require_hdr       *require;      // The first Require header.
      pjsip_parser_err_report  parse_err;    // List of parser errors.
   } msg_info;
```

```
    // This part is updated after the rx_data reaches endpoint.
    struct
    {
        pj_str_t                key;                        // Transaction key.
        void                   *mod_data[PJSIP_MAX_MODULE]; // Module specific data.
    } endpt_info;
};
```

接受数据缓冲区结构声明

5.1.2 传输数据缓冲区 Transmit Data Buffer (pjsip_tx_data)

当 PJSIP 消息想往外发送消息时，它必须创建一个数据缓冲区。数据缓冲区提供了与消息有关的消息字段必须进行内存分配的内存池、引用计数器、锁保护(lock protection)以及传输层用来处理消息的其它信息。

```
struct pjsip_tx_data
{
    /** This is for transmission queue; it's managed by transports. */
    PJ_DECL_LIST_MEMBER(struct pjsip_tx_data);

    /** Memory pool for this buffer. */
    pj_pool_t                   *pool;

    /** A name to identify this buffer. */
    char                        obj_name[PJ_MAX_OBJ_NAME];

    /** Time of the rx request; set by pjsip_endpt_create_response(). */
    pj_time_val                 rx_timestamp;

    /** The transport manager for this buffer. */
    pjsip_tpmgr                 *mgr;

    /** Ioqueue asynchronous operation key. */
    pjsip_tx_data_op_key        op_key;

    /** Lock object. */
    pj_lock_t                   *lock;

    /** The message in this buffer. */
    pjsip_msg                   *msg;

    /** Contigous buffer containing the packet. */
    pjsip_buffer                buf;

    /** Reference counter. */
    pj_atomic_t                 *ref_cnt;

    /** Being processed by transport? */
    int                         is_pending;

    /** Transport manager internal. */
    void                        *token;
    void                        (*cb)(void*, pjsip_tx_data*, pj_ssize_t);

    /** Transport info, only valid during on_tx_request() and on_tx_response() */
    struct {
       pjsip_transport          *transport;     /**< Transport being used.    */
       pj_sockaddr              dst_addr;       /**< Destination address.     */
       int                      dst_addr_len;   /**< Length of address.       */
       char                     dst_name[16];   /**< Destination address.     */
       int                      dst_port;       /**< Destination port.        */
    } tp_info;
};
```

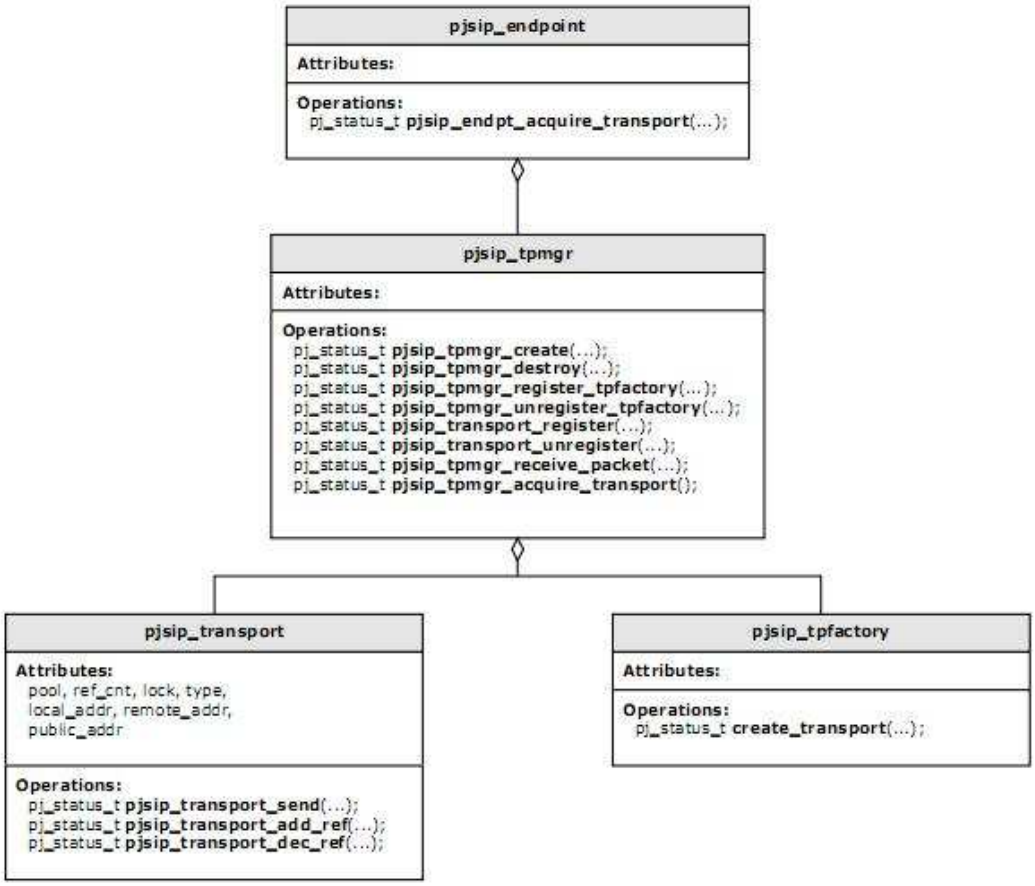
第六章 传输层

传输方式(Transports)用来从网络发送/接受消息。PJSIP 的传输框架是可扩展，意味着程序可以注册自己的传输方式来传输消息。

6.1 传输层设计

6.1.1 “类图”

下图显示了传输层实例之间的关系。



传输层”类图”

6.1.2 传输管理

传输管理(pjsip_tpmgr)管理着所有的传输对象和工厂。它提供了如下的功能：

- 使用传输引用计数器(transport’s reference counter)和空闲定时器(idle timer)管理传输的生命期。
- 管理传输工厂(transport factories)

- 从传输接受包，解析包，向 endpoint 提交消息。
- 根据传输类型和远程地址，找到匹配的传输方式来发送 SIP 消息到特定的目标。
- 如果无可用的传输方式来发送 SIP 消息到新目标地址，那么动态创建新的传输。

一个 endpoint 只有一个传输管理器。传输管理器通常是应用程序不可见的。应用程序应该使用 endpoint 提供的函数与传输管理器交互。

6.1.3 传输工厂(Transport Factory)

传输工厂(pjsip_tpfactory)用来向远程 endpoint 建立动态连接。这种类型连接的一个例子是 TCP 传输，需要为每一目标地址建立一个 TCP 传输连接。

当传输管理器检测到需要为一个新目标地址创建传输方式时，它从传输工厂查找匹配的传输方式并要求传输工厂建立连接。

传输工厂对象会在下面声明。

6.1.4 传输(Transport)

传输对象表现为 pjsip_transport 结构。此结构的每一个实例通常表现为一种 socket 句柄(handle)(如 UDP、TCP)，尽管传输层也支持非 socket 传输方式。

通用传输操作

从框架的角度来看，传输对象是一个活动对象。框架中不存在轮询传输对象的机制。相反，传输对象必须找到自己的方式从网络上接受数据包，及为了进一步处理将数据包提交到传输管理器。

为了达到上述目标的推荐方式是注册传输的 socket 句柄到 endpoint 的 I/O 队列(pj_ioqueue_t)。这样，当 endpoint 轮询 I/O 队列时，传输对象就会从网络上接受到数据包。

一旦传输对象接受到数据包，它必须调用 pjsip_tpmgr_receive_packet()方法将数据包提交到传输管理器。这样数据包才会被解析并分派到协议栈的其余部分。传输对象必须初始化接受数据缓冲区的 tp_info 和 pkt_info 成员。

每一个传输对象拥有一个函数指针来发送数据到网络(即传输对象的属性 send_msg())。程序(或协议栈)调用 pjsip_transport_send()函数发送消息到网络，消息最终到达传输对象，然后 send_msg()函数被调用。包的发送可能是安全异步的，如果是这样，传输必须返回在 send_msg()中返回 PJ_EPENDING，并在消息发送到目标后调用参数中指定的回调函数。

传输对象声明

下面的代码显示了传输对象的声明：

