

# OTP 认证引擎开发指南



V1.0

2012-08

坚石诚信科技有限公司

网址: [www.jansh.com.cn](http://www.jansh.com.cn)

## 章节目录

<b>第 1 章 关于本文档 .....</b>	<b>1</b>
1.1 读者对象 .....	1
1.2 反馈意见 .....	1
1.3 文档约定 .....	1
<b>第 2 章 OTP 认证引擎简介 .....</b>	<b>2</b>
2.1 头文件 .....	2
2.2 库文件 .....	2
2.3 示例程序 .....	3
<b>第 3 章 应用开发接口说明 .....</b>	<b>4</b>
3.1 OTP 认证引擎接口一览表 .....	4
3.2 数据结构说明 .....	5
3.3 种子文件功能接口说明 .....	8
3.3.1 init_pskc .....	8
3.3.2 read_pskc_rec .....	8
3.3.3 uninit_pskc .....	9
3.4 私有数据功能接口说明 .....	9
3.4.1 set_key .....	10
3.4.2 encode_pdata .....	10
3.4.3 decode_pdata .....	10
3.4.4 dump_pdata .....	11
3.5 业务驱动接口说明 .....	11
3.5.1 enable_token .....	11
3.5.2 disable_token .....	12
3.5.3 set_nextmode_threshold .....	12
3.5.4 set_wnd .....	13
3.5.5 set_otp_reused .....	14
3.5.6 set_chpass_reused .....	15
3.5.7 set_user_login .....	15
3.5.8 set_pin .....	16
3.5.9 get_passcode_time .....	16
3.5.10 check_password .....	16
3.5.11 genotp .....	17
3.5.12 resynch_token .....	18
3.5.13 resynch2_token .....	18
3.5.14 gen_chlg .....	19
3.5.15 check_chpass .....	20
3.5.16 gen_chpass .....	20
3.5.17 gen_hacode .....	21
3.5.18 get_puk .....	22
3.5.19 get_puk2 .....	22
3.5.20 check_sigpass .....	23
3.5.21 gen_sigpass .....	24
3.5.22 update_key .....	24
3.5.23 check_chap_pass .....	25
3.5.24 check_mschap_pass .....	26
3.5.25 gen_ac .....	27
3.5.26 gen_ac_cipher .....	28
<b>第 4 章 令牌业务驱动流程及示例程序 .....</b>	<b>29</b>

4.1 应用开发流程 .....	29
4.1.1 系统测试 .....	29
4.1.2 应用环境 .....	29
4.1.3 示例程序使用流程 .....	30
4.1.4 应用开发 .....	32
4.2 令牌业务驱动流程及示例程序 .....	32
4.2.1 导入种子 .....	32
4.2.2 激活令牌 .....	33
4.2.3 注销令牌 .....	35
4.2.4 更新密钥 .....	36
4.2.5 获取解锁码 .....	37
4.2.6 同步令牌 .....	39
4.2.7 验证动态口令 .....	41
4.2.8 生成动态口令 .....	43
4.2.9 生成挑战值 .....	44
4.2.10 验证应答动态口令 .....	45
4.2.11 生成应答动态口令 .....	46
4.2.12 验证签名动态口令 .....	47
4.2.13 生成签名动态口令 .....	48
4.2.14 生成主机认证码 .....	49
4.2.15 验证 CHAP 动态口令 .....	51
4.2.16 验证 MSCHAP 动态口令 .....	52
4.2.17 生成手机令牌激活码 .....	54
4.2.18 生成手机令牌加密激活码 .....	55
<b>第 5 章 返回值列表 .....</b>	<b>57</b>

# 第1章 关于本文档

本文档的目的是帮助开发人员使用 OTP 认证引擎 SDK 提供的接口与应用系统进行集成，实现在应用系统中增加动态口令身份认证功能来提高应用系统的安全性。本文档的主要内容包括 OTP 认证引擎简介、应用开发接口说明、令牌业务驱动流程及示例程序等。

## 1.1 读者对象

本文档的读者对象为使用 OTP 认证引擎 SDK 提供的接口进行系统集成的技术开发人员、二次开发人员、测试人员以及其它相关人员。

## 1.2 反馈意见

我们非常欢迎并重视您对本文档所发表的意见和建议。如果您愿意，您可以将您的意见或建议通过电子邮件发送给我们。

- 国内技术支持：[techsup@jansh.com.cn](mailto:techsup@jansh.com.cn)

## 1.3 文档约定

为了方便读者阅读和理解本文档，在文档编写过程中，我们遵循如下约定：

(1) 本文档所述内容包括在 Windows、Linux 平台上，使用 OTP 认证引擎的 C/C++ 语言开发接口进行应用开发。除特殊说明以外，所描述的内容适合这里提到的所有平台。

(2) 本文档中的示例程序的编译、调试和运行以前，假设已经安装了编译器，如：Microsoft Visual C++、GNU C 等。

(3) 本文档以 C 语言说明 OTP 认证引擎的使用方法，在一定规则下，同样适用于 C++ 环境。

## 第2章 OTP 认证引擎简介

OTP 认证引擎是支持各种令牌业务的 API 函数库，由于其具备功能简单、易于集成和部署等特点，已在基于动态口令认证的应用场合里得到了广泛应用。

OTP 认证引擎 SDK 包括下列文件。

### 2.1 头文件

头文件中包含 OTP 认证引擎所提供的接口的定义，位于“include”目录下。开发人员在进行 OTP 认证引擎应用开发时，应将头文件包含到自己的应用程序中。

**ftauthng.h 文件** 定义了 OTP 认证引擎提供的私有数据功能接口和业务驱动接口、特殊类型以及返回值。

**ftngpskc.h 文件** 定义了 OTP 认证引擎提供的种子文件功能接口。

### 2.2 库文件

不同的操作系统环境下使用的库文件是不同的，因此本节将针对不同的平台分别进行说明。

#### 【Windows 平台】

Windows 平台上的库文件位于“lib/libftauthng\_win32.zip”（适用于 Windows 32 位操作系统）和“lib/libftauthng\_win64.zip”（适用于 Windows 64 位操作系统）目录下，在 Windows 平台上进行应用开发以前，应该将库文件拷贝到目标计算机的适当目录（比如系统目录或者是当前目录）。Windows 平台上的库文件主要包括下面几个文件：

**libftauthng.dll 文件** OTP 认证引擎的动态库文件。

**libftauthng.lib 文件** OTP 认证引擎的动态库链接文件。

#### 【Linux 平台】

Linux 平台上的库文件位于“lib/libftauthng\_linux2.6\_x86.tar.gz”（适用于 Linux 32 位操作系统）和“libftauthng\_linux2.6\_x64.tar.gz”（适用于 Linux 64 位操作系统）目录下，在 Linux 平台上进行应用开发以前，应该将库文件拷贝到目标计算机的适当目录（比如系统目录或者是当前目录）。Linux 平台上的库文件主要包括以下文件：

**libftauthng.so.1.0.1 文件** OTP 认证引擎实际的动态库文件。

**libftauthng.so** 文件 指向 OTP 认证引擎动态库的软链接。

**libftauthng.so.1** 文件 指向 OTP 认证引擎动态库的软链接。

## 2.3 示例程序

示例程序位于“**test**”目录下，开发人员可以编辑、编译和调试该程序。通过该程序，开发人员可以初步了解 OTP 认证引擎的用法。提供的示例程序文件具体包括：

**test\_pskc.c** 文件 提供了 OTP 认证引擎中与种子文件操作相关的接口的示例程序。

**test\_stat.c** 文件 提供了 OTP 认证引擎中与令牌业务（如令牌的激活、注销，动态口令最大重试次数、窗口值的设置，解锁码的生成等）相关的接口的示例程序。

**test\_auth.c** 文件 提供了 OTP 认证引擎中与认证相关的接口的示例程序，包括动态口令验证接口、应答动态口令验证接口、签名动态口令验证接口、主机认证码生成接口等。

**test\_chap.c** 文件 提供了 OTP 认证引擎中与 CHAP 形式、MSCHAP 形式的动态口令认证相关的接口的示例程序。

**test\_sync.c** 文件 提供了 OTP 认证引擎中与令牌同步相关的接口的示例程序。

**test\_mobile.c** 文件 提供了 OTP 认证引擎中与手机激活码/加密激活码生成相关的接口的示例程序。

## 第3章 应用开发接口说明

OTP 认证引擎主要用于实现动态令牌的各种业务操作，其提供的外部接口为驱动接口，各驱动接口是由种子文件解析模块，私有数据编、解码模块，以及业务驱动模块共同实现的。通过使用这些接口，开发人员可以开发出符合自己需要的认证服务器和管理工具，实现和应用系统的无缝集成。

### 3.1 OTP 认证引擎接口一览表

种子文件功能接口	
init_pskc	初始化种子导入接口
read_pskc_rec	从解密的数据中逐个读取令牌种子信息
uninit_pskc	反初始化种子导入接口
私有数据功能接口	
set_key	设置私有数据保护密钥
encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中
decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
dump_pdata	输出令牌结构数据
业务驱动接口	
enable_token	令牌首次使用时激活令牌，以及令牌的解锁
disable_token	注销令牌
set_nextmode_threshold	设置动态口令的最大重试次数
set_wnd	设置窗口值
set_otp_reused	设置动态口令是否可重复使用
set_chpass_reused	设置挑战应答或签名是否可重复使用
set_user_login	在令牌注册时设定令牌的用户名
set_pin	设定 PIN 码（认证时使用）
get_passcode_time	获取系统当前时间
check_password	验证动态口令

genotp	生成当前的动态口令
resynch_token	同步令牌（需要两次调用此函数）
resynch2_token	同步令牌（仅需调用一次，需要同时传入两个连续的动态口令）
gen_chlg	生成挑战值
check_chpass	验证应答动态口令
gen_chpass	生成当前的应答动态口令
gen_hacode	生成主机认证码
get_puk	获取一级解锁码
get_puk2	获取二级解锁码
check_sigpass	验证签名动态口令
gen_sigpass	生成当前的签名动态口令
update_key	更新种子密钥
check_chap_pass	验证 CHAP 形式的动态口令
check_mschap_pass	验证 MSCHAP 形式的动态口令
gen_ac	生成手机令牌激活码
gen_ac_cipher	生成手机令牌密文激活码

