HarmonyOS Connect 直连方案（Wi-Fi+Cmobo+BLE+SLE）

文档版本 01

发布日期 2025-05-29

华为终端有限公司

商标声明


、 

、 

、华为，以上为华为公司的商标（非详尽清单），未经华为公司书面事先明示许可，任何第三方不得以任何形式使用。

注意

华为会不定期对本文档的内容进行更新。

本文档仅作为使用指导，文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为终端有限公司

地址： 广东省东莞市松山湖园区新城路 2 号

网址： https://consumer.huawei.com

目 录

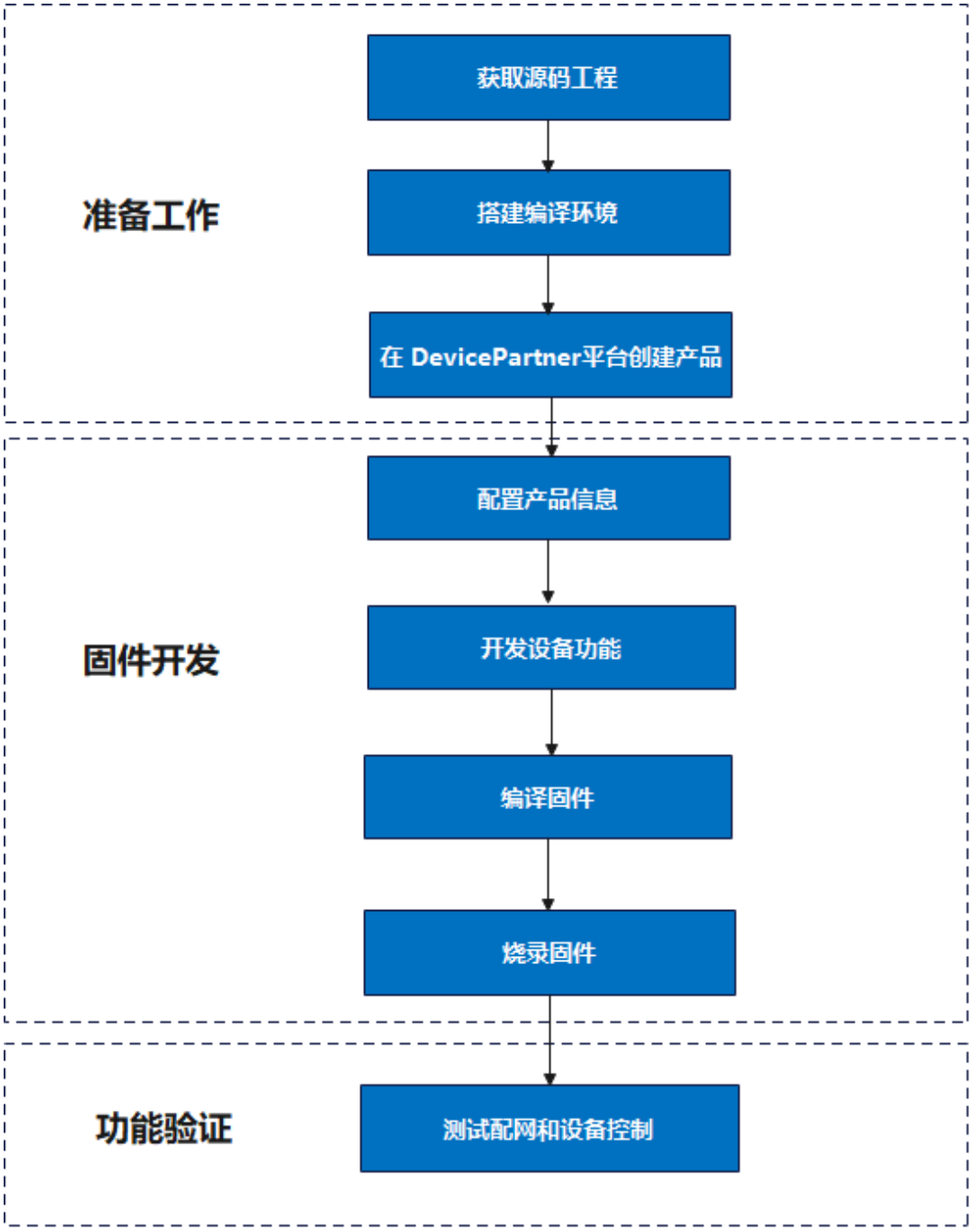
1 概述

简介

本文档为 HarmonyOS Connect 生态产品合作伙伴提供集成开发指导，该方案采用三种配网方式，分别为双联双控配网，SoftAp Wi-Fi 配网和蓝牙辅助配网，旨在帮助伙伴快速进行产品开发，完成产品信息配置、配网和设备控制等功能开发，并基于智慧生活 App 进行配网测试和设备控制测试。

开发流程

图1-1 设备集成开发流程



2 文档更新记录

日期	修订版本	修改描述
2025-05-12	4.0	【变更】新增双联双控特性和星闪相关接口操作说明信息。
2023-6-07	3.0	【变更】移除 Kit Framework 相关操作说明信息。
2022-5-30	2.0	1.【变更】SDK 升级后，接口和产品信息配置有调整，详见 4.2.1 配置 hilink_device.c 中产品信息 和 4.2.2 配置 hilink_device.c 中的服务信息。 2.【新增】设备发现方式增加蓝牙碰一碰和蓝牙靠近发现，详见 4.2.3 设备发现。 3.【新增】设备控制样例代码，详见 4.2.4 实现设备控制功能。 4.【变更】存在 MCU 的情况下，软件可见版本号需要需要包含 MCU 版本，详见 zh-cn_topic_0000001618863593.xml。 5.【变更】使用 Debug 版本智慧生活调测需要设置调测环境，详见配置智慧生活 App 测试环境。
2021-10-01	1.0	首次发布。

3 准备工作

步骤 1 联系模组商获取以下工具和信息。

表3-1 工具与信息

资料名称	说明
HarmonyOS Connect 服务包源码工程	用于开发固件，所集成的鸿蒙智联 SDK 需要为 release 版本。
编译工具链和使用指导	用于搭建编译环境。不同芯片采用的编译工具链不同，请联系模组商获取。
烧录工具	用于烧录固件。不同芯片采用的驱动和烧录工具不同，请联系模组商获取。
串口驱动	用于固件烧录过程中 PC 和设备的通信。不同芯片采用的串口驱动不同，请联系模组商获取。
SDK 集成路径	用于配置鸿蒙智联 SDK 所需产品信息。

