

# W217 CMS-OTA 组件移植使用文档

---

Version: 1.5

Date: 2023-04-11



[iot.10086.cn](http://iot.10086.cn)

# 目 录

<b>1 概述</b>	<b>1</b>
1.1 CMS	1
1.2 OTA	1
<b>2 静态代码组织</b>	<b>3</b>
2.1 组件代码分布	3
<b>3 接口说明</b>	<b>4</b>
3.1 系统适配接口	4
3.1.1 flash 接口	4
3.1.2 kernel 列表	7
3.1.3 sys 列表	16
3.2 OTA 接口	22
3.2.1 API 列表	22
3.2.2 API 说明	24
3.3 CMS 接口	28
3.3.1 cms_id_set_did	28
3.4 功能控制定义	29
3.4.1 CMS 控制	29
3.4.2 OTA 控制	29
<b>4 升级操作流程</b>	<b>31</b>
4.1 云平台创建项目	31
4.2 本地配置	33
4.3 生成差分包	33
4.3.1 本地打包	33
4.3.2 平台差分	34
4.4 升级测试	38
4.4.1 一键升级 demo	38

4.4.2 查询是否有新版本 .....	39
4.4.3 有新版本后下载 .....	39
<b>5 FAQ.....</b>	<b>40</b>
5.1 如何调用升级接口? .....	40
5.2 调用 cmiot_upgrade()接口总是超时是什么原因? .....	42
5.3 如何设置设备 device ID? .....	42
5.4 假如文件下载快完成了, 突然中断了, 下次下载会从头开始下载吗? .....	42
5.5 cmiot_upgrade 内部返回码 .....	42
<b>参考资料.....</b>	<b>44</b>

# 1 概述

CMIOT FOTA 组件包括两部分：CMS 和 OTA。CMS 提供云平台连接功能，OTA 提供升级包下载存储及升级结果上报等功能。这两部分内容都需要移植到终端才能实现完整的 FOTA 功能。

## 1.1 CMS

CMS，全称是 China Mobile Service，即中国移动服务框架，是中国移动 OneOS 操作系统服务合集，包含端侧 CMS 基础组件和云端 CMS Cloud 服务。CMS 基础组件包含 CMS Connect 组件、CMS ID 组件及 CMS core lib，负责 CMS 基础的接入、传输、认证等功能。

CMS 作为 OTA 传输通道，只需要移植即可，无须用户调用相关接口。

## 1.2 OTA

远程升级（Firmware Over The Air）可通过无线网络实现固件升级。OneOS 的 OTA 为差分升级，采用此方式可极大降低升级包的大小。

CMS 云平台中的 OTA 服务提供制作差分包、差分包测试及发布、版本管理、查询升级日志等功能。

OTA 组件包括 BootLoader 和 APP，BootLoader 负责保存引导、差分还原，APP 负责制作差分包压缩文件（前置版本压缩包、目标版本压缩包）、版本监测、固件差分包下载、版本校验、日志上报。

本文档包含 CMS 云平台 OTA 服务的使用说明及 OTA APP 组件移植说明。

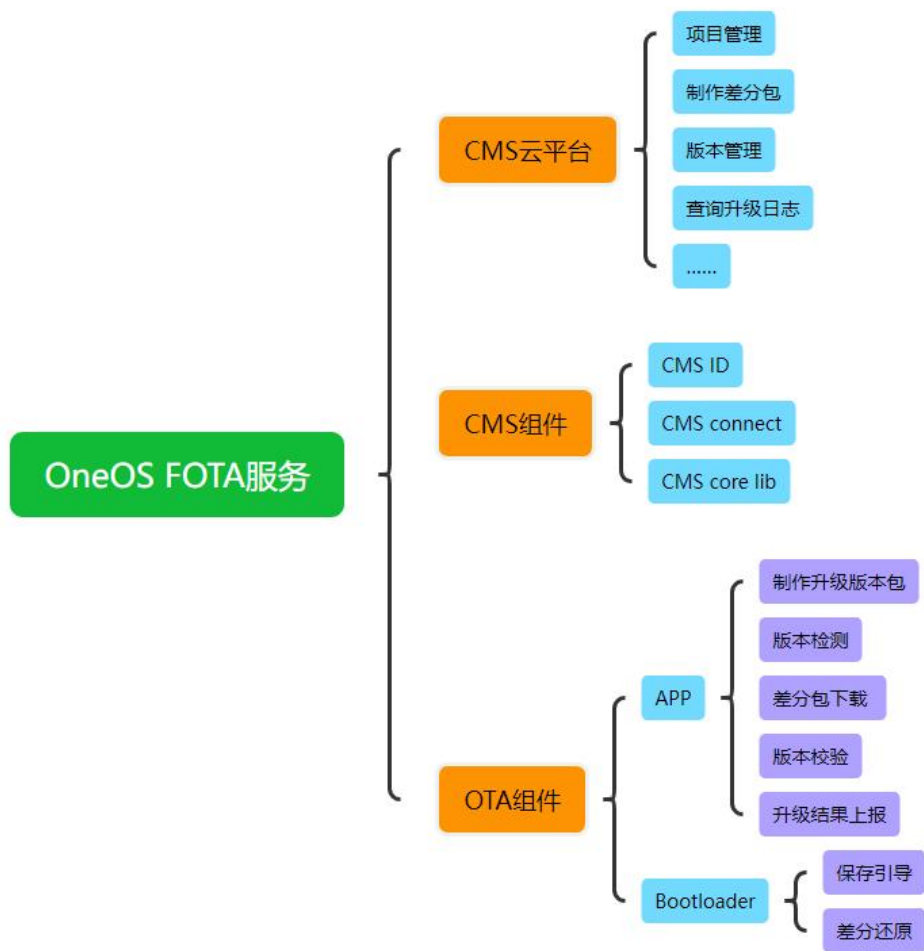


图 1.1 FOTA 组成

## 2 静态代码组织

### 2.1 组件代码分布

CMIOT FOTA 代码分为以下几部分：

1. **cms** 目录：包含 CMS 用户调用的接口声明及 lib 依赖接口实现，其中 **atest** 目录仅供测试参考，可以忽略；

`\Third-party\oneos\cms`

2. **ota** 目录：包含 OTA 用户调用接口声明及 lib 依赖接口实现，另外 **pack** 目录存放用于制作差分升级包压缩文件的脚本；

`\Third-party\oneos\ota\cmiot`

3. **os\_port** 适配目录：包含移植时需要适配的系统接口及定义；

`\external\oneos`

4. **oneos\_config.h** 用于组件配置；

`\Third-party\oneos\oneos_config.h`

具体使用说明请参照以下章节。

## 3 接口说明

### 3.1 系统适配接口

需要适配的接口包括 fal(flash 操作接口), kernel(内存、mutex、semaphore、task、netdb 等), sys(socket 相关内容)三部分, **以下接口如果未适配, 则需要用户在新平台上自行实现。**