## 3.2 数据结构说明

otp\_pdata 结构的定义如下：

```
typedef struct otp_pdata
{
    unsigned int token_type; //令牌类型
    unsigned int stage1;    //0 初始状态, 1 已绑定
    unsigned int stage2;    //0 停用, 1 启动, 2 已锁定, 3 已挂失, 4 第一次同步成功
    unsigned int stage3;    //0 非 NEXT, 1 NEXT
    unsigned int birth;     //令牌密钥初始化时间, 相对于 EPOCH 的秒数
    unsigned int death;     //令牌自动废止时间, 相对于 EPOCH 的秒数, 超出则作废

    /*****
    动态口令算法相关
    *****/
    unsigned int algid;     //算法标识
    unsigned int otp_len;   //OTP 长度
}
```



```

unsigned int t0;           //起始参考时间(短信 OTP 时间戳)
unsigned int interval;    //间隔时间(短信 OTP 的存活周期)
int drift;                //漂移次数

//小窗口(认证窗口), 事件型默认为 40, 时间型默认为 2
unsigned int auth_wnd;

//中窗口, 事件默认为 100, 时间默认为 4
unsigned int medium_wnd;

//大窗口(同步窗口), 事件型默认值为 200, 时间型默认值为 40
unsigned int sync_wnd;

//带事件的挑战应答使用的窗口
unsigned int cr_auth_wnd;

unsigned int adj_period;   //认证窗口调整为中窗口大小的周期, 默认为 2 周

unsigned int login_err;    //登录错误次数
unsigned int max_repeat;   //动态口令最大重试次数
unsigned int lock_time;    //上次锁定时间, stage2 状态变化时间戳
unsigned int lock_interval; //锁定最大维持周期(秒), 默认为永久锁定
unsigned int drft_time;    //上次校准时钟漂移的时间戳
unsigned int last_succ;    //上次认证成功系统本地时间

unsigned int puk_itv;      //一级解锁码时钟周期
unsigned int puk2_itv;     //二级解锁码时钟周期
unsigned int upresp_len;   //密钥更新响应长度
unsigned int st_new;       //是否为密钥更新完成待首次认证状态

uint64_t auth_base;        //认证基数, 或 TOTP 上次登录成功令牌时间因子
uint64_t cr_tmbase;        //带时间的挑战应答令牌上次成功令牌时间因子
uint64_t cr_base;         //带时间/事件的挑战应答中的认证基数

unsigned int key_len;       //共享密钥(二进制格式)长度
unsigned char key[MAX_TOKENKEY_SIZE]; //共享密钥(二进制格式)
unsigned char key_hash[KEY_HASH_SIZE]; //共享密钥摘要

char sn[MAX_TOKENSN_LEN + 1]; //令牌序列号

char cr_suite[MAX_OCRASUITE_LEN + 1]; //挑战应答 SUITE
char sig_suite[MAX_OCRASUITE_LEN + 1]; //签名 SUITE
char ha_suite[MAX_OCRASUITE_LEN + 1]; //主机认证 SUITE

```

```

char userid[MAX_USERID_LEN + 1];      //令牌注册时的用户名
char pin[MAX_PIN_LEN + 1];            //PIN 码

unsigned int k2_len;                   //新共享密钥长度
unsigned char k2[MAX_TOKENKEY_SIZE];   //新共享密钥
unsigned char k2_hash[KEY_HASH_SIZE];  //新共享密钥摘要

unsigned int otp_lifecycle;            //动态口令的存活周期
unsigned int set_otp;                  //动态口令保存时的时间戳
char otp[OTP_SIZE];                   //动态口令

unsigned int chp_lifecycle;            //挑战应答口令的存活周期
unsigned int set_chp;                  //挑战应答口令保存时的时间戳
char chlg[CHLG_SIZE];                 //挑战值
char chp[OTP_SIZE];                   //挑战应答口令
} otp_pdata_t;

```

StTokenData 结构的定义如下:

```

typedef struct StTokenData
{
    otp_pdata_t pdata;                //令牌数据结构
    char priv_data[MAX_PRIVATE_DATA_SIZE]; //私有数据
} StTokenData;

```

其中:

**pdata** 表示令牌私有数据的结构体表示, 其中包括令牌私有数据的各个成员的定义。

**priv\_data[MAX\_PRIVATE\_DATA\_SIZE]** 表示令牌私有数据, 该数据做为 OTP 引擎接口的私有数据, 是 OTP 引擎接口的基础数据, 该数据的最大长度为 1024 字节。

MSCHAP 的版本的定义如下:

```

typedef enum
{
    MSCHAP_V1 = 1,
    MSCHAP_V2 = 2
} mschap_ver_t;

```

## 3.3 种子文件功能接口说明

OTP 认证引擎提供的种子文件功能接口主要是用于解析令牌种子文件中的令牌种子信息，并转化为令牌私有数据字符串。它包括三个接口：`init_pskc` 接口用于初始化，指定令牌文件（XML 格式）；`read_pskc_rec` 接口用于逐个读取指定的种子信息；`uninit_pskc` 接口用于实现反初始化，种子导入完成以后调用该接口用于资源释放。

### 3.3.1 `init_pskc`

`init_pskc` 函数主要是用于初始化种子导入接口，主要操作包括打开 XML 种子文件，进行格式分解，以及返回空间和总令牌数。

#### 【语法】

```
int init_pskc(  
    char *file,  
    char *keyfile,  
    char *pass,  
    void **ctx,  
    unsigned int *recs  
);
```

#### 【参数】

**file** 令牌文件的名称。

**keyfile** 密钥文件的名称。可以是对称密钥或 RSA 私钥。

**pass** 密码。可以是 PBE 密码或 RSA 私钥的保护密码。

**ctx** 解密的空间。使用 `uninit_pskc` 函数进行释放。

**recs** 总共的令牌数。

#### 【返回值】

该函数的返回值为一个整数，如果返回 `FT_API_SUCC(0)`，表示初始化成功；否则，表示初始化失败，返回值详细说明信息请参考本文档的“返回值列表”。

### 3.3.2 `read_pskc_rec`

`read_pskc_rec` 函数主要是用于从解密的数据中逐个读取令牌种子信息。

**【语法】**

```
int read_pskc_rec(
void *ctx,
StTokenData *pdata,
unsigned int i
);
```

**【参数】**

**ctx** 解密的空间。

**pdata** 组装成 PDATA。

**i** 记录的索引号，从 0 开始。

**【返回值】**

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示操作成功；否则，表示操作失败，返回值详细说明信息请参考本文档的“返回值列表”。

### 3.3.3 uninit\_pskc

uninit\_pskc 函数主要用于种子导入接口的反初始化，在种子导入完成以后调用该接口，以清除 XML 种子文件解密的内存空间。

**【语法】**

```
int uninit_pskc(void *ctx);
```

**【参数】**

**ctx** 解密的空间。

**【返回值】**

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示反初始化成功；否则，表示反初始化失败，返回值详细说明信息请参考本文档的“返回值列表”。

## 3.4 私有数据功能接口说明

私有数据功能接口主要是用于私有数据的编码和解码。

### 3.4.1 set\_key

set\_key 函数主要是用于设置私有数据保护密钥，一般在解析种子文件之前调用。在调用具体业务驱动之前，也必须再次使用该函数设置与先前解析种子文件时相同的保护密钥。如果不使用该函数，则认证引擎内部将使用默认的保护密钥。

注意：私有数据保护密钥仅在解析种子文件前设置，后续阶段不允许按需更新。此函数不允许多线程调用。

#### 【语法】

```
int set_key(unsigned char key[PROTECT_KEY_SIZE]);
```

#### 【参数】

key 保护密钥。该参数必须不小于 16 字节。

#### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示设置私有数据保护密钥成功；否则，表示设置失败，返回值详细说明信息请参考本文档的“返回值列表”。

### 3.4.2 encode\_pdata

encode\_pdata 函数主要用于将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv\_data 成员中。

#### 【语法】

```
int encode_pdata(StTokenData *pdata);
```

#### 【参数】

pdata 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

#### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示操作成功；否则，表示操作失败，返回值详细说明信息请参考本文档的“返回值列表”。

### 3.4.3 decode\_pdata

decode\_pdata 函数主要用于将 StTokenData 结构的 priv\_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中。

#### 【语法】

```
int decode_pdata(StTokenData *pdata);
```

**【参数】**

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**【返回值】**

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示操作成功；否则，表示操作失败，返回值详细说明信息请参考本文档的“返回值列表”。

### 3.4.4 dump\_pdata

dump\_pdata 函数主要用于输出令牌结构数据。

**【语法】**

```
void dump_pdata(otp_pdata_t *pdata);
```

**【参数】**

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**【返回值】**

无

**【注释】**

该函数要求 pdata 参数必须有效，并且在调用该函数之前已经调用过 decode\_pdata 函数。

## 3.5 业务驱动接口说明

业务驱动接口是 OTP 认证引擎接口的关键组成部分，它主要是用于实现动态令牌的各种业务操作，如令牌的启用、注销、认证、同步、密钥更新等。

注意：由于业务驱动接口所做的操作都是针对内存中的数据进行的，所以在接口调用完成以后，应将修改过的数据保存到数据库中，否则会导致信息丢失。

### 3.5.1 enable\_token

enable\_token 函数主要是用于令牌首次使用时激活令牌，以及令牌的解锁等操作。

**【语法】**

```
int enable_token(StTokenData *pdata);
```

**【参数】**

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**【返回值】**

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示令牌激活或令牌解锁成功；否则，表示操作失败，返回值详细说明信息请参考本文档的“返回值列表”。

