

关于 PJSIP:

PJSIP 是一个小巧而性能优异的协议栈。

请访问: <http://www.pjsip.org> 获得更多信息。

关于本文档:

这是一份自由文档。每一个人都有限按原样（verbatim copies）复制与分发此文档，但不允许修改。

PJSUA API—高级软电话的 API

为构建 SIP UA 应用而设计的具有高水平的 API。

模块

[PJSUA-API 的基本 API](#)

基本应用程序的创建、初始化、日志配置等。

[PJSUA-API 的信令传输](#)

管理 SIP 传输的 API。

[PJSUA-API 的账户管理](#)

PJSUA 帐户的管理。

[PJSUA-API 呼叫管理](#)

呼叫操作。

[PJSUA-API 的好友、用户状态和即时消息](#)

好友管理，好友的状态和即时消息。

[PJSUA-API 的媒体操作](#)

媒体操作。

详细说明

一个为 C / C++ 编程提供的 SIP 用户代理 API

PJSUA-API 是构建 SIP 多媒体用户代理应用而提供的非常高水平的 API。它把信令和媒体功能包装在一起放入一个易于使用的调用的 API 中，提供帐户管理，好友管理，状态，即时消息以及多媒体功能例如会议、文件流、本地回放、录音等。

C / C++ 绑定

应用程序必须与 PJSUA-lib 链接来使用这些 API。此外，该库依赖于以下库：

pjsip-ua,

pjsip-simple,

pjsip-core,

pjmedia,

pjmedia-codec,

pjlib-util,

pjlib,

应用程序还必须链接这些库。有关详细信息，请参阅入门 [Getting Started with PJSIP](#)。

pjsua_samples

提供了一些例程：

例程：Simple PJSUA

很简单的 SIP 用户代理，具有注册、打电话和媒体建立的功能，使用 PJSUA 的 API，

在 200 行代码内实现。

PJSUA

这是 PJSIP 协议和 PJMEDIA 的参考实现。PJSUA 是基于控制台的应用程序，设计简单、易读，但功能强大，足以体现 PJSIP 和 PJMEDIA 提供的所有功能。

使用 PJSUA API

关于如何创建和初始化 API 请参阅 PJSUA-API Basic API。

PJSUA-API Basic API

[PJSUA API - 高级软电话的 API]

基本的应用程序的创建、初始化，日志配置等。

数据结构

```
struct pjsua\_logging\_config
struct pjsua\_mwi\_info
struct pjsua\_reg\_info
struct pjsua\_callback
struct pjsua\_config
struct pjsua\_msg\_data
struct pj\_stun\_resolve\_result
```

定义

```
#define PJSUA\_INVALID\_ID (-1)
#define PJSUA\_DEFAULT\_USE\_SRTP PJMEDIA\_SRTP\_DISABLED
#define PJSUA\_DEFAULT\_SRTP\_SECURE\_SIGNALING 1
#define PJSUA\_ADD\_ICE\_TAGS 1
#define PJSUA\_ACQUIRE\_CALL\_TIMEOUT 2000
#define pjsip\_cred\_dup pjsip_cred_info_dup
```

Typedefs

```
typedef int pjsua\_call\_id
typedef int pjsua\_acc\_id
typedef int pjsua\_buddy\_id
typedef int pjsua\_player\_id
typedef int pjsua\_recorder\_id
typedef int pjsua\_conf\_port\_id
typedef pj\_stun\_resolve\_cb (const pj\_stun\_resolve\_result *result)
void(*
```

枚举

```
enum pjsua\_create\_media\_transport\_flag
{ PJSUA\_MED\_TP\_CLOSE\_MEMBER = 1 }
enum pjsua\_sip\_timer\_use { PJSUA\_SIP\_TIMER\_INACTIVE,
PJSUA\_SIP\_TIMER\_OPTIONAL, PJSUA\_SIP\_TIMER\_REQUIRED,
PJSUA\_SIP\_TIMER\_ALWAYS }
enum pjsua\_100rel\_use { PJSUA\_100REL\_NOT\_USED,
PJSUA\_100REL\_MANDATORY, PJSUA\_100REL\_OPTIONAL }
enum pjsua\_destroy\_flag { PJSUA\_DESTROY\_NO\_RX\_MSG = 1,
```

[PJSUA_DESTROY_NO_TX_MSG](#) = 2, [PJSUA_DESTROY_NO_NETWORK](#) }

功能函数

void	pjsua_logging_config_default (pjsua_logging_config *cfg)
void	pjsua_logging_config_dup (pj_pool_t *pool, pjsua_logging_config *dst, const pjsua_logging_config *src)
void	pjsua_config_default (pjsua_config *cfg)
void	pjsua_config_dup (pj_pool_t *pool, pjsua_config *dst, const pjsua_config *src)
void	pjsua_msg_data_init (pjsua_msg_data *msg_data)
pj_status_t	pjsua_create (void)
pj_status_t	pjsua_init (const pjsua_config *ua_cfg, const pjsua_logging_config *log_cfg, const pjsua_media_config *media_cfg)
pj_status_t	pjsua_start (void)
pj_status_t	pjsua_destroy (void)
pj_status_t	pjsua_destroy2 (unsigned flags)
int	pjsua_handle_events (unsigned msec_timeout)
pj_pool_t *	pjsua_pool_create (const char *name, pj_size_t init_size, pj_size_t increment)
pj_status_t	pjsua_reconfigure_logging (const pjsua_logging_config *c)
pjsip_endpoint *	pjsua_get_pjsip_endpt (void)
pjmedia_endpt *	pjsua_get_pjmedia_endpt (void)
pj_pool_factory *	pjsua_get_pool_factory (void)
pj_status_t	pjsua_detect_nat_type (void)
pj_status_t	pjsua_get_nat_type (pj_stun_nat_type *type)
pj_status_t	pjsua_resolve_stun_servers (unsigned count, pj_str_t srv[], pj_bool_t wait, void *token, pj_stun_resolve_cb cb)
pj_status_t	pjsua_cancel_stun_resolution (void *token, pj_bool_t notify_cb)
pj_status_t	pjsua_verify_sip_url (const char *url)
pj_status_t	pjsua_verify_url (const char *url)
pj_status_t	pjsua_schedule_timer (pj_timer_entry *entry, const pj_time_val *delay)
void	pjsua_cancel_timer (pj_timer_entry *entry)
void	pjsua_perror (const char *sender, const char *title, pj_status_t status)
void	pjsua_dump (pj_bool_t detail)

详细说明

基础的 PJSUA-API 控制 PJSUA 的创作、初始化、启动，同时还提供各种辅助功能。

使用 PJSUA 库

创建 PJSUA

在任何事情之前，应用程序必须调用 `pjsua_create()` 创建 PJSUA。除其他事项外，这将初始化 PJLIB，PJLIB-util，并创建一个 SIP endpoint，在调用任何 PJLIB 功

能之前这是至关重要的一步。

这个函数被调用后，应用程序可以创建一个内存池（使用 `pjsua_pool_create()`），并从命令行或文件中读取配置，初始化 PJSUA 如下。

初始化 PJSUA

创建 PJSUA 后，应用程序可以通过调用 `pjsua_init()` 初始化 PJSUA。如果应用程序要设置它们，这个函数有几个参数可选。

PJSUA-LIB 初始化（C 语言）

初始化 PJSUA 的示例（C 代码）：

```
#include <pjsua-lib/pjsua.h>
#define THIS_FILE __FILE__
static pj\_status\_t app_init(void)
{
    pjsua\_config ua_cfg;
    pjsua\_logging\_config log_cfg;
    pjsua\_media\_config media_cfg;
    pj\_status\_t status;
    // 再做任何事情之前必须创建 PJSUA!
    status = pjsua\_create();
    if (status != PJ_SUCCESS) {
        pjsua\_perror(THIS_FILE, "Error initializing pjsua", status);
        return status;
    }

    // 使用默认配置
    pjsua\_config\_default(&ua_cfg);
    pjsua\_logging\_config\_default(&log_cfg);
    pjsua\_media\_config\_default(&media_cfg);
    // 应用程序若要被重写，则回调 pjsua_config:
    ua_cfg.cb.on\_incoming\_call = ...
    ua_cfg.cb.on\_call\_state = ..
    ...
    // 自定义其他设置或用配置文件初始化它们
    ...
    // 初始化 pjsua
    status = pjsua\_init(&ua_cfg, &log_cfg, &media_cfg);
    if (status != PJ_SUCCESS) {
        pjsua\_perror(THIS_FILE, "Error initializing pjsua", status);
        return status;
    }
    .
    ...
}
```

其他初始化

PJSUA 初始化后，应用程序将通常需要/想要执行下列任务：

使用 `pjsua_transport_create()` 创建 SIP 传输。应用程序将为它支持每个传输类

型（例如 UDP、TCP 和 TLS）调用 `pjsua_transport_create()`。更多信息，请参阅 PJSUA-API 的信令传输。

使用 `pjsua_acc_add()` 或 `pjsua_acc_add_local()` 创建一个或多个 SIP 帐号。SIP 帐户用于向 SIP 服务器注册。更多信息，请参阅 PJSUA-API 的帐务管理。

使用 `pjsua_buddy_add()` 添加一个或多个好友。更多信息，请参阅 PJSUA-API 的好友、用户状态和即时消息。

选择配置声音设备，设置编解码器，设置其他媒体。更多信息，请参阅 PJSUA-API 的媒体操作。

运行 PJSUA

完成初始化后，应用程序必须调用 `pjsua_start()` 开始 PJSUA。此功能将检查所有的设置是否正确配置，如果他们没有将采用默认设置，或者如果它是无法从丢失的设置中恢复则报告错误状态。

大多数设置都可以在运行时改变。例如，应用程序可以在运行时添加，修改，或删除帐户，好友，或改变媒体设置。

开始 PJSUA 的例子 C 代码

示例代码：

```
static pj\_status\_t app_run(void)
{
    pj\_status\_t status;
    // 运行 pjsua
    status = pjsua\_start();
    if (status != PJ_SUCCESS) {
        pjsua\_destroy();
        pjsua\_perror(THIS_FILE, "Error starting pjsua", status);
        return status;
    }
    // 循环执行应用程序
    while (1) {
        char choice[10];

        printf("Select menu: ");
        fgets(choice, sizeof(choice), stdin);
        ...
    }
}
```

Define Documentation

`#define PJSUA_INVALID_ID (-1)`

无效 ID

`#define PJSUA_DEFAULT_USE_SRTP PJMEDIA_SRTP_DISABLED`

最多代理账户的数目，SRTP 模式下的默认值。有效值：PJMEDIA_SRTP_DISABLED，PJMEDIA_SRTP_OPTIONAL，and PJMEDIA_SRTP_MANDATORY.

`#define PJSUA_DEFAULT_SRTP_SECURE_SIGNALING 1`

SRTP 安全信令要求的默认值。有效值为：0: SRTP 的不需要安全信号；1: SRTP

需要安全传输如 TLS; 2: SRTP 需要安全的终端到终端传输 (SIPS)

#define PJSUA_ADD_ICE_TAGS 1

如果在配置中 ICE 使能, 控制 PJSUA-LIB 是否应该添加 ICE 媒体功能的标签参数 ("sip.ice") 到联系首部, 默认: 1

#define PJSUA_ACQUIRE_CALL_TIMEOUT 2000

在一次特定呼叫中获取互斥锁的超时值

默认: 2000 ms

#define pjsip_cred_dup pjsip_cred_info_dup

它的实现已经被转移至 sip_auth.h

Typedef Documentation

typedef [int pjsua_call_id](#)

呼叫标识

typedef [int pjsua_acc_id](#)

账户标识

typedef [int pjsua_buddy_id](#)

好友标识

typedef [int pjsua_player_id](#)

文件的播放器标识

typedef [int pjsua_recorder_id](#)

文件记录标识

typedef [int pjsua_conf_port_id](#)

会议端口标识

typedef void(* [pj_stun_resolve_cb](#))(const [pj_stun_resolve_result](#) *result)

定义的回调类型用于 [pjsua_resolve_stun_servers\(\)](#).