当前已通过认证的芯片和模组参见 HarmonyOS Connect > 芯片与模组。

步骤 2 搭建编译环境。

步骤 3 安装开发板编译环境。

根据开发板实际型号，参考对应的章节进行配置。

步骤 4 在 Device Partner 平台创建产品时根据不同产品选不同的通信类型见表 3-2。

- “产品开发 > 产品定义 > 软硬件定义 > 极简连接”选择相应的方案类型。
- 导出产品信息，用于固件开发中的信息配置

表3-2

通信类型	极简连接	模组
Wi-Fi	极速常规配网 辅助配网（BLE/SLE）	支持所有鸿蒙模组，建议使用推荐模组，模组型号以实际页面显示为准。
Wi-Fi+BLE combo	蓝牙辅助配网 双连双控（WiFi+BLE）	
Wi-Fi+BLE+SLE	双连双控（Wi-Fi/蜂窝+BLE+SLE）	仅支持推荐模组，模组型号以实际页面显示为准。

步骤 5 在 Device Partner 平台的“产品开发”页面，单击“导出”可以获得产品基础信息，如图 3-1 所示。

图3-1 点击右上角箭头引导处导出产品信息

----结束

4 固件开发

4.1 简介

4.1.1 鸿蒙智联 SDK 开发包

鸿蒙智联 SDK 开发包含有 Wi-Fi 与 BLE/SLE 两部分，根据模组型号不同其内容不同。

- Wi-Fi 开发包文件说明：

目录	文件名	说明
lib/debug	libhilinkdevicesdk.a libhilinkota.a	鸿蒙智联 SDK 静态库文件 Debug 版本用于设备调测时集成 HOTA 升级特性静态库文件(仅支持使用华为 HOTA 云的产品） Debug 版本用于设备调测时集成
lib/release	libhilinkdevicesdk.a libhilinkota.a	Release 版本用于商用设备发布时集成 Release 版本用于商用设备发布时集成
include	hilink.h hilink_custom.h hilink_log_manage.h	鸿蒙智联主流程框架集成头文件 鸿蒙智联 SDK 定制化接口头文件 鸿蒙智联日志级别管理头文件
adapter/include	hilink_sal_aes.h hilink_mbedtls_utils.h hilink_network_adapter.h	AES 加解密适配层接口头文件 鸿蒙智联 SDK 软件适配层 TLS 基于 mbedTLS 实现内部接口头文件 网络适配层接口头文件

目录	文件名	说明
	hilink_tls_client.h hilink_kv_adapter.h hilink_open_ota_mcu_adapter.h hilink_time_adapter.h hilink_sal_drbg.h hilink_sys_adapter.h hilink_sal_kdf.h hilink_thread_adapter.h hilink_sal_base64.h hilink_sal_mpi.h hilink_sal_md.h hilink_stdio_adapter.h hilink_open_ota_adapter.h hilink_mem_adapter.h hilink_softap_adapter.h hilink_sal_rsa.h hilink_socket_adapter.h hilink_sal_defines.h hilink_str_adapter.h	TLS 客户端常用操作头文件，包括会话的创建、销毁等 鸿蒙智联 KV 头文件 MCU OTA 适配层接口头文件 时间适配层接口头文件 安全随机数适配层接口头文件 系统适配层接口头文件 密钥派生算法适配层接口头文件 线程适配层接口头文件 base64 编码解码适配层接口头文件 多精度整数适配层接口头文件 消息摘要算法适配层接口头文件 系统适配层标准输出接口头文件 OTA 适配层接口头文件 系统适配层内存接口头文件 SoftAP 适配层接口头文件 RSA 加解密适配层接口头文件 系统适配层网络 Socket 接口头文件 鸿蒙智联 SDK 软件适配层相关定义头文件 系统适配层字符串接口头文件
product	hilink_device.h hilink_device.c	鸿蒙智联产品适配头文件 鸿蒙智联产品适配实现源文件
demo	hilink_demo.c	鸿蒙智联 SDK 提供的 demo 示例代码
adapter	hilink_network_adapter.c hilink_socket_adapter.c hilink_open_ota_mcu_adapter.c hilink_sal_rsa.c hilink_thread_adapter.c hilink_mem_adapter.c hilink_sal_mpi.c hilink_sys_adapter.c hilink_tls_client.c hilink_softap_adapter.	网络适配实现 系统适配层网络 Socket 接口实现 外挂 MCU 升级适配实现 RSA 加解密适配层接口 mbedTLS 实现 线程适配层接口 cmsis2 实现源文件 系统适配层内存接口实现源文件 多精度整数 系统适配层接口实现 TLS 客户端常用操作，包括会话的创建、销毁等 SoftAP 适配实现 消息摘要算法适配层接口 mbedTLS 实现

目录	文件名	说明
	c hilink_sal_md.c hilink_kv_adapter.c hilink_time_adapter.c hilink_sal_kdf.c hilink_sal_drbg.c hilink_sal_aes.c hilink_sal_base64.c hilink_str_adapter.c hilink_stdio_adapter.c hilink_open_ota_adapter.c	鸿蒙智联配置保存适配源文件 时间适配层接口实现 密钥派生算法适配层接口 mbedTLS 实现 安全随机数适配层接口 mbedTLS 实现 AES 加解密适配层接口 mbedTLS 实现 base64 编码解码适配层接口 mbedTLS 实现 系统适配层字符串接口实现 系统适配层标准输出接口实现 OTA 适配实现