**【注释】**

如果调用此接口返回成功，该接口会将令牌认证重试次数设置为 0，清除以前的认证重试次数记录，同时记录令牌的激活时间。

### 3.5.2 disable\_token

disable\_token 函数主要用于注销令牌。

**【语法】**

```
int disable_token(StTokenData *pdata);
```

**【参数】**

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**【返回值】**

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示注销令牌成功；否则，表示注销失败，返回值详细说明信息请参考本文档的“返回值列表”。

**【注释】**

如果调用该接口返回成功，该接口在注销令牌的同时，还将记录令牌的注销时间。

### 3.5.3 set\_nextmode\_threshold

set\_nextmode\_threshold 主要用于设置动态口令的最大重试次数。

**【语法】**

```
int set_nextmode_threshold(StTokenData *pdata, unsigned int repeat);
```

**【参数】**

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**repeat** 动态口令最大重试次数。该参数的有效范围为 0 到 256，如果设置的值超出了该有效范围，将会被自动设置为默认值 DEFAULT\_MAX\_REPEAT（10）。如果设置为 INFINITE\_MAX\_REPEAT（65535），表示认证或同步失败重试时不限制最大重试次数，即无论连续重试失败多少次，令牌都不会被锁定。

**【返回值】**

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示设置成功；否则，表示设置失败，返回值详细说明信息请参考本文档的“返回值列表”。

**3.5.4 set\_wnd**

为了提高令牌的易用性和安全性，同时为了满足应用系统的安全性、易用性和效率，OTP 认证引擎提供了小窗口、中窗口和大窗口检测和处理机制。

set\_wnd 函数主要用于设置小、中、大窗口的窗口值。

**【语法】**

```
int set_wnd(
    StTokenData *pdata,
    unsigned int auth_wnd,
    unsigned int med_wnd,
    unsigned int sync_wnd
);
```

**【参数】**

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**auth\_wnd** 小窗口值。小窗口（认证窗口），事件型默认值为 40，时间型默认值为 2。

**med\_wnd** 中窗口值。中窗口，事件型默认值为 100，时间型默认值为 4。

**sync\_wnd** 大窗口值。大窗口（同步窗口），事件型默认值为 200，时间型默认值为 40。

**【返回值】**

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示设置成功；否则，表示设置失败，返回值详细说明信息请参考本文档的“返回值列表”。

**【注释】**

小窗口是指能够一次性认证通过允许的时间相差的范围。小窗口在增强了认证系统易用性的同时，也



降低了口令被穷举猜中的几率。

中窗口是指系统允许通过连续二次输入正确动态口令来通过认证，并保持时间同步所允许的时间偏差。在时间发生较大偏移的情况下，输入的动态口令不能在小窗口内认证通过，系统会继续在中窗口范围内继续进行认证，如果在中窗口范围内找到，则提示用户输入下一个动态口令，如果下一个动态口令在中窗口范围内找到，则用户认证成功，服务器自动校准令牌的漂移，从而保持与令牌的同步。中窗口可以保证在令牌的时钟偏差超出小窗口时，通过输入下一个动态口令，使得令牌同步成功并校准漂移。

大窗口是指能够保持自动同步的最大的时间偏移值。大窗口的作用主要包括以下几个：（1）作为小窗口的扩展极限，也就是一次能认证通过的最大的偏差；（2）作为中窗口的扩展极限，也就是进入下一令牌码模式的最大偏差；（3）作为首次认证通过的窗口，对于第一次在系统中使用的令牌，系统将采用大窗口进行认证，并自动同步。时间偏差超过大窗口的令牌，不能通过自动的方式保持同步，只能通过手工同步的方式保存同步。大窗口在令牌首次认证及令牌手工同步时采用，可以保证令牌在首次使用或长时间不使用时，令牌仍能够同步成功并继续使用。

如果令牌漂移偏差超过大窗口，则通过手工同步，可以使时间偏差值更大范围内的令牌恢复同步。手工同步允许的偏差很大，一般情况下用于服务器时间调整带来的失去同步的情况。

通过手工同步不能恢复同步的令牌，作为报废处理。一般情况下很少出现这种情况，出现这种情况往往意味着令牌严重质量问题，应该进行报废。

### 3.5.5 set\_otp\_reused

set\_otp\_reused 函数主要是用于设置普通型动态口令在规定的时间周期内是否可以重复使用进行认证，以满足对在一定的时间周期内，需要使用同一个动态口令进行认证的需求。

#### 【语法】

```
int set_otp_reused(StTokenData *pdata, unsigned int itv);
```

#### 【参数】

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**itv** 可重复使用的时间周期。设置的时间周期不超过 MAX\_OTP\_LIFE（120 分钟），若该值超出 MAX\_OTP\_LIFE，则自动设置为不可重复使用（0 值表示不可重复使用）。

#### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示设置成功；否则，表示设置失败，返回值详细说明信息请参考本文档的“返回值列表”。

### 3.5.6 set\_chpass\_reused

set\_chpass\_reused 函数主要是用于设置挑战应答型动态口令或签名动态口令在规定的时间内是否可以重复使用进行认证，以满足对在一定的时间内，需要使用同一个动态口令进行认证的需求。

#### 【语法】

```
int set_chpass_reused(StTokenData *pdata, unsigned int itv);
```

#### 【参数】

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**itv** 可重复使用的时间周期。设置的时间周期不超过 MAX\_OTP\_LIFE（120 分钟），若该值超出 MAX\_OTP\_LIFE，则自动设置为不可重复使用（0 值表示不可重复使用）。

#### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示设置成功；否则，表示设置失败，返回值详细说明信息请参考本文档的“返回值列表”。

### 3.5.7 set\_user\_login

set\_user\_login 函数主要用于在令牌注册时设定令牌的用户名，该用户名一般设定为令牌序列号，以方便令牌的统一管理。

#### 【语法】

```
int set_user_login(StTokenData *pdata, const char *name);
```

#### 【参数】

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**name** 用户名。一般设定为令牌序列号。

#### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示设置成功；否则，表示设置失败，返回值详细说明信息请参考本文档的“返回值列表”。

#### 【注释】

name 参数的最大长度为 MAX\_USERID\_LEN（32），超出部分将被忽略，并且返回值不会体现是否有数据被忽略。

### 3.5.8 set\_pin

set\_pin 函数主要用于设定 PIN 码（认证时使用）。如果设定了 PIN 码，在认证时需要同时提供 PIN 码和动态口令进行认证。只有在 PIN 码和动态口令均正确的情况下，才允许用户访问。

#### 【语法】

```
int set_pin(StTokenData *pdata, const char *pin);
```

#### 【参数】

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**pin** 需要设定的 PIN 码（认证时使用）。如果该参数为空或是空字符（pin[0]为'\0'），则进行认证时，PIN 码参数也必须与该参数值一致。

#### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示设置成功；否则，表示设置失败，返回值详细说明信息请参考本文档的“返回值列表”。

#### 【注释】

设置 PIN 时，PIN 的最大长度为 MAX\_PIN\_LEN（32）个字符，如果长度超过这个值，超出的字符将会被忽略，并且返回值不会体现是否有数据被忽略。

### 3.5.9 get\_passcode\_time

get\_passcode\_time 函数主要用于获取系统当前时间。当使用 check\_password、resynch\_token 等需要系统当前时间参与的接口时，需要使用到此函数。

#### 【语法】

```
long get_passcode_time();
```

#### 【参数】

无

#### 【返回值】

该函数将返回当前时间相对于指定时间点 EPOCH (1970-01-01 00:00:00 UTC) 的时间差（单位：秒）。

### 3.5.10 check\_password

check\_password 函数主要用于验证动态口令。

注意：在短信令牌应用中，需要先调用 `genotp` 函数生成一个动态口令，并通过短信网关以短信的方式发送到用户的手机之后，再调用 `check_password` 函数进行认证。

#### 【语法】

```
int check_password(
    long tm,
    StTokenData *pdata,
    const char *otp,
    const char *pin
);
```

#### 【参数】

**tm** 系统当前时间。时间的有效范围应该是在令牌的生命周期以内，即大于令牌密钥初始化时间，小于令牌过期时间，否则会导致该接口调用失败。

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 **pdata** 为空，将会导致此接口调用失败。

**otp** 当前的动态口令（由令牌设备产生）。如果 **otp** 为空，将会导致该接口调用失败。

**pin** PIN 码（认证时使用）。无论令牌私有数据中的 **pin** 以及这里的参数 **pin** 是否为空，只要所提供的 **pin** 与令牌私有数据中保存的 **pin** 是一致的，就认为 **pin** 验证通过。

#### 【返回值】

该函数的返回值为一个整数，如果返回 `FT_API_SUCC(0)`，表示验证成功；否则，表示验证失败，返回值详细说明信息请参考本文档的“返回值列表”。

### 3.5.11 genotp

`genotp` 函数主要是用于生成当前的动态口令。

#### 【语法】

```
int genotp(
    long tm,
    StTokenData *pdata,
    char otp[OTP_SIZE]
);
```

#### 【参数】

**tm** 系统当前时间。时间的有效范围应该是在令牌的生命周期以内，即大于令牌密钥初始化时间，小于令牌过期时间，否则会导致该接口调用失败。

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**otp** 生成的当前动态口令（由认证引擎产生）。

#### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示当前动态口令生成成功；否则，表示生成失败，返回值详细说明信息请参考本文档的“返回值列表”。