Enumeration Type Documentation

enum [pjsua_create_media_transport_flag](#)

此枚举指定自定义媒体传输的选项

Enumerator:

<i>PJSUA_MED_TP_CLOSE_MEMBER</i>	此标志表示当 pjmedia_transport_close() 被调用时, 媒体传输层必须关闭其“成员”或“子”传输层。如果这个标志没有被指定, 那么媒体传输层不能调用它的成员传输层的 pjmedia_transport_close() 函数。
----------------------------------	---

enum [pjsua_sip_timer_use](#)

此枚举指定 SIP 会话计时器的使用

Enumerator:

<i>PJSUA_SIP_TIMER_INACTIVE</i>	当此标志被指定时, 除非在远程请求时被明确要求, 会话计时器将不被用于任何会话。
<i>PJSUA_SIP_TIMER_OPTIONAL</i>	当此标志被指定时, 当远程会话支持和

	使用时会话计时器时，它将被用于所有会话。
<i>PJSUA_SIP_TIMER_REQUIRED</i>	当此标志被指定时，建立远程会话必须支持会话计时器。
<i>PJSUA_SIP_TIMER_ALWAYS</i>	当此标志被指定时，会话计时器将被用于所有会话，无论远程会话支持/使用与否。

enum [pjsua_100rel_use](#)

这个常数控制 100rel 扩展的使用

Enumerator:

<i>PJSUA_100REL_NOT_USED</i>	不使用，对于 UAC，支持 100rel 将被显示在支持首部，这样其他人可以选择使用它；对于 UAS，即使 UAC 支持此功能，100rel 仍不会被使用。
<i>PJSUA_100REL_MANDATORY</i>	强制性。UAC 将 100rel 置于 Require 首部，UAS 拒绝支持首部不包含 100rel 的来电。
<i>PJSUA_100REL_OPTIONAL</i>	可选。与 <i>PJSUA_100REL_NOT_USED</i> 相似，但对于 UAS，如果 UAC 支持此功能，100rel 将被使用。

enum [pjsua_destroy_flag](#)

标志可用于 [pjsua_destroy2\(\)](#)

Enumerator:

<i>PJSUA_DESTROY_NO_RX_MSG</i>	允许发送传出的消息（如 unregistration, event unpublication, BYEs, unsubscription 等），但不等待响应。有时“尽力而为”是非常有用的。
<i>PJSUA_DESTROY_NO_TX_MSG</i>	不发送任何传出的消息。如果应用程序知道要发送消息的网络目前不佳时，这个标志是有用的。
<i>PJSUA_DESTROY_NO_NETWORK</i>	在摧毁过程中，不要发送或接收消息。这个标志速记为 <i>PJSUA_DESTROY_NO_RX_MSG</i> + <i>PJSUA_DESTROY_NO_TX_MSG</i> 。

Function Documentation

```
void pjsua_logging_config_default ( pjsua\_logging\_config* cfg )
```

使用此函数来初始化日志配置

Parameters:

cfg 要初始化的日志配置

```
void pjsua_logging_config_dup ( pj\_pool\_t* pool,
                                pjsua\_logging\_config* dst,
                                const pjsua\_logging\_config* src
                                )
```


使用此函数来重新进行日志配置

Parameters:

pool 使用的内存池
dst 目的配置
src 源配置

```
void pjsua_config_default ( pjsua\_config * cfg )
```

使用此函数初始化 PJSUA 配置

Parameters:

cfg 要初始化的 pjsua 配置

```
void pjsua_config_dup ( pj\_pool\_t * pool,  
                       pjsua\_config * dst,  
                       const pjsua\_config * src  
                       )
```

重置 [pjsua_config](#).

Parameters:

pool 内存池
dst 目的配置
src 源配置

```
void pjsua_msg_data_init ( pjsua\_msg\_data * msg_data )
```

初始化信息数据

Parameters:

msg_data 要被初始化的信息数据

```
pj\_status\_t pjsua_create ( void )
```

实例化 PJSUA 应用。应用程序调用任何其他功能之前必须调用这个函数，以确保正确初始化底层库。一旦这个函数返回成功，在退出应用程序之前，必须调用 `pjsua_destroy()`。

Returns:

PJ_SUCCESS (成功)，或相应的错误代码。

```
pj\_status\_t pjsua_init ( const pjsua\_config * ua_cfg,  
                        const pjsua\_logging\_config * log_cfg,  
                        const pjsua\_media\_config * media_cfg  
                        )
```

初始化 PJSUA 指定的设置。所有的设置是可选的，当配置未指定时，将使用默认值

请注意，调用此函数之前必须调用 `pjsua_create()`

Parameters:

ua_cfg 用户代理配置
log_cfg 可选的日志设置
media_cfg 可选的媒体配置

Returns:

PJ_SUCCESS (成功)，或相应的错误代码。

```
pj\_status\_t pjsua_start ( void )
```

应用建议所有初始化完成后调用此函数，使库可以做额外的创建检查

应用调用 `pjsua_init()` 后可随时使用这个函数。

Returns:

`PJ_SUCCESS` (成功), 或相应的错误代码。

pj_status_t <code>pjsua_destroy</code>	(<code>void</code>)
--	---	-------------------	---

摧毁 PJSUA。应用建议执行正常关闭 (如注销 SIP 服务器的帐户, 终止在线状态, 结束回话) 后调用这个函数, 当然, 如果发现需要被终止活动的会话, 这个函数将做所有这些。此函数将滞后一秒钟, 等待来自远程的答复

如果不保持跟踪它的状态, 应用程序可以安全地多次调用这个函数

另见:

[pjsua_destroy2\(\)](#)

Returns:

`PJ_SUCCESS` (成功), 或相应的错误代码。

pj_status_t <code>pjsua_destroy2</code>	(<code>unsigned</code>	<i>flags</i>)
---	---	-----------------------	--------------	---

使用其他标志灵活摧毁 PJSUA

Parameters:

`flags` 枚举型 `pjsua_destroy_flag` 的组合

Returns:

`PJ_SUCCESS` (成功), 或相应的错误代码。

int <code>pjsua_handle_events</code>	(<code>unsigned</code>	<i>msec_timeout</i>)
--	---	-----------------------	---------------------	---

轮询 PJSUA 的事件, 如果必要块的调用者线程可指定最大时间间隔 (以毫秒为单位)

应用通常不需要调用这个函数, 如果在 `pjsua_config` 结构中, 它已经配置了工作线程 (`thread_cnt` field), 因为轮询将由这些工作线程完成

Parameters:

`msec_timeout` 等待的最长时间, 以毫秒为单位

Returns:

在轮询中已处理事件的数量。负值表示错误, 应用程序可以检索错误 (`status = -return_value`)

pj_pool_t* <code>pjsua_pool_create</code>	(<code>const char *</code>	<i>name,</i>
		pj_size_t	<i>init_size,</i>
		pj_size_t	<i>increment</i>
)		

创建应用程序使用的内存池。一旦应用程序使用完池, 它必须用 `pj_pool_release()` 释放

Parameters:

`name` 可选池名称
`init_size` 初始化内存池大小.
`increment` 增量的大小

Returns:

内存池, 或当没有内存时返回 `NULL`

pj_status_t <code>pjsua_reconfigure_logging</code>	(<code>const</code>	pjsua_logging_config *	<i>c</i>)
--	---	--------------------	--	----------	---

应用程序可以在任何时间 (当然在 `pjsua_create()` 之后) 调用此功能更改日志设置

Parameters:

c 记录配置

Returns:

PJ_SUCCESS (成功), 或相应的错误代码。

[pjsip_endpoint](#)* pjsua_get_pjsip_endpt (void)

内部函数来获取 PJSUA 的 SIP endpoint 实例, 有时这是必须的例如登记模块, 创建传输等, 其只有 pjsua_init () 后调用有效

Returns:

SIP endpoint 实例

[pjmedia_endpt](#)* pjsua_get_pjmedia_endpt (void)

内部函数来获取 media endpoint 实例, 其只有 [pjsua_init\(\)](#) 后调用有效

Returns:

Media endpoint 实例

[pj_pool_factory](#)* pjsua_get_pool_factory (void)

内部函数来获取 PJSUA 内存池工厂, 其只有在 [pjsua_create\(\)](#) 后调用有效

Returns:

PJSUA 目前使用的内存池

[pj_status_t](#) pjsua_detect_nat_type (void)

函数在 endpoint 前面检测 NAT 类型。一旦成功调用, 这个函数将异步执行并报告 pjsua_callback 回调的 on_nat_detect () 中的结果

在 NAT 检测后执行回调, 应用程序可以通过调用 pjsua_get_nat_type () 检测 NAT 类型。应用程序也可以在稍后时间再次通过调用 pjsua_detect_nat_type () 执行 NAT 检测

请注意, STUN 必须启用来成功地运行此函数。

Returns:

PJ_SUCCESS (成功), 或相应的错误代码。

[pj_status_t](#) pjsua_get_nat_type ([pj_stun_nat_type](#)* type)

与 [pjsua_detect_nat_type\(\)](#) 函数相似得到 NAT 类型。这个函数在 [pjsua_detect_nat_type\(\)](#) 成功执行和 [on_nat_detect\(\)](#) 回调后, 仅返回有用的 NAT 类型

Parameters:

type NAT 类型

Returns:

在检测过程中, 函数将返回 PJ_EPENDING, 类型将被设置为 PJ_STUN_NAT_TYPE_UNKNOWN。在 NAT 类型成功检测后, 函数将返回 PJ_SUCCESS, 类型将被置于正确值。其他返回值表示错误, 类型将被置于 PJ_STUN_NAT_TYPE_ERR_UNKNOWN

另见:

[pjsua_call_get_rem_nat_type\(\)](#)

pj_status_t	pjsua_resolve_stun_servers	(unsigned	count,
			pj_str_t	srv[],
			pj_bool_t	wait,
			void *	token,
			pj_stun_resolve_cb	cb

)

辅助功能，以解决和联系每个 STUN 服务器项目（按顺序），找出哪些是可用的。
调用此函数之前必须已调用 `pjsua_init()`

Parameters:

`count` STUN 服务器项目的数量
`srv` STUN 服务器项目的列表。请参阅在 `pjsua_config` 文档中关于项目格式的 `stun_srv` 部分
`wait` 指定非零阻碍改函数，直到它得到的结果。在这种情况下，当正在做决议时该函数将被阻止，这个函数返回之前回调将被调用。
`token` 在回调中任意令牌将被传回应用程序
`cb` 回调将被调用来通知函数结果

Returns:

等待参数非零时，如果一个可用的 STUN 服务器被发现将返回 `PJ_SUCCESS`。否则它将始终返回 `PJ_SUCCESS`，应用程序将在回调中通知结果

pj_status_t <code>pjsua_cancel_stun_resolution</code>	(<code>void *</code>	<code>token,</code>
		pj_bool_t	<code>notify_cb</code>
)		

取消挂起的符合指定令牌的 STUN 决议

Parameters:

`token` 匹配的令牌。令牌将用于 `pjsua_resolve_stun_servers()`
`notify_cb` 布尔型来控制回调是否应该为取消的决议调用，当回调被调用时，结果的状态将被设为 `PJ_ECANCELLED`

Returns:

如果有至少有一个悬而未决的 STUN 决议被取消返回 `PJ_SUCCESS`

如果没有匹配的或其他错误返回 `PJ_ENOTFOUND`

pj_status_t <code>pjsua_verify_sip_url</code>	(<code>const char *</code>	<code>url</code>)
---	---	---------------------------	------------------	---

函数来验证 SIP URL 的有效性。如果 URL 是个有效的 SIP/SIPS,将返回 `PJ_SUCCESS`

Parameters:

`url` URL（以 NULL 结尾的字符串）

Returns:

`PJ_SUCCESS`（成功），或相应的错误代码

另见:

[pjsua_verify_url\(\)](#)

pj_status_t <code>pjsua_verify_url</code>	(<code>const char *</code>	<code>url</code>)
---	---	---------------------------	------------------	---

函数来验证 URL 的有效性。与 [pjsua_verify_sip_url\(\)](#)不同，如果使用了 `tel:URL`,则返回 `PJ_SUCCESS`

Parameters:

`url` URL（以 NULL 结尾的字符串）

Returns:

`PJ_SUCCESS`（成功），或相应的错误代码

另见:

[pjsua_verify_sip_url\(\)](#)

pj_status_t <code>pjsua_schedule_timer</code>	(pj_timer_entry *	<code>entry,</code>
		<code>const pj_time_val *</code>	<code>delay</code>
)		

)

启动一个定时器。请注意，定时器的回调可能由不同的线程来执行，这取决于是否启用工作线程

Parameters:

entry 定时器实体
delay 时间间隔

Returns:

PJ_SUCCESS（成功），或相应的错误代码

另见:

[pjsip_endpt_schedule_timer\(\)](#)

void pjsua_cancel_timer ([pj_timer_entry](#) * entry)

删除先前的定时器

Parameters:

entry 定时器实体

另见:

[pjsip_endpt_cancel_timer\(\)](#)

void pjsua_perror (const char * sender, const char * title, [pj_status_t](#) status)

