LightDuer SDK介绍

LightDuer SDK是基于FreeRTOS的轻量级设备解决方案。LightDuer SDK提供的API可以大大简化设备端开发者的工作，只需少量的代码即可完成设备端连接云平台，录音，播放媒体文件等功能。LightDuer SDK主要包括DCS framework 和Connagent（Connection Agent），Ntp，Voice_engine，OTA等模块。

1 SDK模块介绍

1.1 主要模块

1.1.1 Connagent 模块

该模块主要提供**设备**通过**profile**接入到**设备云**的能力。**设备**在连接上**设备云**后，可以通过相关接口向云端上报数据，并接收云端下发的指令。完成设备**数据上报**、**查询**及**控制**等能力。该模块头文件如下：

```
#include "lightduer_connagent.h"
```

1.1.2 Ntp模块

从网络端获取时间，供设备端做时间校准。
该模块的头文件如下：

```
#include "Lightduer_net_ntp.h"
```

1.1.3 Voice_engine模块

该模块主要提供设备的录音事件上报功能。设备在连接上设备云后，可以通过相关接口将录音上传到云端。该模块头文件如下：

```
#include "lightduer_voice.h"
```

1.1.4 OTA模块

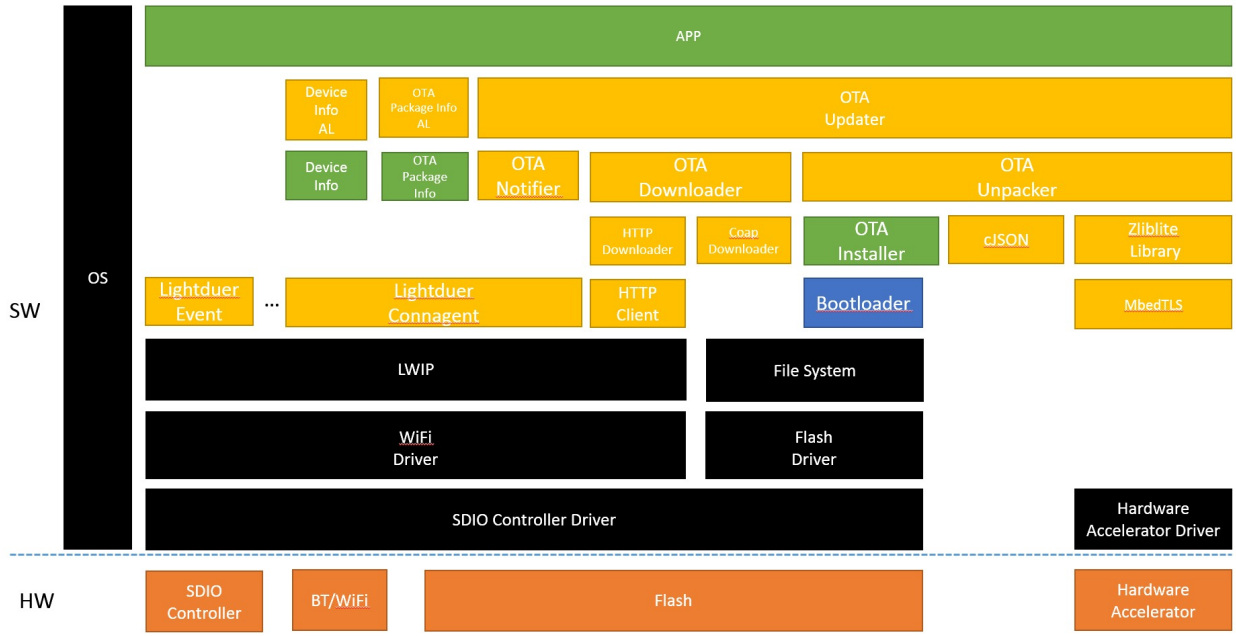
OTA模块提供了设备升级功能。开发者可以在（DuerOS开放平台-控制台-对应产品的编辑-OTA升级）自定义升级策略，再配合设备端的OTA功能，以自动完成升级工作。设备端的OTA功能如下：

- 支持多种下载协议下载升级包
- 解压缩升级包
- 校验升级包（通过SHA-1和服务器公钥校验升级包）
- 提供用户升级包的配置信息（此功能尚未开放，需要开放，请联系产品经理）
- 提供用户原始上传的文件信息

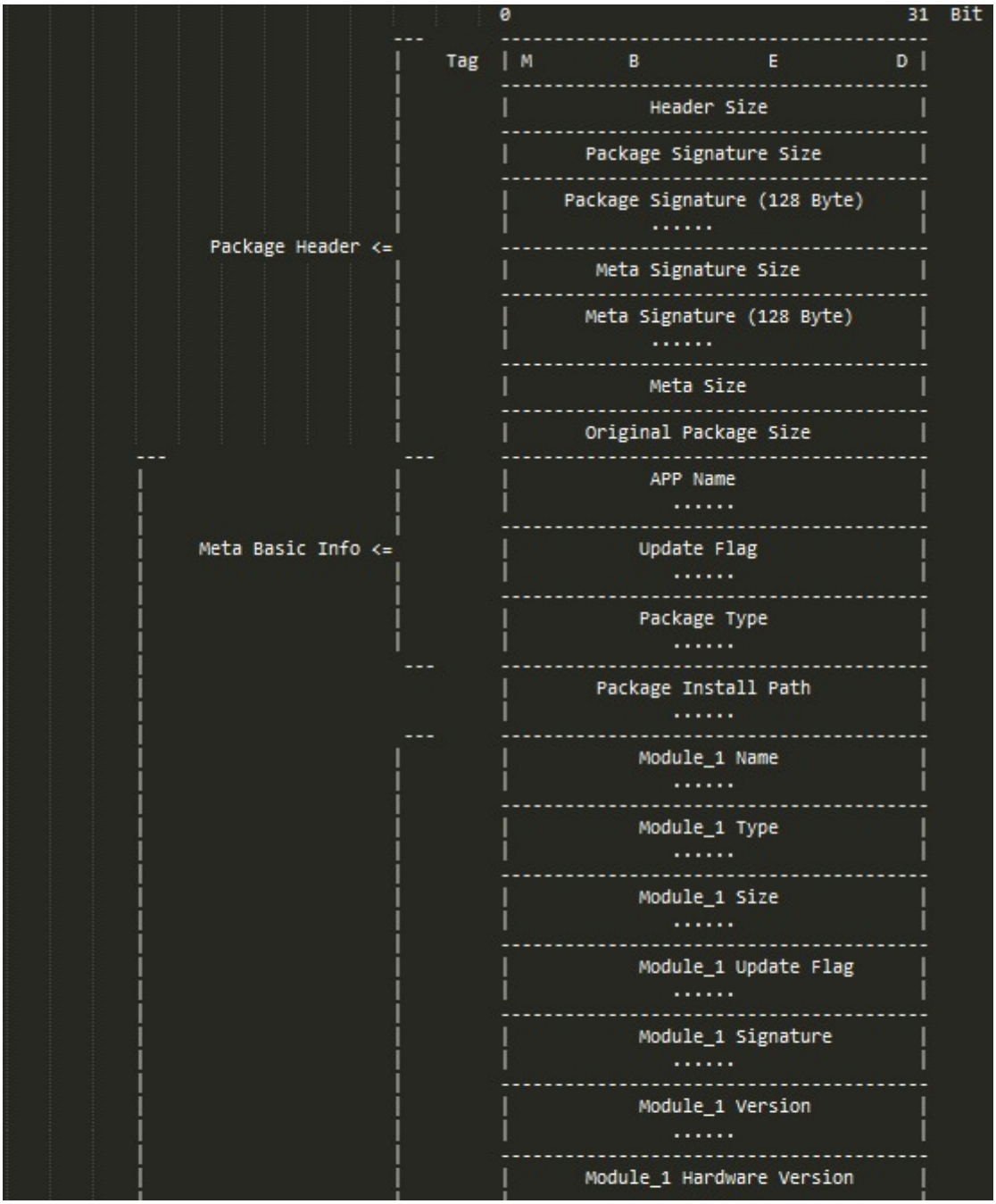
该模块的头文件如下：

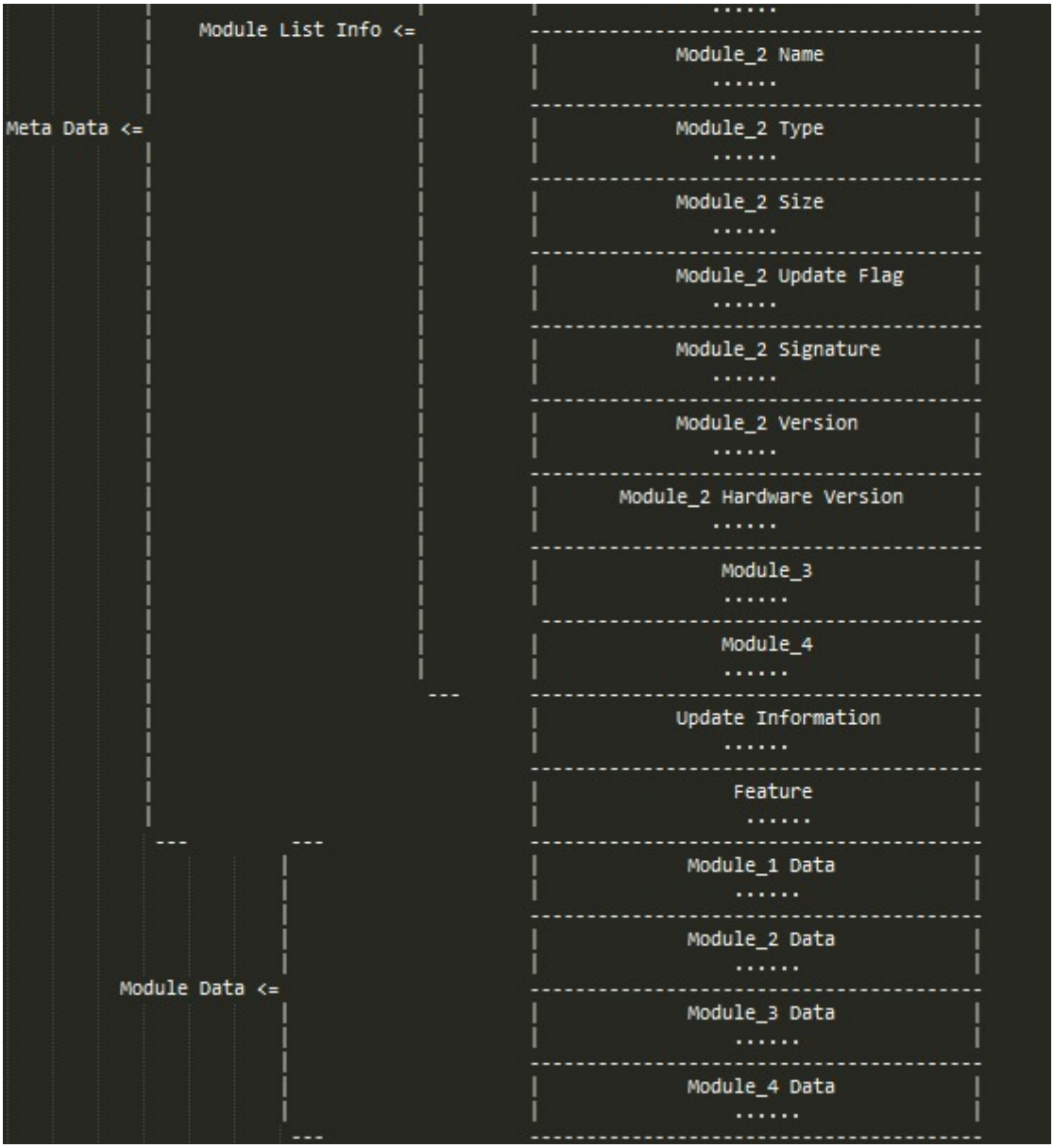
```
#include "lightduer_ota_updater.h"
#include "lightduer_ota_unpack.h"
#include "lightduer_ota_notifier.h"
#include "lightduer_dev_info.h"
#include "lightduer_ota_installer.h"
```