### 3.5.12 resynch\_token

resynch\_token 函数主要是用于同步令牌。采用该函数进行令牌同步操作时，需要连续两次调用该函数，才能完成令牌的同步操作。

#### 【语法】

```
int resynch_token(
    long tm,
    StTokenData *pdata,
    char *otp
);
```

#### 【参数】

**tm** 系统当前时间。时间的有效范围应该是在令牌的生命周期以内，即大于令牌密钥初始化时间，小于令牌过期时间，否则会导致该接口调用失败。

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**otp** 当前的动态口令（由令牌设备产生）。如果 otp 为空，则会导致该接口调用失败。

#### 【返回值】

连续两次调用此函数，执行成功时返回值不同：第一次执行成功时，该函数返回 FT\_API\_NEXTOTP(400)；第二次执行成功时，该函数返回 FT\_API\_SUCC(0)。

其它返回值详细说明信息请参考本文档的“返回值列表”。

### 3.5.13 resynch2\_token

resynch2\_token 函数主要用于同步令牌。该函数与 resynch\_token 函数的功能相同，不同之处在于该函数需要同时传入两个连续的动态口令参与操作，如果匹配成功，则会校准漂移，完成令牌的同步操作，而无需像 resynch\_token 函数一样需要连续调用函数两次才能完成令牌的同步操作。

#### 【语法】

```
int resynch2_token(
    long tm,
    StTokenData *pdata,
    char *otp_prev,
    char *otp
);
```

#### 【参数】

**tm** 系统当前时间。时间的有效范围应该是在令牌的生命周期以内，即大于令牌密钥初始化时间，小于令牌过期时间，否则会导致该接口调用失败。

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**otp\_prev** 前一个动态口令（由令牌设备产生）。如果 otp\_prev 为空，则会导致该接口调用失败。

**otp** 当前的动态口令（由令牌设备产生）。如果 otp 为空，则会导致该接口调用失败。

#### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示同步成功；否则，表示同步失败，返回值详细说明信息请参考本文档的“返回值列表”。

### 3.5.14 gen\_chlg

gen\_chlg 函数主要用于生成挑战值。

#### 【语法】

```
int gen_chlg(StTokenData *pdata, char chlg[CHLG_SIZE]);
```

#### 【参数】

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**chlg** 生成的挑战值（由认证引擎产生）。

#### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示生成挑战值成功；否则，表示生成失败，返回值详细说明信息请参考本文档的“返回值列表”。

#### 【注释】

该函数支持的令牌类型为挑战应答型。

### 3.5.15 check\_chpass

check\_chpass 函数主要用于验证应答动态口令。

#### 【语法】

```
int check_chpass(  
    long tm,  
    StTokenData *pdata,  
    const char *otp,  
    const char *pin,  
    const char *chlg  
);
```

#### 【参数】

**tm** 系统当前时间。时间的有效范围应该是在令牌的生命周期以内，即大于令牌密钥初始化时间，小于令牌过期时间，否则会导致该接口调用失败。

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**otp** 应答动态口令，由令牌设备根据认证引擎生成的挑战值产生的当前应答动态口令。

**pin** PIN 码（认证时使用）。

**chlg** 挑战值（由认证引擎产生）。

#### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示验证成功；否则，表示验证失败，返回值详细说明信息请参考本文档的“返回值列表”。

#### 【注释】

该函数支持的令牌类型为挑战应答型。

### 3.5.16 gen\_chpass

gen\_chpass 函数主要用于生成当前的应答动态口令。

#### 【语法】

```
int gen_chpass(  
    long tm,  
    StTokenData *pdata,  
    char otp[RESP_SIZE],  
    const char *chlg  
);
```

```
);
```

#### 【参数】

**tm** 系统当前时间。时间的有效范围应该是在令牌的生命周期以内，即大于令牌密钥初始化时间，小于令牌过期时间，否则会导致该接口调用失败。

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 **pdata** 为空，将会导致此接口调用失败。

**otp** 应答动态口令，由认证引擎根据传入的挑战值产生的当前应答动态口令。

**chlg** 挑战值。

#### 【返回值】

该函数的返回值为一个整数，如果返回 **FT\_API\_SUCC(0)**，表示应答动态口令生成成功；否则，表示生成失败，返回值详细说明信息请参考本文档的“返回值列表”。

#### 【注释】

该函数支持的令牌类型为挑战应答型。

### 3.5.17 gen\_hacode

**gen\_hacode** 函数主要是用于生成主机认证码。

#### 【语法】

```
int gen_hacode(
    StTokenData *pdata,
    const char *chlg,
    char otp[RESP_SIZE]
);
```

#### 【参数】

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 **pdata** 为空，将会导致此接口调用失败。

**chlg** 挑战值（由令牌设备产生）。

**otp** 生成的主机认证码（由认证引擎根据挑战值产生）。

#### 【返回值】

该函数的返回值为一个整数，如果返回 **FT\_API\_SUCC(0)**，表示主机认证码生成成功；否则，表示生成失败，返回值详细说明信息请参考本文档的“返回值列表”。

#### 【注释】



该函数支持的令牌类型为挑战应答型。

### 3.5.18 get\_puk

`get_puk` 函数主要用于获取一级解锁码。挑战应答型令牌提供两级解锁的功能。当令牌设备开机密码连续输入错误次数达到开机密码允许重试的最大次数时，令牌将被锁定。令牌锁定之后，用户需要向服务器获取一个一级解锁码，并输入到令牌设备中进行解锁。如果输入的一级解锁码正确，则令牌解锁成功。

#### 【语法】

```
int get_puk(StTokenData *pdata, const char *chlg, char puk[PUK_SIZE]);
```

#### 【参数】

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 **pdata** 为空，将会导致此接口调用失败。

**chlg** 解锁挑战码。当令牌设备的解锁模式为挑战应答模式时，此参数值为令牌设备实际产生的解锁挑战码；若为其它模式时，则忽略该参数。

**puk** 一级解锁码（由认证引擎产生）。

#### 【返回值】

该函数的返回值为一个整数，如果返回 `FT_API_SUCC(0)`，表示获取一级解锁码成功；否则，表示获取失败，返回值详细说明信息请参考本文档的“返回值列表”。

#### 【注释】

该函数支持的令牌类型为挑战应答型。

### 3.5.19 get\_puk2

`get_puk2` 函数主要用于获取二级解锁码。挑战应答型令牌提供两级解锁的功能。在令牌设备已锁定的状态下，当一级解锁码连续输入错误次数达到一级解锁码允许重试的最大次数时，一级解锁码将被锁定。一级解锁码锁定之后，用户需要向管理员请求分发一个二级解锁码进行解锁，管理员通过服务器获取到一个二级解锁码，并发送给用户，由用户输入到令牌中。如果输入的二级解锁码正确，则令牌解锁成功。

#### 【语法】

```
int get_puk2(StTokenData *pdata, const char *chlg, char puk[PUK_SIZE]);
```

#### 【参数】

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 **pdata** 为空，将会导致此接口调用失败。

**chlg** 解锁挑战码。当令牌设备的解锁模式为挑战应答模式时，此参数值为令牌设备实际产生的解锁挑战码；若为其它模式时，则忽略该参数。

**puk** 二级解锁码（由认证引擎产生）。

#### 【返回值】

该函数的返回值为一个整数，如果返回 `FT_API_SUCC(0)`，表示获取二级解锁码成功；否则，表示获取失败，返回值详细说明信息请参考本文档的“返回值列表”。

#### 【注释】

该函数支持的令牌类型为挑战应答型。

### 3.5.20 check\_sigpass

`check_sigpass` 函数主要用于验证签名动态口令。

#### 【语法】

```
int check_sigpass(
long tm,
StTokenData *pdata,
const char *otp,
const char *pin,
const char *sigdata
);
```

#### 【参数】

**tm** 系统当前时间。时间的有效范围应该是在令牌的生命周期以内，即大于令牌密钥初始化时间，小于令牌过期时间，否则会导致该接口调用失败。

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 **pdata** 为空，将会导致此接口调用失败。

**otp** 签名动态口令，由令牌设备根据签名数据产生的当前签名动态口令。

**pin** PIN 码（认证时使用）。

**sigdata** 签名数据。

#### 【返回值】

该函数的返回值为一个整数，如果返回 `FT_API_SUCC(0)`，表示验证成功；否则，表示验证失败，返回值详细说明信息请参考本文档的“返回值列表”。

#### 【注释】

该函数支持的令牌类型为挑战应答型。

### 3.5.21 gen\_sigpass

gen\_sigpass 函数主要用于生成当前的签名动态口令。

#### 【语法】

```
int gen_sigpass(  
    long tm,  
    StTokenData *pdata,  
    char otp[RESP_SIZE],  
    const char *sigdata  
);
```

#### 【参数】

**tm** 系统当前时间。时间的有效范围应该是在令牌的生命周期以内，即大于令牌密钥初始化时间，小于令牌过期时间，否则会导致该接口调用失败。

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**otp** 签名动态口令，由认证引擎根据签名数据产生的当前签名动态口令。

**sigdata** 签名数据。

#### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示签名动态口令生成成功；否则，表示生成失败，返回值详细说明信息请参考本文档的“返回值列表”。

#### 【注释】

该函数支持的令牌类型为挑战应答型。

### 3.5.22 update\_key

update\_key 函数主要用于种子密钥更新，同时还会产生一个密钥更新响应值，用户需要将此响应值输入到令牌设备中，以完成设备端的密钥更新操作。

#### 【语法】

```
int update_key(  
    StTokenData *pdata,  
    char *r_tk,
```

```
char *tran_info,
char up_resp[MAX_UP_RESP_SIZE]
);
```