代码路径: external\oneos\

#### 3.1.1 flash 接口

位于\inc\drivers\fal\_part.h、fal.h

##### 3.1.1.1 重要结构体

```
typedef struct fal_part_info
{
    char name[FAL_DEV_NAME_MAX]; /* flash 分区名, 必须要定义 */
    char flash_name[FAL_DEV_NAME_MAX]; /* 该分区所属 flash 名 */
    os_uint32_t offset; /* 该分区在 flash 的偏移地址, 必须要定义 */
    os_uint32_t size; /* 该分区字节大小, 必须要定义 */
    os_uint32_t flags; /* 未使用 */
}fal_part_info_t; /* 分区信息表 */

struct fal_flash /* flash 信息表 */
{
    char name[FAL_DEV_NAME_MAX]; /* flash 名 */
    os_uint32_t page_size; /* flash 最小的读写字节大小, 必须要定义 */
    os_uint32_t block_size; /* flash 最小擦除字节大小, 必须要定义 */
    os_uint32_t capacity; /* flash 容量字节大小, 必须要定义 */
    struct fal_flash_ops ops; /* flash 读写擦除操作函数指针 */
}
```

```

    os_list_node_t list;

    void *priv;
};

typedef struct fal_part
{
    fal_part_info_t *info; /* 指向分区信息表，必须要定义 */
    fal_flash_t      *flash; /* 指向包含本分区的 flash 信息表，必须要定义 */
    os_uint32_t flags; /* 未使用 */
} fal_part_t;

```

若平台不支持 FAL，可以自定义 `fal_part_t` 结构体变量，手动填入必须要定义的成员，实现相关接口。

### 3.1.1.2 fal\_part\_find

该函数用于根据分区名获取 flash 分区信息。原型如下：

```

fal_part_t * fal_part_find(const char *name);

```

参数	说明
name	分区名，详见 3.4.2 节 CMIOT_ONEOS_UPDATE.
返回	说明
<code>fal_part_t *</code>	分区详细信息,详见结构体定义
NULL	获取失败

### 3.1.1.3 fal\_part\_read

该函数用于读取 flash 分区特定地址数据。原型如下：

```

int fal_part_read(fal_part_t *part, os_uint32_t addr, os_uint8_t *buf, os_size_t size);

```

参数	说明
part	Flash 分区结构体



参数	说明
addr	读取数据偏移地址（相对于 part 分区首地址偏移）
buf	数据存储缓冲区
size	读取数据长度
返回	说明
>0	读取数据长度,一般为 size 长度
其他	读取数据失败

#### 3.1.1.4 fal\_part\_write

该函数用于向 flash 分区特定地址写入数据。原型如下：

```
int fal_part_write(fal_part_t *part, os_uint32_t addr, const os_uint8_t *buf, os_size_t size);
```

参数	说明
part	Flash 分区结构体
addr	写数据偏移地址（相对于 part 分区首地址偏移）
buf	数据存储缓冲区
size	写入数据长度
返回	说明
>0	写入数据长度,一般为 size 长度
其他	写入数据失败

#### 3.1.1.5 fal\_part\_erase

该函数用于擦除 flash 分区某块地址。原型如下：

```
int fal_part_erase(fal_part_t *part, os_uint32_t addr, os_size_t size);
```

参数	说明
----	----

参数	说明
part	Flash 分区结构体
addr	擦除数据偏移地址（相对于 part 分区首地址偏移）
size	擦除块长度
返回	说明
<0	擦除失败
其他	擦除成功

### 3.1.2 kernel 列表

位于\inc\kernel\os\_assert.h

#### 3.1.2.1 OS\_ASSERT

ASSERT 异常处理函数。

```
void OS_ASSERT(os_bool_t condition);
```

参数	说明
condition	异常判断条件
返回	说明
无	无

位于\inc\kernel\os\_clock.h

#### 3.1.2.2 os\_tick\_get

获取当前系统时间，以 tick 计算。

```
os_tick_t os_tick_get(void);
```

参数	说明
无	无
返回	说明
os_tick_t	当前系统时间，以 tick 为单位

### 3.1.2.3 os\_tick\_from\_ms

将单位 ms 转换为 tick。

```
os_tick_t os_tick_from_ms(os_uint32_t ms);
```

参数	说明
ms	ms 时间
返回	说明
os_tick_t	tick 时间

位于\inc\kernel\os\_memory.h

### 3.1.2.4 os\_malloc

从 heap 中申请分配指定大小 byte 的内存空间。

```
void *os_malloc(os_size_t size)
```

参数	说明
Size	申请内存大小，单位 byte
返回	说明
Null	申请失败

参数	说明
非 null	分配的内存首地址

### 3.1.2.5 os\_free

释放指定地址的内存

```
void os_free(void *ptr)
```

参数	说明
ptr	待释放内存地址
返回	说明
无	无

### 3.1.2.6 os\_realloc

调整指定内存块大小，如果 size 为 0 则等同于 os\_free，如果指定内存地址为 null 则等同于 os\_malloc.

```
void *os_realloc(void *ptr, os_size_t size)
```

参数	说明
ptr	指定内存块地址
Size	调整后大小
返回	说明
Null	调整失败
其他	调整后的内存地址

### 3.1.2.7 os\_calloc

从 heap 申请一块内存组，每个元素内存大小相同，并初始化为 0.

```
void *os_alloc(os_size_t count, os_size_t size)
```