OTA软件架构图如下：



OTA的升级包的格式信息如下：



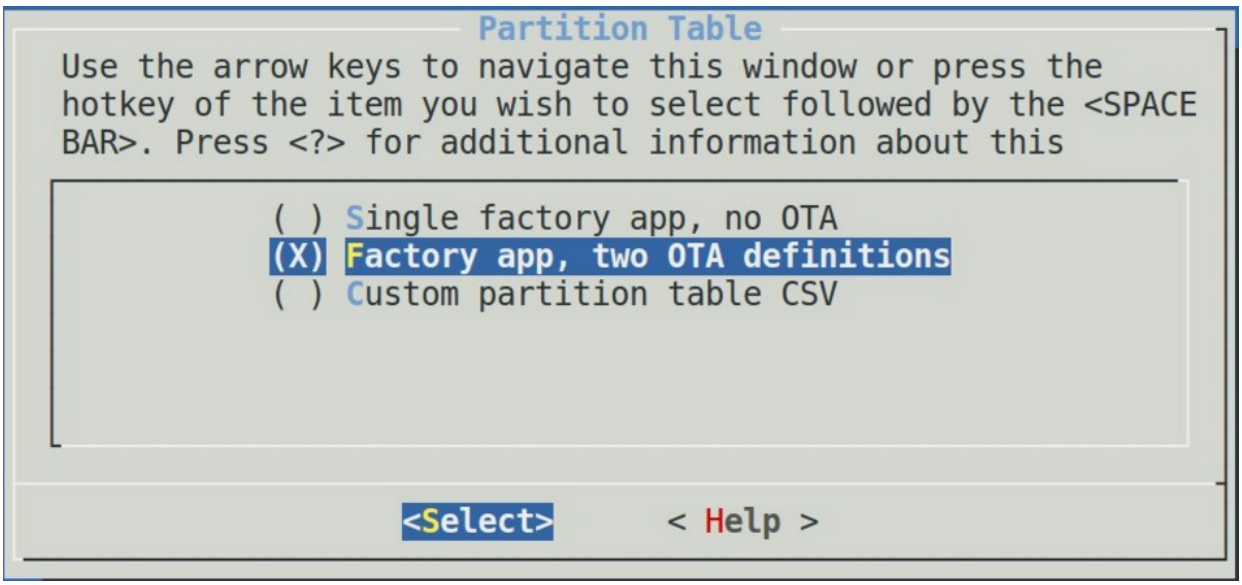


开启OTA功能，需要注意两处配置:

- 在配置文件（libduer-device/build/device/freertos/esp32/sdkconfig.mk）中做如下配置：

```
modules_module_HTTP=y
external_module_Zliblite=y
external_module_mbedtls=y
module_framework=y
modules_module_coap=y
modules_module_connagent=y
modules_module_cjson=y
modules_module_Device_Info=y
modules_module_OTA=y
```

- 在menuconfig配置时请选择Factory app，two OTA definitions ，如下图所示：



开发提示：

1. OTA 依赖Zliblite库，Zliblite库包含了types.h头文件，由于IAR,Keil编译器的C库没有提供这个头文件，所以会导致编译失败。
   解决方法： 在zconf.h头文件里自行定义off_t, 32位系统声明为long int,64位系统声明为long long int。
2. ESP32平台默认的分区表，支持最大的烧录Bin文件是1.5MB，如果开发者烧录的Bin文件大于1.5MB会导致OTA升级失败。
   解决办法： 开发者自行增加一个partitions.csv文件，根据开发板的Flash大小修改分区表信息，并在sdkconfig里添加用户自定义的分区表信息（$ make menuconfig
   Partition Table —> Partition Table (Single factory app, no OTA) —> ( ) Custom partition table CSV ）。
3. 参考demo(libduer_device/modules/OTA/Readme.md[5])中升级包信息参数的定义，os_version & staged_version 的版本号信息必须与云端填写的版本号信息一致。

```
static duer_package_info_t s_package_info = {
    .product = "ESP32 Demo",
    .batch   = "12",
    .os_info = {
        .os_name        = "FreeRTOS",
        .developer      = "Allen",
        .os_version     = "1.0.0.0", // This version is that you write in the Duer Cloud
        .staged_version = "1.0.0.0", // This version is that you write in the Duer Cloud
    }
};
```

1.2 DCS framework介绍

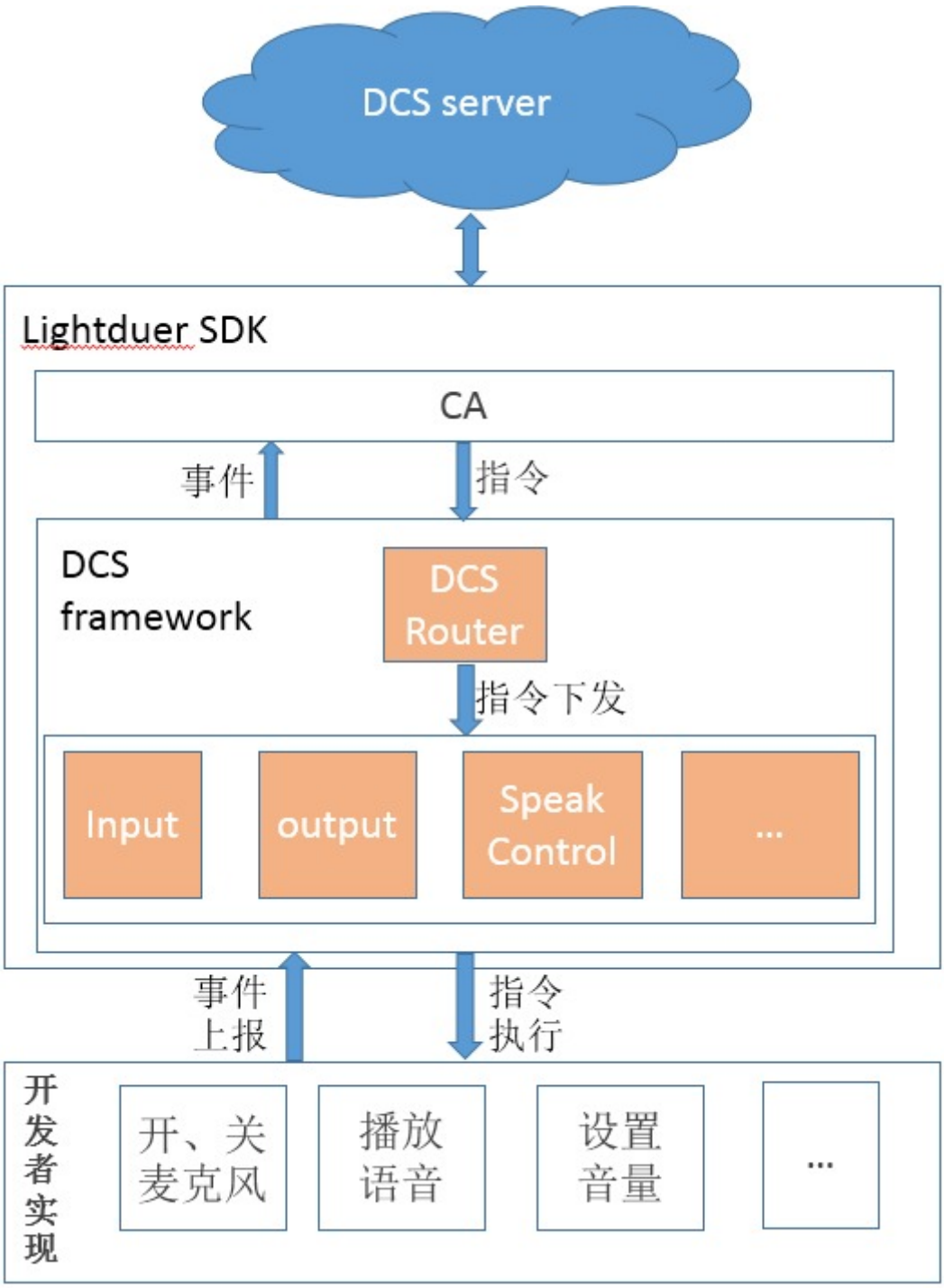
1.2.1 DCS简介

- DuerOS Conversational Service(以下简称DCS)是DuerOS对外免费开放的智能语音交互服务API。
- DCS是面向开发者（硬件厂商）的一个标准化接入协议，允许开发者使用麦克风、扬声器等客户端能力，通过语音来连接由DuerOS提供的服务(如音乐播放，计时器和闹钟，天气查询，知识解答等)。
- DCS主要包含三部分：
  指令(directive)：服务端下发给设备端，设备端需要执行相应的操作，比如播放语音（Speak指令）。
  事件(event)：需要设备端上报给服务端，通知服务端在设备端发生的事情，比如音乐播放开始了（PlaybackStarted事件）等。
  端状态(clientContext) ：当前设备上的所有状态，比如当前音量、是不是正在播放音乐、是否有闹钟等。

1.2.2 DCS模块设计说明

为了简化开发者的工作，lightduer SDK中提供了DCS framework，主要提供以下功能：

1. 接收并解析server端下发的指令，调用接口（由开发者实现）完成相应操作，例如播放音乐、设置闹钟等。
2. 为开发者提供接口，实现事件和端状态的上报
   DCS framework的总体设计如下图：



1.2.3 使用方法

根据DCS协议，DCS framework主要分为语音输入、语音输出、播放控制等7个接口（具体参照接口说明部分）。开发者可以根据需要，选择自己需要使用的接口。DCS 要求的功能，DCS framework中有的已经实现，有的提供了函数给开发者调用（例如事件上报），还有一些需要开发者自己实现（例如音频播放）。

提醒：在上传语音过程中，若被按键操作（上一曲、下一曲、暂停等）打断，此段语音将被设备端抛弃。

DCS framework提供了两个头文件：

lightduer_dcs.h

该头文件中声明了dcs framework提供给开发者或需要开发者实现的所有函数（除了闹钟接口相关函数）。

lightduer_dcs_alert.h
由于闹钟功能比较独立，只有部分开发者可能会用到，所以把闹钟接口相关的函数单独放到了该文件中。
我们提供了demo来展示如何使用或实现相关的函数，可供开发者参考。

2 主要接口介绍

2.1 ConnAgent

2.1.1 类型定义

2.1.1.1 duer_event_id

该类型为一个枚举类型，表示当前设备的连接状态，原型定义如下：

```
typedef enum _duer_event_id_enum {
    DUER_EVENT_STARTED,
    DUER_EVENT_STOPPED
} duer_event_id;
```

状态值说明：

状态	说明
DUER_EVENT_STARTED	与设备云连接成功
DUER_EVENT_STOPPED	与设备云连接断开

2.1.1.2 duer_event_t

该类型为一个结构体，用于存储当前事件消息，其原型定义如下：

```
typedef struct _duer_event_data_s {
    int     _event;     //< Event id, see in @{link duer_event_id}
    void *  _object;    //< The return content
} duer_event_t;
```

结构说明：

成员	类型	说明

_event	int	当前事件状态，值参照duer_event_id
_object	void *	预留

2.1.1.3 duer_event_callback

该类型为一个函数指针类型，用于接收事件消息，其原型如下：

```
typedef void (*duer_event_callback)(duer_event_t *);
```