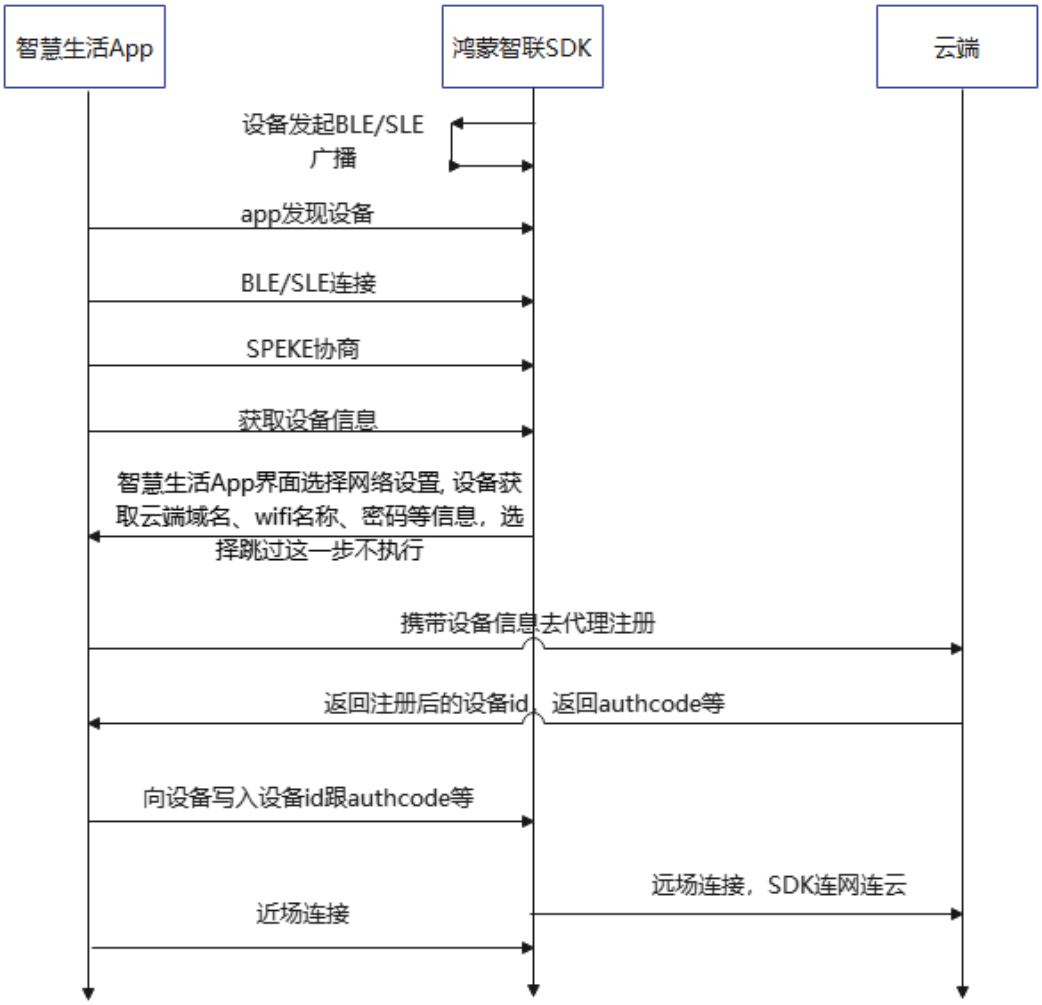
- BLE/SLE 开发包文件说明：

目录	文件名	说明
lib/debug	libhilinkbtsdk.a	Debug 版本用于设备调测时集成
lib/release	libhilinkbtsdk.a	Release 版本用于商用设备发布时集成
demo	hilink_demo.c	BLE SDK 提供 demo 示例代码
include	hilink_bt_function.h hilink_bt_api.h hilink_sle_api.h ble_cfg_net_api.h hilink_bt_custom.h hilink_bt_netcfg_api.h	BLE SDK 功能函数头文件 BLE SDK API 头文件 SLE SDK 接口头文件 BLE 辅助配网 SDK API 头文件 鸿蒙智联 SDK 定制化接口头文件 BLE SDK 配网 API 头文件

4.1.2 三种配网方案

1）双联双控配网

双联双控配网方案采用 BLE/SLE 代理激活方案，支持 Wi-Fi、BLE（SLE）两种连接方式。



配网过程：

图4-1 网络设置界面

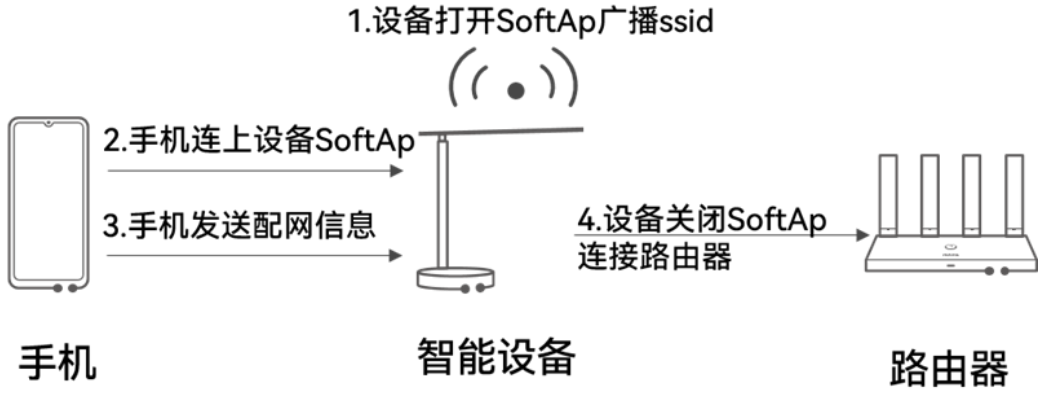
图4-2 用户自行更新配网信息界面

- 设备待配网时发起 BLE 广播+SLE 广播，智慧生活 App 通过扫描或者蓝牙靠近发现设备；
- 智慧生活 App 通过 BLE 或 SLE 连接设备，发送设备信息，按照蓝牙代理激活的方式由手机代理注册到云端，注册完成后将 Authcode 等信息下发到设备保存；
- 用户在网络设置时选择跳过，代理注册完成后，BLE/SLE 连接，设备管理页面进行近场连接，可以进行蓝牙本地控；
- 用户在设网络设置时配置 Wi-Fi 和密码，选择下一步，则完成代理激活后，鸿蒙智联 SDK 根据智慧生活 App 下发的网络配置进行连网连云操作，设备云在线，设备管理页面走远场控制；
- 用户未进行网络设置或连接设备云失败或云端离线情况下，设备管理页面走近场控制；
- 设备管理页面给设备预留下发 Wi-Fi 信息接口，设备网络信息更新时可以自行更新配网信息（图 2）；
- 双联双控设备分享后，被分享设备蓝牙或星闪无法连接，不能进行近场控制；