```
struct pjsip_transport
{
    char                        obj_name[PJ_MAX_OBJ_NAME];   // Name.

    pj_pool_t                  *pool;          // Pool used by transport.
    pj_atomic_t                *ref_cnt;       // Reference counter.
    pj_lock_t                  *lock;          // Lock object.
    int                         tracing;       // Tracing enabled?

    pjsip_transport_type_e      type;          // Transport type.
    char                        type_name[8]; // Type name.
    unsigned                    flag;          // See #pjsip_transport_flags_e

    pj_sockaddr                 local_addr;    // Bound address.
    pjsip_host_port             addr_name;     // Published name (e.g. STUN address).
    pj_sockaddr                 rem_addr;      // Remote addr (zero for UDP)

    pjsip_endpoint             *endpt;         // Endpoint instance.
    pjsip_tpmgr                *tpmgr;         // Transport manager.
    pj_timer_entry              idle_timer;    // Timer when ref cnt is zero.

    /* Function to be called by transport manager to send SIP messages. */
    pj_status_t                 (*send_msg)( pjsip_transport *transport,
                                             pjsip_tx_data *tdata,
                                             const pj_sockaddr_in *rem_addr,
                                             void *token,
                                             void (*callback)( pjsip_transport*,
                                                               void *token,
                                                               pj_ssize_t sent));

    /* Called to destroy this transport. */
    pj_status_t                 (*destroy)( pjsip_transport *transport );