参数说明：

类型	说明
duer_event_t *	触发事件的消息载体，详见duer_event_t

2.1.1.4 duer_resource_mode_e

该类型为一个枚举类型，标明当前资源类型，其定义如下：

```
typedef enum {
    DUER_RES_MODE_STATIC,            // Static resources have some value
 that doesn't change
    DUER_RES_MODE_DYNAMIC,           // Dynamic resources are handled in
 application
} duer_resource_mode_e;
```

状态说明：

值	说明
DUER_RES_MODE_STATIC	标明静态资源，形态一般为常量字符串等
DUER_RES_MODE_DYNAMIC	标明动态资源，通过回调函数的方式来获取资源内容

2.1.1.5 duer_resource_operation_e

该类型为一个枚举类型，标明资源被请求的Method，其定义如下：：

```
typedef enum {
    DUER_RES_OP_GET      = 0x01,     // Get operation allowed
    DUER_RES_OP_PUT      = 0x02,     // Put operation allowed
    DUER_RES_OP_POST     = 0x04,     // Post operation allowed
    DUER_RES_OP_DELETE   = 0x08      // Delete operation allowed
} duer_resource_operation_e;
```

方法说明：

值	说明
DUER_RES_OP_GET	GET方法
DUER_RES_OP_PUT	PUT方法
DUER_RES_OP_POST	POST方法
DUER_RES_OP_DELETE	DELETE方法

2.1.1.6 duer_msg_type_e

该类型为一个枚举类型，表明消息类型，具体作用参看CoAP - Messaging Model，其定义如下：

```
/**
 * CoAP Message type, used in CoAP Header
 */
typedef enum {
    DUER_MSG_TYPE_CONFIRMABLE       = 0x00, // Reliable Request messages
    DUER_MSG_TYPE_NON_CONFIRMABLE   = 0x10, // Non-reliable Request and R
esponse messages
    DUER_MSG_TYPE_ACKNOWLEDGEMENT   = 0x20, // Response to a Confirmable
Request
    DUER_MSG_TYPE_RESET             = 0x30  // Answer a Bad Request
} duer_msg_type_e;
```

2.1.1.7 duer_msg_code_e

该类型为一个枚举类型，表明消息请求方法或响应代码状态代码，其定义如下：

```
/*
 * The message codes.
 */
typedef enum {

    // Request Method
    DUER_MSG_REQ_GET                        = 1,
    DUER_MSG_REQ_POST                       = 2,
    DUER_MSG_REQ_PUT                        = 3,
    DUER_MSG_REQ_DELETE                     = 4,

    // Response Code
    DUER_MSG_RSP_CREATED                    = 65,   // 2.01
    DUER_MSG_RSP_DELETED                    = 66,   // 2.02
    DUER_MSG_RSP_VALID                      = 67,   // 2.03
    DUER_MSG_RSP_CHANGED                    = 68,   // 2.04
    DUER_MSG_RSP_CONTENT                    = 69,   // 2.05
```

```
    DUER_MSG_RSP_CONTINUE                           = 95,   // 2.31
    DUER_MSG_RSP_BAD_REQUEST                        = 128,  // 4.00
    DUER_MSG_RSP_UNAUTHORIZED                       = 129,  // 4.01
    DUER_MSG_RSP_BAD_OPTION                         = 130,  // 4.02
    DUER_MSG_RSP_FORBIDDEN                          = 131,  // 4.03
    DUER_MSG_RSP_NOT_FOUND                          = 132,  // 4.04
    DUER_MSG_RSP_METHOD_NOT_ALLOWED                 = 133,  // 4.05
    DUER_MSG_RSP_NOT_ACCEPTABLE                     = 134,  // 4.06
    DUER_MSG_RSP_REQUEST_ENTITY_INCOMPLETE          = 136,  // 4.08
    DUER_MSG_RSP_PRECONDITION_FAILED                = 140,  // 4.12
    DUER_MSG_RSP_REQUEST_ENTITY_TOO_LARGE           = 141,  // 4.13
    DUER_MSG_RSP_UNSUPPORTED_CONTENT_FORMAT         = 143,  // 4.15
    DUER_MSG_RSP_INTERNAL_SERVER_ERROR              = 160,  // 5.00
    DUER_MSG_RSP_NOT_IMPLEMENTED                    = 161,  // 5.01
    DUER_MSG_RSP_BAD_GATEWAY                        = 162,  // 5.02
    DUER_MSG_RSP_SERVICE_UNAVAILABLE                = 163,  // 5.03
    DUER_MSG_RSP_GATEWAY_TIMEOUT                    = 164,  // 5.04
    DUER_MSG_RSP_PROXYING_NOT_SUPPORTED             = 165,  // 5.05

} duer_msg_code_e;
```

2.1.1.8 duer_msg_t

该类型为一个结构体，其中包含了基于CoAP协议的请求及响应消息，其定义如下：

```
typedef struct _duer_message_s {
    duer_u16_t token_len;

    duer_u8_t  msg_type;
    duer_u8_t  msg_code;
    duer_u16_t msg_id;

    duer_u16_t path_len;
    duer_u16_t query_len;
    duer_u16_t payload_len;

    duer_u8_t* token;
    duer_u8_t* path;
    duer_u8_t* query;
    duer_u8_t* payload;
} duer_msg_t;
```

结构说明，可参考rfc7252 section 3 Message Format：

成员	类型	说明
token_len	duer_u16_t	Token值的长度
msg_type	duer_u8_t	消息类型，取值见duer_msg_type_e

msg_code	duer_u8_t	消息代码，取值见duer_msg_code_e
msg_id	duer_u16_t	消息id，每条消息唯一，由系统自动生成
path_len	duer_u16_t	Uri-Path的长度
query_len	duer_u16_t	Uri-Query的长度
payload_len	duer_u16_t	消息负荷的长度
token	duer_u8_t *	Token值，每条请求/响应内唯一
path	duer_u8_t *	Uri-Path字串内容，长度由path_len限定
query	duer_u8_t *	Uri-Query字串内容，长度由query_len限定
payload	duer_u8_t *	消息负荷

2.1.1.9 duer_notify_f

该类型为一个函数指针类型，用于接收事件消息，其原型如下：

```
typedef duer_status_t (*duer_notify_f)(duer_context ctx, duer_msg_t *msg,
 duer_addr_t *addr);
```

参数说明：

类型	说明
duer_context	预留
duer_msg_t *	服务端的请求数据
duer_addr_t *	预留

2.1.1.10 duer_res_t

该类型为一个结构体，用于存储访问资源，其定义如下：

```
typedef struct _duer_resource_s {
    duer_u8_t    mode: 2;    // the resource mode, SEE in ${link duer_res
ource_mode_e}
    duer_u8_t    allowed: 6; // operation permission, SEE in ${link duer_
resource_operation_e}
    char *       path;       // the resource path identify

    union {
        duer_notify_f   f_res;   // dynamic resource handle function, NUL
L if static
```

```
        struct {
            void *       data;   // static resource value data, NULL if d
ynamic
            duer_size_t  size;   // static resource size
        } s_res;
    } res;
} duer_res_t;
```

结构说明：

成员	类型	说明
mode	duer_u8_t:2	资源类型，取值见duer_resource_mode_e
allowed	duer_u8_t:6	资源允许的访问权限，取值见duer_resource_operation_e，可以通过“或”运算允许多种不同的权限
path	char *	资源访问路径
res	union	资源内容，根据资源mode的不同，访问不同类型结构的资源

资源union结构说明：

- 静态资源（DUER_RES_MODE_STATIC）
  - 资源存放于s_res结构中，data是资源的起始地址，size是资源的大小；
  - 当云端请求资源时，设备端直接将data指向的内容透传到云端。
- 动态资源（DUER_RES_MODE_DYNAMIC）
  - f_res存放回调函数的地址；
  - 当云端请求资源时，设备端会回调f_res对应的函数；
  - 资源回调函数详见：duer_notify_f

2.1.2 函数接口

2.1.2.1 duer_initialize

该函数用于初始化整个模块所需要的环境，需要在使用其它所有接口前调用，且在设备上电后只需要调用一次。

2.1.2.2 duer_set_event_callback

参数

类型	描述
duer_event_callback	回调函数，通知设备连接状态的改变

返回值
void
示例

```
static void duer_event_hook(duer_event_t *event)
{
    if (!event) {
        DUER_LOGE("NULL event!!!");
        return;
    }

    switch (event->_event) {
    case DUER_EVENT_STARTED:
        // TODO: 成功连接到设备云
        break;
    case DUER_EVENT_STOPPED:
        // TODO: 与设备云连接断开
        break;
    }
}

void main(void) {

    ......

    duer_initialize();

    // Set the Duer Event Callback
    duer_set_event_callback(duer_event_hook);

    ......
}
```

2.1.2.3 duer_start

该函数用于发起连接设备云的事件，连接成功时，通过duer_set_event_callback中设定的回调函数通知到调用者。
参数

类型	描述
const void *	Profile内容存放地址
size_t	Profile内容大小

返回值

int	说明

DUER_OK	触发连接事件成功
DUER_ERR_FAILED	触发连接事件失败，需要重新发起该事件

示例

```
void duer_test_start(const char* uri) {
    // 载入Profile数据到内存中
    const char *data = duer_load_profile(uri);
    if (data == NULL) {
        DUER_LOGE("load profile failed!");
        return;
    }

    // 传入Profile数据后，开始请求连接设备云
    duer_start(data, duer_obtain_profile_size(data));

    // 清除Profile中缓存的数据
    duer_release_profile((void *)data);
}
```

2.1.2.4 duer_add_resources

该函数用于添加访问资源（数据点），该资源可以通过OpenAPI的**设备控制**能力进行访问和请求。接口调用时机为:

- CA启动后，如果CA没有启动成功，调用此接口并不能正确的添加对应控制点。而且在CA断开之后重连后需要重新添加对应的数据点。
- CA启动成功的标志为: 收到通知事件 DUER_EVENT_STARTED

参数

类型	**描述**
const duer_res_t *	资源列表的首地址，资源信息详见duer_res_t说明
size_t	资源长度

返回值
void
示例

```
void volume_up_down(duer_context ctx,duer_msg_t *msg, duer_addr_t *addr)
{
    ... ...
}

void duer_init_voice_ctrl()
```

```
{
        duer_res_t res[] = {
             {DUER_RES_MODE_DYNAMIC, DUER_RES_OP_PUT | DUER_RES_OP_GET, "v
olume",.res.f_res = volume_up_down},
        };
        duer_add_resources(res, sizeof(res) / sizeof(res[0]));
}
```

2.1.2.5 duer_data_available

当有云端下发数据时，调用此函数通知设备准备接收数据

参数

void

返回值

int	说明
DUER_OK	设备端接收数据准备成功
DUER_ERR_FAILED	设备端接收数据准备失败

示例

```
static void duer_transport_notify(duer_transevt_e event)
{
        switch (event) {
        case DUER_TEVT_RECV_RDY:
             //设备准备接收数据
            duer_data_available();
            break;
        case DUER_TEVT_SEND_RDY:
            //设备准备发送数据
            duer_data_send();
            break;
        }
}
```

2.1.2.6 duer_data_report

此接口用于设备向服务器上报数据，使用上报数据点时需要用到此接口将信息进行上报。

参数

类型	描述
const baidu_json *	发送数据点的值，数据内容详见cJson说明

返回值

int	说明
DUER_OK	发送数据成功
DUER_ERR_FAILED	发送数据失败

示例

示例参见duer_voice_start 示例

2.1.2.7 duer_response

对于云端通过数据点下发的请求，设备端处理结束后应给出相应的应答，此接口便是起到响应云端请求功能。

参数

类型	描述
const duer_msg_t *	服务器发送请求消息的结构体，参见duer_msg_t说明
int	云端消息请求的方式，参见duer_msg_code_e说明
const void *	数据包payload
size_t	数据包的大小