函数现实错误代码对应的错误信息。错误信息将被发送到日志中

Parameters:

sender 发送者字段
title 错误消息的标题.
status 状态码

void pjsua_dump ([pj_bool_t](#) detail)

把堆栈状态转储到日志，使用 verbosity level 3

Parameters:

detail 当非零时，将打印出详细的输出(如 SIP 通信列表)

PJSUA-API Signaling Transport

[PJSUA API - 高级软电话的 API]

管理 SIP 传输的 API

数据结构

struct [pjsua_transport_config](#)
struct [pjsua_transport_info](#)

Typedefs

typedef [int](#) [pjsua_transport_id](#)

函数

void [pjsua_transport_config_default](#) ([pjsua_transport_config](#) *cfg)
void [pjsua_transport_config_dup](#) ([pj_pool_t](#) *pool, [pjsua_transport_config](#) *dst, const [pjsua_transport_config](#) *src)
[pj_status_t](#) [pjsua_transport_create](#) ([pjsip_transport_type_e](#) type, const

```

pjsua\_transport\_config *cfg, pjsua\_transport\_id *p_id)
pj\_status\_t pjsua\_transport\_register (pjsip\_transport *tp, pjsua\_transport\_id
    *p_id)
pj\_status\_t pjsua\_enum\_transports (pjsua\_transport\_id id[], unsigned *count)
pj\_status\_t pjsua\_transport\_get\_info (pjsua\_transport\_id id, pjsua\_transport\_info
    *info)
pj\_status\_t pjsua\_transport\_set\_enable (pjsua\_transport\_id id, pj\_bool\_t
    enabled)
pj\_status\_t pjsua\_transport\_close (pjsua\_transport\_id id, pj\_bool\_t force)

```

详细说明

PJSUA-API 支持创建多个传输实例，例如 UDP，TCP 和 TLS。在添加账户之前必须创建 SIP 传输

Typedef Documentation

```
typedef int pjsua\_transport\_id
```

SIP 传输标识

Function Documentation

```
void pjsua_transport_config_default ( pjsua\_transport\_config * cfg )
```

使用默认值初始化 UDP 配置

Parameters:

cfg 将要初始化的 UDP 配置

```
void pjsua_transport_config_dup ( pj\_pool\_t * pool,
    pjsua\_transport\_config * dst,
    const pjsua\_transport\_config * src
)
```

重设传输配置

Parameters:

pool 内存池
dst 目的配置
src 源配置

```

pj\_status\_t pjsua_transport_create ( pjsip\_transport\_type\_e type,
    const pjsua\_transport\_config * cfg,
    pjsua\_transport\_id * p_id
)

```

根据具体的设置，创建并启动一个新的 SIP 传输

Parameters:

type 传输类型
cfg 传输配置
p_id 可选的指针来接受传输 ID

Returns:

PJ_SUCCESS (成功)，或相应的错误代码

```

pj\_status\_t pjsua_transport_register ( pjsip\_transport * tp,
    pjsua\_transport\_id * p_id
)

```

注册已被应用程序创建的传输。如果应用程序要实现自定义的 SIP 传输时，该函数是有用的

Parameters:

tp 传输实例
p_id 可选的指针来接受传输 ID

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_enum_transports	(pjsua_transport_id	<i>id[],</i>
		unsigned *	<i>count</i>
)		

枚举当前系统创建的所有传输。函数将返回所有传输 ID，应用程序可以调用 [pjsua_transport_get_info\(\)](#) 得到有关某个传输的详细信息

Parameters:

id 接受传输 ID 的数组
count 输入是指定的元素的最大数量；返回它实际包含元素的数目

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_transport_get_info	(pjsua_transport_id	<i>id,</i>
		pjsua_transport_info *	<i>info</i>
)		

得到关于传输的信息

Parameters:

id 传输 ID.
info 接受传输信息的指针.

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_transport_set_enable	(pjsua_transport_id	<i>id,</i>
		pj_bool_t	<i>enabled</i>
)		

禁止传输或者重新启动它。默认情况下，传输在创建后就启动。禁止传输不一定要关闭端口，只是丢弃传入的信息并阻止传输发送消息

Parameters:

id 传输 ID
enabled 非零启动，零禁止

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_transport_close	(pjsua_transport_id	<i>id,</i>
		pj_bool_t	<i>force</i>
)		

关闭传输。如果传输被强制关闭，它将立即关闭，任何正在使用传输的挂起事务可能异常终止（甚至崩溃）。否则系统将等待所有事物完毕后安全关闭传输

Parameters:

id 传输 ID
force 非零立即关闭传输（不推荐）

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

PJSUA-API Accounts Management

[PJSUA API – 高级软电话 API]

PJSUA 的账户管理

数据结构

```
struct pjsua\_acc\_config
struct pjsua\_acc\_info
```

定义

```
#define PJSUA\_MAX\_ACC 8
#define PJSUA\_REG\_INTERVAL 300
#define PJSUA\_UNREG\_TIMEOUT 4000
#define PJSUA\_PUBLISH\_EXPIRATION PJSIP_PUBC_EXPIRATION_NOT_SPECIFIED
#define PJSUA\_DEFAULT\_ACC\_PRIORITY 0
#define PJSUA\_SECURE\_SCHEME "sip"
#define PJSUA\_UNPUBLISH\_MAX\_WAIT\_TIME\_MSEC 2000
#define PJSUA\_REG\_RETRY\_INTERVAL 300
#define PJSUA\_CONTACT\_REWRITE\_METHOD 2
#define PJSUA\_REG\_USE\_OUTBOUND\_PROXY 1
#define PJSUA\_REG\_USE\_ACC\_PROXY 2
#define PJSUA\_CALL\_HOLD\_TYPE\_DEFAULT PJSUA_CALL_HOLD_TYPE_RFC3264
```

枚举型

```
enum pjsua\_call\_hold\_type { PJSUA\_CALL\_HOLD\_TYPE\_RFC3264,
PJSUA\_CALL\_HOLD\_TYPE\_RFC2543 }
```

函数

```
void pjsua\_acc\_config\_default (pjsua\_acc\_config *cfg)
void pjsua\_acc\_config\_dup (pj\_pool\_t *pool, pjsua\_acc\_config *dst,
const pjsua\_acc\_config *src)
unsigned pjsua\_acc\_get\_count (void)
pj\_bool\_t pjsua\_acc\_is\_valid (pjsua\_acc\_id acc_id)
pj\_status\_t pjsua\_acc\_set\_default (pjsua\_acc\_id acc_id)
pjsua\_acc\_id pjsua\_acc\_get\_default (void)
pj\_status\_t pjsua\_acc\_add (const pjsua\_acc\_config *acc_cfg, pj\_bool\_t
is_default, pjsua\_acc\_id *p_acc_id)
pj\_status\_t pjsua\_acc\_add\_local (pjsua\_transport\_id tid, pj\_bool\_t is_default,
pjsua\_acc\_id *p_acc_id)
pj\_status\_t pjsua\_acc\_set\_user\_data (pjsua\_acc\_id acc_id, void *user_data)
void * pjsua\_acc\_get\_user\_data (pjsua\_acc\_id acc_id)
pj\_status\_t pjsua\_acc\_del (pjsua\_acc\_id acc_id)
pj\_status\_t pjsua\_acc\_modify (pjsua\_acc\_id acc_id, const pjsua\_acc\_config
*acc_cfg)
pj\_status\_t pjsua\_acc\_set\_online\_status (pjsua\_acc\_id acc_id, pj\_bool\_t
```

	is_online)
pj_status_t	pjsua_acc_set_online_status2 (pjsua_acc_id acc_id, pj_bool_t is_online, const pj_rpid_element *pr)
pj_status_t	pjsua_acc_set_registration (pjsua_acc_id acc_id, pj_bool_t renew)
pj_status_t	pjsua_acc_get_info (pjsua_acc_id acc_id, pjsua_acc_info *info)
pj_status_t	pjsua_enum_accs (pjsua_acc_id ids[], unsigned *count)
pj_status_t	pjsua_acc_enum_info (pjsua_acc_info info[], unsigned *count)
pjsua_acc_id	pjsua_acc_find_for_outgoing (const pj_str_t *url)
pjsua_acc_id	pjsua_acc_find_for_incoming (pjsip_rx_data *rdata)
pj_status_t	pjsua_acc_create_request (pjsua_acc_id acc_id, const pjsip_method *method, const pj_str_t *target, pjsip_tx_data **p_tdata)
pj_status_t	pjsua_acc_create_uac_contact (pj_pool_t *pool, pj_str_t *contact, pjsua_acc_id acc_id, const pj_str_t *uri)
pj_status_t	pjsua_acc_create_uas_contact (pj_pool_t *pool, pj_str_t *contact, pjsua_acc_id acc_id, pjsip_rx_data *rdata)
pj_status_t	pjsua_acc_set_transport (pjsua_acc_id acc_id, pjsua_transport_id tp_id)

详细说明

PJSUA 帐户提供正在使用应用程序用户的特征（或身份）。在 SIP 结构中，身份在请求头的 FROM 字段被使用。

PJSUA API 支持创建和管理多个帐户。帐户的最大数量由时间常数 PJSUA_MAX_ACC 决定。

帐户可以进行有与它相关联的客户端注册。当该帐户发送 SIP 请求消息将使用与之相应的路由设置和身份验证。一个帐户也有当前的在线状态，当远程好友检查帐户状态时将被上传，或如果版本支持上传到服务器上。

在应用程序中，必须创建至少一个帐户。如果没有需要的账号，应用程序可以调用 `pjsua_acc_add_local()` 创建 `userless`（无使用者）帐户。一个 `userless` 帐户替代特定的用户来标识本地 `endpoint`，它对应一个特定的传输实例。

另外一个帐户必须设置为默认的帐户，当 PJSUA 未能将请求与其他帐户相匹配时，该帐户将被使用。

当发送传出 SIP 请求（如拨打电话或发送即时消息）时，通常 PJSUA 需要的应用程序指定正在使用哪个帐户。如果没有指定帐户，PJSUA 可能是可以选择与目标域名相匹配的帐户，并没有找到匹配时使用默认帐户。

Define Documentation

#define PJSUA_MAX_ACC 8

最大用户数

#define PJSUA_REG_INTERVAL 300

默认的注册时间间隔

#define PJSUA_UNREG_TIMEOUT 4000

默认等待注销事务完成的最长时间

默认: 4000 (4 秒)

#define PJSUA_PUBLISH_EXPIRATION PJSIP_PUBC_EXPIRATION_NOT_SPECIFIED

默认版本有效期

#define PJSUA_DEFAULT_ACC_PRIORITY 0

默认账户优先权

#define PJSUA_SECURE_SCHEME "sip"

当使用传输如 TLS 时，这个宏指定的 URI 将在 Contact 头中使用。应用程序可以指定"sip" 或 "sips"

#define PJSUA_UNPUBLISH_MAX_WAIT_TIME_MSEC 2000

在发送注销前，在关闭过程中等待 unpublication 事务完成的最长时间。在关闭过程中库尽力等待在向未注册账户发送 REGISTER 请求之前完成。如果只设置太小，在 unpublication 完成之前，注销信息可能会被发送，这将导致 unpublication 请求失败

默认: 2000 (2 秒)

#define PJSUA_REG_RETRY_INTERVAL 300

默认的自动重试重新登记的时间间隔，以秒为单位。设置为 0 以禁用此功能。应用可以使用 pjsua_acc_config.reg_retry_interva 为每个帐户分别设置

默认: 300 (5 分钟)

#define PJSUA_CONTACT_REWRITE_METHOD 2

此宏指定 pjsua_acc_config 中 contact_rewrite_method 的默认值。如果在帐户配置中启用 allow_contact_rewrite。指定在注册过程中如何完成 contact 更新，如果设置为 1，Contact 更新将通过向目前登记的 Contact 发送注销，同时发送新登记（使用不同的 Call-ID）来完成

如果设置为 2，Contact 更新将独立完成，目前注册会议通过删除当前绑定（通过设置其 Contact 的过期参数为零），并添加一个新的 Contact 绑定，使用独立的请求

值 1 是传统的行为

默认值: 2

#define PJSUA_REG_USE_OUTBOUND_PROXY 1

位值在 pjsua_acc_config.reg_use_proxy 使用来表明，全球对外代理列表应该被添加到 REGISTER 请求中

#define PJSUA_REG_USE_ACC_PROXY 2

位值在 pjsua_acc_config.reg_use_proxy 使用来表明，账户的代理列表应该被添加到 REGISTER 请求中

#define PJSUA_CALL_HOLD_TYPE_DEFAULT PJSUA_CALL_HOLD_TYPE_RFC3264

指定 pjsua_acc_config 使用的默认呼叫持有类型

默认是 PJSUA_CALL_HOLD_TYPE_RFC3264，而且也没有理由去改变，除非你与老/非标准的同行交流

Enumeration Type Documentation

enum [pjsua_call_hold_type](#)