参数	说明
count	申请的内存组的元素数量
Size	每个元素内存大小
返回	说明
Null	申请失败
非 null	分配的内存首地址

位于\inc\kernel\os\_mutex.h

### 3.1.2.8 os\_mutex\_create

创建一个互斥量 mutex，注意：必须支持可递归的 recursive。

```
os_mutex_t *os_mutex_create(const char *name, os_bool_t recursive)
```

参数	说明
name	mutex 名称
recursive	是否可递归
返回	说明
Null	创建失败
非 null	Mutex 地址

### 3.1.2.9 os\_mutex\_destroy

销毁指定的 mutex。

```
os_err_t os_mutex_destroy(os_mutex_t *mutex)
```

参数	说明
----	----

参数	说明
mutex	mutex 地址
返回	说明
0	销毁成功
非 0	销毁失败

### 3.1.2.10 os\_mutex\_recursive\_lock

递归的锁 mutex，如果其他任务已经锁住，则一直 block 到 mutex 可用或超时为止。

```
os_err_t os_mutex_recursive_lock(os_mutex_t *mutex, os_tick_t timeout)
```

参数	说明
mutex	mutex 地址
Timeout	超时时间，单位 tick
返回	说明
0	锁成功
非 0	锁失败

### 3.1.2.11 os\_mutex\_recursive\_unlock

递归的释放 mutex.

```
os_err_t os_mutex_recursive_unlock(os_mutex_t *mutex)
```

参数	说明
mutex	mutex 地址
返回	说明

参数	说明
0	释放成功
非 0	释放失败

位于\inc\kernel\os\_sem.h

### 3.1.2.12 os\_sem\_create

创建信号量 semaphore

```
os_sem_t *os_sem_create(const char *name, os_uint32_t value, os_uint32_t max_value)
```

参数	说明
name	信号量名称
Value	信号量初始值
Max_value	信号量最大值
返回	说明
Null	创建失败
非 null	创建成功的信号量地址

### 3.1.2.13 os\_sem\_destroy

销毁信号量

```
os_err_t os_sem_destroy(os_sem_t *sem)
```

参数	说明
Sem	信号量地址
返回	说明
0	销毁成功

参数	说明
非 0	销毁失败

#### 3.1.2.14 os\_sem\_wait

等待信号量，如果信号量值大于 0，则减 1 并立刻返回，否则一直等待或超时返回

```
os_err_t os_sem_wait(os_sem_t *sem, os_tick_t timeout)
```

参数	说明
Sem	信号量地址
Timeout	等待超时时间，单位 tick
返回	说明
0	等待成功
非 0	等待失败

#### 3.1.2.15 os\_sem\_post

释放信号量，信号量值会增加 1.

```
os_err_t os_sem_post(os_sem_t *sem)
```

参数	说明
Sem	信号量地址
返回	说明
0	释放成功
非 0	释放失败

位于\inc\kernel\os\_task.h



### 3.1.2.16 os\_task\_t

任务控制块结构体，此结构体用于识别每个任务，在任务操作时传入，用户可根据实际情况决定是否使用相关成员。

### 3.1.2.17 os\_task\_create

动态创建一个任务，任务控制块空间及栈空间需要此接口分配。任务创建后处于初始状态，需要通过 os\_task\_startup 接口启动任务。

```
os_task_t *os_task_create(const char *name,  
                           void      (*entry)(void *arg),  
                           void      *arg,  
                           os_uint32_t stack_size,  
                           os_uint8_t priority)
```

参数	说明
name	任务名称
entry	任务启动函数
arg	任务启动函数的参数
stack_size	任务栈大小
priority	任务优先级，值越小，优先级越高
返回	说明
Null	创建失败
非 null	创建成功的任务地址

### 3.1.2.18 os\_task\_destroy

销毁指定任务。

```
os_err_t os_task_destroy(os_task_t *task)
```

参数	说明
----	----

参数	说明
task	任务地址
返回	说明
0	销毁成功
非 0	销毁失败

### 3.1.2.19 os\_task\_startup

启动指定任务

```
os_err_t os_task_startup(os_task_t *task)
```

参数	说明
task	任务地址
返回	说明
0	启动成功
非 0	启动失败

### 3.1.2.20 os\_task\_msleep

使当前任务进入休眠状态

```
os_err_t os_task_msleep(os_uint32_t ms)
```

参数	说明
Ms	休眠时间，单位 ms
返回	说明
0	休眠成功
非 0	休眠失败

位于\inc\kernel\os\_utils.h

### 3.1.2.21 os\_kprintf

该函数用于打印日志，其函数原型如下：

```
void os_kprintf(const char *fmt, ...);
```

参数	说明
fmt	标准打印参数
返回	说明
无	无

位于\inc\kernel\ring\_buff.h

### 3.1.2.22 Ring buffer 接口

目前 ring buffer 已经实现在\src\kernel\ring\_buff.c。如果需要修改，请实现所有的接口。

## 3.1.3 sys 列表

目前 select.h 暂未使用，不用修改。socket 接口位于\inc\sys\socket.h

### 3.1.3.1 os\_socket

该函数用于创建一个 socket 链接。原型如下：

```
int os_socket(int domain, int type, int protocol);
```

参数	说明
domain	协议族，一般使用 AF_INET
type	Socket 类型，包括以下三类： #define SOCK_STREAM 1 #define SOCK_DGRAM 2 #define SOCK_RAW 3

参数	说明
protocol	通常为 0
返回	说明
$\geq 0$	创建成功的 socket 描述符 fd
-1	创建失败

### 3.1.3.2 os\_closesocket

该函数用于销毁一个 socket 链接。原型如下：

```
int os_closesocket(int fd);
```

参数	说明
fd	socket 描述符
返回	说明
0	销毁成功
其他	销毁失败，查看返回值确定原因

### 3.1.3.3 os\_connect

该函数用于和远端建立 tcp 连接。原型如下：

```
int os_connect(int fd, const struct sockaddr *name, socklen_t namelen);
```