#### 【参数】

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 **pdata** 为空，将会导致此接口调用失败。

**r\_tk** 令牌随机数（由令牌设备产生，仅在密钥更新模式为令牌设备/后台服务器共同主导或者令牌主导时，使用该值）。

**tran\_info** 业务信息数据。如果为空，则使用业务系统内部随机生成的数据。

**up\_resp** 密钥更新响应值（由认证引擎产生）。如果密钥更新模式为后台服务器主导或者令牌设备/后台服务器共同主导时，则成功更新后认证引擎会返回实际值。

提示：目前密钥更新主要有三种模式：

1. 令牌设备主导模式：由令牌设备产生一组数据（令牌随机数），输入到后台服务器中完成密钥更新操作；
2. 后台服务器主导模式：由后台服务器产生一组数据（密钥更新响应），输入到令牌设备中完成密钥更新操作；
3. 令牌设备/后台服务器共同主导模式：令牌设备和后台服务器共同主导的密钥更新模式，实际采用的是一种挑战应答的方式，更新步骤如下：先由令牌设备产生一组数据（令牌随机数），传送至后台服务器，后台服务器根据令牌随机数产生一组数据（密钥更新响应），将该组数据输入到令牌设备中完成更新操作。

#### 【返回值】

该函数的返回值为一个整数，如果返回 **FT\_API\_SUCC(0)**，表示密钥更新成功；否则，表示更新失败，返回值详细说明信息请参考本文档的“返回值列表”。

#### 【注释】

该函数支持的令牌类型为挑战应答型。

### 3.5.23 check\_chap\_pass

**check\_chap\_pass** 函数主要用于验证 CHAP 形式的动态口令。

#### 【语法】

```
int check_chap_pass(
long tm,
```

```
StTokenData *pdata,
const unsigned char *chlg,
unsigned int chlg_len,
const unsigned char *pass,
unsigned int pass_len
);
```

#### 【参数】

**tm** 系统当前时间。时间的有效范围应该是在令牌的生命周期以内，即大于令牌密钥初始化时间，小于令牌过期时间，否则会导致该接口调用失败。

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 **pdata** 为空，将会导致此接口调用失败。

**chlg** 挑战值，由使用 CHAP 密码验证的客户端（如 VPN 设备）产生。

**chlg\_len** 挑战值的长度。

**pass** CHAP 口令，由 CHAP 客户端根据用户实际输入的动态口令（或包含了 PIN 码）生成。

**pass\_len** CHAP 口令的长度。

#### 【返回值】

该函数的返回值为一个整数，如果返回 **FT\_API\_SUCC(0)**，表示验证成功；否则，表示验证失败，返回值详细说明信息请参考本文档的“返回值列表”。

### 3.5.24 check\_mschap\_pass

check\_mschap\_pass 函数主要是用于验证 MSCHAP 形式的动态口令。

#### 【语法】

```
int check_mschap_pass(
long tm,
StTokenData *pdata,
mschap_ver_t ver,
const unsigned char *chlg,
unsigned int chlg_len,
const unsigned char *pass,
unsigned int pass_len,
unsigned char pswd_hash[16],
unsigned char hash_hash[16]
);
```

#### 【参数】

**tm** 系统当前时间。时间的有效范围应该是在令牌的生命周期以内，即大于令牌密钥初始化时间，小于令牌过期时间，否则会导致该接口调用失败。

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**ver** MSCHAP 的版本。

**chlg** 挑战值，由使用 MSCHAP 密码验证的客户端（如 VPN 设备）产生。

**chlg\_len** 挑战值的长度。

**pass** MSCHAP 口令，由 MSCHAP 客户端根据用户实际输入的动态口令（或包含了 PIN 码）生成。

**pass\_len** MSCHAP 口令的长度。

**pswd\_hash** 密码 HASH 缓冲区。

**hash\_hash** 密码 HASH 的摘要缓冲区。

### 【返回值】

该函数的返回值为一个整数，如果返回 FT\_API\_SUCC(0)，表示验证成功；否则，表示验证失败，返回值详细说明信息请参考本文档的“返回值列表”。

## 3.5.25 gen\_ac

gen\_ac 函数主要用于生成手机令牌激活码。用户在其手机设备上成功安装手机令牌应用程序之后，还需要通过使用激活码激活手机令牌，才能正常使用手机令牌的功能。

### 【语法】

```
int gen_ac(
    StTokenData *pdata,
    const char *udid,
    const char *ap,
    char ac[AC_MAX_SIZE]
);
```

### 【参数】

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 pdata 为空，将会导致此接口调用失败。

**udid** 手机令牌标识码，由手机令牌应用程序产生，它是用于限制待分发的手机令牌只能在指定手机的手机令牌应用程序中激活的一个重要参数。

**ap** 激活密码（由管理员设置或系统随机产生）。

**ac** 激活码（由认证引擎产生）。

**【返回值】**

该函数的返回值为一个整数，如果返回 `FT_API_SUCC(0)`，表示激活码生成成功；否则，表示生成失败，返回值详细说明信息请参考本文档的“返回值列表”。

**【注释】**

该函数仅适用于手机令牌。

### 3.5.26 gen\_ac\_cipher

`gen_ac_cipher` 函数主要用于生成手机令牌密文激活码。

**【语法】**

```
int gen_ac_cipher(  
    StTokenData *pdata,  
    const char *udid,  
    const char *ap,  
    const char *rand_buf,  
    char ac[AC_MAX_SIZE]  
);
```

**【参数】**

**pdata** 令牌私有数据及可见数据。此参数允许回写。如果 **pdata** 为空，将会导致此接口调用失败。

**udid** 手机令牌标识码，由手机令牌应用程序产生，它是用于限制待分发的手机令牌只能在指定手机的手机令牌应用程序中激活的一个重要参数。

**ap** 激活密码（由管理员设置或系统随机产生）。

**rand\_buf** 随机数。

**ac** 激活码（由认证引擎产生）。

**【返回值】**

该函数的返回值为一个整数，如果返回 `FT_API_SUCC(0)`，表示密文激活码生成成功；否则，表示生成失败，返回值详细说明信息请参考本文档的“返回值列表”。

**【注释】**

该函数仅适用于手机令牌。

## 第4章 令牌业务驱动流程及示例程序

为了帮助开发人员快速、直观、方便地理解和掌握 OTP 认证引擎接口的应用开发流程，本节将详细介绍令牌业务驱动流程，并提供示例程序，使开发人员可以及时、准确地了解到 OTP 认证引擎接口应用开发的基本步骤，为进一步使用 OTP 认证引擎接口进行应用开发奠定基础。

### 4.1 应用开发流程

使用 OTP 认证引擎进行应用开发时涉及到认证服务器和管理工具的业务逻辑和业务流程，对这些业务逻辑和业务流程的理解有利于帮助开发人员理解如何使用 OTP 认证引擎提供的接口进行应用开发。

#### 4.1.1 系统测试

在进行 OTP 认证引擎应用开发以前，建议开发人员首先安装和配置认证服务器和管理工具，并进行相关功能的测试，只有在认证服务器测试通过以后，才可以开始进行 OTP 认证引擎的应用开发。

通过对认证服务器的安装、配置和测试，了解相关功能之间的逻辑关系，对正确地使用 OTP 认证引擎进行应用开发具有非常重要的参考作用。对理解示例程序也有很大的帮助。

#### 4.1.2 应用环境

当应用系统的运行环境不同时，所需要用到的开发包也不一样，主要体现在操作系统的差别上。OTP 认证引擎可以在 Windows、Linux 平台下与应用系统进行集成。下表列出的操作系统及版本已经过详细测试，而其它版本的操作系统尚未进行详细测试。

操作系统 & 版本	备注
Windows2003 (32 位 / 64 位)	已测试
Windows2008 (32 位 / 64 位)	已测试
RedHat Linux AS 4 / 5 / 6 (x86 / x64)	已测试
Suse Linux 10 / 11 (x86 / x64)	已测试
Suse Enterprise Linux 10 / 11 (x86 / x64)	已测试
Ubuntu 9 / 10 / 11 (x86 / x64)	已测试
Debian 4 / 5 / 6 (x86 / x64)	已测试
CentOS 4 / 5 / 6 (x86 / x64)	已测试



### 4.1.3 示例程序使用流程

为了帮助开发人员快速、直观、方便地理解 OTP 认证引擎接口的用法，OTP 认证引擎 SDK 提供了示例程序，该示例程序演示了使用 OTP 认证引擎接口进行应用开发的基本过程。

#### 4.1.3.1 头文件

头文件中包含 OTP 认证引擎提供的接口的定义，通常在调用 OTP 认证引擎接口的工程的源码文件中包含头文件：

```
# include "ftauthng.h"
```

#### 4.1.3.2 库文件

当使用头文件提供的接口进行应用开发时，还需要在工程中添加对应的库文件。由于不同的操作系统环境下所使用的库文件是不同的，所以本节将针对不同的平台分别进行说明。

##### 【Windows 平台】

OTP 认证引擎提供的 Windows 系统平台上的动态库文件为 libftauthng.dll，动态库链接文件（Lib 文件）为 libftauthng.lib。

##### 1. 添加动态库文件

开发人员在使用示例程序进行认证测试以前应首先将 libftauthng.dll 动态库文件拷贝到示例程序能够找到的位置（比如 Windows 系统目录或当前目录），然后再通过开发工具对示例程序进行编辑、编译和链接。

##### 2. 添加动态库链接文件

开发人员可以通过两种方式来使用 OTP 认证引擎动态库：一种方式是头文件和 Lib 文件配合下的动态库功能调用，另一种方式是不需要头文件和 Lib 文件的前提下直接动态加载动态库来调用其中的功能。不管采用哪种方式，动态库链接文件都应该存在工程能够访问的目录下，比如工程当前目录、操作系统目录或者已经添加到系统 PATH 变量中的目录。