此枚举指定，我们应该如何向远程好友提供呼叫保持请求。默认值由时间常数 PJSUA_CALL_HOLD_TYPE_DEFAULT 的设置，应用程序可以通过 pjsua_acc_config 中的 call_hold_type 控制每个帐户的设置

Enumerator:

PJSUA_CALL_HOLD_TYPE_RFC3264	这将遵循 RFC 3264 建议使用 a= sendonly,
-------------------------------------	---------------------------------

	a= recvonly, 和=a=inactive,a 意味着信号呼叫保持状态
<i>PJSUA_CALL_HOLD_TYPE_RFC2543</i>	这将使用在 RFC 2543 中规定的旧的和过时的方法, 并会在 SDP 提供 C = 0.0.0.0, 使用这个有很多缺点, 如当呼叫被搁置时不能保持媒体传输, 应该只用于远程不理解 RFC 3264 标准的情况

Function Documentation

void pjsua_acc_config_default	(pjsua_acc_config *	cfg)
-------------------------------	---	------------------------------------	-----	---

用默认值初始化账户配置

Parameters:

cfg 将要初始化的账户

void pjsua_acc_config_dup	(pj_pool_t *	pool,
		pjsua_acc_config *	dst,
		const pjsua_acc_config *	src
)		

重新配置账户

Parameters:

pool 内存池
dst 目的配置
src 源配置

unsigned pjsua_acc_get_count	(void)
------------------------------	---	------	---

获得当前用户的数量

Returns:

当前用户数

pj_bool_t pjsua_acc_is_valid	(pjsua_acc_id	acc_id)
--	---	------------------------------	--------	---

检测指定账户是否有效

Parameters:

acc_id 账户 ID.

Returns:

如果账户有效非零值

pj_status_t pjsua_acc_set_default	(pjsua_acc_id	acc_id)
---	---	------------------------------	--------	---

设置默认账户, 用于收到/发送消息不符合任何账户的情况

Parameters:

acc_id 作为默认账户的 ID

Returns:

PJ_SUCCESS (成功)

pjsua_acc_id pjsua_acc_get_default	(void)
--	---	------	---

设置默认账户, 用于来电不符合任何账户的情况

Returns:

默认账户 ID, 或如果没有默认账户被配置 PJSUA_INVALID_ID

pj_status_t pjsua_acc_add	(const pjsua_acc_config *	acc_cfg,
		pj_bool_t	is_default,

```

        pjsua\_acc\_id *      p_acc_id
    )

```

添加一个新的账户。在调用这个函数前 PJSUA 必须初始化 ([pjsua_init\(\)](#))。配置账户的注册信息，函数将启动 SIP 注册，注册将通过库内部维护，应用程序不需要做任何事情

Parameters:

`acc_cfg` 账户配置
`is_default` 如果非零，账户将被设置为默认账户。根据建议，默认账户设置为 local/LAN 账户。
`p_acc_id` 接受新账户 ID 的指针

Returns:

PJ_SUCCESS (成功)，或相应的错误代码。

```

pj\_status\_t pjsua_acc_add_local      ( pjsua\_transport\_id  tid,
                                         pj\_bool\_t          is_default,
                                         pjsua\_acc\_id *      p_acc_id
                                         )

```

添加一个本地用户。本地用户用开识别本地 endpoint，而不是特定用户，因为这个原因传输 ID 需要获得本地地址信息

Parameters:

`tid` 生成账户地址的传输 ID
`is_default` 如果非零，账户将被设置为默认账户
`p_acc_id` 接受新账户 ID 的指针

Returns:

PJ_SUCCESS (成功)，或相应的错误代码

```

pj\_status\_t pjsua_acc_set_user_data ( pjsua\_acc\_id  acc_id,
                                         void *      user_data
                                         )

```

设置与账户相关的数据

Parameters:

`acc_id` 账户 ID
`user_data` 用户/应用程序数据。

Returns:

PJ_SUCCESS (成功)，或相应的错误代码

```

void* pjsua_acc_get_user_data      ( pjsua\_acc\_id  acc_id )

```

检索与账户关联的数据

Parameters:

`acc_id` 账户 ID.

Returns:

用户数据，ID 无效时返回 NULL

```

pj\_status\_t pjsua_acc_del      ( pjsua\_acc\_id  acc_id )

```

删除一个账户。这将从服务器注销，如果必要终止与服务器端的关联

Parameters:

`acc_id` 账户 ID

Returns:

PJ_SUCCESS（成功），或相应的错误代码

pj_status_t pjsua_acc_modify	(pjsua_acc_id	<i>acc_id,</i>
		const pjsua_acc_config *	<i>acc_cfg</i>
)		

修改账户信息

Parameters:

acc_id 账户 ID
acc_cfg 新的账户配置

Returns:

PJ_SUCCESS（成功），或相应的错误代码

pj_status_t pjsua_acc_set_online_status	(pjsua_acc_id	<i>acc_id,</i>
		pj_bool_t	<i>is_online</i>
)		

修改用户当前状态。如果服务器端存在账户信息，这将触发发送 NOTIFY 请求，如果账户启动 publication，将发送 PUBLISH

另见:

[pjsua_acc_set_online_status2\(\)](#)

Parameters:

acc_id 账户 ID
is_online True/false.

Returns:

PJ_SUCCESS（成功），或相应的错误代码

pj_status_t pjsua_acc_set_online_status2	(pjsua_acc_id	<i>acc_id,</i>
		pj_bool_t	<i>is_online,</i>
		const pjrpid_element *	<i>pr</i>
)		

修改用户当前状态。如果服务器端存在账户信息，这将触发发送 NOTIFY 请求，如果账户启动 publication，将发送 PUBLISH

See also:

[pjsua_acc_set_online_status\(\)](#)

Parameters:

acc_id 账户 ID
is_online True / false.
pr 扩展信息（使用 RPID 格式的子集），允许设置自定义的文本信息。

Returns:

PJ_SUCCESS（成功），或相应的错误代码

pj_status_t pjsua_acc_set_registration	(pjsua_acc_id	<i>acc_id,</i>
		pj_bool_t	<i>renew</i>
)		

更新注册或者执行注销。如果配置了账户注册，当执行 [pjsua_acc_add\(\)](#)时将初始化 SIP REGISTER

Parameters:

acc_id 账户 ID
renew 如果更新参数是 0，将启动注销

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_acc_get_info	(pjsua_acc_id	<i>acc_id,</i>
		pjsua_acc_info *	<i>info</i>
)		

获取指定账户的信息

Parameters:

acc_id 账户 ID

info 接受账户信息的指针

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_enum_accs	(pjsua_acc_id	<i>ids[],</i>
		unsigned *	<i>count</i>
)		

枚举所有账户.结果将放在 *ids* 数组里, 应用程序可以使用 [pjsua_acc_get_info\(\)](#) 查询账户信息

另见:

[pjsua_acc_enum_info\(\)](#).

Parameters:

ids 初始化的账户 ID 数组

count 输入是指定的元素的最大数量; 返回它实际包含元素的数目

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_acc_enum_info	(pjsua_acc_info	<i>info[],</i>
		unsigned *	<i>count</i>
)		

枚举账户信息

Parameters:

info 初始化的账户 ID 数组

count 输入是指定的元素的最大数量; 返回它实际包含元素的数目

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pjsua_acc_id pjsua_acc_find_for_outgoing	(const pj_str_t *	<i>url</i>)
--	---	----------------------------------	--------------

找到合适的账户, 以达到指定的 URL

Parameters:

url 要达到的远程 URL

Returns:

账户 ID

pjsua_acc_id pjsua_acc_find_for_incoming	(pjsip_rx_data *	<i>rdata</i>)
--	---	---------------------------------	----------------

找到最合适的账户, 以处理来电

Parameters:

rdata 接受的请求信息

Returns:

账户 ID

```

pj\_status\_t pjsua_acc_create_request ( pjsua\_acc\_id          acc_id,
                                         const pjsip\_method *    method,
                                         const pj\_str\_t *        target,
                                         pjsip\_tx\_data **        p_tdata
                                         )

```

使用账户创建请求。应用程序使用该函数创建对话外的辅助请求，比如 OPTIONS，并使用通话或存在的 API 创建与对话相关的请求

Parameters:

`acc_id` 账户 ID
`method` 请求的 SIP 方法
`target` 目标 URI
`p_tdata` 接受请求的指针

Returns:

PJ_SUCCESS 或错误代码

```

pj\_status\_t pjsua_acc_create_uac_contact ( pj\_pool\_t *      pool,
                                             pj\_str\_t *      contact,
                                             pjsua\_acc\_id      acc_id,
                                             const pj\_str\_t *    uri
                                             )

```

根据特定的目标 URI 为特定账户创建一个合适的 Contact 头部值

Parameters:

`pool` 内存池
`contact` Contact 将被存储的字符串
`acc_id` 账户 ID
`uri` 请求的目标 URI

Returns:

PJ_SUCCESS (成功) 或错误

```

pj\_status\_t pjsua_acc_create_uas_contact ( pj\_pool\_t *      pool,
                                             pj\_str\_t *      contact,
                                             pjsua\_acc\_id      acc_id,
                                             pjsip\_rx\_data *    rdata
                                             )

```

根据收到请求的信息创建一个合适的 Contact 头部

Parameters:

`pool` 内存池
`contact` Contact 将被存储的字符串.
`acc_id` 账户 ID
`rdata` 传入的请求

Returns:

PJ_SUCCESS (成功) 或错误

```

pj\_status\_t pjsua_acc_set_transport ( pjsua\_acc\_id          acc_id,
                                         pjsua\_transport\_id    tp_id
                                         )

```

把账户绑定/锁定到一个特定的传输/监听。通常应用程序不需要这样，账户将根

据目的自动选择传输

当账户被绑定到特定传输，这个账户的所有传出请求将使用指定的传输（包括 SIP 注册，会话和其他请求如 MESSAGE）

请注意，`transport_id` 也可在 [pjsua_acc_config](#) 中指定

Parameters:

`acc_id` 账户 ID.

`tp_id` 传输 ID.

Returns:

PJ_SUCCESS（成功）

PJSUA-API Calls Management

[PJSUA API – 高级软电话 API]

呼叫操作

数据结构

struct [pjsua_call_info](#)

定义

#define [PJSUA_MAX_CALLS](#) 32

#define [PJSUA_XFER_NO_REQUIRE_REPLACES](#) 1

枚举型

enum [pjsua_call_media_status](#) {
[PJSUA_CALL_MEDIA_NONE](#), [PJSUA_CALL_MEDIA_ACTIVE](#),
[PJSUA_CALL_MEDIA_LOCAL_HOLD](#), [PJSUA_CALL_MEDIA_REMOTE_HOLD](#),
[PJSUA_CALL_MEDIA_ERROR](#)
}

enum [pjsua_call_flag](#) { [PJSUA_CALL_UNHOLD](#) = 1, [PJSUA_CALL_UPDATE_CONTACT](#)
= 2 }