参数	说明
fd	socket 描述符
name	远端服务器地址，详见 struct sockaddr
namelen	地址长度
返回	说明
0	连接成功

参数	说明
-1	连接失败

#### 3.1.3.4 os\_send

该函数用于发送 tcp 数据。原型如下：

```
int os_send(int fd, const void *data, os_size_t size, int flags);
```

参数	说明
fd	socket 描述符
data	待发送数据 buffer
size	待发送数据字节长度
flags	一般为 0
返回	说明
>0	发送数据长度
-1	发送失败

#### 3.1.3.5 os\_sendto

该函数用于发送 udp 数据。原型如下：

```
int os_sendto(int fd, const void *data, os_size_t size, int flags, const struct sockaddr *to, socklen_t tolen);
```

参数	说明
fd	socket 描述符
data	待发送数据 buffer
size	待发送数据字节长度
flags	一般为 0
to	数据发送目的地

参数	说明
tolen	地址长度
返回	说明
>0	发送数据长度
-1	发送失败

### 3.1.3.6 os\_select

该函数用于确定一个或多个 socket 状态。原型如下：

```
int os_select(int fd, os_bool_t isreadset, os_bool_t iswriterset, os_bool_t isexceptset, int timeout_ms)
```

参数	说明
fd	socket 描述符
isreadset	是否需要确定可读：1:是，0:不是。
iswriterset	是否需要确定可写：1:是，0:不是。
isexceptset	是否需要确定异常：1:是，0:不是。
timeout_ms	select 等待时间
返回	说明
-1	失败
>=0	检测到需要监测的状态

### 3.1.3.7 os\_setsockopt

该函数用于设置 socket 选项。原型如下：

```
int os_setsockopt(int fd, int level, int optname, const void *optval, socklen_t optlen);
```

参数	说明
----	----

fd	socket 描述符
level	选项定义的层次,目前仅使用 SOL_SOCKET。
optname	待设置选项,目前仅使用 SO_RCVTIMEO
optval	指向存放选项值缓冲区
optlen	缓冲区长度
返回	说明
0	设置成功
其他	设置失败

### 3.1.3.8 os\_recv

该函数用于接收 tcp 数据。原型如下：

```
int os_recv(int fd, void *mem, os_size_t len, int flags);
```

参数	说明
fd	socket 描述符
mem	接收数据 buffer
len	接收数据 buffer 字节长度
flags	一般为 0
返回	说明
>0	接收数据长度
-1	接收失败

### 3.1.3.9 os\_recvfrom

该函数用于接收 tcp 数据。原型如下：

```
int os_recvfrom(int fd, void *mem, os_size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
```

参数	说明
fd	socket 描述符
mem	接收数据 buffer
len	接收数据 buffer 字节长度
flags	一般为 0
from	数据来源地址
fromlen	作为输入时，fromlen 常置为 sizeof(struct sockaddr)；当输出时，fromlen 包含实际存入 buf 中的数据字节数
返回	说明
>0	接收数据长度
-1	接收失败

### 3.1.3.10 os\_gethostbyname

该函数用于根据主机名获取主机地址。原型如下：

```
struct hostent * os_gethostbyname(const char *name);
```

参数	说明
name	主机名字符串
返回	说明
struct hostent *	主机地址,详见结构体定义
NULL	获取失败

### 3.1.3.11 os\_getaddrinfo

该函数将主机名、主机地址、服务名和端口的字符串表示转换成套接字地址结构体。原型如下：

```
int os_getaddrinfo( const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);
```



参数	说明
Node	节点名可以是主机名，也可以是数字地址
service	包含十进制数的端口号或服务名
hints	一个空指针或指向一个 <code>addrinfo</code> 结构的指针，由调用者填写关于它所想返回的信息类型的线索
res	存放返回 <code>addrinfo</code> 结构链表的指针
返回	说明
0	获取成功
非 0	获取失败

### 3.1.3.12 os\_freeaddrinfo

释放 `os_getaddrinfo` 所返回的 `addrinfo` 链表。原型如下：

```
void os_freeaddrinfo(struct addrinfo *ai);
```

参数	说明
ai	存放返回 <code>addrinfo</code> 结构链表的指针
返回	说明
无	无

## 3.2 OTA 接口

### 3.2.1 API 列表

本章节只列出较为重要的一些接口。注：以下说明中“**无需调用**”表示该接口无需 OTA 使用人员主动调用，OTA 组件在需要的时候会自动调用，使用者只用根据自己的需求来重写。“**用户可更改**”表示用户可以根据平台实际情况移植时修改。

代码路径：\Third-party\oneos\ota\cmiot

文件名	接口	说明
cmiot_config.h	cmiot_get_try_time	OTA 组件 socket 的接收超时时间，单位毫秒，无需调用
cmiot_config.h	cmiot_get_try_count	OTA 组件在解码服务器的数据失败时，或端侧处理失败时，重新向服务器请求的次数，无需调用
cmiot_config.h	cmiot_get_utc_time	当前的 UTC 时间戳，暂时未用到，目前可保持默认值，无需调用
cmiot_log.h	cmiot_printf	OTA 组件日志输出接口，当前默认用 os_kprintf，用户可更改，无需调用
cmiot_user.h	cmiot_get_network_type	OTA 组件使用的通信网络类型，用于在服务端显示，目前可保持默认值，无需调用
cmiot_user.h	cmiot_download_name	差分包下载分区名字，用户可更改，但要和 fal_part_init 一致，无需调用
cmiot_user.h	cmiot_msleep	OTA 延时接口，用户可更改，无需调用
cmiot_user.h	cmiot_upgrade	检查新版本并下载固件包,供用户调用
cmiot_user.h	cmiot_report_upgrade	上报升级结果,供用户调用
cmiot_user.h	cmiot_check_version	检查是否有新版本,供用户调用
cmiot_user.h	cmiot_only_upgrade	存在新版本后调用此接口下载新版本,供用户调用
cmiot_client.h	cmiot_init	ota 初始化函数，无需调用

3.2.2 API 说明

3.2.2.1 cmiot\_get\_try\_time

该函数用于获取 socket 的接收超时时间，单位毫秒，可根据实际网络情况调整，其函数原型如下：

```
cmiot_uint32 cmiot_get_try_time(void);
```