返回值

int	说明
DUER_OK	设备端成功响应请求
DUER_ERR_FAILED	设备端未响应请求

示例

```
void volume_up_down(duer_context ctx,duer_msg_t *msg, duer_addr_t *addr)
{
    DUER_LOGI("volume_up_down  \n");
    duer_handler handler = (duer_handler)ctx;
    if (handler && msg) {
        if (msg->msg_code == DUER_MSG_REQ_GET ) {
                DUER_LOGI("volume DUER_MSG_REQ_GET ");
        }else if (msg->msg_code == DUER_MSG_REQ_PUT ) {
                DUER_LOGI("volume DUER_MSG_REQ_PUT ");
                ... ...
        }
        duer_response(msg, msg_code, NULL, 0);
    }
```

```
}
```

2.2 Voice_engine

2.2.1 类型定义

2.2.1.1 duer_voice_result_func

该类型所指向的回调函数，用于录音上传云端，云端返回消息被解析后执行，其原型如下：

```
typedef void (*duer_voice_result_func)(baidu_json *result);
```

参数说明：

类型	说明
baidu_json *	数据点的值，数据内容详见详见cJson说明

2.2.1.2 duer_voice_delay_func

回调函数指针，用于设置发送数据超时是的回调函数，其原型如下：

```
typedef void (*duer_voice_delay_func)(duer_u32_t);
```

参数说明：

类型	说明
duer_u32_t	发送语音片段使用的时间，单位：ms

2.2.1.3 duer_voice_mode

上传语音模式的枚举类型，用于表示语音交互的模式，其原型如下：

```
typedef enum _duer_voice_mode_enum {
    DUER_VOICE_MODE_DEFAULT,              // 普通对话模式
    DUER_VOICE_MODE_CHINESE_TO_ENGLISH,   // 汉语翻译成英语
    DUER_VOICE_MODE_ENGLISH_TO_CHINESE    // 英语翻译成汉语
} duer_voice_mode;
```

2.2.2 接口说明

2.2.2.1 duer_voice_start

按参数中的采样率，开始采集语音。
注：请确保duer_voice_start(),duer_voice_send(),duer_voice_stop()三个函数在同一线程中。

参数

类型	描述
int	采样率（建议采样率为16k）

返回值

int	说明
DUER_OK	开始采集成功
DUER_ERR_FAILED	开始采集失败

示例

```
static void duer_record_event_callback(int event, const void *data, size_
t param)
{
    DUER_LOGD("record callback : %d", event);
    switch (event) {
        case REC_START:
        {
            DUER_LOGD("start send voice: %s", (const char *)data);
            baidu_json *path = baidu_json_CreateObject();
            baidu_json_AddStringToObject(path, "path", (const char *)dat
a);
            duer_data_report(path);
            baidu_json_Delete(path);
            //采集语音开始
            duer_voice_start(param);
            break;
        }
           ... ...
    }
}
```

2.2.2.2 duer_voice_send

发送语音数据到云端。

参数

类型	描述
void *	语音数据（WAV格式语音数据参数：16k、16bit、单声道、PCM）
size_t	数据大小

返回值

int	说明
DUER_OK	传输数据成功
DUER_ERR_FAILED	传输数据失败

示例

```
static void duer_record_event_callback(int event, const void *data, size_
t param)
{
    DUER_LOGD("record callback : %d", event);
    switch (event) {
        ... ...
        case REC_DATA:
            duer_voice_send(data, param);
            break;
        ... ...
    }
}
```

2.2.2.3 duer_voice_stop

停止语音采集。

参数
void
返回值

int	说明
DUER_OK	停止录音操作成功
DUER_ERR_FAILED	停止录音操作失败

示例

```
static void duer_record_event_callback(int event, const void *data, size_
t param)
{
    DUER_LOGD("record callback : %d", event);
    switch (event) {
        ... ...
        case REC_STOP:
            DUER_LOGD("stop send voice: %s", (const char *)data);
            duer_voice_stop();
            break;
        ... ...
    }
}
```

2.2.2.4 duer_voice_set_delay_threshold

用于设置监控发送语音数据的时间的阈值，并且注册回调函数。
参数

类型	名称	说明
duer_u32_t	delay	发送语音数据的时间的阈值，发送时间超过阈值时触发注册的回调函数。默认值是 `-1` ，表示最大值
duer_voice_delay_func	-	注册的回调函数，发送语音数据超时时调用此函数，开发者根据自己的需求实现相应功能

返回值
void

2.2.2.5 duer_voice_terminate

用于设备端主动终止发送语音数据。
参数
void
返回值
void

2.2.2.6 duer_voice_set_mode

此接口用于设置语音交互模式，模式分为：普通模式、中翻英、英翻中模式。默认为普通模式。从普通模式切换到翻译模式时，如果正在播放audio，应先发送DCS_PAUSE_CMD事件停止audio播放，并且建议先调用duer_dcs_close_multi_dialog接口退出多轮。
参数

类型	名称	说明

duer_voice_mode	mode	语音交互模式

返回值
void

2.2.2.7 duer_voice_get_mode

此接口用于获得当前的语音交互模式。
参数
void
返回值

类型	说明
duer_voice_mode	语音交互模式

2.3 OTA

2.3.1 类型定义

2.3.1.1 duer_ota_init_ops_t

该类型为一个结构体，存储了三个用于OTA初始化时注册的函数，其原型定义如下：

```
typedef struct _duer_ota_init_ops_s {
    int (*init_updater)(duer_ota_updater_t *updater);
    int (*uninit_updater)(duer_ota_updater_t *updater);
    int (*reboot)(void *arg);
} duer_ota_init_ops_t;
```

结构说明：

成员	类型	说明
init_updater	函数指针	初始化更新器的函数指针
uninit_updater	函数指针	卸载更新器的函数指针
reboot	函数指针	设备重启函数的函数指针

2.3.1.2 duer_ota_installer_t

该类型为一个结构体，用于存储OTA安装器所需要的七个回调函数，其原型定义如下：
注：此结构体中的回调函数不允许做非常耗时（超过10分钟）的操作。

```
typedef struct _duer_ota_installer_s{
    int (*notify_data_begin)(void *cxt);
    int (*notify_meta_data)(void *cxt, duer_ota_package_meta_data_t *met
a);
    int (*notify_module_data)(void *cxt, unsigned int offset, unsigned ch
ar *data, unsigned int size);
    int (*notify_data_end)(void *cxt);
    int (*update_img_begin)(void *cxt);
    int (*update_img)(void *cxt);
    int (*update_img_end)(void *cxt);
} duer_ota_installer_t;
```

结构说明：

成员	类型	说明
notify_data_begin	函数指针	开始解压升级包时的通知函数
notify_meta_data	函数指针	接收升级包描述信息函数
notify_module_data	函数指针	接收用户上传到云端的原始文件
notify_data_end	函数指针	升级包解压缩完成通知函数
update_img_begin	函数指针	开始更新升级包通知函数
update_img	函数指针	更新升级包函数
update_img_end	函数指针	更新升级包完毕通知函数

2.3.1.3 DevInfoOps

该类型为一个结构体，定义在lightduer_dev_info.h中，包含4个用于获取设备信息的函数指针

```
struct DevInfoOps {
    int (*get_firmware_version)(char *firmware_version);
    int (*get_chip_version)(char *chip_version);
    int (*get_sdk_version)(char *sdk_version);
    int (*get_network_info)(struct NetworkInfo *info);
};
```

结构说明：

成员	类型	说明

get_firmware_version	函数指针	获取固件版本号
get_chip_version	函数指针	获取芯片型号
get_sdk_version	函数指针	获取SDK版本号
get_network_info	函数指针	获取网络信息

2.3.1.4 duer_ota_switch

该类型为一个枚举类型，表示OTA使能状态。其原型如下：

```
typedef enum _duer_ota_switch {
    ENABLE_OTA  =  1,
    DISABLE_OTA = -1,
} duer_ota_switch;
```

状态值说明：

状态	说明
ENABLE_OTA	OTA升级使能
DISABLE_OTA	OTA升级不使能

2.3.1.5 init_updater

该类型为一个函数指针，初始化升级器时调用此指针指向的函数，开发者需要在函数中实现以下功能：

- 调用duer_ota_unpack_register_installer() 注册安装器
- 申请用户自定义资源
- 调用duer_ota_unpack_set_private_data()设置传递给安装器的自定义资源
  其原型如下：

```
int (*init_updater)(duer_ota_updater_t *updater);
```

参数说明：

类型	说明
duer_ota_updater_t *	用户创建的更新器实例,详情参见duer_ota_updater_t

2.3.1.6 uninit_updater

该类型为一个函数指针，该指针指向的函数用于销毁升级器时调用，开发者需要在此函数内释放用户自定义资源。
可使用duer_ota_unpack_get_private_data接口获取updater中包含的用户自定义资源。
其原型如下：

```
int (*uninit_updater)(duer_ota_updater_t *updater);
```

参数说明：

类型	描述
duer_ota_updater_t *	用户创建的更新器实例,详情参见duer_ota_updater_t

2.3.1.7 reboot

该类型为一个函数指针，该指针指向的函数用于升级完成时调用此函数，开发者需要在此函数内实现设备的重启工作。
其原型如下：

```
int (*reboot)(void *arg);
```

参数说明：

类型	描述
void*	保留参数，目前无实际意义

2.3.1.8 notify_data_begin

该类型为一个函数指针，开始解压升级包时的通知该指针指向的函数。在开始下载解压升级包前调用此函数。
用户可在这个函数里，创建文件，初始化Flash，使能DMA，擦除Flash等操作。
其原型如下：

```
int (*notify_data_begin)(void *cxt);
```

参数说明：

类型	描述
void*	用户传递给安装器的自定义参数

2.3.1.9 notify_meta_data

该类型为一个函数指针，该指针指向的函数用于接收升级包描述信息，此接口暂时无效，需要使用此接口请联系我们。
其原型如下：

```
int (*notify_meta_data)(void *cxt, duer_ota_package_meta_data_t *meta);
```

参数说明：

类型	描述
void*	用户自定义参数
duer_ota_package_meta_data_t*	升级包的描述信息

2.3.1.10 notify_module_data

该类型为一个函数指针，该指针指向的函数用于接收用户上传到云端的原始文件，可以将原始文件保存到文件系统或直接写入到Flash里。升级包会分为多个包传送给设备，此函数会被多次调用。
其原型如下：

```
int (*notify_module_data)(  void *cxt,
                            unsigned int offset,
                            unsigned char *data,
                            unsigned int size );
```

参数说明：

类型	名称	描述
void*	cxt	用户自定义参数
unsigned int	offset	data指向的数据在整个数据包中的偏移量
unsigned char *	data	升级包数据
unsigned int	size	data指向数据的大小

2.3.1.11 notify_data_end

该类型为一个函数指针，该指针指向的函数用于升级包解压缩完成通知函数，升级包发送完整后调用此函数。
用户可以在这个函数里，释放通知函数申请的资源，并校验收到的数据。
其原型如下：

```
int (*notify_data_end)(void *cxt);
```

参数说明：

类型	描述
void*	用户自定义参数

2.3.1.12 update_img_begin

该类型为一个函数指针，该指针指向的函数用于开始更新升级包通知函数，安装升级包前的资源申请和初始化。
其原型如下：

```
int (*update_img_begin)(void *cxt);
```

参数说明：

类型	描述
void*	用户自定义参数

2.3.1.13 update_img

该类型为一个函数指针，该指针指向的函数用于更新升级包函数，用户开始安装升级包。
其原型如下：

```
int (*update_img)(void *cxt);
```

参数说明：

类型	描述
void*	用户自定义参数

2.3.1.14 update_img_end

该类型为一个函数指针，该指针指向的函数用于更新升级包完毕通知函数，用户可以校验安装是否成功。
其原型如下：

```
int (*update_img_end)(void *cxt);
```