添加 lib 文件的通常做法是先将 OTP 认证引擎提供的 Lib 文件 libftauthng.lib 拷贝到工程相关目录下，然后开发人员可以通过下面三种方式来添加对该文件的引用。

三种引用方式都可以达到同一个目的，开发人员只需要选择使用其中一种方式即可。

（1）通过程序代码添加对 Lib 文件的引用，代码示例如下：

```
#pragma comment(lib, "libftauthng.lib")
```

通过该方式引用 Lib 文件时，要求 Lib 文件应保存在工程目录下，如果 Lib 文件在其它目录下，链接时可能会提示找不到对应的文件。

(2) 将 Lib 文件添加到工程中，添加完成以后，可以在工程的文件视图中看到已添加的 Lib 文件。通过该方式引用 Lib 文件时，Lib 文件可以不在工程目录下，这是因为在添加 Lib 文件时，已经记录了 Lib 文件的路径。

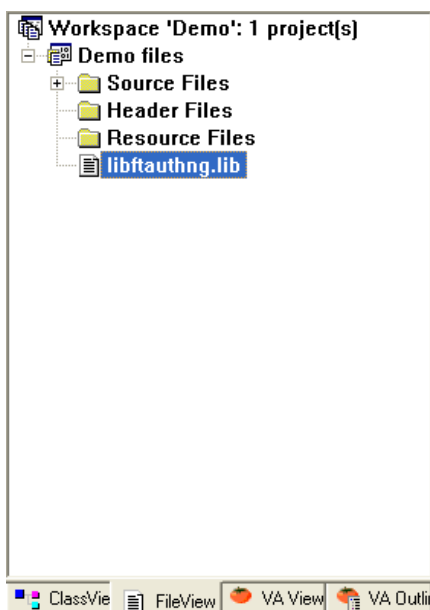


图 1 通过文件引用

(3) 将 Lib 文件名添加到工程的“Object/library modules”设置中，如下图所示。通过该方式引用 Lib 文件，要求 Lib 文件应保存在工程目录下，如果 Lib 文件在其它目录下，链接时可能会提示找不到该文件。

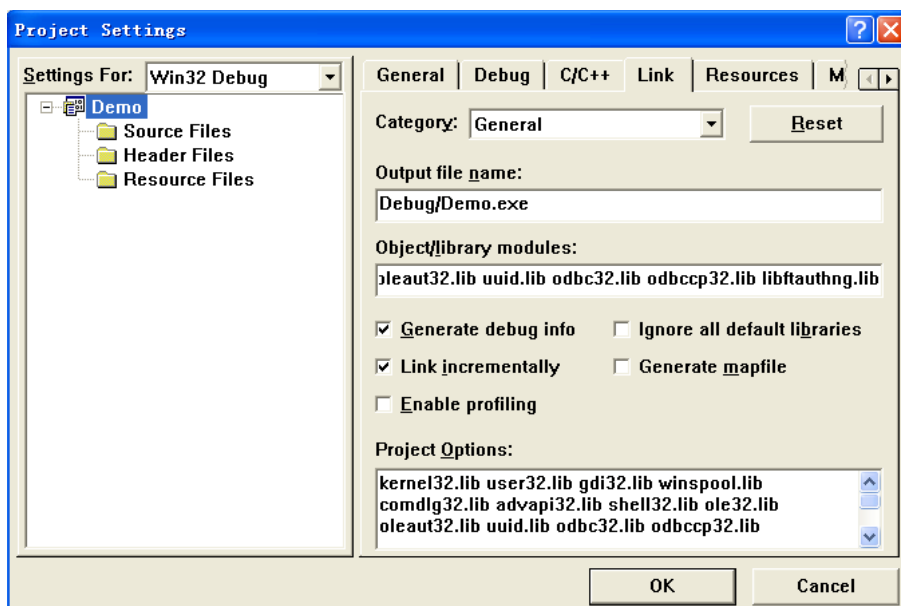


图 2 通过工程链接选项引用动态库链接文件

### 【Linux 平台】

OTP 认证引擎提供的 Linux 系统平台上的动态库文件为 libftauthng.so.1.0.1，同时提供了两个软链接文件：libftauthng.so 及 libftauthng.so.1。

开发人员在使用示例程序进行测试以前应将库文件 `libftauthng.so.1.0.1` 及软链接文件 `libftauthng.so`、`libftauthng.so.1` 拷贝到 `/usr/lib` 或其他目录下，然后就可以在示例程序所在目录下，直接调用 `make` 命令来编译和链接示例程序。如果在编译或运行时出现找不到动态库的错误提示，可以执行下面的命令，将 OTP 认证引擎动态库文件实际所在的目录添加到环境变量 `LD_LIBRARY_PATH` 中（假定动态库文件是拷贝到 `/usr/lib/` 目录下）：

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib
```

#### 4.1.3.3 示例程序

示例程序位于 `“test”` 目录下，开发人员可以编辑、编译和调试该程序。通过该程序，开发人员可以初步了解使用 OTP 认证引擎提供的接口进行应用开发的方法。

#### 4.1.4 应用开发

本文档中的“应用开发接口”和“令牌业务驱动流程及示例程序”两部分对使用 OTP 认证引擎接口进行应用开发的开发人员具有重要的参考作用。

### 4.2 令牌业务驱动流程及示例程序

令牌的关键业务包括令牌的导入、激活、认证（包括各种认证模型的认证，如动态口令、挑战应答、主机认证码、交易签名等）、同步、注销、解锁、密钥更新、手机令牌激活码/加密激活码生成等。OTP 认证引擎接口内部严格按照令牌业务流程和业务逻辑进行实现。

#### 4.2.1 导入种子

导入种子需要调用 OTP 认证引擎提供的种子文件功能接口，主要是将含有多个令牌信息的令牌文件（XML 格式）转换为令牌的私有数据。

导入种子时，接口的调用顺序如下：

序号	函数名称	说明
1	<code>init_pskc</code>	初始化种子导入接口
2	<code>read_pskc_rec</code>	从解密的数据中逐个读取令牌种子信息
3	<code>uninit_pskc</code>	反初始化种子导入接口

以下是导入种子的示例代码：

```
void *tk_ctx;
unsigned int i, recs;
StTokenData rec;
int ret;

//按明文方式解析
if ((ret = init_pskc("//令牌文件名称, NULL, NULL, &tk_ctx, &recs)) !=
    FT_API_SUCC)
{
    //加载种子文件失败, 出错处理
}

for (i = 0; i < recs; i++)
{
    if ((ret = read_pskc_rec(tk_ctx, &rec, i)) != FT_API_SUCC)
    {
        //警告: 获取种子失败
        continue;
    }
}

uninit_pskc(tk_ctx);

//令牌种子导入成功

//返回执行结果
```

#### 4.2.2 激活令牌

在令牌的首次使用时，需要先对令牌进行激活之后，才能正常使用令牌的其它功能。

首次激活令牌时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	set_nextmode_threshold	设定动态口令最大重试次数
3	set_user_login	设定令牌注册用户名，一般是使用令牌序列号作为令牌注册用户名

4	enable_token	激活令牌
5	set_pin	设定 PIN 码（认证时使用）
6	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

注意：如果没有调用 set\_pin 设置 PIN 码，则沿用原有的 PIN 码；令牌首次使用时 PIN 码为空。

以下是首次激活令牌的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库中），sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败，出错处理
}

if ((ret = set_nextmode_threshold(&pdata, ""//OTP 最大重试次数)) != FT_API_SUCC)
{
    //设置重试次数失败，出错处理
}

if ((ret = set_user_login(&pdata, ""// 令牌注册的用户名)) != FT_API_SUCC)
{
    //设置令牌用户名失败，出错处理
}

if ((ret = enable_token(&pdata)) != FT_API_SUCC)
{
    //启用令牌失败，出错处理
}

if ((ret = set_pin(&pdata, ""//PIN 码)) != FT_API_SUCC)
{
    //设置 PIN 失败，出错处理
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{

```

```
        //令牌私有数据重新编码失败,出错处理
    }

    //将私有数据重新写入数据库(略)

    //启用令牌成功

    //返回执行结果
```

4.2.3 注销令牌

如果暂不使用某个已激活的令牌，可以将其注销。注销之后该令牌还存在于数据库中，但是无法进行认证、同步等操作，可以通过调用 enable\_token 函数重新启用该令牌。

注销令牌时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	disable_token	注销令牌
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是令牌注销的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库中），sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败,出错处理
}

if ((ret = disable_token(&pdata)) != FT_API_SUCC)
{
    //注销令牌失败,出错处理
}
```

```
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败，出错处理
}

//将私有数据重新写入数据库（略）

//注销令牌成功

//返回执行结果
```

4.2.4 更新密钥

为了提高令牌种子密钥的安全性，挑战应答型令牌提供了更新种子密钥的功能。

更新种子密钥时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	update_key	更新种子密钥
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是更新种子密钥的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;
char up_resp[MAX_UP_RESP_SIZE];

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库中，sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败，出错处理
}
```

```

// r_tk 为随机数, tran_info 为业务信息数据, up_resp 为密钥更新响应值
if ((ret = update_key(&pdata, r_tk, tran_info, up_resp)) != FT_API_SUCC)
{
    //密钥更新失败,出错处理
}
else
{
    //密钥更新成功, 输出密钥更新响应值, 成功后认证引擎设置服务器端实际的密钥值
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败, 出错处理
}

//将私有数据重新写入数据库 (略)

// 返回执行结果

```

#### 4.2.5 获取解锁码

挑战应答型令牌提供两级解锁的功能。当令牌设备开机密码连续输入错误次数达到开机密码允许重试的最大次数时, 令牌将被锁定。令牌锁定之后, 用户需要向服务器获取一个一级解锁码, 并输入到令牌设备中进行解锁。如果输入的一级解锁码正确, 则令牌解锁成功。