参数	说明
无	无
返回	说明
cmiot_uint32	socket 接收超时时间

3.2.2.2 cmiot\_get\_try\_count

该函数用于获取在解码服务器的数据失败时，或端侧处理失败时，重新向服务器请求的次数，函数原型如下：

```
cmiot_uint8 cmiot_get_try_count(void);
```

参数	说明
无	无
返回	说明
cmiot_uint8	重试次数

3.2.2.3 cmiot\_get\_utc\_time

该函数用于获取当前的 UTC 时间戳，函数原型如下：

```
cmiot_uint32 cmiot_get_utc_time(void);
```

参数	说明
无	无
返回	说明

参数	说明
cmiot_uint32	当前 UTC 时间戳

#### 3.2.2.4 cmiot\_printf

该函数用于日志输出，该函数原型如下：

```
void cmiot_printf(cmiot_char *data, cmiot_uint32 len);
```

参数	说明
data	需要输出的字符串
len	字符串长度
返回	说明
无	无

#### 3.2.2.5 cmiot\_get\_network\_type

该函数用于获取网络类型，其函数原型如下：

```
cmiot_char *cmiot_get_network_type(void);
```

参数	说明
无	无
返回	说明
cmiot_char *	网络类型，用户自定义，不能为空

#### 3.2.2.6 cmiot\_download\_name

该函数用于获取 DOWNLOAD 分区的名字，该分区存储下载的升级包，该函数原型如下：

```
cmiot_char *cmiot_download_name(void);
```

参数	说明
----	----

参数	说明
无	无
返回	说明
cmiot_char *	DOWNLOAD 分区名字，不能为空

### 3.2.2.7 cmiot\_msleep

该函数用于延时，该函数原型如下：

```
void cmiot_msleep(cmiot_uint32 time);
```

参数	说明
time	延时，单位毫秒
返回	说明
无	无

### 3.2.2.8 cmiot\_upgrade

该函数检测新版本，如果新版本存在会下载固件升级包，其函数原型如下：

```
cmiot_int8 cmiot_upgrade(void);
```

参数	说明
无	无
返回	说明
E_CMIOT_SUCCESS	检测到新版本并下载成功，重启终端后会进入还原流程
E_CMIOT_FAILURE	无网络等异常原因
E_CMIOT_NOMEMORY	内存不足，初始化失败
E_CMIOT_LAST_VERSION	当前版本已经是最新
E_CMIOT_PACKAGE_TOO_LARGE	检测到新版本，但是版本数据过大，无法升级

### 3.2.2.9 cmiot\_report\_upgrade

该函数用于上报升级结果，其函数原型如下：

```
cmiot_int8 cmiot_report_upgrade(void);
```

参数	说明
无	无
返回	说明
E_CMIOT_SUCCESS	升级结果上报成功
E_CMIOT_FAILURE	无网络等异常原因上报失败
E_CMIOT_NO_UPGRADE	未检测到升级，无需上报

### 3.2.2.10 cmiot\_check\_version

该函数用于检测是否存在新版本，其函数原型如下：

```
cmiot_int8 cmiot_check_version(char version[CMIOT_VERSION_NAME_MAX]);
```

参数	说明
version	如果新版本检测成功，存放新版本号，字符长度最大 CMIOT_VERSION_NAME_MAX
返回	说明
E_CMIOT_SUCCESS	检测成功
E_CMIOT_FAILURE	无网络等异常原因
E_CMIOT_NOMEMORY	内存不足，初始化失败
E_CMIOT_LAST_VERSION	当前版本已经是最新
E_CMIOT_PACKAGE_TOO_LARGE	检测到新版本，但是版本数据超限，无法升级
E_CMIOT_NEW_VERSION_EXIST	新版本已下载，请重启升级

### 3.2.2.11 cmiot\_only\_upgrade

该函数用于检测是否存在新版本，其函数原型如下：

```
cmiot_int8 cmiot_only_upgrade(void);
```

参数	说明
无	无
返回	说明
E_CMIOT_SUCCESS	下载成功，重启终端后会进入还原流程
E_CMIOT_FAILURE	无网络等异常原因
E_CMIOT_NOMEMORY	内存不足，初始化失败

## 3.3 CMS 接口

CMS 组件作为 OTA 连接云平台的通道，只需要修改设置设备 ID 的接口，其他接口不用调用。

代码路径：\Third-party\oneos\cms

### 3.3.1 cms\_id\_set\_did

位于\src\cms\_config.h

该函数返回端侧的设备 ID 给调用者，用于 OTA 升级时区分不同的终端设备，**需要**用户根据实际情况修改，其函数原型如下：

```
int cms_id_set_did(uint8_t *did, size_t did_max_len, size_t *did_len)
```

参数	说明
*did	存储本设备 ID 的 buffer
did_max_len	存储设备 ID buffer 的大小
did_len	设备 ID 的实际大小
返回	说明

参数	说明
0	获取成功
其他	获取失败

## 3.4 功能控制定义

### 3.4.1 CMS 控制

相关控制宏定义于 `oneos_config.h`

```
#define CMS_CONNECT_MQTT /* OTA 使用 http 传输则配置为 MQTT，如果使用 udp 则配置为 COAP*/
```

/\* 以下两项要根据平台配置修改，详见第 4.1 章 \*/

```
#define CMS_ID_SET_PID "RHpkm3417N"
```

```
#define CMS_ID_KEY "6E05C5A6F0788A5B5757B7CF33A3D2AB"
```

### 3.4.2 OTA 控制

相关控制宏定义于 `oneos_config.h`

```
#define CMIOT_DEFAULT_TRY_TIME 5000 /* socket 连接超时值，单位 ms */
```

```
#define CMIOT_DEFAULT_NETWORK_PROTOCOL 2 /* 网络类型选择：1.udp, 2.http，需要和 CMS 同步*/
```

```
#define CMIOT_FOTA_ALGORITHM 1 /* 差分还原算法，不能修改！ */
```

```
#define CMIOT_FOTA_AVAILABLE_RAM_SIZE 0x80000 /* OTA 还原可用 RAM 空间，平台使用*/
```

```
#define CMIOT_DEFAULT_SEGMENT_SIZE_INDEX 3 /* 差分包下载块大小，范围 1-5 */
```

```
#define CMIOT_UPDATE_SIZE 0xA0000 /* 下载差包分区大小平台使用*/
```

```
#define CMIOT_FOTA_CONTRAST_BLOCK_SIZE 64 /* 差分包对比大小，单位 KB，需要和还原算法一致*/
```