说明

此类配置针对可移动的非家居类产品设计（如：台灯），建议家居类产品使用蓝牙辅助配网或极速常规配网。

2）SoftAp Wi-Fi 配网



配网过程：

- SoftAp Wi-Fi 配网模式下，设备会打开热点广播特殊的 SSID；
- 点击智慧生活 App 添加设备按钮会扫描到特殊的 SSID 并通知用户发现新设备；
- 手机连接设备 Open 状态的 SoftAp 热点；
- 配网数据传递完成后，智慧生活 App 主动断开 SoftAp 热点连接，切换到配网前连接的 Wi-Fi 信号；
- 设备关闭 AP 热点，连接设备云端注册连云。

3） BLE 蓝牙辅助配网

蓝牙辅助配网下设备将会发起 BLE+SLE 广播，用户可以通过鸿蒙智联 Svc 弹窗或者智慧生活 App 扫描发现设备。

配网过程：

- 设备待配网时发起 BLE 广播+SLE 广播，App 通过扫描或者蓝牙靠近发现设备；
- 智慧生活 App 通过 BLE 或 SLE 连接设备蓝牙信号，发送配网信息，实现完成智慧生活 App 和设备进行配网数据交互；
- 配网数据传递完成后，手机侧主动断开蓝牙或星闪配对连接；
- 设备侧注册连云。

4.2 开发设备功能

4.2.1 配置 hilink_device.c 中产品信息

1. wifi 配置产品基本信息，用于设备配网和设备注册。

```
/* 设备产品 ID */
static const char *PRODUCT_ID = "9ABC"; // 产品 ID，必须和产品真实信息一致
/* 设备产品子型号 ID */
static const char *SUB_PRODUCT_ID = ""; // 产品子型号，无需填写
/* 设备类型 ID */
static const char *DEVICE_TYPE_ID = "xxx";// 产品类型 ID，必须和产品真实信息一致
/* 设备类型英文名称 */
static const char *DEVICE_TYPE_NAME = "xxxxx"; // 品类英文名，与 Device Partner 平台的“集成开发”页面中“无线网络名称（SSID）”的品类英文名保持一致
/* 设备制造商 ID */
static const char *MANUAFACTURER_ID = "xxx"; // 厂商 ID，必须和产品真实信息一致
/* 设备制造商英文名称 */
static const char *MANUAFACTURER_NAME = "XXXX";  // 品牌名，与 Device Partner 平台的“集成开发”页面中“无线网络名称（SSID）”的品牌名保持一致
/* 设备型号 */
static const char *PRODUCT_MODEL = "test123456"; // 产品型号，必须和产品真实信息一致
/* 设备 SN */
static const char *PRODUCT_SN = ""; // 设备 SN 号，建议与 GetSerial()接口返回指保持一致
/* 设备固件版本号 */
static const char *FIRMWARE_VER = "1.0.0"; // 模组固件版本号
/* 设备硬件版本号 */
static const char *HARDWARE_VER = "1.0.0"; // 模组硬件版本号。对应智慧生活 App 中设备版本信息显示的硬件版本。需要与产品硬件版本号（OHOS_HARDWARE_MODEL）保持一致
/* 设备软件版本号 */
static const char *SOFTWARE_VER = "1.0.0"; // 对应智慧生活 App 中设备版本信息显示的 SDK 版本，即 HiLink SDK 版本。例如 12.0.0.303（默认值即可，无需修改，HiLink SDK 会自动替换该版本号）
```

2. 蓝牙设备基本信息适配

```
/*
 * 设备基本信息根据设备实际情况填写
 * 与鸿蒙智联 sdk 相同定义，双模组模式只需一份，已提供给第三方厂家，暂不按编程规范整改
 */
#define DEVICE_SN "12345678"
#define PRODUCT_ID "2XXX"
#define DEVICE_MODEL "model-X"
#define DEVICE_TYPE "000"
#define MANUAFACTURER "000"
#define DEVICE_MAC "AABBCCDDEEFF"
#define DEVICE_HIVERSION "1.0.0"
#define DEVICE_PROT_TYPE 12
#define DEVICE_TYPE_NAME "other"
#define MANUAFACTURER_NAME "HW"

/* 蓝牙 sdk 单独使用的定义 */
#define SUB_PRODUCT_ID "00"
#define ADV_TX_POWER 0xF8
#define BLE_ADV_TIME 60
/* 厂商自定义蓝牙广播，设备型号信息 */
```



```
#define BT_CUSTOM_INFO "12345678"
#define DEVICE_MANU_ID "017"
static HILINK_BT_DevInfo g_btDevInfo;
/*
 * 功能：获取设备 sn 号
 * 参数[in]： len sn 的最大长度， 39 字节
 * 参数[out]： sn 设备 sn
 * 注意： sn 指针的字符串长度为 0 时将使用设备 mac 地址作为 sn
 */
void HILINK_GetDeviceSn(unsigned int len, char *sn)
{
    if (sn == NULL) {
        return;
    }
    const char *ptr = DEVICE_SN;
    int tmp = MIN_LEN(len, sizeof(DEVICE_SN));
    for (int i = 0; i < tmp; i++) {
        sn[i] = ptr[i];
    }
    return;
}
/*
 * 获取设备的子型号，长度固定两个字节
 * subProdId 为保存子型号的缓冲区，len 为缓冲区的长度
 * 如果产品定义有子型号，则填入两字节子型号，并以'\0'结束，返回 0
 * 没有定义子型号，则返回-1
 * 该接口需设备开发者实现
 */
int HILINK_GetSubProdId(char *subProdId, int len)
{
    if (subProdId == NULL) {
        return -1;
    }
    const char *ptr = SUB_PRODUCT_ID;
    int tmp = MIN_LEN((unsigned int)len, sizeof(SUB_PRODUCT_ID));
    for (int i = 0; i < tmp; i++) {
        subProdId[i] = ptr[i];
    }
    return 0;
}
/*
 * 获取设备表面的最强点信号发射功率强度，最强点位置的确定以及功率测试方
 * 法，参照 hilink 认证蓝牙靠近发现功率设置及测试方法指导文档，power 为出参，
 * 单位 dbm，下一次发送广播时生效
 */
int HILINK_BT_GetDevSurfacePower(char *power)
{
    if (power == NULL) {
        return -1;
    }
    *power = ADV_TX_POWER;
    return 0;
}
/* 获取蓝牙 SDK 设备相关信息 */
HILINK_BT_DevInfo *HILINK_BT_GetDevInfo(void)
```



```
{
    printf("HILINK_BT_GetDevInfo\n");
    g_btDevInfo.manuName = MANUAFACTURER;
    g_btDevInfo.devName = DEVICE_TYPE_NAME;
    g_btDevInfo.productId = PRODUCT_ID;
    g_btDevInfo.mac = DEVICE_MAC;
    g_btDevInfo.model = DEVICE_MODEL;
    g_btDevInfo.devType = DEVICE_TYPE;
    g_btDevInfo.hiv = DEVICE_HIVERSION;
    g_btDevInfo.protType = DEVICE_PROT_TYPE;
    g_btDevInfo.sn = DEVICE_SN;
    return &g_btDevInfo;
}

/*
 * 若厂商发送广播类型为 BLE_ADV_CUSTOM 时才需适配此接口
 * 获取厂商定制信息，由厂家实现
 * 返回 0 表示获取成功，返回其他表示获取失败
 */
int HILINK_GetCustomInfo(char *customInfo, unsigned int len)
{
    if (customInfo == NULL || len == 0) {
        return -1;
    }
    const char *ptr = BT_CUSTOM_INFO;
    int tmp = MIN_LEN(len, strlen(BT_CUSTOM_INFO));
    for (int i = 0; i < tmp; i++) {
        customInfo[i] = ptr[i];
    }
    return 0;
}
/*
 * 若厂商发送广播类型为 BLE_ADV_CUSTOM 时才需适配此接口
 * 获取厂家 ID，由厂家实现
 * 返回 0 表示获取成功，返回其他表示获取失败
 */
int HILINK_GetManuId(char *manuId, unsigned int len)
{
    if (manuId == NULL || len == 0) {
        return -1;
    }
    const char *ptr = DEVICE_MANU_ID;
    int tmp = MIN_LEN(len, strlen(DEVICE_MANU_ID));
    for (int i = 0; i < tmp; i++) {
        manuId[i] = ptr[i];
    }
    return 0;
}
/*
 * 获取蓝牙 mac 地址，由厂家实现
 * 返回 0 表示获取成功，返回其他表示获取失败
 */
int HILINK_BT_GetMacAddr(const char *mac, unsigned int len)
{
    (void)mac;
```



```
    (void)len;
    return 0;
}
/* 填写固件、软件和硬件版本号，APP 显示版本号以及 OTA 版本检查与此相关 */
int getDeviceVersion(char* *firmwareVer, char* *softwareVer, char* *hardwareVer)
{
    *firmwareVer = "1.0.0";
    *softwareVer = "1.1.0";
    *hardwareVer = "1.1.1";
    return 0;
}
```