函数

unsigned	pjsua_call_get_max_count (void)
unsigned	pjsua_call_get_count (void)
pj_status_t	pjsua_enum_calls (pjsua_call_id ids[], unsigned *count)
pj_status_t	pjsua_call_make_call (pjsua_acc_id acc_id, const pj_str_t *dst_uri, unsigned options , void * user_data , const pjsua_msg_data *msg_data, pjsua_call_id *p_call_id)
pj_bool_t	pjsua_call_is_active (pjsua_call_id call_id)
pj_bool_t	pjsua_call_has_media (pjsua_call_id call_id)
pjmedia_session *	pjsua_call_get_media_session (pjsua_call_id call_id)
pjmedia_transport *	pjsua_call_get_media_transport (pjsua_call_id cid)
pjsua_conf_port_id	pjsua_call_get_conf_port (pjsua_call_id call_id)
pj_status_t	pjsua_call_get_info (pjsua_call_id call_id, pjsua_call_info *info)
pjsip_dialog_cap_status	pjsua_call_remote_has_cap (pjsua_call_id call_id, int htype, const pj_str_t *hname, const pj_str_t *token)
pj_status_t	pjsua_call_set_user_data (pjsua_call_id call_id, void

```

        *user_data)
void *      pjsua_call_get_user_data (pjsua_call_id call_id)
pj_status_t pjsua_call_get_rem_nat_type (pjsua_call_id call_id,
pj_status_t pj_stun_nat_type *p_type)
pj_status_t pjsua_call_answer (pjsua_call_id call_id, unsigned code,
const pj_str_t *reason, const pjsua_msg_data *msg_data)
pj_status_t pjsua_call_hangup (pjsua_call_id call_id, unsigned code,
const pj_str_t *reason, const pjsua_msg_data *msg_data)
pj_status_t pjsua_call_process_redirect (pjsua_call_id call_id,
pjsip_redirect_op cmd)
pj_status_t pjsua_call_set_hold (pjsua_call_id call_id, const
pjsua_msg_data *msg_data)
pj_status_t pjsua_call_reinvite (pjsua_call_id call_id, unsigned
options, const pjsua_msg_data *msg_data)
pj_status_t pjsua_call_update (pjsua_call_id call_id, unsigned options,
const pjsua_msg_data *msg_data)
pj_status_t pjsua_call_xfer (pjsua_call_id call_id, const pj_str_t *dest,
const pjsua_msg_data *msg_data)
pj_status_t pjsua_call_xfer_replaces (pjsua_call_id call_id,
pjsua_call_id dest_call_id, unsigned options, const
pjsua_msg_data *msg_data)
pj_status_t pjsua_call_dial_dtmf (pjsua_call_id call_id, const pj_str_t
*digits)
pj_status_t pjsua_call_send_im (pjsua_call_id call_id, const pj_str_t
*mime_type, const pj_str_t *content, const
pjsua_msg_data *msg_data, void *user_data)
pj_status_t pjsua_call_send_typing_ind (pjsua_call_id call_id,
pj_bool_t is_typing, const pjsua_msg_data *msg_data)
pj_status_t pjsua_call_send_request (pjsua_call_id call_id, const
pj_str_t *method, const pjsua_msg_data *msg_data)
void        pjsua_call_hangup_all (void)
pj_status_t pjsua_call_dump (pjsua_call_id call_id, pj_bool_t
with_media, char *buffer, unsigned maxlen, const char
*indent)

```

Define Documentation

#define PJSUA_MAX_CALLS 32

最大同时呼叫数

#define PJSUA_XFER_NO_REQUIRE_REPLACES 1

标志表示"Require: replaces"不能放在由 [pjsua_call_xfer_replaces\(\)](#) 创建的 REFER 引起的传出 INVITE 请求中

Enumeration Type Documentation

enum [pjsua_call_media_status](#)

此枚举型指定呼叫的媒体状态，是 [pjsua_call_info](#) 结构的一部分

Enumerator:

<i>PJSUA_CALL_MEDIA_NONE</i>	当前呼叫没有媒体
<i>PJSUA_CALL_MEDIA_ACTIVE</i>	媒体是有效的
<i>PJSUA_CALL_MEDIA_LOCAL_HOLD</i>	目前媒体由本地 endpoint 暂时搁置
<i>PJSUA_CALL_MEDIA_REMOTE_HOLD</i>	媒体由远程 endpoint 暂时搁置
<i>PJSUA_CALL_MEDIA_ERROR</i>	媒体错误(如 ICE 谈判)

enum [pjsua_call_flag](#)

标志可用于各种呼叫 APIs. 通过设定位, 多个标志可以被指定

Enumerator:

<i>PJSUA_CALL_UNHOLD</i>	当呼叫被搁置, 指定此标志, 以取消保持它。对于 <code>pjsua_call_reinvite()</code> 这个标志是唯一有效。注: 出于兼容性的原因, 此标志必须有值 1, 因为以前的取消保持选项被指定为布尔值
<i>PJSUA_CALL_UPDATE_CONTACT</i>	根据接收的 <code>contact</code> URI 更新当地 <code>invite</code> 的 <code>contact</code> 。对于 <code>pjsua_call_reinvite()</code> 和 <code>pjsua_call_update()</code> 这个标志是唯一有效。在 IP 地址的变化情况下这个标志是有用的。

Function Documentation

<code>unsigned pjsua_call_get_max_count</code>	<code>(void)</code>
--	-----------------------

得到 pjsua 配置的最大呼叫数目

Returns:

配置呼叫的最大数目

<code>unsigned pjsua_call_get_count</code>	<code>(void)</code>
--	-----------------------

获取当前活动的电话数目

Returns:

当前活跃呼叫的数目

<code>pj_status_t pjsua_enum_calls</code>	<code>(pjsua_call_id ids[], unsigned * count)</code>
---	--

枚举所有活动呼叫。应用程序可以通过调用 [pjsua_call_get_info\(\)](#) 查询每个呼叫的信息和状态

Parameters:

`ids` 账户 ID 数组

`count` 输入为指定的最大元素数量, 输出为实际包含的元素数目

Returns:

PJ_SUCCESS (成功), 或相应的错误代码。

<code>pj_status_t pjsua_call_make_call</code>	<code>(pjsua_acc_id acc_id, const pj_str_t * dst_uri, unsigned options, void * user_data, const pjsua_msg_data * msg_data, pjsua_call_id * p_call_id)</code>
---	--

)

使用特定的账户呼叫指定的 URI

Parameters:

acc_id 要使用的账户
 dst_uri 放在头部的 URI (通常与目标 URI 相同).
 options 选项 (目前必须为 0).
 user_data 用户数据, 可以稍后被检索
 msg_data 被添加到 INVITE 请求的可选标题等, 如果没有自定义头为 NULL
 p_call_id 接受 call identification 的指针

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

[pj_bool_t](#) pjsua_call_is_active ([pjsua_call_id](#) call_id)

检测是否特定的呼叫有 INVITE 且 INVITE 未断开

Parameters:

call_id Call identification.

Returns:

如果呼叫活跃返回非零

[pj_bool_t](#) pjsua_call_has_media ([pjsua_call_id](#) call_id)

检测呼叫是否有活跃的媒体

Parameters:

call_id Call identification.

Returns:

如果是返回非零

[pjmedia_session*](#) pjsua_call_get_media_session ([pjsua_call_id](#) call_id)

检索与呼叫相关的媒体会话。请注意, 根据当前呼叫的媒体状态 (在 [pjsua_call_info](#) 中 pjsua_call_media_status 信息), 媒体可能无法使用。应用程序可以使用媒体会话来检索更多关于呼叫的媒体的详细信息

Parameters:

call_id Call identification.

Returns:

呼叫媒体会话

[pjmedia_transport*](#) pjsua_call_get_media_transport ([pjsua_call_id](#) cid)

检索呼叫使用的媒体传输实例。应用程序必须使用媒体传输来获得更多关于媒体传输的信息

Parameters:

cid Call identification (the call_id).

Returns:

呼叫媒体传输

[pjsua_conf_port_id](#) pjsua_call_get_conf_port ([pjsua_call_id](#) call_id)

获取与当前呼叫相关的会议 port ID

Parameters:

call_id Call identification.

Returns:

会议 port ID, 当媒体奖励失败或不活跃时返回 `PJSUA_INVALID_ID`

```
pj\_status\_t pjsua_call_get_info ( pjsua\_call\_id      call_id,
                                   pjsua\_call\_info *    info
                                   )
```

获取当前呼叫的详细信息

Parameters:

`call_id` Call identification
`info` 初始化的呼叫信息

Returns:

`PJ_SUCCESS` (成功), 或相应的错误代码

```
pjsip\_dialog\_cap\_status pjsua_call_remote_has_cap ( pjsua\_call\_id      call_id,
                                                       int                  htype,
                                                       const pj\_str\_t *    hname,
                                                       const pj\_str\_t *    token
                                                       )
```

检测远程好友是否支持特定功能

Parameters:

`call_id` Call identification.
`htype` 检测的头类型, 可以是:

- `PJSIP_H_ACCEPT`
- `PJSIP_H_ALLOW`
- `PJSIP_H_SUPPORTED`

`hname` 如果 `htype` 指定 `PJSIP_H_OTHER`, 那么头名必须由该字段指定. 否则为 `NULL`.
`token` 检测能力令牌. 例如如果 `htype` 是 `PJSIP_H_ALLOW`, 令牌指定方法名; 如果 `htype` 是 `PJSIP_H_SUPPORTED`, 令牌指定扩展名如 "100rel".

Returns:

如果明确支持指定的功能返回 `PJSIP_DIALOG_CAP_SUPPORTED`

```
pj\_status\_t pjsua_call_set_user_data ( pjsua\_call\_id      call_id,
                                         void *            user_data
                                         )
```

为呼叫设定用户数据. 应用程序可以调用 [pjsua_call_get_user_data\(\)](#) 来检查此数据

Parameters:

`call_id` Call identification.
`user_data` 连接到呼叫的数据

Returns:

用户数据

```
void* pjsua_call_get_user_data ( pjsua\_call\_id      call_id )
```

得到先前由 [pjsua_call_set_user_data\(\)](#) 设置的用户数据

Parameters:

`call_id` Call identification.

Returns:

用户数据

```

pj_status_t pjsua_call_get_rem_nat_type (    pjsua_call_id    call_id,
                                             pj_stun_nat_type *  p_type
                                             )

```

得到远端 endpoint 的 NAT 类型. 这是 PJSUA-LIB 的专有功能, 当在 [pjsua_config](#) 中设置 *nat_type_in_sdp* 时, 其在 SDP 中发送 NAT 类型

函数只有在接收到远端 SDP 后被调用. 这意味着, 对于来电, 呼叫一旦接收到包含 SDP 的传入即可调用函数; 对于呼叫, 只有 SDP (通常是对 INVITE 的 200/OK 回应) 被接受到函数才可被使用. 再一般情况下, 应用程序可以在 *on_call_media_state()* 回调后使用此函数

Parameters:

call_id Call identification

p_type 存储 NAT 类型的指针. 应用程序可以通过 *pj_stun_get_nat_name()* 检索 NAT 类型的字符串描述

Returns:

PJ_SUCCESS (成功)

另见:

[pjsua_get_nat_type\(\)](#), *nat_type_in_sdp*

```

pj_status_t pjsua_call_answer (    pjsua_call_id    call_id,
                                   unsigned           code,
                                   const pj_str_t *    reason,
                                   const pjsua_msg_data * msg_data
                                   )

```

想到来的 INVITE 请求发送回应. 根据参数指定的状态代码, 该函数可以发送临时响应, 建立呼叫或终止通话

Parameters:

call_id 来电 call identification.

code 状态码, (100-699).

reason 可选原因字段如果为 NULL, 默认文本将被使用.

msg_data 添加到传出响应信息的可选头列表.

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```

pj_status_t pjsua_call_hangup (    pjsua_call_id    call_id,
                                   unsigned           code,
                                   const pj_str_t *    reason,
                                   const pjsua_msg_data * msg_data
                                   )

```

根据会话状态, 使用适当方法挂掉电话. 这个函数不仅回应呼叫使用 3xx-6xx 相应 (使用 [pjsua_call_answer\(\)](#)), 函数将挂掉电话无论呼叫的状态和作用, 而 [pjsua_call_answer\(\)](#) 仅在来电处于 EARLY 状态时工作

Parameters:

call_id Call identification.

code 当拒绝来电时发送的可选状态 如果为零, 发送 "603/Decline".

reason 当拒绝来电时发送的可选原因字段. 如果为 NULL, 发送默认文本

msg_data 添加到 request/response 消息中的可选头列表.

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_call_process_redirect	(pjsua_call_id	<i>call_id,</i>
		pjsip_redirect_op	<i>cmd</i>
)		

接受或拒绝重定向响应.当回调 *on_call_redirected()* 标定 *PJSIP_REDIRECT_PENDING* 后, 应用程序必须调用此函数, 来表明接受还是拒绝对于当前目标的重定向. 应用程序可以使用 *on_call_redirected()* 中 *PJSIP_REDIRECT_PENDING* 命令的组合, 函数在重定向前得到使用者的许可

请注意, 如果应用程序选择拒绝或停止重定向 (分别使用 *PJSIP_REDIRECT_REJECT* 或 *PJSIP_REDIRECT_STOP*), 这个函数返回之前通话断线回调将被调用. 如果应用程序拒绝目标, 如有另一种尝试的目标在这个函数返回之前 *on_call_redirected()* 回调也可能被调用

Parameters:

call_id call ID.

cmd 应用到当前目标的重定向操作. 其语义与 *on_call_redirected()* 回调中的描述相似, 除了 *PJSIP_REDIRECT_PENDING* 在这里不被接受

Returns:

成功操作 PJ_SUCCESS

pj_status_t pjsua_call_set_hold	(pjsua_call_id	<i>call_id,</i>
		const pjsua_msg_data *	<i>msg_data</i>
)		

保持指定的呼叫. 这将发送 re-INVITE 和适当的 SDP 来通知远端呼叫正被保持. 请求的最终状态将在 *on_call_media_state()* 回调中给出, 来报告应用程序呼叫的媒体状态已被改变

Parameters:

call_id Call identification.

msg_data 与请求一起发送的可选消息组件

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_call_reinvite	(pjsua_call_id	<i>call_id,</i>
		unsigned	<i>options,</i>
		const pjsua_msg_data *	<i>msg_data</i>
)		

发送 re-INVITE 来释放搁置 (保持). 请求的最终状态将在 *on_call_media_state()* 回调中给出, 来报告应用程序呼叫的媒体状态已被改变

Parameters:

call_id Call identification.

options *pjsua_call_flag* 常量的位掩码. 考虑到兼容性, 指定 *PJ_TRUE* 与指定 *PJSUA_CALL_UNHOLD* 相同.

msg_data 与请求一起发送的可选消息组件.

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_call_update	(pjsua_call_id	<i>call_id,</i>
---	---	-------------------------------	-----------------


```

        unsigned
        const pjsua\_msg\_data *      options,
                                   msg_data
    )

```

发送 UPDATE 请求

Parameters:

call_id Call identification
 options pjsua_call_flag 常量的位掩码
 msg_data 与请求一起发送的可选消息组件

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```

pj\_status\_t pjsua_call_xfer ( pjsua\_call\_id      call_id,
                               const pj\_str\_t *    dest,
                               const pjsua\_msg\_data * msg_data
                             )

```

初始化呼叫转移到特定的地址。函数将发送 REFER 请求来指示远端呼叫向指定的目标初始化, 发起新的 INVITE 会议

如果应用程序有兴趣检测转移请求的进展, 可以执行 *on_call_transfer_status()*

Parameters:

call_id 被转移的 call id
 dest 连接的新目的地址.
 msg_data 与请求一起发送的可选消息组件.