参数说明：

类型	描述

void*	用户自定义参数

2.3.1.15 get_firmware_version

该类型为一个函数指针，该指针指向的函数用于获取固件版本号，需要开发者在此函数内将固件版本号存入firmware_version的地址下，firmware_version指针有16个字节可用，固件版本号所占用空间不能大于16个字节。
其原型如下：

```
int (*get_firmware_version)(char *firmware_version);
```

参数说明：

类型	描述
char*	保存固件版本号信息

2.3.1.16 get_chip_version

该类型为一个函数指针，该指针指向的函数用于获取芯片型号，需要开发者在此函数内将芯片型号存入chip_version的地址下，chip_version指针有20个字节可用，芯片型号所占用空间不能大于20个字节。
其原型如下：

```
int (*get_chip_version)(char *chip_version);
```

参数说明：

类型	描述
char*	保存芯片型号信息

2.3.1.17 get_sdk_version

该类型为一个函数指针，该指针指向的函数用于获取SDK版本号，需要开发者在此函数内将SDK版本号存入sdk_version的地址下，sdk_version指针有15个字节可用，SDK版本号所占用空间不能大于15个字节。
其原型如下：

```
int (*get_sdk_version)(char *sdk_version);
```

参数说明：

类型	描述
char*	保存SDK版本号信息

2.3.1.18 get_network_info

该类型为一个函数指针，该指针指向的函数用于获取网络信息，需要开发者在此函数内将网络信息存入info的地址下，info指针有18个字节可用，网络信息所占用空间不能大于18个字节。
其原型如下：

```
int (*get_network_info)(struct NetworkInfo *info);
```

参数说明：

类型	描述
struct NetworkInfo *	保存网络信息

2.3.1.19 get_package_info

该类型为一个函数指针，该指针指向的函数用于获取安装包信息。
其原型如下：

```
int (*get_package_info)(duer_package_info_t *info);
```

参数说明：

类型	描述
duer_package_info_t *	保存安装包信息

2.3.2 主要接口

2.3.2.1 duer_init_ota

此函数主要用于初始化OTA模块，同时注册3个回调函数（函数功能在参数中介绍）。**注：** 此函数需要在CA启动后调用。该函数所属头文件为：

```
#include "lightduer_ota_updater.h"
```

参数

类型	描述
OTAInitOps	设置OTA初始化的回调函数，详情参见OTAInitOps

返回值

int	说明
DUER_OK	初始化成功
Other	初始化失败

示例

```
//初始化升级器的具体实现
static int duer_ota_init_updater(duer_ota_updater_t *ota_updater)
{
    int ret = DUER_OK;

    //申请用户自定义资源
    struct OTAInstallHandle *ota_install_handle = NULL;
    ota_install_handle = (struct OTAInstallHandle *)malloc(sizeof(*ota_in
stall_handle));
    if (ota_install_handle == NULL) {
        DUER_LOGE("OTA Unpack OPS: Malloc failed");
        ret = DUER_ERR_MEMORY_OVERLOW;
        goto out;
    }
    ota_install_handle->updater = ota_updater;

    //注册安装器
    ret = duer_ota_unpack_register_installer(ota_updater->unpacker, &upda
ter);
    if (ret != DUER_OK) {
        DUER_LOGE("OTA Unpack OPS: Register updater failed ret:%d", ret);
        goto out;

    //设置传递给安装器的自定义资源
    ret = duer_ota_unpack_set_private_data(ota_updater->unpacker, ota_ins
tall_handle);
    if (ret != DUER_OK) {
        DUER_LOGE("OTA Unpack OPS: Set private data failed ret:%d", ret);
    }
out:
    return ret;
}
//卸载升级器的具体实现
static int duer_ota_uninit_updater(duer_ota_updater_t *ota_updater)
{
    struct OTAInstallHandle *ota_install_handle = NULL;
```

```
    ota_install_handle = duer_ota_unpack_get_private_data(ota_updater->un
packer);
    if (ota_install_handle == NULL) {
        DUER_LOGE("OTA Unpack OPS: Get private data failed");
        return DUER_ERR_INVALID_PARAMETER;
    }
   free(ota_install_handle);
    return DUER_OK;
}
//升级完成后的系统重启功能实现
static int reboot(void *arg)
{
    int ret = DUER_OK;
    DUER_LOGE("OTA Unpack OPS: Prepare to restart system");
    esp_restart();
    return ret;
}

static duer_ota_init_ops_t s_ota_init_ops = {
    .init_updater = duer_ota_init_updater,
    .uninit_updater = duer_ota_uninit_updater,
    .reboot = reboot,
};

int duer_initialize_ota(void)
{
    int ret = DUER_OK;
    ret = duer_init_ota(&s_ota_init_ops);
    if (ret != DUER_OK) {
        DUER_LOGE("Init OTA failed");
    }

   ...
}
```

2.3.2.2 duer_ota_unpack_register_installer

注册安装器，安装器中包含7个回调函数，需要开发者自行实现，回调函数的具体实现可以参照SDK中的Demo。所属头文件：

```
#include "lightduer_ota_unpack.h"
```

参数

类型	描述
duer_ota_unpacker_t*	指向解包模块实例的指针

duer_ota_installer_t*	duer_ota_installer_t是定义在lightduer_ota_installer.h中的结构，包含7个函数指针，详情参见duer_ota_installer_t

返回值

int	说明
DUER_OK	成功
other	失败

示例
请参考init_updater示例

2.3.2.3 duer_ota_unpack_set_private_data

此函数用于设置需要传递给安装器的资源，在init_updater回调函数内使用。所属头文件：

```
#include "lightduer_ota_unpack.h"
```

参数

类型	描述
duer_ota_unpacker_t*	指向解包模块实例的指针
void*	用户自定义参数

返回值

int	说明
DUER_OK	成功
other	失败

示例

```
// 申请资源
struct rda_ota_update_context *ota_install_handle = NULL;
ota_install_handle = (struct rda_ota_update_context *)malloc(sizeof(*ota_
install_handle));

// 整合原有资源和新申请资源
ota_install_handle->updater = ota_updater;

// 设置安装器使用的资源
```

```
ret = duer_ota_unpack_set_private_data(ota_updater->unpacker, ota_install
_handle);
```

2.3.2.4 duer_ota_unpack_get_private_data

获取用户传递给安装器的资源，可用于释放资源。所属头文件：

```
#include "lightduer_ota_unpack.h"
```

参数

类型	描述
duer_ota_unpacker_t*	指向解包模块实例的指针

返回值

void *	说明
NULL	获取资源地址失败
Other	获取到的资源地址

示例

```
struct rda_ota_update_context *ota_install_handle = NULL;

ota_install_handle = duer_ota_unpack_get_private_data(ota_updater->unpack
er);
```

2.3.2.5 duer_ota_register_package_info_ops

注册安装包信息，参数包含一个回调函数指针。开发者需要在这个回调函数中将安装包信息存入此回调函数的参数中。所属头文件:

```
#include "lightduer_ota_notifier.h"
```

参数

类型	描述
duer_package_info_ops_t*	PackageInfoOPS中包含1个函数指针，指向一个获取升级包信息的函数，参见get_package_info

返回值

int	说明
DUER_OK	成功
other	失败

示例

```
//获取升级包信息的具体实现
static int get_package_info(duer_package_info_t *info)
{
    int ret = DUER_OK;
    char firmware_version[FIRMWARE_VERSION_LEN + 1];

    if (info == NULL) {
        DUER_LOGE("Argument Error");

        ret = DUER_ERR_INVALID_PARAMETER;

        goto out;
    }

    memset(firmware_version, 0, sizeof(firmware_version));

    ret = duer_get_firmware_version(firmware_version);
    if (ret != DUER_OK) {
        DUER_LOGE("Get firmware version failed");

        goto out;
    }

    strncpy((char *)&s_package_info.os_info.os_version,
            firmware_version,
            FIRMWARE_VERSION_LEN + 1);
    memcpy(info, &s_package_info, sizeof(*info));

out:
    return ret;
}

static duer_package_info_ops_t s_package_info_ops = {
    .get_package_info = get_package_info,
};

duer_ota_register_package_info_ops(&s_package_info_ops);
```

2.3.2.6 duer_register_device_info_ops

注册设备信息，设备信息分为4种，通过4个回调函数传入此接口。所属头文件:

```
#incllude "lightduer_dev_info.h"
```

参数

类型	描述
struct DevInfoOps*	DevInfoOps是定义在lightduer_dev_info.h中的结构，包含4个函数指针

返回值

int	说明
DUER_OK	成功
other	失败

示例

```
static struct DevInfoOps dev_info_ops = {
    .get_firmware_version = get_firmware_version,
    .get_chip_version     = get_chip_version,
    .get_sdk_version      = get_sdk_version,
    .get_network_info     = get_network_info,
};
duer_register_device_info_ops(&dev_info_ops);
```

2.3.2.7 duer_ota_set_switch

设置是否使能OTA。所属头文件

```
#include "lightduer_ota_updater.h"
```

参数

类型	描述
enum duer_ota_switch	ENABLE_OTA：使能，DISABLE_OTA：不使能

返回值

int	说明
DUER_OK	成功

DUER_ERR_FAILED	失败

2.3.2.8 duer_ota_get_switch

获取OTA是否开启。所属头文件:

```
#include "lightduer_ota_updater.h"
```

参数

void

返回值

int	说明
DUER_OK	成功
DUER_ERR_FAILED	失败

2.3.2.9 duer_ota_set_reboot

设置OTA更新完后重启状态。所属头文件

```
#include "lightduer_ota_updater.h"
```

参数

类型	描述
enum duer_ota_reboot	ENABLE_REBOOT：重启使能，DISABLE_REBOOT：重启关闭

返回值

int	说明
DUER_OK	成功
DUER_ERR_FAILED	失败

示例

```
ret = duer_ota_set_reboot(ENABLE_REBOOT);
```

2.3.2.10 duer_ota_get_reboot

获取OTA更新完后重启状态。所属头文件:

```
#include "lightduer_ota_updater.h"
```

参数

void

返回值

int	说明
DUER_OK	成功
DUER_ERR_FAILED	失败

2.3.2.11 duer_ota_notify_package_info

调用此接口可将升级包信息通知给Duer云端，需在duer_ota_register_package_info_ops接口以后调用，且用户在重启设备后，必须调用 duer_ota_notify_package_info() 上报升级包信息。
所属头文件：

```
lightduer_ota_notifier.h
```

参数

void

返回值

int	说明
DUER_OK	成功
Other	失败

3 DCS framework接口说明

3.1 初始化DCS framework

要使用DCS framework，开发者首先需要调用duer_dcs_framework_init函数进行初始化。

3.1.1 duer_dcs_framework_init

该函数用于初始化DCS framework。
参数
none
返回值
none

3.2 语音输入(voice input)接口

该接口定义语音输入相关的功能，如果设备有麦克风，应该实现该接口，接入语音输入的能力。该接口包括设备端进行语音请求（上传语音），服务端下发开始监听、停止监听指令等。