4.2.2 配置 hilink_device.c 中的服务信息

为了确保设备控制功能的正常使用，需要将 Profile 中定义的设备功能，配置在 hilink_device.c 中。Profile 中默认添加的功能（例如 ota、netinfo 等），无需在 hilink_device.c 文件中配置。

须知

请确保工程代码实现与 DP 平台物模型定义中的信息配置一致，否则会导致设备信息校验失败，出现设备反复上线/下线的现象。

步骤 1 获取产品 Profile 文件。

1. 在 Device Partner 平台的“产品开发”页面，选择对应产品。
2. 在“产品定义 > 物模型定义”页面，单击右侧的“下载 Profile”。

步骤 2 在 hilink_device.c 文件中配置设备功能。

以门锁为例介绍如何配置设备功能，假定产品 profile 信息如表 4-1 所示，则对应的工程代码示例如下：

表4-1 产品 profile 信息（节选）

服务 sid	服务(中文)	服务类型 ServiceType
lockState	门在线/离线	state
lockMode	防护模式状态	mode

```
/* 服务信息定义 */
static const HILINK_SvcInfo SVC_INFO[] =
{
    { "state", "lockState"},
    { "mode", "lockMode"}
};
```

须知

定义服务信息时的结构顺序必须遵循：先服务类型 **ServiceType**、后服务 **sid** 的顺序。否则，会导致功能无法生效。

关于结构体 HILINK_SvcInfo 的定义，可以在“hilink_device.h”中查看。

```
typedef struct {
    char svcType[32];  /* 服务类型，长度范围(0, 32] */
    char svcId[64];    /* 服务 ID，长度范围(0, 64] */
} HILINK_SvcInfo;
```

----结束

4.2.3 设备发现

4.2.3.1 蓝牙设备发现

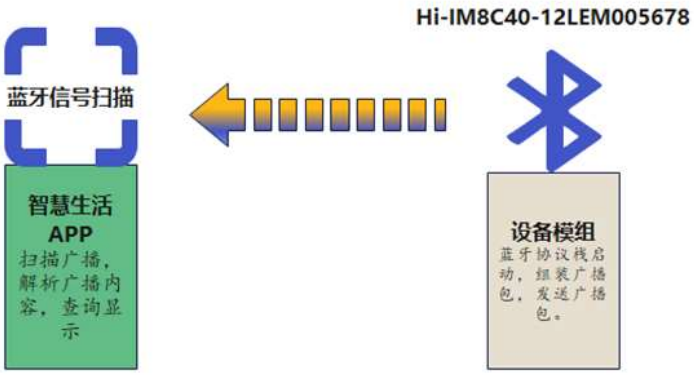
蓝牙设备信号名称：伙伴根据产品品类配置适配，鸿蒙智联 SDK 内部组装，传递给模组蓝牙模块，启动蓝牙广播信号，发送蓝牙广播发现数据包。

蓝牙广播格式："HI"（或"Hi"）+ ‘-’ + 'MANUAFACTURER_NAME' + 'DEVICE_TYPE_NAME（MANUAFACTURER_NAME + DEVICE_TYPE_NAME 取 10 字节）'+ ‘-’ + “1” + 'PRODUCT_ID' + 'SUB_PRODUCT_ID '+ 'PRODUCT_SN'（获取设备 SN 序列号后四位 ）。