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```

pj\_status\_t pjsua_call_xfer_replaces ( pjsua\_call\_id      call_id,
                                         pjsua\_call\_id      dest_call_id,
                                         unsigned            options,
                                         const pjsua\_msg\_data * msg_data
                                       )

```

初始化现存的呼叫转移. 函数将发送 REFER 请求来指示远端呼叫向 *dest_call_id* 的 URI, 发起新的 INVITE 会议. *dest_call_id* 应该与 REFER 接受者建立会话"取代"原有会话

Parameters:

call_id 被转移的 call id
 dest_call_id 被替代的 call id
 options 应用程序应指定 PJSUA_XFER_NO_REQUIRE_REPLACES 来避免将"Require: replaces"加入到被 REFER 请求创建的 INVITE 请求。
 msg_data 与请求一起发送的可选消息组件

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```

pj\_status\_t pjsua_call_dial_dtmf ( pjsua\_call\_id      call_id,
                                     const pj\_str\_t *    digits
                                   )

```

使用 RFC 2833 有效负载格式向远端发送 DTMF

Parameters:

call_id Call identification
 digits 发送的 DTMF 数字串

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_call_send_im ( pjsua\_call\_id call_id,
                                const pj\_str\_t * mime_type,
                                const pj\_str\_t * content,
                                const pjsua\_msg\_data * msg_data,
                                void * user_data
                                )
```

在 INVITE 会话里发送即时信息

Parameters:

call_id Call identification
 mime_type 可选 MIME 类型. 如果 NULL, 设定为"text/plain"
 content 消息内容
 msg_data 包含在传出请求中的可选头列表. 在 msg_data 中的 body 描述符被忽略.
 user_data 可选用户数据, 当 IM 回调时将被返回

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_call_send_typing_ind ( pjsua\_call\_id call_id,
                                           pj\_bool\_t is_typing,
                                           const pjsua\_msg\_data * msg_data
                                           )
```

在 INVITE 会话里发送 IM 指示

Parameters:

call_id Call identification
 is_typing 非零想远端指示本地用户正在键入一个 IM
 msg_data 包含在传出请求中的可选头列表

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_call_send_request ( pjsua\_call\_id call_id,
                                       const pj\_str\_t * method,
                                       const pjsua\_msg\_data * msg_data
                                       )
```

发送任意请求. 例如发送 INFO 请求. 请注意, 应用程序不能使用该函数发送改变 invite 会话状态的请求, 比如 re-INVITE, UPDATE, PRACK, 和 BYE

Parameters:

call_id Call identification.
 method 请求的 SIP 方法
 msg_data 包含在传出请求中的可选消息体或头列表

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
void pjsua_call_hangup_all ( void )
```

终止所有会话. 为所有现在活跃的会话启用 [pjsua_call_hangup\(\)](#)

pj_status_t pjsua_call_dump	(pjsua_call_id	<i>call_id,</i>
		pj_bool_t	<i>with_media,</i>
		char *	<i>buffer,</i>
		unsigned	<i>maxlen,</i>
		const char *	<i>indent</i>
)		

转储呼叫和媒体统计为字符串

Parameters:

[call_id](#) Call identification
[with_media](#) 非零包含媒体信息
[buffer](#) 统计将被写入的缓冲区
[maxlen](#) 缓冲区最大长度
[indent](#) 左缩进空间

Returns:

PJ_SUCCESS (成功)

PJSUA-API Buddy, Presence, and Instant Messaging

[PJSUA API – 高级软电话 API]

好友管理, 好友状态和即时消息

数据结构

struct	pjsua_buddy_config
struct	pjsua_buddy_info

定义

#define	PJSUA_MAX_BUDDIES	256
#define	PJSUAPres_TIMER	300

枚举型

enum	pjsua_buddy_status	{	PJSUA_BUDDY_STATUS_UNKNOWN ,
		PJSUA_BUDDY_STATUS_ONLINE , PJSUA_BUDDY_STATUS_OFFLINE	}

函数

void	pjsua_buddy_config_default	(pjsua_buddy_config *cfg)
unsigned	pjsua_get_buddy_count	(void)
pj_bool_t	pjsua_buddy_is_valid	(pjsua_buddy_id buddy_id)
pj_status_t	pjsua_enum_buddies	(pjsua_buddy_id ids[], unsigned *count)
pjsua_buddy_id	pjsua_buddy_find	(const pj_str_t *uri)
pj_status_t	pjsua_buddy_get_info	(pjsua_buddy_id buddy_id, pjsua_buddy_info *info)
pj_status_t	pjsua_buddy_set_user_data	(pjsua_buddy_id buddy_id, void * user_data)
void *	pjsua_buddy_get_user_data	(pjsua_buddy_id buddy_id)
pj_status_t	pjsua_buddy_add	(const pjsua_buddy_config *buddy_cfg, pjsua_buddy_id *p_buddy_id)
pj_status_t	pjsua_buddy_del	(pjsua_buddy_id buddy_id)

```

pj\_status\_t      pjsua\_buddy\_subscribe\_pres    (pjsua\_buddy\_id    buddy_id,
pj\_bool\_t subscribe)
pj\_status\_t      pjsua\_buddy\_update\_pres (pjsua\_buddy\_id buddy_id)
pj\_status\_t      pjsua\_pres\_notify    (pjsua\_acc\_id    acc_id,    pjsua\_srv\_pres
*srv_pres, pjsip\_evsub\_state state, const pj\_str\_t *state_str,
const pj\_str\_t *reason, pj\_bool\_t with_body, const
pjsua\_msg\_data *msg_data)
void              pjsua\_pres\_dump (pj\_bool\_t verbose)
pj\_status\_t      pjsua\_im\_send    (pjsua\_acc\_id acc_id, const pj\_str\_t *to, const
pj\_str\_t *mime_type, const pj\_str\_t *content, const
pjsua\_msg\_data *msg_data, void *user\_data)
pj\_status\_t      pjsua\_im\_typing (pjsua\_acc\_id acc_id, const pj\_str\_t *to,
pj\_bool\_t is_typing, const pjsua\_msg\_data *msg_data)

```

变量

```

const          pjsip\_message\_method
pjsip\_method

```

详细说明

本节介绍 PJSUA 伙伴管理，存在管理，即时消息的 API。

Define Documentation

```
#define PJSUA_MAX_BUDDIES 256
```

好友列表的最大伙伴数目

```
#define PJSUA_PREP_TIMER 300
```

在重试 SUBSCRIBE 请求之前，库应该等待的时间，没有规则自动重新 subscribe。这也控制着在 PUBLISH 请求重试前的时间
默认: 300 秒

Enumeration Type Documentation

```
enum pjsua\_buddy\_status
```

基本好友的在线状态

Enumerator:

PJSUA_BUDDY_STATUS_UNKNOWN 在线状态未知(可能因为没有 subscription 建立).

PJSUA_BUDDY_STATUS_ONLINE 好友在线.

PJSUA_BUDDY_STATUS_OFFLINE 好友离线.

Function Documentation

```
void pjsua\_buddy\_config\_default    ( pjsua\_buddy\_config *    cfg )
```

好友配置设为默认值

```
unsigned pjsua\_get\_buddy\_count      (    void    )
```

得到好友总数目

Returns:

好友数目

```
pj\_bool\_t pjsua\_buddy\_is\_valid    ( pjsua\_buddy\_id    buddy_id )
```

检查好友 ID 是否有效

Parameters:

buddy_id 好友 ID.

Returns:

如果好友 ID 有效返回非零

```
pj\_status\_t pjsua_enum_buddies ( pjsua\_buddy\_id ids[],
                                unsigned *      count
                                )
```

枚举好友列表中所有 ID. 应用程序可以用 [pjsua_buddy_get_info\(\)](#) 来得到每个好友 ID 的相关信息

Parameters:

ids ID 数组

count 输入是数组的最大值, 返回时初始化的实际数量

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pjsua\_buddy\_id pjsua_buddy_find ( const pj\_str\_t * uri )
```

查找具有指定 URI 的好友 ID

Parameters:

uri 好友 URI.

Returns:

好友 ID, 如果没有返回 PJ_SUA_INVALID_ID

```
pj\_status\_t pjsua_buddy_get_info ( pjsua\_buddy\_id buddy_id,
                                   pjsua\_buddy\_info * info
                                   )
```

得到好友详细信息

Parameters:

buddy_id 好友 ID

info 接受好友信息的指针

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_buddy_set_user_data ( pjsua\_buddy\_id buddy_id,
                                         void *      user_data
                                         )
```

设置与好友相关的用户数据

Parameters:

buddy_id 好友 ID

user_data 用户数据

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
void* pjsua_buddy_get_user_data ( pjsua\_buddy\_id buddy_id )
```

得到与好友相关的用户数据

Parameters:

buddy_id 好友 ID

Returns:

应用程序数据

```
pj\_status\_t pjsua_buddy_add ( const pjsua\_buddy\_config * buddy_cfg,
                               pjsua\_buddy\_id * p_buddy_id
                             )
```

向好友列表添加好友.如果现在的 subscription 对于该好友是积极的,函数将立即启动当前的 subscription 会话

Parameters:

buddy_cfg 好友配置
p_buddy_id 接受好友 ID 的指针

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_buddy_del ( pjsua\_buddy\_id buddy_id )
```

删除指定好友. 任何现存的关于该好友的 subscription 将被终止

Parameters:

buddy_id 好友 ID

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_buddy_subscribe_pres ( pjsua\_buddy\_id buddy_id,
                                           pj\_bool\_t subscribe
                                         )
```

启动/禁止好友的现存检测.一旦好友的现状被订阅 (subscribed), 应用程序将通过 [on_buddy_state\(\)](#) 被告知好友现存状态的变化

Parameters:

buddy_id 好友 ID
subscribe 指定非零, 向特定用户激活存在的 subscription

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_buddy_update_pres ( pjsua\_buddy\_id buddy_id )
```

为好友更新状态信息.虽然库定期为所有好友更新存在的 subscription, 一些应用程序可能想要立即刷新, 这种情况可通过该函数完成

请注意, 如果检测启动, 好友的当前认购才启动. 详情参见 [pjsua_buddy_subscribe_pres\(\)](#). 如果的好友当前认购已经是活跃的, 这个函数将不作任何事情

一旦好友的当前订阅被成功启动, 应用程序可通过 [on_buddy_state\(\)](#) 被告知好友现状

Parameters:

buddy_id 好友 ID.