`#define CMIOT_FIRMWARE_VERSION "1.0" /* 指示版本号，升级只能从低版本往高版本方向进行，如 1.1->1.2 */`

`/*以下配置需要进行升级的 bin 文件及期对应地址，按以下格式新增或删除 bin 文件，最多支持 20 个 bin 文件进行升级，在打包升级包时会使用，详见 4.3.1*/`

`#define CMIOT_FOTA_FILENAME1 "mmi_res_240x240_child_circular.bin"`

`#define CMIOT_FOTA_ADDRESS1 0x90000004`

`#define CMIOT_FOTA_FILE_MORE1`

`#define CMIOT_FOTA_FILENAME2 "UWS6121E_WC_1H00_delta_nv.bin"`

`#define CMIOT_FOTA_ADDRESS2 0x90000100`

`#define CMIOT_FOTA_FILE_MORE2`

`#define CMIOT_FOTA_FILENAME3 "UWS6121E_UWS6121E_WC_1H00_cp.bin"`

`#define CMIOT_FOTA_ADDRESS3 0x90000101`

`#define CMIOT_FOTA_FILE_MORE3`

`#define CMIOT_FOTA_FILENAME4 "UWS6121E_UWS6121E_WC_1H00_stone.img.bin"`

`#define CMIOT_FOTA_ADDRESS4 0x80000003`

## 4 升级操作流程

### 4.1 云平台创建项目

确保账号有权限后，登录 cms 中台 CMS 中台地址，在 cms 主界面点击产品开发-产品管理-新建产品按钮：



图 4.1 CMS 平台入口

填写产品名称，选择芯片型号，选择所要使用的服务、通信方式以及协议，完成产品创建：

1

基础配置

2

完成

\* 产品名称:

CMS\_OTA

\* 操作系统:

☒ OneOS

☐ Linux

\* 芯片型号:

SOC / 紫光展锐 / W217 / W217

\* 支持服务:

☒ License授权管理服务

☒ 远程调试服务

☒ RTC实时音视频服务

☒ OnePos定位服务

☒ OTA升级服务

【CODESYS】服务需芯片适配支持，如有需要请 [提交工单](#)咨询

\* 通信方式:

☒ 2G/3G/4G/5G

☐ NB-IoT

☐ WiFi

☐ Zigbee

☐ 蓝牙

☐ 其他

\* 协议选择:

 MQTT

 CoAP

 TCP

 UDP

产品描述:

最多不超过100个字符

完成创建

取消

图 4.2 CMS 项目创建

完成产品创建后可以在 cms 产品界面查看刚刚创建的产品的详细信息，其中产品 ID 与产品秘钥要配置到端侧 onecos\_config 中：

CMS\_OTA

编辑删除

产品ID: g1i3zV1okW

操作系统: OneOS

产品密钥: 55FD426FD4D481226881AE7C1A853F0F 复制

支持服务:

✔ OTA升级服务

✔ OnePos定位服务

✔ RTC实时音视频服务

✔ 远程调试服务

✔ License授权管理服务

芯片型号: SOC/紫光展锐/W217/W217

通信方式: 2G/3G/4G/5G

协议选择: TCP

产品描述:

图 4.3 CMS 产品信息

## 4.2 本地配置

根据项目信息修改 `oneos_config.h` 中相关宏定义:

```
#define CMS_ID_SET_PID "g1i3zV1okW" /* 对应平台中的产品 ID */
```

```
#define CMS_ID_KEY "55FD426FD4D481226881AE7C1A853F0F" /* 对应平台中的产品密  
钥 */
```

## 4.3 生成差分包

### 4.3.1 本地打包

整个工程编译完成后, 通过脚本或 `cmd` 手动执行命令, 进到:

Third-party\oneos\cms


执行 `pack_cmd.bat <目录名称>`, 得到当前编译的 OTA 版本:

```

PS E:\03code\w217\oneos-ecosys-2X\Third-party\oneos> .\pack_cmd.bat v1.0
package source: v1.0
..\..\build\UWS6121E_WC_1H00_build\img\fota_bootloader.bin
..\..\build\UWS6121E_WC_1H00_build\img\mmi_res_240x240_child_circular.bin
..\..\build\UWS6121E_WC_1H00_build\img\nor_bootloader.bin
..\..\build\UWS6121E_WC_1H00_build\img\nor_bootloader.img
..\..\build\UWS6121E_WC_1H00_build\img\nor_fdl.bin
..\..\build\UWS6121E_WC_1H00_build\img\nor_fdl.img
..\..\build\UWS6121E_WC_1H00_build\img\nor_fdl1.bin
..\..\build\UWS6121E_WC_1H00_build\img\nor_fdl1.img
..\..\build\UWS6121E_WC_1H00_build\img\nvitem.bin
..\..\build\UWS6121E_WC_1H00_build\img\NV_GROUP.txt
..\..\build\UWS6121E_WC_1H00_build\img\preload_img.bin
..\..\build\UWS6121E_WC_1H00_build\img\UWS6121E_UWS6121E_WC_1H00.axf
..\..\build\UWS6121E_WC_1H00_build\img\UWS6121E_UWS6121E_WC_1H00.map
..\..\build\UWS6121E_WC_1H00_build\img\UWS6121E_UWS6121E_WC_1H00_cp.bin
..\..\build\UWS6121E_WC_1H00_build\img\UWS6121E_UWS6121E_WC_1H00_stone.bin
..\..\build\UWS6121E_WC_1H00_build\img\UWS6121E_UWS6121E_WC_1H00_stone.img
..\..\build\UWS6121E_WC_1H00_build\img\UWS6121E_WC_1H00.pac
..\..\build\UWS6121E_WC_1H00_build\img\UWS6121E_WC_1H00_delta_nv.bin
已复制      18 个文件。
UWS6121E_UWS6121E_WC_1H00_stone.img.bin
已复制      1 个文件。
删除文件 - E:\03code\w217\oneos-ecosys-2X\Third-party\oneos\info.txt
complete