3.2.1 duer_dcs_voice_input_init

该函数用于初始化voice input接口，如果开发者需要使用语音输入相关的功能，则必须先调用该函数。
参数
none
返回值
none

3.2.2 duer_dcs_on_listen_started

该函数用于发送ListenStarted事件给DCS服务端。当用户开始发起语音请求时（上传语音之前），应该调用该函数通知DCS。如果有audio正在播放，则DCS framework会调用duer_dcs_audio_pause_handler暂停audio，待speak交互完成后调用duer_dcs_audio_resume_handler继续播放audio。
参数
none
返回值

int	说明
DUER_OK	执行成功
DUER_ERR_FAILED	执行失败

3.2.3 duer_dcs_stop_listen_handler

该函数需要开发者实现，当DCS framework收到StopListen指令后，会调用该函数用于关闭麦克风。

注意：该函数需要立即返回，不能被阻塞
参数
none
返回值
none

3.2.4 duer_dcs_on_listen_stopped

使用翻译模式的语音交互时，设备端主动停止发送语音后调用此接口通知DCS模块。
参数
none
返回值

int	说明
0	执行成功
非0	执行失败

3.2.5 duer_dcs_listen_handler

该函数内需要开发者实现打开麦克风功能，当需要自动打开麦克风时会被调，如：多轮交互。
注意：该函数需要立即返回，不能被阻塞
参数
none
返回值
none

3.3 语音输出(voice output)接口

该接口定义语音输出相关的功能，如果设备有扬声器，则应该实现该接口，接入语音输出的能力。

3.3.1 duer_dcs_voice_output_init

该函数用于初始化voice output接口，如果开发者需要使用语音输出相关的功能，则必须先调用该函数。
参数
none
返回值
none

3.3.2 duer_dcs_speech_on_finished

当设备播放完server端发送的语音后，应该调用该函数。调用该函数后，DCS framework会向DCS服务端发送SpeechFinished事件，并调用duer_dcs_audio_resume_handler播放被打断的audio。
参数
none
返回值
none

3.3.3 duer_dcs_speak_handler

开发者需要实现该函数，当DCS framework收到server端下发的Speak指令后，会调用该函数播放语音。
注意：该函数需要立即返回，不能被阻塞
参数

char *	说明
url	需要播放的语音资源的url

返回值
none

3.4. 扬声器控制(speaker controller)接口

该接口定义扬声器控制相关的功能，如音量的调节、静音设置等等。

3.4.1 duer_dcs_speaker_control_init

该函数用于初始化speaker controller接口，如果开发者需要使用扬声器控制相关的功能，则必须先调用该函数。
参数
none
返回值
none

3.4.2 duer_dcs_on_volume_changed

当设备上的音量发生变化时，调用该函数进行事件上报。
参数

none
返回值

int	说明
DUER_OK	事件上报成功
DUER_ERR_FAILED	事件上报失败

3.4.3 duer_dcs_on_mute

当设备发生静音/取消静音等操作时，调用该函数进行事件上报。
参数
none
返回值

int	说明
DUER_OK	事件上报成功
DUER_ERR_FAILED	事件上报失败

3.4.4 duer_dcs_get_speaker_state

开发者需要实现该函数，用于DCS framework获取当前播放控制器的状态，包括音量、是否静音等。
注意：该函数需要立即返回，不能被阻塞
参数

名称	类型	说明
volume	int	用来存储当前音量值
is_mute	bool *	用来存储当前静音状态

返回值
none

3.4.5 duer_dcs_volume_set_handler

开发者需要实现该函数，当DCS server端下发设置音量的指令后，DCS framework会调用该函数进行音量设置。
注意：该函数需要立即返回，不能被阻塞
参数

名称	类型	说明
volume	int	要设置的音量绝对值，[0~100]

返回值
none

3.4.6 duer_dcs_volume_adjust_handler

开发者需要实现该函数，当DCS server端下发调整音量的指令后，DCS framework会调用该函数调整音量。
注意：该函数需要立即返回，不能被阻塞
参数

名称	类型	说明
volume	int	音量调整相对值，[-100, 100]

返回值
none

3.4.7 duer_dcs_mute_handler

开发者需要实现该函数，当DCS server端下发设置或取消静音的指令后，DCS framework会调用该函数设置或取消静音。
注意：该函数需要立即返回，不能被阻塞
参数

名称	类型	说明
is_mute	bool	true代表设置静音，false代表取消静音

返回值
none

3.5 音频播放器(audio player)接口

该接口提供了播放音频相关的函数，通过这些函数可以控制设备端上音频资源的播放。

3.5.1 duer_dcs_audio_player_init

该函数用于初始化audio player接口，如果开发者需要使用音频播放器，则必须先调用该函数。
参数
none
返回值
none

3.5.2 duer_dcs_audio_on_finished

当一个audio播放完毕后，开发者调用该函数通知DCS framework。DCS framework会向DCS server端上报，server端根据需要下发下一个audio播放指令。
参数
none
返回值
none

3.5.3 duer_dcs_audio_play_handler

由开发者调用其media播放器来实现该函数，当DCS server下发audio play指令时，DCS framework会调用该函数来播放audio。
注意：该函数需要立即返回，不能被阻塞
参数

名称	类型	说明
url	const char *	需要播放的audio的url

返回值
none

3.5.4 duer_dcs_audio_stop_handler

由开发者来实现该函数，当DCS server下发stop指令时，DCS framework会调用该函数来停止audio的播放。
注意：该函数需要立即返回，不能被阻塞
参数
none
返回值
none

3.5.5 duer_dcs_audio_resume_handler

由开发者来实现该函数，用来实现恢复audio播放，与duer_dcs_audio_pause_handler相对应。例如一个audio在播放的过程中，用户进行了语音交互，则audio需要暂停播放（通过调用duer_dcs_audio_pause_handler），等语音交互结束后，DCS framework会调用duer_dcs_audio_resume_handler进行续播。
注意：该函数需要立即返回，不能被阻塞
参数

名称	类型	说明
url	const char *	需要播放的audio的url
offset	int	从该位置开始播放

返回值
none

3.5.6 duer_dcs_audio_pause_handler

由开发者来实现该函数。当audio播放过程中出现对话等更高优先级的事情时，DCS framework会调用该接口来暂停audio的播放。
注意：该函数需要立即返回，不能被阻塞
参数
none
返回值
none

3.5.7 duer_dcs_audio_get_play_progress

由开发者来实现该函数，DCS framework会调用该接口来获取当前audio的播放进度。
注意：该函数需要立即返回，不能被阻塞
参数
none
返回值

类型	说明
int	audio的播放进度

3.5.8 duer_dcs_audio_on_stuttered

设备在播放audio时发生卡顿调用此接口进行上报。
注意：当出现卡顿时必须调用此接口上报状态。
参数

名称	类型	说明
is_stuttered	bool	true：播放卡顿，false：播放恢复正常

返回值

int	说明
DUER_OK	事件上报成功
DUER_ERR_FAILED	事件上报失败

3.5.9 duer_dcs_audio_play_failed

设备在播放audio失败时，调用此接口通知dcs模块并进行上报。
注意：当出现播放audio失败时必须调用此接口上报状态。

参数

名称	类型	说明
type	duer_dcs_audio_error_t	错误类型
msg	const char *	设备播放出错的具体信息

duer_dcs_audio_error_t 定义如下：

```
typedef enum {
    DCS_MEDIA_ERROR_UNKNOWN,                // 未知错误类型
    DCS_MEDIA_ERROR_INVALID_REQUEST,        // 服务器无效响应，如请求不好，禁
止，未找到.etc等
    DCS_MEDIA_ERROR_SERVICE_UNAVAILABLE,    // 设备无法连接服务器
    DCS_MEDIA_ERROR_INTERNAL_SERVER_ERROR, // 服务器无法处理设备的请求
    DCS_MEDIA_ERROR_INTERNAL_DEVICE_ERROR, // 设备内部错误
} duer_dcs_audio_error_t;
```

返回值

int	说明
0	事件上报成功
非0	事件上报失败

3.5.10 duer_dcs_audio_report_metadata

用于上报播放资源的元数据，设备端通过云端下发的url中获取到的资源数据中可能携带资源相关的元数据，例如：歌曲名、作曲家、演唱者等。设备端如果能够获取到资源内部的元数据，可通过此接口上报到云端，有助于排查可能有资源引发的问题。

参数

名称	类型	说明
metadata	baidu_json *	json格式的元数据信息

返回值

int	说明
0	事件上报成功
非0	事件上报失败

3.6 播放控制 (Playback Controller)接口

该接口用来支持用户通过控制按钮等方式来控制音乐播放（上一首、下一首、播放、暂停等）。

3.6.1 duer_dcs_play_control_cmd_t

该枚举类型定义了播放控制事件。

```
typedef enum {
    DCS_PLAY_CMD, // 当用户按了设备端上的播放按钮时，上报该事件，DCS server会下发play指令，duer_dcs_audio_play_handler会被调用播放audio
    DCS_PAUSE_CMD, // 当用户按了设备端上的暂停按钮时，上报该事件，DCS server会下发stop指令，duer_dcs_audio_stop_handler会被调用停止正在播放的audio
    DCS_PREVIOUS_CMD, // 当用户按了设备端上的上一首按钮时，上报该事件，DCS server会下发play指令播放前一个audio，duer_dcs_audio_play_handler会被调用
    DCS_NEXT_CMD, // 当用户按了设备端上的下一首按钮时，上报该事件，DCS server会下发play指令播放下一个audio，duer_dcs_audio_play_handler会被调用
    DCS_PLAY_CONTROL_EVENT_NUMBER,
} duer_dcs_play_control_cmd_t;
```

3.6.2 duer_dcs_send_play_control_cmd

该函数用来上报播放控制事件。例如，如果用户按了下一首的按键，则应上报DCS_NEXT_CMD事件，DCS server收到该事件后，会下发一个新的audio给设备进行播放。

参数

名称	类型	说明
cmd	duer_dcs_play_control_cmd_t	需要上报的事件类型

返回值

int	说明
DUER_OK	事件上报成功
DUER_ERR_FAILED	事件上报失败
DUER_ERR_MEMORY_OVERLOW	内存不足，事件上报失败

3.7 闹钟（Alerts）接口

该接口提供设置、取消闹钟的功能。闹钟功能的使用流程如下：

1. 用户通过“今天晚上8点提醒我喝水”之类的话术，由设备传给云端
2. 云端解析后下发设置闹钟的指令，其中包括token、time、type等内容，开发者需要把这些信息保存到本地，并设定本地闹钟
3. 闹钟触发后，删除本地存储的该闹钟的一些信息

3.7.1 duer_dcs_alert_init

该函数用于初始化alert接口，如果开发者需要使用闹钟功能，则必须先调用该函数。

3.7.2 duer_alert_event_type

该枚举类型定义了闹钟事件类型。

```
typedef enum {
    SET_ALERT_SUCCESS,     // 设置闹钟成功时上报该事件
    SET_ALERT_FAIL,        // 设置闹钟失败时上报该事件
    DELETE_ALERT_SUCCESS, // 删除闹钟成功时上报该事件
    DELETE_ALERT_FAIL,     // 删除闹钟失败时上报该事件
    ALERT_START,           // 闹钟触发时上报该事件
    ALERT_STOP,            // 闹钟停止时上报该事件
} duer_alert_event_type;
```

3.7.3 duer_dcs_alert_info_type

定义了闹钟详细参数的结构体。

```
typedef struct {
    char *type;     // 闹钟类型，"ALARM" 或 "TIMER"
    char *time;     // 闹钟的触发时间，ISO 8601格式，根据这个time，通过NTP或RTC校准时间，然后设置定时器
    char *token;    // 闹钟的token，是闹钟的唯一标识
} duer_dcs_alert_info_type;
```

3.7.4 duer_dcs_report_alert_event

该函数用于上报闹钟事件。当闹钟设置（或删除）成功（或失败）、闹钟触发或被停止时，都应该上报相应的事件给DCS server，然后DCS server会下发相应的语音给设备。例如上报SET_ALERT_SUCCESS事件后，设备会收到类似”成功为您设置了闹钟“之类的语音播报。
参数

名称	类型	说明
token	const char *	该闹钟的token
type	duer_alert_event_type	需要上报的事件类型

3.7.5 duer_dcs_alert_set_handler

开发者需要实现该函数，用于设置闹钟。
注意：该函数需要立即返回，不能被阻塞
参数

名称	类型	说明
alert_info	const duer_dcs_alert_info_type *	闹钟的详细信息，具体可参照 duer_dcs_alert_info_type