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_pres_notify ( pjsua\_acc\_id acc_id,
                                pjsua\_srv\_pres * srv_pres,
                                pjsip\_evsub\_state state,
                                const pj\_str\_t * state_str,
                                const pj\_str\_t * reason,
                                pj\_bool\_t with_body,
```

```

        const pjsua\_msg\_data *    msg_data
    )

```

发送 NOTIFY 来通知账户现在状态或者终止服务端当前 subscription. 如果应用程序想拒绝到来请求, 它应当设置状态为 `PJSIP_EVSUB_STATE_TERMINATED`

Parameters:

acc_id 账户 ID.
 srv_pres 服务器现存 subscription 实例.
 state 新的状态.
 state_str 如果不是 "active", "pending", 或 "terminated", 指定的状态名称.
 reason 如果新状态是 `PJSIP_EVSUB_STATE_TERMINATED`, 选择性地指定原因
 with_body 如果新状态是 `PJSIP_EVSUB_STATE_TERMINATED`, 指定是否 NOTIFY 请求应该包含具有账户当前信息的消息体.
 msg_data 要发送 NOTIFY 请求的可选头列表.

Returns:

`PJ_SUCCESS` (成功)

```

void pjsua_pres_dump ( pj\_bool\_t verbose )

```

转储当前 subscriptions 到日志

Parameters:

verbose Yes / no.

```

pj\_status\_t pjsua_im_send ( pjsua\_acc\_id          acc_id,
                             const pj\_str\_t *      to,
                             const pj\_str\_t *      mime_type,
                             const pj\_str\_t *      content,
                             const pjsua\_msg\_data * msg_data,
                             void *                user_data
                           )

```

发送即时信息 (对话外), 使用特定路由设置和权限的用户

Parameters:

acc_id 发送请求的账户 ID
 to 远程 URI
 mime_type 可选的 MIME 类型. 如果为空, 则选择 "text/plain"
 content 信息内容
 msg_data 包含在传出请求的头列表. 忽略在 msg_data 的 body 描述符
 user_data 可选用户数据, 当执行 IM 回调时将被返回

Returns:

`PJ_SUCCESS` (成功), 或相应的错误代码

```

pj\_status\_t pjsua_im_typing ( pjsua\_acc\_id          acc_id,
                              const pj\_str\_t *      to,
                              pj\_bool\_t            is_typing,
                              const pjsua\_msg\_data * msg_data
                            )

```

发送 typing 暗示 (对话外)

Parameters:

acc_id 发送请求的账户 ID

to 远程 URI

is_typing 如果非零,它将告诉远程人,本地用户正在撰写一个 IM

msg_data 被添加到传出请求中的可选头列表

Returns:

PJ_SUCCESS (成功), 或相应的错误代码.

Variable Documentation

const [pjsip_method](#) [pjsip_message_method](#)

MESSAGE 方法(在 pjsua_im.c 中定义)

PJSUA-API Media Manipulation

[PJSUA API – 高级软电话 API]

媒体操纵

数据结构

struct	pjsua_media_config
struct	pjsua_codec_info
struct	pjsua_conf_port_info
struct	pjsua_media_transport

定义

#define	PJSUA_MAX_CONF_PORTS 254
#define	PJSUA_DEFAULT_CLOCK_RATE 16000
#define	PJSUA_DEFAULT_AUDIO_FRAME_PTIME 20
#define	PJSUA_DEFAULT_CODEC_QUALITY 8
#define	PJSUA_DEFAULT_ILBC_MODE 30
#define	PJSUA_DEFAULT_EC_TAIL_LEN 200
#define	PJSUA_MAX_PLAYERS 32
#define	PJSUA_MAX_RECORDERS 32

函数

void	pjsua_media_config_default (pjsua_media_config *cfg)
unsigned	pjsua_conf_get_max_ports (void)
unsigned	pjsua_conf_get_active_ports (void)
pj_status_t	pjsua_enum_conf_ports (pjsua_conf_port_id id[], unsigned *count)
pj_status_t	pjsua_conf_get_port_info (pjsua_conf_port_id port_id, pjsua_conf_port_info *info)
pj_status_t	pjsua_conf_add_port (pj_pool_t *pool, pjmedia_port *port, pjsua_conf_port_id *p_id)
pj_status_t	pjsua_conf_remove_port (pjsua_conf_port_id port_id)
pj_status_t	pjsua_conf_connect (pjsua_conf_port_id source, pjsua_conf_port_id sink)
pj_status_t	pjsua_conf_disconnect (pjsua_conf_port_id source, pjsua_conf_port_id sink)
pj_status_t	pjsua_conf_adjust_tx_level (pjsua_conf_port_id slot, float level)

pj_status_t	pjsua_conf_adjust_rx_level (pjsua_conf_port_id slot, float level)
pj_status_t	pjsua_conf_get_signal_level (pjsua_conf_port_id slot, unsigned *tx_level, unsigned *rx_level)
pj_status_t	pjsua_player_create (const pj_str_t *filename, unsigned options , pjsua_player_id *p_id)
pj_status_t	pjsua_playlist_create (const pj_str_t file_names[], unsigned file_count, const pj_str_t *label, unsigned options , pjsua_player_id *p_id)
pjsua_conf_port_id	pjsua_player_get_conf_port (pjsua_player_id id)
pj_status_t	pjsua_player_get_port (pjsua_player_id id, pjmedia_port **p_port)
pj_status_t	pjsua_player_set_pos (pjsua_player_id id, pj_uint32_t samples)
pj_status_t	pjsua_player_destroy (pjsua_player_id id)
pj_status_t	pjsua_recorder_create (const pj_str_t *filename, unsigned enc_type, void *enc_param, pj_ssize_t max_size, unsigned options , pjsua_recorder_id *p_id)
pjsua_conf_port_id	pjsua_recorder_get_conf_port (pjsua_recorder_id id)
pj_status_t	pjsua_recorder_get_port (pjsua_recorder_id id, pjmedia_port **p_port)
pj_status_t	pjsua_recorder_destroy (pjsua_recorder_id id)
pj_status_t	pjsua_enum_aud_devs (pjmedia_aud_dev_info info [], unsigned *count)
pj_status_t	pjsua_enum_snd_devs (pjmedia_snd_dev_info info [], unsigned *count)
pj_status_t	pjsua_get_snd_dev (int *capture_dev, int *playback_dev)
pj_status_t	pjsua_set_snd_dev (int capture_dev, int playback_dev)
pj_status_t	pjsua_set_null_snd_dev (void)
pjmedia_port *	pjsua_set_no_snd_dev (void)
pj_status_t	pjsua_set_ec (unsigned tail_ms, unsigned options)
pj_status_t	pjsua_get_ec_tail (unsigned *p_tail_ms)
pj_bool_t	pjsua_snd_is_active (void)
pj_status_t	pjsua_snd_set_setting (pjmedia_aud_dev_cap cap, const void *pval, pj_bool_t keep)
pj_status_t	pjsua_snd_get_setting (pjmedia_aud_dev_cap cap, void *pval)
pj_status_t	pjsua_enum_codecs (pjsua_codec_info id[], unsigned *count)
pj_status_t	pjsua_codec_set_priority (const pj_str_t *codec_id, pj_uint8_t priority)
pj_status_t	pjsua_codec_get_param (const pj_str_t *codec_id, pjmedia_codec_param *param)
pj_status_t	pjsua_codec_set_param (const pj_str_t *codec_id, const pjmedia_codec_param *param)
pj_status_t	pjsua_media_transports_create (const pjsua_transport_config *cfg)

[pj_status_t](#) [pjsua_media transports attach](#) ([pjsua_media transport](#) tp[], unsigned count, [pj_bool_t](#) auto_delete)

详细说明

PJSUA 具有强大的媒体功能,其基于 PJMEDIA 会议桥建立. 基本上,所有的媒体 "ports" (比如呼叫, WAV 播放器, WAV 播放列表, 文件录制, 声音设备, 音频发生等) 都有会议桥完成,应用程序对此可以自由操纵

会议桥为应用程序提供强大的交换和混合功能. 使用会议桥, 每一个会议槽 (比如呼叫) 可以传送到多个目的地, 一个终端可以接受多个源. 如果在一个槽终止多个媒体终端, 会议桥将自动混合信号

应用程序可以通过调用 [pjsua_conf_connect\(\)](#) 连接两个媒体终端. 这将建立从源到 sink 终端的单向媒体流. 为了建立双向连接, 应用程序需要再次调用 [pjsua_conf_connect\(\)](#), 这次对调源和目的参数

例如, 传送 WAV 流到远方呼叫, 应用程序需要以下步骤:

```
pj\_status\_t stream_to_call( pjsua\_call\_id call_id )
{
    pjsua\_player\_id player_id;

    status = pjsua\_player\_create("mysong.wav", 0, NULL, &player_id);
    if (status != PJ_SUCCESS)
        return status;

    status = pjsua\_conf\_connect( pjsua\_player\_get\_conf\_port(),
                                pjsua\_call\_get\_conf\_port() );
}
```

PJSUA 媒体的其他特点:

- 高效的从 N 到 M 媒体终端之间的连接.
- 可以连接到自身, 创建回环媒体.
- 媒体终端可以有不同的时钟速率, 通过会议桥自动采样.
- 媒体终端可以有不同的框架; 会议桥将建立过滤调整二者差距.
- 当终端移除桥时, 互连自动终止.
- 声音设备可以随时改变.
- 从 RTCP 分组交换包正确报告呼叫的媒体质量 (在 [pjsua_call_dump\(\)](#))

Define Documentation

#define PJSUA_MAX_CONF_PORTS 254

会议桥的最多端口. 设置为 [pjsua_media config.max_media_ports](#) 的默认值

#define PJSUA_DEFAULT_CLOCK_RATE 16000

会议桥的默认时间速率. 设置为 [pjsua_media config.clock_rate](#) 的默认值

#define PJSUA_DEFAULT_AUDIO_FRAME_PTIME 20

会议桥的默认帧长度. 设置为 [pjsua_media config.audio_frame_ptime](#) 的默认值

#define PJSUA_DEFAULT_CODEC_QUALITY 8

默认编解码质量. 设置为 [pjsua_media config.quality](#) 的默认值

#define PJSUA_DEFAULT_ILBC_MODE 30

默认的 iLBC 模式. 设置为 [pjsua_media config.ilbc_mode](#) 的默认值

```
#define PJSUA_DEFAULT_EC_TAIL_LEN 200
```

默认的回波抵消尾长. 设置为 [pjsua_media_config.ec_tail_len](#) 的默认值

```
#define PJSUA_MAX_PLAYERS 32
```

文件播放器最大数

```
#define PJSUA_MAX_RECORDERS 32
```

文件播放器最大数

Function Documentation

```
void pjsua_media_config_default ( pjsua\_media\_config * cfg )
```

初始化媒体配置

Parameters:

cfg 媒体配置.

```
unsigned pjsua_conf_get_max_ports ( void )
```

获得会议端口的最大数目

Returns:

会议桥的端口最大数目

```
unsigned pjsua_conf_get_active_ports ( void )
```

获得会议桥中当前活跃端口数目

Returns:

数值.

```
pj\_status\_t pjsua_enum_conf_ports ( pjsua\_conf\_port\_id id[],  
                                   unsigned * count  
                                   )
```

枚举所有会议端口

Parameters:

id 会议端口 ID 数组

count 输入为指定的数组最大元素, 输出为它实际初始化的元素数目

Returns:

PJ_SUCCES (成功), 或相应的错误代码

```
pj\_status\_t pjsua_conf_get_port_info ( pjsua\_conf\_port\_id port_id,  
                                       pjsua\_conf\_port\_info * info  
                                       )
```

得到指定会议端口的信息

Parameters:

port_id 端口 ID

info 存储端口信息的指针

Returns:

PJ_SUCCES (成功), 或相应的错误代码