    /* Application may extend this structure. */
};
```

传输对象声明

传输管理(Transport Management)

传输调用 pjsip_transport_register()将自身注册到传输管理器。在这个函数调用之前，传输结构的所有成员必须被初始化。

传输的生命期由传输管理器自动管理。每次传输的引用计数器到达 0，一个空闲定时器(idle timer)就会启动。如果空闲定时器过期时引用计数器值仍然为 0，那么传输管理器就会调用 pjsip_transport_unregister()来销毁传输对象。这个函数会从传输管理器的哈希表中注销掉传输对象，最终销毁此传输对象。

一些传输对象需要一直存在，即使没有对象使用此传输对象(如，UDP 传输对象，这是一个单件实例)。为了防止这种传输对象被删除，在初始时它必须设置引用计数为 1，这样引用计数永远不会为 0。

传输错误处理(Transport Error Handling)

传输中的任何错误(如发送数据包时出错或连接被重置)都由 transport user 来处理。传输对象不需要处理此类错误，除了从函数返回值中报告此错误。特别地，它也不需要重新尝试地建立失败或关闭了的连接。

6.2 使用传输对象

6.2.1 函数参照(Function Reference)

- **pj_status_t pjsip_endpt_acquire_transport(pjsip_endpoint *endpt,pjsip_transport_type_e t_type, const pj_sockaddr_t *remote_addr, int addrlen,pjsip_transport **p_transport);**

  获取一个 *t_type* 的传输对象用来发送消息到目标地址 *remote_addr*。注意如果传输对象获取成功了，那么此传输对象的引用计数会增加 1。

- **pj_status_t pjsip_transport_add_ref(pjsip_transport *transport);**

增加 *transport* 的引用计数。此函数会防止 *transport* 被销毁，同时如果空闲定时器是活动的话，函数会取消此定时器。

- **pj_status_t pjsip_transport_dec_ref(pjsip_transport *transport);**

  对 *transport* 有引用计数减 1。当 *transport* 的引用计数到达 0 时，一个空闲定时器会启动。如果空闲定时器到期后，*transport* 的引用计数仍然为 0，那么传输管理器就会销毁 *transport*。

- **pj_status_t pjsip_transport_send(pjsip_transport *transport,pjsip_tx_data *tdata,const pj_sockaddr_t *remote_addr,int addrlen,void *token,void (*cb)(void *token, pjsip_tx_data *tdata, pj_ssize_t bytes_sent));**

  使用 *transport* 将 *tdata* 中的消息发送到 *remote_addr*。如果函数立即结束且消息发送成功，函数返回 PJ_SUCCESS。如果函数立即结束但有错误发生，函数会返回一个非 0 值的错误码。这两种情况下，回调函数都不会调用。

  如果函数不能立即结束(例如底层的 socket 缓冲区已满)，函数会返回 PJ_PENDING。调用者将通过回调函数 *cb* 来得知消息发送完成。如果未决的发送操作完成时有错误发生，错误码将会以负值的形式在回调函数的参数 bytes_sent 中指示出来(要得到错误码，请这样做 pj_status_t status = -bytes_sent)。

  这个函数将会以原样的方式发送消息，不会对消息做任何验证。函数也不会 Via 头域做任何修改。

6.3 扩展传输对象(Extending Transports)

PJSIP 的传输组件可以使用自定义的传输对象进行扩展。理论上，任何类型的传输方式，不限于 TCP/IP，都可以添加到架构管理框架中。请参见头文件<pjsip/sip_transport.h>及 sip_transport_udp.[hc] 获取得更多信息。

6.4 初始化传输对象(Initializing Transports)

PJSIP 默认不会启动任何传输对象(甚至内置的传输对象) ;应用程序自己来负责初始化和启动想使用的传输对象。

下面是内置的 UDP 和 TCP 传输对象的初始化函数。

6.4.1 UDP 传输对象初始化

PJSIP 提供两种初始化和启动 UDP 传输对象的选择。这些函数声明在<pjsip/sip_transport_udp.h>中。

- **pj_status_t pjsip_udp_transport_start(pjsip_endpoint *endpt,const pj_sockaddr_in *local_addr,const pj_sockaddr_in *pub_addr,unsigned async_cnt,pjsip_transport **p_transport);**

  创建、初始化、注册、启动一个新的 UDP 传输对象。UDP socket 将会绑定到 *local_addr*。如果 endpoint 位于防火墙/NAT 或其它端口转发设备后面 ,那么 *pub_addr* 可以用来作为此传输对象的公共地址；否则 *pub_addr* 应该和 *local_addr* 相同。参数 *async_cnt* 指定在这个传输对象上可以允许多少并发操作，且为了性能最大化，这个参数的值应该等于节点上的处理器数目。

  如果传输对象成功启动了，函数返回 PJ_SUCCESS 且传输对象返回到参数 *p_transport* 中。程序可以马上使用此对象。程序不需要注册此传输对象到传输管理器，此函数返回成功时已经做了此工作。

  如果出现错误，此函数函数返回非 0 错误代码。

- **pj_status_t pjsip_udp_transport_attach(pjsip_endpoint *endpt,pj_sock_t sock,const pj_sockaddr_in *pub_addr,unsigned async_cnt,pjsip_transport **p_transport);**

  使用此方法在 UDP socket 已经可用的情况下创建、初始化、注册、启动一个新的 UDP 传输对象。这在如下的情况下非常有用 :应用程序已经使用 STUN 解析了

socket 的公用地址，这时不是关闭它后再重新建立一个新 socket，而是为 SIP 传输对象重用此 socket。