返回值
none

3.7.6 duer_dcs_alert_delete_handler

开发者需要实现该函数，用于删除闹钟。
注意：该函数需要立即返回，不能被阻塞

参数

名称	类型	说明
token	const char *	要删除的闹钟的token

返回值
none

3.7.7 duer_insert_alert_list

开发者在duer_dcs_get_all_alert函数中使用该函数，将现在设备上所有的闹钟信息逐个上报给DCS framework.
参数

名称	类型	说明
alert_list	baidu_json *	用于存储所有的闹钟相关信息，该参数由duer_dcs_get_all_alert函数传入
alert_info	duer_dcs_alert_info_type *	闹钟的详细信息，具体可参照duer_dcs_alert_info_type
is_active	bool	是否正在响铃

返回值
none

3.7.8 duer_dcs_get_all_alert

该函数需要由开发者实现，DCS framework通过调用该函数获取当前设备上所有的闹钟信息。开发者可以在该函数中循环调用duer_dcs_report_alert函数，来实现这些信息的上报。
参数

名称	类型	说明
alert_array	baidu_json *	用于存储所有的闹钟相关信息

返回值
none

3.8 系统（System）接口

该接口包含了一些系统级别的接口，DCS framework已经做了相应接口的实现，开发者只需要在设备启动时调用duer_dcs_sync_state函数，把设备上当前的状态上报给DCS server端即可。

3.8.1 duer_dcs_sync_state

该函数用于当设备启动时把设备的状态发送给DCS server。
参数
none
返回值
none

3.8.2 duer_dcs_close_multi_dialog

此接口用于主动退出多轮。多轮是指：猜谜语、词语接龙、听声辨物等。
参数
none
返回值
none

3.9 文字上屏接口

该接口用于将使用者与设备的交互内容以文本的形式表示出来。用于有屏幕的设备展示交互内容。

3.9.1 duer_dcs_screen_init

该函数用于初始化文字上屏接口，如果开发者需要使用文字上屏功能，则必须先调用该函数。
参数
none
返回值
none

3.9.2 duer_dcs_render_card_handler

开发者需要实现该函数，将云端下发的信息表示出来。
参数

名称	类型	说明
payload	baidu_json *	云端返回的具体信息

返回值

duer_status_t	说明
DUER_OK	文字上屏成功
DUER_ERR_FAILED	文字上屏失败
DUER_MSG_RSP_BAD_REQUEST	Json解析失败时，返回此值

参数格式说明

```
"directive": {
    "header": {
    "namespace": "ai.dueros.device_interface.screen",
    "name": "RenderCard",
    "messageId": "{{STRING}}",
    "dialogRequestId": "{{STRING}}"
    },
    "payload": {
    "token": "{{STRING}}",
    "type": "TextCard",
    "content": "{{STRING}}",
    "link": {{LinkStructure}},
    "skillInfo": {{SkillInfoStructure}}
    }
}
```

content 字段是云端发送给设备的具体文本信息。

3.9.3 duer_dcs_input_text_handler

该接口用于接收语音识别过程中的识别结果，并将其表示出来。开发者需要实现该函数。
注意：该函数需要立即返回，不能被阻塞。
参数

名称	类型	说明
text	const char *	识别过程中文字
type	const char *	“INTERMEDIATE”: 中间结果，”FINAL”: 最终结果

返回值

duer_status_t	说明
DUER_OK	成功
DUER_ERR_FAILED	失败

DUER_MSG_RSP_BAD_REQUEST	type字段不是”INTERMEDIATE”、”FINAL”时，返回此值

3.10 蓝牙模块

该模块包含使用蓝牙功能的相关接口

3.10.1 duer_dcs_device_control_init

该函数用于初始化蓝牙接口，如果开发者需要使用蓝牙功能，则必须先调用该函数。

3.10.2 duer_dcs_bluetooth_set_handler

开发者需要实现此接口，此接口功能是打开或者关闭蓝牙功能。
注意：该函数需要立即返回，不能被阻塞。
参数

名称	类型	说明
is_switch	bool	true：开启蓝牙功能，false：关闭蓝牙功能
target	const char *	预留参数

返回值
none

3.10.3 duer_dcs_bluetooth_connect_handler

开发者需要实现此接口，此接口功能是连接或者断开蓝牙功能。
注意：该函数需要立即返回，不能被阻塞。
参数

名称	类型	说明
is_connect	bool	true：连接蓝牙，false：断开蓝牙
target	const char *	预留参数

返回值
none

4 libduer-device平台移植说明

4.1 移植简介

libduer-device库是设备端接入dueros的通用库。只需要针对不同的平台进行简单的适配即可使用。
`void baidu_ca_adapter_initialize()` ：此函数会在CA模块启动时被调用，用于初始化各种平台自定义资源。
开发者根据各平台的资源实现以下相关功能：

- 内存分配
- 线程锁
- debug信息
- socket
- 获取线程id
- 系统时间戳
- sleep
- 随机数

4.2 移植适配接口

4.2.1 baidu_ca_memory_init —— 内存分配

此接口用于注册实现内存分配的回调函数。通过此接口注册三个回调函数分别实现标准c函数的malloc、realloc、free功能。

接口参数

名称	类型	说明
context	duer_context	用户自定义参数，会在其他三个回调函数调用时以参数形式传入回调函数
f_malloc	duer_malloc_f	回调函数指针，指向实现标准c函数的malloc功能的函数
f_realloc	duer_realloc_f	回调函数指针，指向实现标准c函数的realloc功能的函数
f_free	duer_free_f	回调函数指针，指向实现标准c函数的free功能的函数

参数类型说明

- **duer_context 定义：**

```
typedef void* duer_context;
```

用户自定义参数，`void*`类型，方便各种用户自定义类型指针来传递参数。

- **duer_malloc_f 定义：**

```
typedef void* (*duer_malloc_f)(duer_context context, duer_size_t siz
e);
```

参数：

名称	类型	说明
context	duer_context	用户自定义参数
size	duer_size_t	分配内存的大小，此参数是 unsigned int 类型

返回值：

类型	说明
void *	指向分配内存的指针

- **duer_realloc_f 定义：**

```
typedef void* (*duer_realloc_f)(duer_context context, void*, duer_siz
e_t size);
```

参数：

名称	类型	说明
context	duer_context	用户自定义参数
-	void *	指向需要重新分配内存的指针
size	duer_size_t	重新分配内存的大小，此参数是 unsigned int 类型

返回值：

类型	说明
void *	重新指向分配内存的指针

- **duer_free_f 定义：**

```
typedef void (*duer_free_f)(duer_context context, void* p);
```

参数：

名称	类型	说明
context	duer_context	用户自定义参数
p	void *	指向需要释放内存的指针

返回值：
无

接口返回值

无

接口调用示例

```
// 此示例适用于linux平台
void *bcamem_alloc(duer_context ctx, duer_size_t size)
{
    return malloc(size);
}

void *bcamem_realloc(duer_context ctx, void *ptr, duer_size_t size)
{
    return realloc(ptr, size);
}

void bcamem_free(duer_context ctx, void *ptr)
{
    free(ptr);
}

void baidu_ca_adapter_initialize()
{
    ...
    baidu_ca_memory_init(NULL, bcamem_alloc, bcamem_realloc, bcamem_fre
e);
    ...
}
```

4.2.2 baidu_ca_mutex_init —— 线程锁

此接口用于注册实现线程锁的回调函数。通过此接口注册四个回调函数分别实现 创建锁 、 获取锁 、 解锁 、 删除锁 功能。

接口参数

名称	类型	说明
f_create	duer_mutex_create_f	回调函数指针，指向实现创建锁功能的函数
f_lock	duer_mutex_lock_f	回调函数指针，指向实现获取锁功能的函数
f_unlock	duer_mutex_unlock_f	回调函数指针，指向实现解锁功能的函数
f_destroy	duer_mutex_destroy_f	回调函数指针，指向实现删除锁功能的函数

参数类型说明

- **duer_mutex_create_f 定义：**

```
typedef void* duer_mutex_t;
typedef duer_mutex_t (*duer_mutex_create_f)();
```

参数：
无
返回值：

类型	说明
duer_mutex_t	指向创建的线程锁的指针

- **duer_mutex_lock_f 定义：**

```
typedef int            duer_status_t;
typedef duer_status_t (*duer_mutex_lock_f)(duer_mutex_t mtx);
```

参数：

名称	类型	说明
mtx	duer_mutex_t	指向线程锁的指针，传入回调函数内实现获取锁功能

返回值：

类型	说明
duer_status_t	0：获取锁成功，其他：获取锁失败

- **duer_mutex_unlock_f 定义：**

```
typedef duer_status_t (*duer_mutex_unlock_f)(duer_mutex_t mtx);
```

参数：

名称	类型	说明
mtx	duer_mutex_t	指向线程锁的指针，传入回调函数内实现解锁功能

返回值：

类型	说明
duer_status_t	0：解锁成功，其他：解锁失败

- **duer_mutex_destroy_f 定义：**

```
typedef duer_status_t (*duer_mutex_destroy_f)(duer_mutex_t mtx);
```

参数：

名称	类型	说明
mtx	duer_mutex_t	指向线程锁的指针，传入回调函数内实现删除锁功能

返回值：

类型	说明
duer_status_t	0：删除锁成功，其他：删除锁失败

参数返回值

无
接口调用示例

```
// 此示例适用于linux平台
duer_mutex_t create_mutex(void)
{
    duer_mutex_t mutex;

    pthread_mutexattr_t attr;
    int ret = pthread_mutexattr_init(&attr);
    if (ret) {
        DUER_LOGW("pthread_mutexattr_init fail!, ret:%d", ret);
        return NULL;
    }
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);// support
 recursive
```

```
    pthread_mutex_t* p_mutex = (pthread_mutex_t*)DUER_MALLOC(sizeof(pthre
ad_mutex_t));
    if (!p_mutex) {
        DUER_LOGW("malloc mutext fail!");
        return NULL;
    }

    ret = pthread_mutex_init(p_mutex, &attr);
    if (ret) {
        free(p_mutex);
        p_mutex = NULL;
        DUER_LOGW("pthread_mutex_init fail!, ret:%d", ret);
        return NULL;
    }
    mutex = (duer_mutex_t)p_mutex;
    pthread_mutexattr_destroy(&attr);

    return mutex;
}

duer_status_t lock_mutex(duer_mutex_t mutex)
{
    pthread_mutex_t* p_mutex;

    if (!mutex) {
        return DUER_ERR_FAILED;
    }

    p_mutex = (pthread_mutex_t*)mutex;

    int ret = pthread_mutex_lock(p_mutex);
    if(ret) {
        DUER_LOGI("pthread_mutex_lock fail!, ret:%d", ret);
        return DUER_ERR_FAILED;
    }
    return DUER_OK;
}

duer_status_t unlock_mutex(duer_mutex_t mutex)
{
    pthread_mutex_t* p_mutex;

    if (!mutex) {
        return DUER_ERR_FAILED;
    }

    p_mutex = (pthread_mutex_t*)mutex;

    int ret = pthread_mutex_unlock(p_mutex);
    if(ret) {
        DUER_LOGW("pthread_mutex_unlock fail!, ret:%d", ret);
        return DUER_ERR_FAILED;
```

```
    }
    return DUER_OK;
}

duer_status_t delete_mutex_lock(duer_mutex_t mutex)
{
    pthread_mutex_t* p_mutex;
    if (!mutex) {
        return DUER_ERR_FAILED;
    }

    p_mutex = (pthread_mutex_t*)mutex;
    pthread_mutex_destroy(p_mutex);
    DUER_FREE(p_mutex);

    return DUER_OK;
}

void baidu_ca_adapter_initialize()
{
    ...
    baidu_ca_mutex_init(create_mutex, lock_mutex, unlock_mutex, delete_mu
tex_lock);
    ...
}
```