示例名称：

```
设备已注册："HI-Huaweiindu-12LK2000000"
设备未注册："Hi-Huaweiindu-12LK2000000"
/* 设备产品 ID */
static const char *PRODUCT_ID = "2LK2";
/* 设备产品子型号 ID */
static const char *SUB_PRODUCT_ID = "";
设备产品子型号 ID 为空，拼装默认“00”
/* 设备类型英文名称 */
static const char *DEVICE_TYPE_NAME = "induction cooker";
/* 设备制造商英文名称 */
static const char *MANUAFACTURER_NAME = "Huawei";
/* 设备 SN */
static const char *PRODUCT_SN = "000606000000";
```

鸿蒙智联 SDK 调用模组 BLE 模块接口，启动固定名称的蓝牙广播信号，智慧生活 App 通过扫描蓝牙广播信号报文，发现蓝牙广播设备。设备发现方式包括 NFC 碰一碰和蓝牙靠近发现两种。伙伴需要根据产品定义时选择的极简交互方式不同，配置不同的信息。



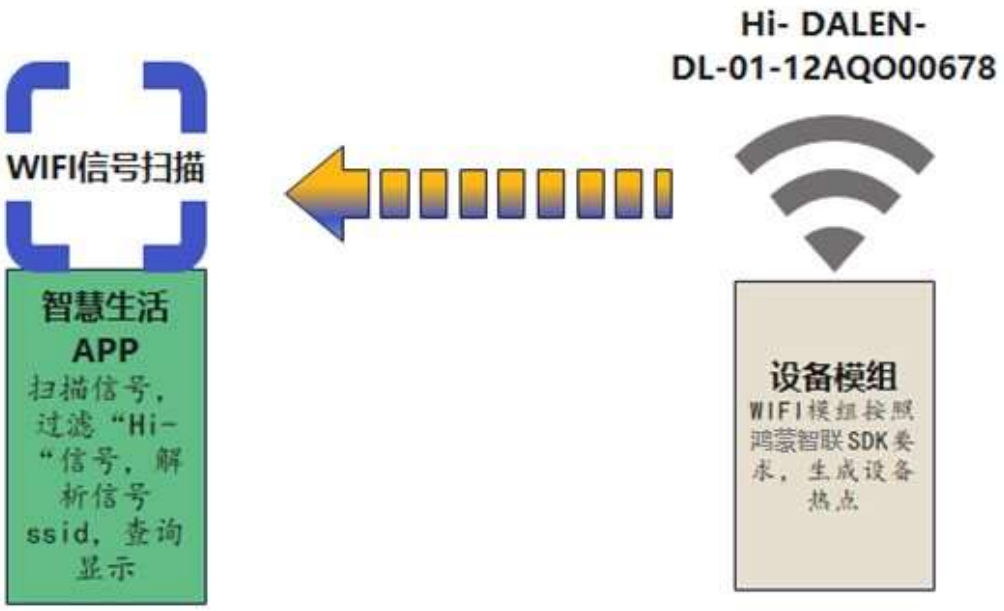
4.2.3.2 Wi-Fi 设备发现

设备 AP 热点名称，由伙伴根据产品品类配置适配，鸿蒙智联 SDK 内部组装，传递给模组 WIFI 模块，启动热点。

SoftAp 热点名称组装规则：snprintf_s(ssidOut, HI_D_SOFTAP_SSID_LEN, strLen, "%s%s%c%s%c%c%s%s%s",HI_D_SOFTAP_SSID_PREFIX, manuName, HI_D_SOFTAP_DASH, devName, HI_D_SOFTAP_DASH,HI_D_SOFTAP_VER_CHAR, deviceId, GetDeviceSubProdId(), deviceSn)

- 拼装后的 SSID 长度为固定 32 位，不足 32 位后面全部补英文空格" "
- HI_D_SOFTAP_SSID_PREFIX：Hi- 。
- manuName 和 devName 各取前 5 位。
- HI_D_SOFTAP_VER_CHAR：默认为 1，当项目使用 HiChain3.0 时为 2。
- deviceId: 项目品类 ProdId。
- GetDeviceSubProdId()：子设备 ProdId，默认“00”。
- deviceSn：设备 SN 后三位。

鸿蒙智联 SDK 调用模组 WIFI 模块接口，启动固定 SSID 规则的 SoftAp 热点信号，智慧生活 App 通过扫描过滤“Hi- xxxx”热点信号，发现 SoftAp 热点设备。



4.2.4 实现设备控制功能

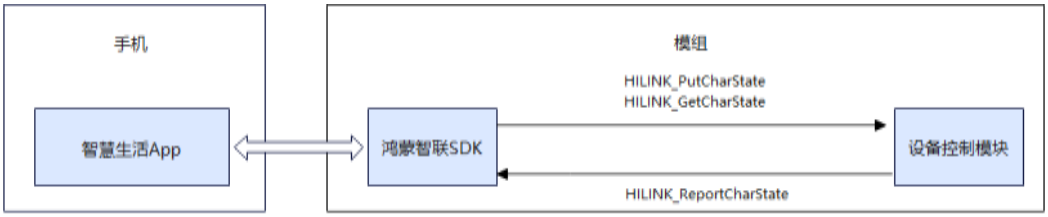
1. 远程控制

需要开发者实现 hilink_device.c 中的 HILINK_PutCharState 和 HILINK_GetCharState 两个函数，并结合鸿蒙智联 SDK 提供的接口 HILINK_ReportCharState 实现开发设备控制和状态上报功能。

表4-2 设备控制相关函数

函数名称	是否需要开发者实现	说明
HILINK_PutCharState	是	云端下发控制指令后会通过鸿蒙智联 SDK 调用此函数，伙伴需要再对服务进行识别、分发和处理。
HILINK_GetCharState	是	云端通过鸿蒙智联 SDK 获取设备状态信息或者鸿蒙智联 SDK 主动调用接口获取设备状态信息。
HILINK_ReportCharState	否	鸿蒙智联 SDK 报文上报接口——用于上报设备状态信息（根据设备功能按需调用）。