```

一般升级至少要有两个版本，一个当前设备的版本，另一个升级目标的版本。可以通过版本号来识别，修改 oneos\_config.h 中 CMIOT\_FIRMWARE\_VERSION 定义后，执行打包命令就可以看到对应版本号及时间戳的升级压缩包。两个版本对应两个压缩包，然后再到平台上制作差分包。

 1.0\_20221101\_164853.zip  
 2.0\_20221101\_164841.zip

### 4.3.2 平台差分

回到 OTA 平台，点击 FOTA 升级-版本管理-添加版本：

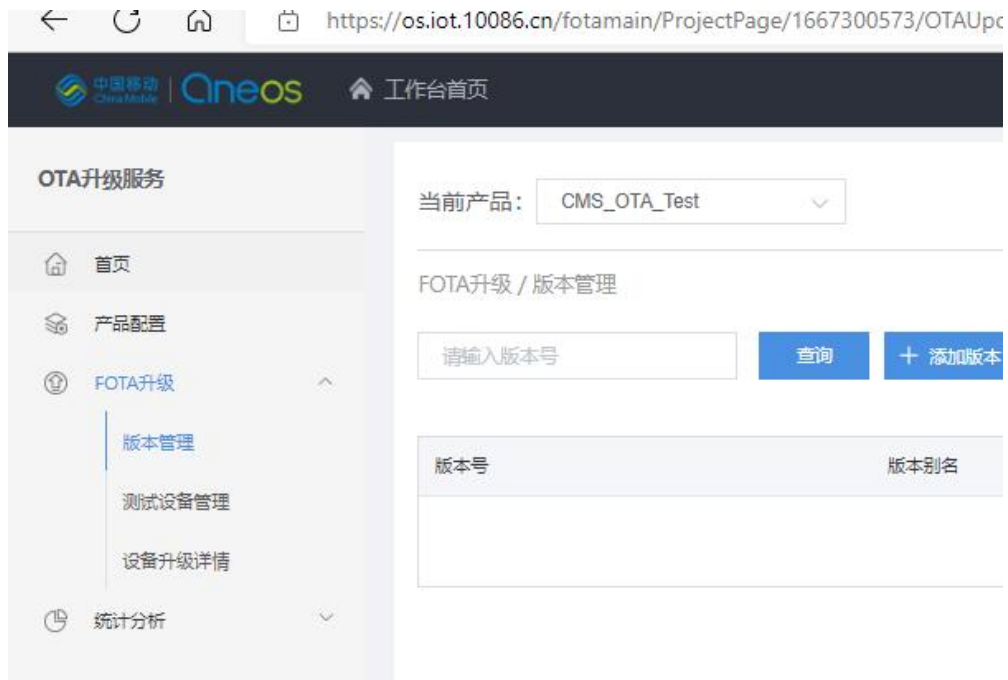


图 4.4 FOTA 版本管理入口

输入版本号，别名，描述，点击上传上一步打包的压缩文件，点确定：

添加版本

✕

版本号

1.0

版本别名

1.0

发布说明

📅

2022-11-01

增加语言

中文(中国) 

🌐

中文(中国)

1. 优化系统- 修复错误 

✎

描述

```
init_version
```

固件文件

点击上传

📁

1.0\_20221101\_164853.zip

取消

确定

图 4.5 FOTA 上传版本

同样上传 2.0 的版本后，可以看到对应的版本信息：

FOTA升级 / 版本管理					
<div>🔍 请输入版本号</div>	<div>📄</div>	<div>+ 添加版本</div>			
版本号	版本别名	固件文件	当前版本 <div>🔗</div>	状态	操作
1.0	1.0	1.0_20221101_164853.zip	无	初始版本	<a href="#">版本详情</a> <div>更多操作</div>
2.0	2.0	2.0_20221101_164841.zip	无	无升级包	<a href="#">配置升级任务</a> <a href="#">版本详情</a> <div>更多操作</div>

图 4.6 FOTA 版本信息

接下来配置升级任务，点击配置升级任务-配置，然后选择升级类型为差分，制作差分包为线上自动差分，可用 RAM 保持默认，RAM 值是根据 oneos\_config.h 中配置获取：

**操作系统**

OneOS

**升级类型**

☒ 差分 ☐ 整包

**制作差分包**

☒ 线上自动差分 ☐ 手动上传差分包

	版本号	固件文件
目标版本	2.0	2.0_20221101_1648...
源版本	1.0	1.0_20221101_1648...

**可用RAM**

4 (KB)

**高级设置 >**

图 4.7 FOTA 版本差分设置

点击确定生成差分升级包，制作完成后回到配置升级任务页，此时升级任务状态为待测试：

配置升级包。编辑策略、测试、发布等操作均5分钟后生效							
目标版本：2.0							
源版本	发布时间	升级包名称	升级包大小	状态	升级包	升级策略	操作
1.0	2022-11-01 17:12:25	dif_update20221101171224.zip.bin	168(0.00MB)	待测试	<a href="#">查看</a>	<a href="#">修改</a> <a href="#">删除</a> <a href="#">编辑</a>	<a href="#">开始测试</a>

图 4.8 FOTA 差分包信息



点击开始测试后，进入测试中状态，此时测试设备可以检测到新的版本（5 分钟后）。测试完成后，点击通过测试，发布版本。

配置升级包、策略策略、测试、发布等操作并5分钟后生效

目标版本: 2.0

源版本	发布时间	升级包名称	升级包大小	状态	升级包	升级策略	操作
1.0	2022-11-01 17:12:25	dif_update20221101171224.zip.bin	168(0.00MB)	发布中	<a href="#">查看</a>	<a href="#">查看</a>	<a href="#">停止发布</a>