4.2.3 baidu_ca_debug_init —— debug信息

此接口用于注册实现打印日志的回调函数。通过此接口注册一个实现打印日志信息功能的回调函数。

接口参数

名称	类型	说明
context	duer_context	用户自定义参数，会在回调函数调用时以参数形式传入回调函数
f_debug	duer_debug_f	回调函数指针，指向实现打印日志功能的函数

参数类型说明

- **duer_debug_f 定义：**

```
    typedef unsigned int   duer_u32_t;
    typedef void (*duer_debug_f)(duer_context ctx,
                                 duer_u32_t level,
                                 const char* file,
                                 duer_u32_t line,
```

```
                                const char* fmt);
```

参数：

名称	类型	说明
ctx	duer_context	用户自定义参数
level	duer_u32_t	打印log的级别。分为5级，1：ERROR，2：Warning，3：Info，4：Debug，5：Verbose
file	const char*	文件名称
line	duer_u32_t	行号
fmt	const char*	打印日志的详细信息

返回值：
无

接口返回值

无

接口调用示例

```
// 此示例适用于linux平台
void bcadbg(duer_context ctx, duer_u32_t level, const char *file, duer_u3
2_t line, const char *msg)
{
    if (file == NULL) {
        file = "unkown";
    }
    printf("[%s]{tid:%d}(%u)%s(%d):%s",
            duer_get_tag(level),
            gettid(),
            duer_timestamp(),
            file,
            line,
            msg);
}

void baidu_ca_adapter_initialize()
{
    ...
    baidu_ca_debug_init(NULL, bcadbg);
    ...
}
```

4.3.4 baidu_ca_transport_init —— socket

此接口用于注册各平台socket相关的回调函数。回调函数分别实现以下功能： 创建socket 、 连接 、 发送数据 、 接收数据 、 断开连接 、 删除socket 。
此接口使用前，应完成平台socket的初始化工作，可以封装一个 bcasoc_initialize() 函数用来对socket进行初始化，如：名称、线程栈的大小、socket队列的大小等等。
接口参数

名称	类型	说明
f_create	duer_soc_create_f	回调函数指针，指向实现创建socket功能的函数，此函数为socket分配并初始化资源
f_conn	duer_soc_connect_f	回调函数指针，指向实现连接功能的函数
f_send	duer_soc_send_f	回调函数指针，指向实现发送数据功能的函数，此回调函数不可阻塞
f_recv	duer_soc_recv_f	回调函数指针，指向实现接收数据功能的函数，f_recv_timeout参数为NULL时会调用此函数，此回调函数不可阻塞
f_recv_timeout	duer_soc_recv_timeout_f	回调函数指针，指向实现带超时机制的接收数据功能的函数，此回调函数不可阻塞
f_close	duer_soc_close_f	回调函数指针，指向实现断开连接功能的函数
f_destroy	duer_soc_destroy_f	回调函数指针，指向实现释放socket资源的函数

注： f_recv和f_recv_timeout为接收数据的回调函数，注册一个即可，不注册的参数传NULL。也可都注册，f_recv_timeout优先调用。两个参数不能同时为NULL，否则将不能接收数据。

参数类型说明

- **duer_soc_create_f 定义：**

```
    typedef void*           duer_socket_t;
    typedef enum _duer_transport_event_enum {
        DUER_TEVT_CONNECT_RDY,  // socket准备连接事件，此时socket资源分配成功，
未进行连接
        DUER_TEVT_SEND_RDY,     // socket准备发送数据事件，此时socket空闲状态，
可以发送数据
        DUER_TEVT_RECV_RDY,     // socket准备接收数据事件，此时socket空闲状态，
可以接收数据
        DUER_TEVT_CLOSE_RDY,    // socket接收关闭事件，此时接收数据异常，需要停
止接收数据
        DUER_TEVT_SEND_TIMEOUT, // socket发送数据超时事件，此时接收数据超时，需
要停止发送数据
        DUER_TEVT_NONE = -1
    } duer_transevt_e;
```

```
typedef void (*duer_transevt_func)(duer_transevt_e event);
typedef duer_socket_t (*duer_soc_create_f)(duer_transevt_func func);
```

参数：

名称	类型	说明
func	duer_transevt_func	回调函数指针，此函数用于将socket的状态事件发送给CA模块，由CA模块进行相关操作

返回值：

类型	说明
duer_socket_t	`void*` 类型，指向被操作的socket的指针

- **duer_soc_connect_f 定义：**

```
typedef unsigned short duer_u16_t;
typedef unsigned char  duer_u8_t;
typedef unsigned int   duer_size_t;
typedef struct _duer_address_s {
    duer_u8_t    type;      // socket的类型
    duer_u16_t   port;      // socket的端口号
    void*        host;      // 指向存储socket的主机地址的内存的首地址
    duer_size_t  host_size; // 存储socket的主机地址的内存的大小
} duer_addr_t;
typedef duer_status_t (*duer_soc_connect_f)(duer_socket_t sock, const duer_addr_t* addr);
```

参数：

名称	类型	说明
sock	duer_socket_t	指向被操作的socket的指针
addr	const duer_addr_t*	指向存储socket详细信息的内存地址

返回值：

类型	说明
duer_status_t	socket连接状态，0：成功，其他：失败

- **duer_soc_send_f 定义：**

```
typedef duer_status_t (*duer_soc_send_f)(duer_socket_t sock,
                                         const void* data,
                                         duer_size_t size,
                                         const duer_addr_t* addr);
```

参数：

名称	类型	说明
sock	duer_socket_t	指向被操作的socket的指针
data	const void*	指向要发送数据的首地址
size	duer_size_t	发送数据的字节数
addr	const duer_addr_t*	指向存储socket详细信息的内存地址

返回值：

类型	说明
duer_status_t	发送数据状态，0：成功，其他：失败

- **duer_soc_recv_f 定义：**

```
typedef duer_status_t (*duer_soc_recv_f)(duer_socket_t sock,
                                         void* data,
                                         duer_size_t size,
                                         duer_addr_t* addr);
```

参数：

名称	类型	说明
sock	duer_socket_t	指向被操作的socket的指针
data	void*	指向需要接收数据的首地址
size	duer_size_t	需要接收数据的字节数
addr	const duer_addr_t*	指向存储socket详细信息的内存地址

返回值：

类型	说明
duer_status_t	接收数据状态，0：成功，其他：失败

- **duer_soc_recv_timeout_f 定义：**

```
typedef duer_status_t (*duer_soc_recv_timeout_f)(duer_socket_t sock,
                                                 void* data,
                                                 duer_size_t size,
                                                 duer_u32_t timeout,
                                                 duer_addr_t* addr);
```

参数：

名称	类型	说明
sock	duer_socket_t	指向被操作的socket的指针
data	void*	指向需要接收数据的首地址
size	duer_size_t	需要接收数据的字节数
timeout	duer_u32_t	接收数据的超时时间
addr	const duer_addr_t*	指向存储socket详细信息的内存地址

返回值：

类型	说明
duer_status_t	接收数据状态，0：成功，其他：失败

- **duer_soc_close_f 定义：**

```
typedef duer_status_t (*duer_soc_close_f)(duer_socket_t sock);
```

参数：

名称	类型	说明
sock	duer_socket_t	指向被操作的socket的指针

返回值：

类型	说明
duer_status_t	socket关闭状态，0：成功，其他：失败

- **duer_soc_destroy_f 定义：**

```
typedef duer_status_t (*duer_soc_close_f)(duer_socket_t sock);
```

参数：

名称	类型	说明
sock	duer_socket_t	指向被操作的socket的指针

返回值：

类型	说明
duer_status_t	socket删除状态，0：成功，其他：失败

接口返回值

无

4.2.5 duer_thread_init —— 线程id

此接口用于注册实现获取线程id的回调函数。非必须接口，开发者按自己的需求进行使用。

接口参数

名称	类型	说明
_f_get_thread_id	duer_get_thread_id_f_t	回调函数指针，指向实现获取当前线程的id功能的函数

参数类型说明

- **duer_get_thread_id_f_t 定义：**

```
typedef duer_u32_t (*duer_get_thread_id_f_t)();
```

参数：
无
返回值：

类型	说明
duer_s32_t	`unsigned int` 类型，表示线程的id

接口返回值

无

接口调用示例

```
// 此示例适用于linux平台
duer_u32_t duer_get_thread_id_impl() {
    pthread_t thread_id = pthread_self();
    return (duer_u32_t)thread_id;
}

void baidu_ca_adapter_initialize()
{
    ...
    baidu_ca_timestamp_init(duer_timestamp_obtain);
    ...
}
```

4.2.6 baidu_ca_timestamp_init —— 系统时间戳

此接口用于注册实现获取系统时间戳的回调函数。通过此接口注册一个实现获取系统时间戳功能的回调函数。
注：此接口系统时间戳的单位应是毫秒（ms），否则会发生不可预知的BUG。

接口参数

名称	类型	说明
f_timestamp	duer_timestamp_f	回调函数指针，指向实现获取系统时间戳功能的函数

参数类型说明

- **duer_debug_f 定义：**

```
typedef duer_u32_t (*duer_timestamp_f)();
```

参数：
无
返回值：

类型	说明
duer_s32_t	`unsigned int` 类型的系统时间戳

接口返回值

无

接口调用示例

```
// 此示例适用于linux平台
static duer_u32_t duer_timestamp_obtain()
{
    DUER_LOGV("duer_timestamp_obtain");
    struct timeval tv;
    int ret = gettimeofday(&tv, NULL);
    if (ret) {
        perror("gettimeofday error");
        return 0;
    }
    unsigned long long time_in_ms = tv.tv_sec * 1000 + tv.tv_usec / 1000;
    return (duer_u32_t)time_in_ms;
}

void baidu_ca_adapter_initialize()
{
    ...
    baidu_ca_timestamp_init(duer_timestamp_obtain);
    ...
}
```

4.2.7 baidu_ca_sleep_init —— sleep

此接口用于注册实现延时的回调函数。通过此接口注册一个实现延时功能的回调函数。
注： 此接口延时的单位应是毫秒（ms），否则会发生不可预知的BUG。

接口参数

名称	类型	说明
_f_sleep	duer_sleep_f_t	回调函数指针，指向实现延时功能的函数

参数类型说明

- **duer_sleep_f_t 定义：**

```
typedef void (*duer_sleep_f_t)(duer_u32_t ms);
```

参数：

名称	类型	说明
ms	duer_u32_t	延时时间，单位：ms

返回值：
无

接口返回值

无

接口调用示例

```
// 此示例适用于linux平台
static void duer_sleep_impl(duer_u32_t ms) {
    usleep(ms * 1000);
}

void baidu_ca_adapter_initialize()
{
    ...
    baidu_ca_sleep_init(duer_sleep_impl);
    ...
}
```

4.2.8 duer_random_init —— 随机数

此接口用于注册实现获取随机数的回调函数。通过此接口注册一个实现获得随机数功能的回调函数。
注：此接口需要实现真随机数，否则会发生不可预知的BUG。

接口参数

名称	类型	说明
f_random	duer_random_f	回调函数指针，指向实现真随机数功能的函数

参数类型说明

- **duer_random_f 定义：**

```
typedef duer_s32_t (*duer_random_f)();
```

参数：
无
返回值：

类型	说明
duer_s32_t	`unsigned int` 类型的真随机数

接口返回值

无

接口调用示例

```
// 此示例适用于linux平台
duer_s32_t duer_random_impl(void) {
    duer_s32_t rs = 0;
    struct timeval tpstart;
    gettimeofday(&tpstart, NULL);
    pid_t tid = gettid();
    //how to generate seed fro rand
    duer_u32_t seed = (tid << 5) | tpstart.tv_usec;
    srand(seed);
    rs = rand();
    return rs;
}
void baidu_ca_adapter_initialize()
{
    ...
    duer_random_init(duer_random_impl);
    ...
}
```