如果一级解锁码连续输入错误次数达到一级解锁码允许重试的最大次数时, 一级解锁码将被锁定。一级解锁码锁定之后, 用户需要向管理员请求分发一个二级解锁码进行解锁, 管理员通过服务器获取到一个二级解锁码, 并发送给用户, 由用户输入到令牌中。如果输入的二级解锁码正确, 则令牌解锁成功。

获取解锁码时, 接口的调用顺序如下:

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构, 并保存在 StTokenData 结构的 pdata 成员中
2	get_puk	获取一级解锁码
	get_puk2	获取二级解锁码
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式, 并保存在 StTokenData 结构的 priv_data 成员中



以下是获取解锁码的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;
char puk[PUK_SIZE];
    char chl[65] = {'\0'};

    strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库
    中），sizeof(pdata.priv_data));
    pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

    if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
    {
        //解码私有数据失败,出错处理
    }

    if ("//判断一级解锁码已被锁定为真)
    {
        ret = get_puk2(&pdata, NULL, puk);
    }
    else
    {
        ret = get_puk(&pdata, NULL, puk);
    }

    if (ret != FT_API_SUCC)
    {
        //获取解锁码失败，出错处理
    }

    if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
    {
        //令牌私有数据重新编码失败，出错处理
    }

    //将私有数据重新写入数据库（略）

    //获取解锁码成功,输出解锁码

    //返回执行结果
```

## 4.2.6 同步令牌

令牌业务驱动接口提供两种同步令牌的方式，供用户选择。

### 4.2.6.1 两步同步方式

两步同步方式主要是连续两次调用 `resynch_token` 接口进行同步，两次传入的动态口令要求为令牌设备连续产生的两个动态口令，否则在第二次调用 `resynch_token` 接口时将报错。两次调用 `resynch_token` 接口，执行成功时返回值不同：第一次调用 `resynch_token` 接口，执行成功时返回 400；第二次调用 `resynch_token` 接口，执行成功时返回 0。

采用两步的方式同步令牌时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	resynch_token	第一次同步
3	resynch_token	第二次同步
4	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是采用两步的方式同步令牌的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库中），sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败，出错处理
}

ret = resynch_token(""//当前时间, &pdata, ""//第一个动态口令);

if (ret == FT_API_NEXTOTP)
{
    //第一次匹配动态口令成功

    //要求输入下一次动态口令,尝试进行第二次匹配
```

```
if((ret = resynch_token("//当前时间,&pdata,"//第二个动态口令) )!= FT_API_SUCC)
{
    //同步失败, 出错处理
}
else
{
    //同步成功
}
}
else
{
    //同步失败, 出错处理
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败, 出错处理
}

//将私有数据重新写入数据库 (略)

//返回执行结果
```

### 4.2.6.2 一步同步方式

一步同步方式则无需连续两次调用 `resynch_token` 接口，只需直接使用 `resynch2_token` 接口一次，即可完成令牌的同步操作。

采用一步的方式同步令牌时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	resynch2_token	同步令牌
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是采用一步的方式同步令牌的示例代码：

```

StTokenData pdata; //引擎驱动接口基本结构
int ret;

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库
中），sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败,出错处理
}

//输入两个连续动态口令
ret = resynch2_token(""//当前时间, &pdata, ""//前一个动态口令, ""//当前动态口令);

if (ret == FT_API_SUCC)
{
    //同步成功
}
else
{
    //同步失败, 出错处理
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败, 出错处理
}

//将私有数据重新写入数据库（略）

//返回执行结果

```

#### 4.2.7 验证动态口令

验证动态口令时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中

2	check_password	验证动态口令
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是验证动态口令的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库
中），sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败，出错处理
}

//tm 为当前时间，otp 为当前动态口令，pin 为 PIN 码（认证时使用）
if ((ret = check_password((long) tm, &pdata, otp, pin)) != FT_API_SUCC)
{
    //验证 OTP & PIN 失败

    if (ret == FT_API_NEED_RESYNC)
    {
        //要求输入下一动态口令进行同步
    }
}
else
{
    //验证 OTP & PIN 成功
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败，出错处理
}

//将私有数据重新写入数据库（略）

//返回执行结果
```

## 4.2.8 生成动态口令

OTP 认证引擎提供动态口令生成功能，该功能用途广泛。在短信令牌的应用中，可以通过此功能来生成动态口令，并通过短信网关以短信的方式发送到用户的手机上，以完成身份认证操作。另外，此功能生成的动态口令还可以作为应急动态口令，可在用户令牌丢失、损坏等情况下，急需使用动态口令登录系统时使用。

生成动态口令时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	genotp	产生当前的动态口令
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是生成当前的动态口令的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;
char otp[OTP_SIZE];

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库中，sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败,出错处理
}

// tm 为系统当前时间，otp 为当前的动态口令
if ((ret = genotp((long) tm, &pdata, otp)) != FT_API_SUCC)
{
    //生成当前动态口令失败，出错处理
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败,出错处理
}
```

```
//将私有数据重新写入数据库（略）
```

```
//生成当前动态口令成功，输出当前的动态口令
```

```
//返回执行结果
```

### 4.2.9 生成挑战值

挑战应答型令牌提供挑战应答动态口令身份认证功能。在使用此功能时，认证引擎需要先生成一个挑战值，然后发送给用户，由用户将此挑战值输入到令牌设备中，参与响应动态口令的计算。

生成挑战值时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	gen_chlg	生成挑战值
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是生成挑战值的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;
char chlg[CHLG_SIZE];

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库中），sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败,出错处理
}

if ((ret = gen_chlg(&pdata, ""//由认证引擎生成的挑战值)) != FT_API_SUCC)
{
    //生成挑战值失败,出错处理
}
```

```

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败,出错处理
}

//将私有数据重新写入数据库 (略)

//生成挑战值成功, 输出挑战值

//返回执行结果

```

#### 4.2.10 验证应答动态口令

对挑战应答型令牌生成的应答动态口令进行验证时, 接口的调用顺序如下:

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构, 并保存在 StTokenData 结构的 pdata 成员中
2	check_chpass	验证应答动态口令
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式, 并保存在 StTokenData 结构的 priv_data 成员中

以下是验证应答动态口令的示例代码:

```

StTokenData pdata; //引擎驱动接口基本结构
int ret;

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值 (可能存在于用户环境的数据库中, sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败,出错处理
}
//tm 为系统当前时间, otp 为应答动态口令, pin 为 PIN 码 (认证时使用), chlg 为挑战值
if ((ret = check_chpass((long) tm, &pdata, (const char *) otp,
    (const char *) pin, (const char *) chlg)) != FT_API_SUCC)
{

```



```
        //应答动态口令验证失败，出错处理
    }
    else
    {
        //应答动态口令验证成功
    }

    if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
    {
        //令牌私有数据重新编码失败,出错处理
    }

    //将私有数据重新写入数据库（略）

    //返回执行结果
```

4.2.11 生成应答动态口令

通过 OTP 认证引擎生成应答动态口令时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	gen_chpass	生成应答动态口令
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是生成应答动态口令的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;
char otp[RESP_SIZE];

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库中, sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败,出错处理
```

```
}

//tm 为系统当前时间，otp 为认证引擎生成的应答动态口令，chlg 为挑战值
if ((ret = gen_chpass((long) tm, &pdata, otp, chlg)) != FT_API_SUCC)
{
    //生成应答值失败，出错处理
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败,出错处理
}

//将私有数据重新写入数据库（略）

//生成应答值成功，输出当前应答值

//返回执行结果
```

4.2.12 验证签名动态口令

挑战应答型令牌提供交易签名功能。用户在执行交易时，将交易关键信息（如帐户、交易金额、交易日期等）输入到令牌设备中，生成签名动态口令，然后再将交易信息和签名动态口令一起发送到认证引擎进行验证。通过交易签名功能，可以有效防止攻击者对交易信息的篡改。在此基础上，再加上动态口令身份认证，可以大大提升身份认证的安全性。

验证签名动态口令时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	check_sigpass	验证签名动态口令
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是验证签名动态口令的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;
```

```

    strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库
    库中, sizeof(pdata.priv_data));
    pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

    if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
    {
        //解码私有数据失败,出错处理
    }

    // tm 为系统当前时间, otp 为令牌生成的签名动态口令
    // pin 为 PIN 码（认证时使用）, sigdata 为签名数据
    if ((ret = check_sigpass((long) tm, &pdata, (const char *) otp,
        (const char *) pin, (const char *) sigdata)) != FT_API_SUCC)
    {
        //验签失败,出错处理
    }
    else
    {
        //验签成功
    }

    if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
    {
        //令牌私有数据重新编码失败,出错处理
    }

    //将私有数据重新写入数据库（略）

    //返回执行结果

```

#### 4.2.13 生成签名动态口令

通过 OTP 认证引擎生成签名动态口令时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	gen_sigpass	生成签名动态口令
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是生成签名动态口令的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;
char otp[RESP_SIZE];

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库中, sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败, 出错处理
}

// tm 为系统当前时间, otp 为认证引擎生成的签名动态口令, sigdata 为签名数据
if ((ret = gen_sigpass((long) tm, &pdata, otp, sigdata)) != FT_API_SUCC)
{
    //生成签名动态口令失败, 出错处理
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败, 出错处理
}

//将私有数据重新写入数据库（略）

//生成签名动态口令成功, 输出当前的签名动态口令

//返回执行结果
```

#### 4.2.14 生成主机认证码

OTP 认证引擎和挑战应答型令牌配合使用，不但提供了认证用户身份的功能，而且还提供了验证服务器的功能。当用户不确定服务器是否真实时，可以通过令牌设备生成用于验证服务器的挑战值，发送到服务器端，根据服务器返回的主机认证码与令牌设备生成的主机认证码是否一致来判断服务器的真伪。