图 4.9 FOTA 版本发布

## 4.4 升级测试

### 4.4.1 一键升级 demo

1. 引用头文件

```
#include <cmiot_user.h>
```

2. 添加上报升级结果和升级更新的代码

```
cmiot_int8 ret = E_CMIOT_FAILURE;
for (int i = 0; i < 5; ++i)
{
    ret = cmiot_report_upgrade();
    if ((ret == E_CMIOT_SUCCESS) || (ret == E_CMIOT_NO_UPGRADE)) // report success or no new upgrade
    {
        break;
    }
}
for (int i = 0; i < 5; ++i)
{
    ret = cmiot_upgrade();
    if (ret == E_CMIOT_SUCCESS) // get new version and download success
    {
        // reset MCU to upgrade
        break;
    }
    else if (ret == E_CMIOT_LAST_VERSION) // no new version
    {
        break;
    }
}
```

注：for 循环重复几次是避免网络问题导致连接失败；升级检测并下载成功后由应用决定是否重启，重启后会进入 bootloader 进行固件升级。

#### 4.4.2 查询是否有新版本

```
os_err_t cmiot_check(int argc, char **argv)
{
    ... cmiot_int8 rst = cmiot_check_version();
    ... os_kprintf("check version result:%d\r\n", rst);
    ... return 0;
}
MSH_CMD_EXPORT(cmiot_check, "CMIOT OTA check if new version exist")
```

#### 4.4.3 有新版本后下载

```
os_err_t cmiot_download(int argc, char **argv)
{
    ... cmiot_int8 rst = cmiot_only_upgrade();
    ... os_kprintf("download version result:%d\r\n", rst);
    ... return 0;
}
MSH_CMD_EXPORT(cmiot_download, "CMIOT OTA download the new version if exist")
```

## 5 FAQ

### 5.1 如何调用升级接口？

OTA 组件只提供机制，策略性的东西需要用户来定，比如可以每天检测一次，也可以每周检测一次，可以下载完就重启升级，也可以等别的业务完成再重启升级。

#### 1、检测是否有新版本

MMI 调用此接口检测是否有新版本，如果有可以进行下载，否则显示当前为最新版本

```
/**
*****
* @brief ..... 检测是否有新版本升级，网络正常才能使用
*
* @param[in] ..... version: 如果新版本检测成功，存放新版本号，字符长度最大CMIOT_VERSION_NAME_MAX
*
* @return ..... E_CMIOT_SUCCESS: 检测成功
* ..... E_CMIOT_FAILURE: 无网络等异常原因
* ..... E_CMIOT_NOMEMORY: 内存不足，初始化失败
* ..... E_CMIOT_LAST_VERSION: 当前版本已经是最新
* ..... E_CMIOT_PACKAGE_TOO_LARGE: 检测到新版本，但是版本数据超限，无法升级
* ..... E_CMIOT_NEW_VERSION_EXIST: 新版本已下载，请重启升级
*****
*/
cmiot_int8 cmiot_check_version(char version[CMIOT_VERSION_NAME_MAX])
```

#### 2、下载新版本

检测到有新版本后，调用此接口进行下载。

```
/**
*****
* @brief ..... 检测到新版本后，可调用此接口下载新版本
*
* @param
*
* @return ..... E_CMIOT_SUCCESS: 下载成功，重启终端后会进入还原流程
* ..... E_CMIOT_FAILURE: 无网络等异常原因
* ..... E_CMIOT_NOMEMORY: 内存不足，初始化失败
*****
*/
cmiot_int8 cmiot_only_upgrade(void)
```

#### 3、下载完成后重启

MMI 调用设备重启接口，具体实现由 SDK 提供

```
/**
 * *****
 * @brief ..... This function for reboot
 * @param[in] ... void
 *
 * @retval ..... void
 * *****
 */
void cmiot_reboot_device(void)
{
    extern void os_hw_cpu_reset(void);
    os_hw_cpu_reset();
}
```

#### 4、一键升级

此接口为一键升级功能，包含检测新版本，下载新版本过程

```
/**
 * *****
 * @brief ..... 一键升级接口,检测到新版本后直接升级
 *
 * @param
 *
 * @return ..... E_CMIOT_SUCCESS: 检测到新版本并下载成功,重启终端后会进入还原流程
 * ..... E_CMIOT_FAILURE: 无网络等异常原因
 * ..... E_CMIOT_NOMEMORY: 内存不足,初始化失败
 * ..... E_CMIOT_LAST_VERSION: 当前版本已经是最新
 * ..... E_CMIOT_PAKAGE_TOO_LARGE: 检测到新版本,但是版本数据过大,无法升级
 * *****
 */
cmiot_int8 cmiot_upgrade(void)
```

#### 5、上报升级结果

升级成功后需要上报升级结果，建议每次开机后调用此接口，如果有升级结果能够及时反馈给云平台。

```

/****
*****
* @brief ----- 升级结果上报接口
*
* @param
*
* @return ----- E_CMIOT_SUCCESS: 升级结果上报成功
* ----- E_CMIOT_FAILURE: 无网络等异常原因上报失败
* ----- E_CMIOT_NO_UPGRADE: 未检测到升级，无需上报
*****
*/
cmiot_int8 cmiot_report_upgrade(void)

```

## 5.2 调用 cmiot\_upgrade()接口总是超时是什么原因？

请根据网络适当加大接收超时时间，对应超时时间的宏为 CMIOT\_DEFAULT\_TRY\_TIME，在 cmiot\_config.h 文件里。

## 5.3 如何设置设备 device ID？

参考 3.3.1 接口，用户自己设定，用于云平台识别终端设备。

## 5.4 假如文件下载快完成了，突然中断了，下次下载会从头开始下载吗？

不会的，会从中断的地方开始下载。

## 5.5 cmiot\_upgrade 内部返回码

内部错误码	错误码说明
1000	success
1001	The productId is invalid
1002	The project does not exist
1003	The param is invalid
1004	Parameter missing

1005	The system is error
1006	JSON parsing exception
1007	Parameters do not conform to the rules
1008	MID Length Error
1009	AES Encryption Error
1010	To the maximum number of visits
1015	To the maximum number of personal user visits
2001	The sign is error
2101	No new updates were found
2103	Equipment not registered
2104	The project has no project team
2201	The download status is not legal
2301	The deltaID is not exist
2302	The upgrade status is not legal

## 参考资料

1. OTA 平台文档中心: <https://os.iot.10086.cn/otaplatform/DocumentCenterViewPage>