```
pj\_status\_t pjsua_conf_add_port ( pj\_pool\_t * pool,  
                                  pjmedia\_port * port,  
                                  pjsua\_conf\_port\_id * p_id  
                                  )
```

向 PJSUA 的会议桥添加端口. 应用程序可以使用该函数添加它创建的媒体端口。

对于 PJSUA-LIB(比如 calls, file player, or file recorder)创建的媒体端口, PJSUA-LIB 将自动添加端口到桥

Parameters:

pool 内存池
port 添加到桥的媒体端口
p_id 接受会议槽 id 的可选指针

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_conf_remove_port	(pjsua_conf_port_id	<i>port_id</i>)
--	---	------------------------------------	----------------	---

从会议桥移除槽. 如果应用程序通过 [pjsua_conf_add_port\(\)](#) 手动注册的端口, 它应该调用此函数

Parameters:

port_id 端口的槽 ID

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_conf_connect	(pjsua_conf_port_id	<i>source,</i>
		pjsua_conf_port_id	<i>sink</i>
)		

建立从源到 Sink 的单向连接. 一个源可以传送到多个目的地/sink. 如果多个源传送到同个 sink, 媒体将自动混合. 源和 sink 可建立有效的循环媒体
如果需要建立双效媒体流, 应用程序需要调用此函数两次

Parameters:

source 源媒体/传送者的 Port ID
sink 目的媒体/接受者的 Port ID

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_conf_disconnect	(pjsua_conf_port_id	<i>source,</i>
		pjsua_conf_port_id	<i>sink</i>
)		

断开从源到目的的媒体流

Parameters:

source 源媒体/传送者的 Port ID
sink 目的媒体/接受者的 Port ID

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

pj_status_t pjsua_conf_adjust_tx_level	(pjsua_conf_port_id	<i>slot,</i>
		float	<i>level</i>
)		

通过加大/减小声音, 调整从桥到指定端口传送信号的级别

Parameters:

slot 会议桥槽数目
level 调整的信号电平. 1 表示没有级别调整, 而 0 表示静音端口

Returns:

PJ_SUCCES (成功), 或相应的错误代码

pj_status_t pjsua_conf_adjust_rx_level	(pjsua_conf_port_id	<i>slot,</i>
		float	<i>level</i>
)		

通过加大/减小声音, 调整从指定端口到桥传送信号的级别

Parameters:

slot 会议桥槽数目

level 调整的信号电平. 1 表示没有级别调整, 而 0 表示静音端口

Returns:

PJ_SUCCES (成功), 或相应的错误代码

pj_status_t pjsua_conf_get_signal_level	(pjsua_conf_port_id	<i>slot,</i>
		unsigned *	<i>tx_level,</i>
		unsigned *	<i>rx_level</i>
)		

获得最后接受信号的级别. 信号级别是从 0 到 255 的整数值, 0 表示没信号, 255 表示最响亮的信号

Parameters:

slot 会议桥槽数

tx_level 从桥接收信号水平的可选参数

rx_level 从端口接收信号水平的可选参数

Returns:

PJ_SUCCESS (成功)

pj_status_t pjsua_player_create	(const pj_str_t *	<i>filename,</i>
		unsigned	<i>options,</i>
		pjsua_player_id *	<i>p_id</i>
)		

创建文件播放器, 自动添加播放器到会议桥

Parameters:

filename 播放文件名. 目前只支持 WAV 文件, WAV 必须为 16 位 PCM 单声道格式 (支持任意时钟速率)

options 可选的标志. 应用程序可指定 PJMEDIA_FILE_NO_LOOP 来避免循环播放

p_id 接收 player ID 的指针

Returns:

PJ_SUCCES (成功), 或相应的错误代码

pj_status_t pjsua_playlist_create	(const pj_str_t	<i>file_names[],</i>
		unsigned	<i>file_count,</i>
		const pj_str_t *	<i>label,</i>
		unsigned	<i>options,</i>
		pjsua_player_id *	<i>p_id</i>
)		

创建一个文件播放列表媒体端口, 自动添加端口到会议桥

Parameters:

file_names 添加到播放列表得文件名数组.注意文件必须有相同的时钟速率, 通道数和每个样本的比特数

file_count 文件数

label 设置媒体端口的可选标签

options 可选标志.应用程序可指定 PJMEDIA_FILE_NO_LOOP 来避免循环

p_id 接收 player ID 的可选指针

Returns:

PJ_SUCCES (成功), 或相应的错误代码

```
pjsua\_conf\_port\_id pjsua_player_get_conf_port ( pjsua\_player\_id id )
```

得到与播放器或播放列表相关的播放端口 ID

Parameters:

id 播放器 ID.

Returns:

会议端口 ID

```
pj\_status\_t pjsua_player_get_port ( pjsua\_player\_id id,
                                     pjmedia\_port\*\* p_port
                                   )
```

得到与播放器或播放列表相关的媒体端口.

Parameters:

id 播放器 ID

p_port 媒体端口

Returns:

PJ_SUCCESS (成功)

```
pj\_status\_t pjsua_player_set_pos ( pjsua\_player\_id id,
                                   pj\_uint32\_t samples
                                   )
```

设置摆放位置.操作对播放列表无效

Parameters:

id 文件播放器 ID.

samples 播放文职,一般应用程序可以指定从零开始重新播放.

Returns:

PJ_SUCCES (成功), 或相应的错误代码

```
pj\_status\_t pjsua_player_destroy ( pjsua\_player\_id id )
```

关闭播放列表文件, 从桥移除播放器,释放相关资源

Parameters:

id 文件播放器 ID

Returns:

PJ_SUCCES (成功), 或相应的错误代码

```
pj\_status\_t pjsua_recorder_create ( const pj\_str\_t* filename,
                                     unsigned enc_type,
                                     void* enc_param,
                                     pj\_ssize\_t max_size,
                                     unsigned options,
```



```

        pjsua\_recorder\_id * p_id
    )

```

创建文件记录器,自动连接录音机到会议桥。记录器目前支持 WAV 文件

Parameters:

filename 输出文件名

enc_type 如果该文件可以支持不同编码的编码器类型,可选指定被用来压缩媒体的方式,。现在这个值必须是零

enc_param 可选的指定编解码器的具体参数。对于.WAV 记录,值必须是 NULL.

max_size 最大文件大小. 指定 0 或-1 除去大小的限制。现在此值必须是零或-1

options 可选的选项

p_id 接收记录实例的指针

Returns:

PJ_SUCCES (成功), 或相应的错误代码

```

pjsua\_conf\_port\_id pjsua_recorder_get_conf_port ( pjsua\_recorder\_id id )

```

得到与记录器相关的会议端口

Parameters:

id recorder ID.

Returns:

会议端口 ID

```

pj\_status\_t pjsua_recorder_get_port ( pjsua\_recorder\_id id,
                                       pjmedia\_port ** p_port
                                     )

```

获取记录器的媒体端口

Parameters:

id recorder ID

p_port 媒体端口

Returns:

PJ_SUCCESS (成功)

```

pj\_status\_t pjsua_recorder_destroy ( pjsua\_recorder\_id id )

```

摧毁录音机(这将完成录音)

Parameters:

id recorder ID

Returns:

PJ_SUCCES (成功), 或相应的错误代码

```

pj\_status\_t pjsua_enum_aud_devs ( pjmedia\_aud\_dev\_info info[],
                                   unsigned * count
                                 )

```

枚举所有安装到系统的声音设备

Parameters:

info 信息的数组

count 输入最大元素数, 输出实际元素数

Returns:

PJ_SUCCES (成功), 或相应的错误代码

pj_status_t pjsua_enum_snd_devs	(pjmedia_snd_dev_info	<i>info[]</i> ,
		unsigned *	<i>count</i>
)		

枚举所有安装到系统的声音设备 (旧 API)

Parameters:

info 信息的数组
count 输入最大元素数, 输出实际元素数

Returns:

PJ_SUCCES (成功), 或相应的错误代码

pj_status_t pjsua_get_snd_dev	(int *	<i>capture_dev</i> ,
		int *	<i>playback_dev</i>
)		

到目前活跃的声音设备. 如果声音设备没被创建 (如未调用 [pjsua_start\(\)](#)) 函数可能返回, 设备 IDs 为 -1

Parameters:

capture_dev 捕获设备的 ID
playback_dev 播放设备的 ID

Returns:

PJ_SUCCES (成功), 或相应的错误代码

pj_status_t pjsua_set_snd_dev	(int	<i>capture_dev</i> ,
		int	<i>playback_dev</i>
)		

选择或更改声音设备, 应用程序将可在任何时候调用它来取代当前声音设备

Parameters:

capture_dev 捕获设备 ID
playback_dev 播放设备 ID

Returns:

PJ_SUCCES (成功), 或相应的错误代码

pj_status_t pjsua_set_null_snd_dev	(void)
--	---	------	---

设置 pjsua 使用空的声音设备. 空的设备仅提供会议桥所需要的时间, 不实现任何硬件连接

Returns:

PJ_SUCCES (成功), 或相应的错误代码

pjmedia_port * pjsua_set_no_snd_dev	(void)
---	---	------	---

从声音设备断开会会议桥, 让应用程序连接桥到自己的声音设备或主端口

Returns:

会议桥的端口连接, 这样应用程序可连接桥到自己的声音设备或主端口

pj_status_t pjsua_set_ec	(unsigned	<i>tail_ms</i> ,
		unsigned	<i>options</i>
)		

改变回波抵消设置

这个函数的行为取决于目前声音设备是否活跃, 如果是这样, 无论设备或软件的

AEC 是否正在使用的

如果目前声音设备是不活跃的, 这只会变更该默认 AEC 设置和设置将在声音设备下一次被打开时应用

Parameters:

`tail_ms` 尾长, 以毫秒为单位. 设置 0 来禁止 AEC
`options` 选项可传递到 `pjmedia_echo_create()`. 通常值为 0

Returns:

PJ_SUCCESS (成功)

pj_status_t <code>pjsua_get_ec_tail</code>	(<code>unsigned *</code>	<code>p_tail_ms</code>)
--	---	-------------------------	------------------------	---

获得当前回波消除尾长

Parameters:

`p_tail_ms` 接收尾巴长度的指针, 以毫秒为单位。如果 AEC 被禁用, 该值将为零.

Returns:

PJ_SUCCESS (成功)

pj_bool_t <code>pjsua_snd_is_active</code>	(<code>void</code>)
--	---	-------------------	---

检查目前声音设备是否活跃. 如果应用程序已设置自动关闭功能为非零 (`pjsua_media_config` 的 `snd_auto_close_time`), 或者没有声音设备通过 [pjsua_set_no_snd_dev\(\)](#) 配置, 声音设备将无效

pj_status_t <code>pjsua_snd_set_setting</code>	(pjmedia_aud_dev_cap	<code>cap,</code>
	<code>const void *</code>	pj_bool_t	<code>pval,</code>
			<code>keep</code>
)		

设置正在使用的声音设备。如果声音设备是目前活跃, 功能设置将立即重设声音设备实例

如果 “keep” 设置为非零值, 设置将保存为未来使用声音设备。如果目前声音设备是无效, 并 “keep” 是假, 函数将返回错误

Parameters:

`cap` 设置改变的声音设备
`pval` 指针值。关于应用于每个设置的值类型请参见 `pjmedia_aud_dev_cap`
`keep` 设置是否供将来使用

Returns:

PJ_SUCCESS (成功) 或相应错误代码。

pj_status_t <code>pjsua_snd_get_setting</code>	(pjmedia_aud_dev_cap	<code>cap,</code>
	<code>void *</code>		<code>pval</code>
)		

获取声音设备设置. 如果目前设备是活跃的, 函数将转发请求到声音设备。如果声音设备不活跃, 函数返回原有设置并保持此设置。否则函数返回错误

Parameters:

`cap` 声音设备
`pval` 接收指针值

Returns:

PJ_SUCCESS (成功) 或相应错误代码

```
pj\_status\_t pjsua_enum_codecs ( pjsua\_codec\_info id[],
                                unsigned * count
                                )
```

枚举系统支持的所有编解码器

Parameters:

id ID 数组.

count 输入为最大元素数, 输出为实际数目

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_codec_set_priority ( const pj\_str\_t * codec_id,
                                       pj\_uint8\_t priority
                                       )
```

改变编码器的优先级

Parameters:

codec_id 编码器 ID, 它是唯一确定编解码器的一个字符串 (比如 "speex/8000")

priority 优先级, 0-255, 0 表示禁用编解码器

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_codec_get_param ( const pj\_str\_t * codec_id,
                                    pjmedia\_codec\_param * param
                                    )
```

获得编解码器的参数

Parameters:

codec_id 编解码器 ID

param 接受编解码器的结构体

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_codec_set_param ( const pj\_str\_t * codec_id,
                                    const pjmedia\_codec\_param * param
                                    )
```

设置编解码器参数

Parameters:

codec_id 编解码器 ID

param 编解码参数. 设置 NUL 来重启默认设置

Returns:

PJ_SUCCESS (成功), 或相应的错误代码

```
pj\_status\_t pjsua_media_transports_create ( const pjsua\_transport\_config * cf,
                                             g
                                             )
```

为所有电话创建 UDP 媒体传输

Parameters:

cfg 媒体传输配置

Returns:

PJ_SUCCE (成功), 或相应的错误代码

pj_status_t	(pjsua_media_transpor	<i>tp[]</i> ,
<code>pjsua_media_transports_attach</code>		t	
		unsigned	<i>count</i> ,
		pj_bool_t	<i>auto_delet</i>
			<i>e</i>
)		

注册自定义的媒体传输

Parameters:

`tp` 媒体传输矩阵

`count` 矩阵中的参数数., 数目必须与当 `pjsua` 创建时配置的最大呼叫数一致

`auto_delete` 标志, 当 `pjsua` 关闭时传输是否销毁

Returns:

PJ_SUCCE (成功), 或相应的错误代码