OTP 认证引擎生成主机认证码时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	gen_hacode	生成主机认证码
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是生成主机认证码的示例代码：

```

StTokenData pdata; //引擎驱动接口基本结构
int ret;
char otp[RESP_SIZE];

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库），sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败,出错处理
}

//chlg 为令牌设备生成的挑战值，otp 为认证引擎生成的主机认证码
ret = gen_hacode(&pdata, chlg , otp);
if (ret != FT_API_SUCC)
{
    //生成主机认证码失败，出错处理
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败，出错处理
}

//将私有数据重新写入数据库（略）

//生成主机认证码成功，输出主机认证码

//返回执行结果

```

### 4.2.15 验证 CHAP 动态口令

CHAP 全称是 PPP（点对点协议）询问握手认证协议（Challenge Handshake Authentication Protocol）。该协议可通过三次握手周期性的校验对端的身份，可在初始链路建立时完成，在链路建立之后重复进行。通过递增改变的标识符和可变的询问值，可防止来自端点的重放攻击，限制暴露于单个攻击的时间。

CHAP 动态口令验证是 CHAP 协议和动态口令身份认证技术的有机结合，它充分利用了两者在认证领域的技术特点和技术优势，进一步提高了身份验证的安全性和可靠性。

验证 CHAP 形式的动态口令时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	check_chap_pass	验证 CHAP 动态口令
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是验证 CHAP 形式的动态口令的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
unsigned char chlg[16];
unsigned char pswd[17];
int ret;

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库中），sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败,出错处理
}

// tm 为系统当前时间
// chlg 为 CHAP 挑战值,由 CHAP 客户端(如 VPN 设备)随机生成
// pswd 为 CHAP 口令,由 CHAP 客户端根据用户实际输入的动态口令（或包含了 PIN 码）生成
if ((ret = check_chap_pass((long) tm, &pdata, chlg, 16, pswd, 17)) != FT_API_SUCC)
{
    //验证 CHAP 口令失败
    if (ret == FT_API_NEED_RESYNC)
    {
        //要求输入下一个动态口令进行同步
    }
}
```

```
    }
}
else
{
    //验证 OTP/PIN 成功
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败,出错处理
}

//将私有数据重新写入数据库（略）

//返回执行结果
```

4.2.16 验证 MSCHAP 动态口令

MSCHAP 是一种不可逆的、加密密码身份验证协议。与 CHAP 类似，MS-CHAP 使用质询—响应机制来防止在身份验证过程中发送密码。将 MSCHAP 协议和动态口令身份认证技术结合起来，极大地提高了远程访问连接的安全性。

验证 MSCHAP 形式的动态口令时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	check_mschap_pass	验证 MSCHAP 动态口令
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是验证 MSCHAP 形式的动态口令的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
unsigned char chlg[8];
unsigned char pswd[50];
unsigned char pswd_hash[16] = {0};
unsigned char hash_hash[16] = {0};
int ret;
```

```

    strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据
    库中），sizeof(pdata.priv_data));
    pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

    if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
    {
        //解码私有数据失败,出错处理
    }

    //tm 为系统当前时间
    //chlg 为 MSCHAP 挑战值,由 MSCHAP 客户端(如 VPN 设备)随机生成
    //pswd 为 MSCHAP 口令, 由 MSCHAP 客户端根据用户实际输入的动态口令（或包含了 PIN 码）生成
    //pswd_hash 为密码 HASH 缓冲区。
    //hash_hash 为密码 HASH 的摘要缓冲区。
    if ((ret = check_mschap_pass((long) tm, &pdata, ver, chlg, 8,
        pswd, 50, pswd_hash, hash_hash)) != FT_API_SUCC)
    {
        //验证 MSCHAP 口令失败
        if (ret == FT_API_NEED_RESYNC)
        {
            //要求输入下一个动态口令进行令牌的同步
        }
    }
    else
    {
        int i;
        //验证 OTP/PIN 成功

        //输出口令 HASH(HEX)
        for (i = 0; i < 16; i++)
        {
            printf("%02x ", pswd_hash[i]);
        }

        //输出口令 HASH 的摘要(HEX)
        for (i = 0; i < 16; i++)
        {
            printf("%02x ", hash_hash[i]);
        }
    }

    if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
    {

```



```
//令牌私有数据重新编码失败,出错处理
}

//将私有数据重新写入数据库（略）

//返回执行结果
```

4.2.17 生成手机令牌激活码

用户在其手机设备上成功安装手机令牌应用程序之后，还需要通过使用激活码激活手机令牌，才能正常使用手机令牌的功能。

生成手机令牌激活码时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	gen_ac	生成手机令牌激活码
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是生成手机令牌激活码的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;
char ac[AC_MAX_SIZE];

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库中，sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败,出错处理
}

//udid 为手机令牌标识码（由手机令牌应用程序产生）
//ap 为激活密码（由管理员设置或系统随机产生）,ac 为激活码（由认证引擎产生）
if ((ret = gen_ac(&pdata, udid, ap, ac)) != FT_API_SUCC)
{
```

```
//生成激活码失败,出错处理
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败,出错处理
}

//将私有数据重新写入数据库(略)

//生成激活码成功,输出激活码

//返回执行结果
```

4.2.18 生成手机令牌加密激活码

生成手机令牌加密激活码时，接口的调用顺序如下：

序号	函数名称	说明
1	decode_pdata	将 StTokenData 结构的 priv_data 字符串格式数据转化为令牌结构，并保存在 StTokenData 结构的 pdata 成员中
2	gen_ac_cipher	生成手机令牌加密激活码
3	encode_pdata	将 StTokenData 结构的 pdata 令牌结构转化为字符串格式，并保存在 StTokenData 结构的 priv_data 成员中

以下是生成手机令牌加密激活码的示例代码：

```
StTokenData pdata; //引擎驱动接口基本结构
int ret;
char ac[AC_MAX_SIZE];

strncpy(pdata.priv_data, ""//实际的某一个令牌的私有数据值（可能存在于用户环境的数据库中，sizeof(pdata.priv_data));
pdata.priv_data[sizeof(pdata.priv_data) - 1] = '\0';

if ((ret = decode_pdata(&pdata)) != FT_API_SUCC)
{
    //解码私有数据失败，出错输出
}
```

```
//udid 为手机令牌标识码（由手机令牌应用程序产生）
//ap 为激活密码（由管理员设置或系统随机产生）
//rand_buf 为随机数
//ac 为激活码（由认证引擎产生）
if ((ret = gen_ac_cipher(&pdata, udid, ap, rand_buf, ac)) != FT_API_SUCC)
{
    //生成加密激活码失败, 出错输出
}

if ((ret = encode_pdata(&pdata)) != FT_API_SUCC)
{
    //令牌私有数据重新编码失败, 出错输出
}

//将私有数据重新写入数据库（略）

//生成加密激活码成功，输出加密激活码

//返回执行结果
```

## 第5章 返回值列表

本章将列举 OTP 认证引擎所提供的接口的返回值。在调用接口函数时，可以根据本章列举的返回值来判断执行失败的原因。

序号	常量定义	常量	说明
1.	FT_API_SUCC	0	成功
2.	FT_API_INVALID_PDATA	1	无效令牌私有数据(结构)
3.	FT_API_INVALID_USERNAME	2	令牌注册的用户名无效
4.	FT_API_INVALID_PIN	3	设置的 PIN 码无效（或 PIN 码验证失败）
5.	FT_API_INVALID_OTP	4	无效动态口令
6.	FT_API_NOT_ASSIGNED	5	令牌未注册用户
7.	FT_API_TOKEN_DISABLED	6	令牌已停用
8.	FT_API_TOKEN_LOST	7	令牌已挂失
9.	FT_API_TOKEN_LOGOFF	8	令牌已注销
10.	FT_API_TOKEN_LOCKED	9	令牌被锁定
11.	FT_API_INVALID_TKTYPE	10	令牌类型暂不支持
12.	FT_API_TOKEN_EXPIRED	11	令牌已过期
13.	FT_API_NEED_RESYNC	12	需要调用同步接口
14.	FT_API_REPLAY_OTP	13	动态口令被重放
15.	FT_API_INVALID_TKKEY	14	令牌的密钥无效（可能被篡改或密钥数据过短） 或保护密钥无效
16.	FT_API_INVALID_OTPLEN	15	动态口令长度无效
17.	FT_API_INVALID_PARAM	16	参数错误
18.	FT_API_CR_ERR FT_API_OCRA_ERR	17	挑战应答算法初始化失败
19.	FT_API_INVALID_CHLG	18	无效挑战值
20.	FT_API_NEED_VERIFY	19	需要验证动态口令
21.	FT_API_OTP_EXHAUSTION	20	卡片令牌包含的动态口令已耗尽
22.	FT_API_INVALID_CHAPCHLG	21	无效CHAP（或MSCHAP）挑战值
23.	FT_API_INVALID_CHAPRESP	22	无效 CHAP（或 MSCHAP）应答值
24.	FT_API_NEED_UPK	23	需要进行密钥更新（当令牌设备要求必须激活后才能使用时，后台系统检测到令牌为未激活状态的情况下的返回值）

25.	FT_API_ALREADY_UPK	24	密钥已成功更新过，不允许重复更新(当规定令牌只能激活一次时，后台系统在接收到令牌激活请求后，检测到令牌已激活的情况下的返回值)
26.	FT_API_NEXTOTP	400	第一次动态口令成功，需要下一次动态口令
27.	FT_API_INVALID_XMLFILE	500	无效的令牌文件（XML）
28.	FT_API_INVALID_TKIDX	501	无效的令牌记录索引（索引越界）