图4-3 手机和模组交互关系



步骤 1 根据 profile 进行产品定义、设备控制和状态查询功能。

- profile 定义设备支持的服务以及服务支持的操作。如：控制、查询、上报等。
- 设备开发必须按照 profile 的定义分别实现对应的功能。

以开关控制（handle_put_switch）和查询（handle_get_switch）功能为例，实现如下：

```
// 设备状态定义
typedef struct{
    unsigned int switch_on;
    unsigned int faltDetection_code;
    unsigned int faltDetection_status;
} t_device_info;

// 分配一个对象记录设备状态
static t_device_info g_device_info = {0};

// 处理从 HILINK_PutCharState 传递过来的信息
int handle_put_switch(const char* svc_id, const char* payload, unsigned int len)
{
    cJSON* pJson = cJSON_Parse(payload);
    if (pJson == NULL){
        printf("JSON parse failed in PUT cmd: ID-%s \r\n", svc_id);
        return INVALID_PACKET;
    }
    cJSON* item = cJSON_GetObjectItem(pJson, "on");
    if (item != NULL) {
        g_device_info.switch_on = item->valueint;
    }
    if (pJson != NULL) {
        cJSON_Delete(pJson);
    }
    printf("handle func:%s, sid:%s \r\n", __FUNCTION__, svc_id);
    return M2M_NO_ERROR;
}

// 处理从 HILINK_GetCharState 传递过来的信息
int handle_get_switch(const char* svc_id, const char* in, unsigned int in_len,
char** out, unsigned int* out_len)
{
    bool on = g_device_info.switch_on;
    *out_len = 20;
    *out = (char*)hilink_malloc(*out_len);
    if (NULL == *out){
        printf("malloc failed in GET cmd: ser %s in GET cmd", svc_id);
        return INVALID_PACKET;
```

```
    }
    *out_len = hilink_sprintf_s(*out, *out_len, "{\"on\":%d}", on);
    printf("hilink_device_ctr.c :%d %s svcId:%s, out:%s\r\n", __LINE__,__FUNCTION__ ,
svc_id, *out);
    return M2M_NO_ERROR;
}
```

步骤 2 注册服务处理信息。

设备开发时，需要根据“HILINK_PutCharState”函数和“HILINK_GetCharState”函数下发的服务 ID，分发指令信息到不同的函数处理。

示例代码如下：

```
// 服务处理函数定义
typedef int (*handle_put_func)(const char* svc_id, const char* payload, unsigned
int len);
typedef int (*handle_get_func)(const char* svc_id, const char* in, unsigned int
in_len, char** out, unsigned int* out_len);
// 服务注册信息定义
typedef struct{
    // service id
    char* sid;
    // HILINK_PutCharState cmd function
    handle_put_func putFunc;
    // handle HILINK_GetCharState cmd function
    handle_get_func getFunc;
} HANDLE_SVC_INFO;

//不支持 HILINK_PutCharState 时，默认实现
int not_support_put(const char* svc_id, const char* payload, unsigned int len)
{
    printf("sid:%s NOT SUPPORT PUT function \r\n", svc_id);
    return 0;
}
// 服务处理信息注册
HANDLE_SVC_INFO g_device_profile[] = {
    {"switch", handle_put_switch, handle_get_switch},
    // 故障不支持 HILINK_PutCharState，配置 not_support_put
    {"faultDetection", not_support_put, handle_get_faultDetection},
};
// 服务总数量
int g_device_profile_count = sizeof(g_device_profile) / sizeof(HANDLE_SVC_INFO);
```

步骤 3 分发服务。

增加服务分发处理函数“handle_put_cmd”、“handle_get_cmd”以及“fast_report”。

- “handle_put_cmd”和“handle_get_cmd”分别用于分发“HILINK_PutCharState”和“HILINK_GetCharState”传递的指令。
- “fast_report”用于快速上报设备状态信息。

示例代码如下：

```
// 辅助函数，用于查找服务注册信息
static HANDLE_SVC_INFO* find_handle(const char* svc_id)
{
```



```
    for(int i = 0; i < g_device_profile_count; i++) {
        HANDLE_SVC_INFO handle = g_device_profile[i];
        if(strcmp(handle.sid, svc_id) == 0) {
            return &g_device_profile[i];
        }
    }
    return NULL;
}
// 分发设备控制指令
int handle_put_cmd(const char* svc_id, const char* payload, unsigned intlen)
{
    HANDLE_SVC_INFO* handle = find_handle(svc_id);
    if(handle == NULL) {
        printf("no service to handle put cmd : %s \r\n", svc_id);
        return INVALID_PACKET;
    }
    handle_put_func function = handle->putFunc;
    if(function == NULL) {
        printf("put function is null for %s \r\n", svc_id);
        return INVALID_PACKET;
    }
    return function(svc_id, payload, len);
}
// 分发服务查询直连
int handle_get_cmd(const char* svc_id, const char* in, unsigned int in_len, char**
out, unsigned int* out_len)
{
    HANDLE_SVC_INFO* handle = find_handle(svc_id);
    if(handle == NULL) {
        printf("no service to handle get cmd : %s \r\n", svc_id);
        return INVALID_PACKET;
    }
    handle_get_func function = handle->getFunc;
    if(function == NULL) {
        printf("get function is null for %s \r\n", svc_id);
        return INVALID_PACKET;
    }
    return function(svc_id, in, in_len, out,out_len);
}
// 快速上报函数，用于上报服务状态信息
int fast_report(const char* svc_id, int task_id)
{
    const char* payload = NULL;
    int len;
    int err = handle_get_cmd(svc_id, NULL, 0, &payload, &len);
    if(err != M2M_NO_ERROR) {
        printf("get msg from %s failed \r\n", svc_id);
        return err;
    }
    err = HILINK_ReportCharState(svc_id, payload, len, task_id);
    printf("report %s result is %d \r\n", svc_id, err);
    return err;
}
```



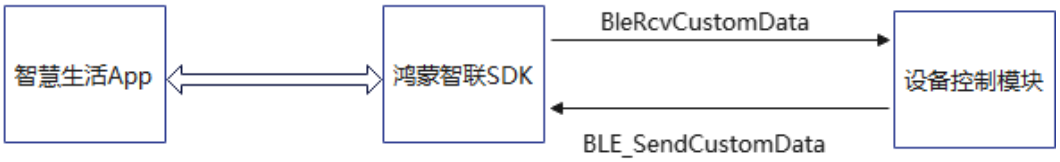
```
// ---------------------------------- //
// 以下两个函数在 hilink_device.c 中      //
// ---------------------------------- //
int HILINK_PutCharState(const char* svc_id,
        const char* payload, unsigned int len){
    int err = M2M_NO_ERROR;
    if(svc_id == NULL) {
        hilink_error("empty service ID in PUT cmd");
        return M2M_SVC_RPT_CREATE_PAYLOAD_ERR;
    }
    if (payload == NULL) {
        hilink_error("empty payload in PUT cmd");
        return M2M_SVC_RPT_CREATE_PAYLOAD_ERR;
    }

    hilink_debug("start handle PUT cmd: ID-%s", svc_id);
    err = handle_put_cmd(svc_id, payload, len);
    hilink_debug("handle PUT cmd end: ID-%s, ret-%d", svc_id, err);
    return err;
}
int HILINK_GetCharState(const char* svc_id, const char* in,
        unsigned int in_len, char** out, unsigned int* out_len){
    int err = M2M_NO_ERROR;
    if(svc_id == NULL){
        hilink_error("empty service ID in GET cmd");
        return M2M_SVC_RPT_CREATE_PAYLOAD_ERR;
    }
    hilink_info("start process GET cmd: ID - %s", svc_id);
    err = handle_get_cmd(svc_id, in, in_len, out, out_len);
    hilink_debug("end process GET cmd: ID - %s, ret - %d", svc_id, err);
    return err;
}
```

----结束

2. 蓝牙/星闪本地控逻辑介绍：

- 在手机同设备蓝牙/星闪连接情况下，智慧生活 App 发送控制命令。
- 手机蓝牙/星闪透传控制命令报文，设备蓝牙接收并透传报文给蓝牙鸿蒙智联 SDK。
- 鸿蒙智联 SDK 解析控制命令，传递给设备业务模块，并原路返回控制命令响应报文给智慧生活 App。
- 设备业务模块控制完成后，调用鸿蒙智联 SDK 接口主动上报控制结果。通过设备蓝牙—手机蓝牙—智慧生活 App 页面，最终显示控制结。

蓝牙本地控接口适配：只适用于双联双控本地控场景。



1. 设备控制：

鸿蒙智联 SDK 提供了智慧生活 App 控制设备时控制命令下发的通道，开发者需要在设备侧挂载控制命令回调函数，接收控制指令并完成控制动作。设备侧接收自定义控制指令需要在启动鸿蒙智联 SDK 时，在入参 BLE_CfgNetCb 类型的结构体中挂载 rcvCustomDataCb 字段的回调函数，当手机侧下发控制指令时，鸿蒙智联 SDK 会通过此回调函数传递自定义控制指令给设备侧。

注意：

- 鸿蒙智联 SDK 提供设备控制通道，控制完成后回调函数的返回值会直接传递给智慧生活 App；
- 由开发者自定义设备物模型，制定每个控制指令数据；
- 一般数据以 json 格式传输，如{"sid":"switch","data":{"on":1}}，其中 sid 对应指令，data 为指令携带的数据；

  设备侧挂载回调函数 demo 如下：

```
/* 智慧生活 App 下发自定义指令时调用此函数，需处理自定义数据，返回 0 表示处理成功 */
static int BleRcvCustomData(unsigned char *buff, unsigned int len)
{
    /* 打印接收到的数据 */
    printf("custom data, len: %u, data: %s\r\n", len, buff);

    /* 处理控制命令 */
    /* code */
}

/* 挂到 BLE_CfgNetCb 下，在初始化时传入 */
static BLE_CfgNetCb g_bleCfgNetCb = {
    .getDevPinCodeCb = NULL,
    .getDeviceInfoCb = NULL,
    .setCfgNetInfoCb = NULL,
    .rcvCustomDataCb = BleRcvCustomData,
    .cfgNetProcessCb = NULL
};
```

2. 设备上报

当设备添加完毕后，如果设备需要主动向智慧生活 App 上报电量、开关状态等信息，可以通过接口 BLE_SendCustomData 进行信息上报，亦可以结合设备控制内容，对接收到的自定义指令进行异步响应。如果接口返回值为非零，表示上报数据失败。

注意：

- 当设备未被手机/网关连接时，无法进行数据上报；
- 数据上报时请注意当前连接会话是否已认证，避免数据泄露。接收设备状态方法见 BLE_CfgNetCb 结构体中 cfgNetProcessCb 字段，回调函数的入参枚举值对应设备各个状态；调用 BLE_SendCustomData 时，dataType 请使用 CUSTOM_SEC_DATA，鸿蒙智联 SDK 会对数据进行加密传输，避免数据泄露；

```
/* BLE 发送用户数据：用户数据发送，与接收回调函数配套使用 */
int BLE_SendCustomData(BLE_DataType dataType, const unsigned char *buff,
unsigned int len);
```


4.2.5 网络优化通用适配

本节介绍如何开启网络优化功能，开发者可选择开启网络优化的功能类型，通过调用HILINK_RegWiFiRecoveryCallback 函数把网络优化相关功能接口注册给鸿蒙智联 SDK 使用。网络优化功能主要包含智能心跳，多扫多连，AP 智选。其中多扫多连和 AP 智选涉及 WiFi 连接机制，需要平台适配对应接口。

表4-3 网络优化相关接口描述，在头文件 hilink_network_adapter.h 中，表 1 接口功能在 SDK 内部实现。

接口名称	是否需要开发者实现	说明
HILINK_ScanAP	否	根据参数扫描周围 AP 信息。
HILINK_GetAPScanResult	否	鸿蒙智联 SDK 调用 HILINK_ScanAP 后，调用此函数获取扫描周围 AP 信息的结果。
HILINK_ConnectWiFiByBssid	否	连接指定 bssid 与配网 ssid 同名的 Wi-Fi。可以通过 bssid 连接到指定的路由器。
HILINK_GetLastConnectResult	否	获取上一次连接 WiFi 失败原因。
HILINK_RestartWiFi	否	重新启动 WiFi 模块。该接口只重启 WiFi 的 STA，不重启设备。
HILINK_RegWiFiRecoveryCallback	否	伙伴实现网络优化相关功能函数，通过此接口注册给鸿蒙智联 SDK 使用。 前提条件：HILINK_Main 初始化之前调用。
HILINK_SetWiFiRecoveryTimesParam	否	设置 WiFi 自恢复时的扫描、连接相关参数。scanTimes 扫描次数，默认 3 次；connectTimes 连接次数，默认 3 次。 前提条件：HILINK_Main 初始化之前调用。
HILINK_SetHeartbeatLimit	否	当环境中路由器 WiFi 信号质量不好时，设置心跳超时离线导致重连 WiFi 的阀值。一段时间内如果心跳超时次数达到该阀值，则进行优化，阀值默认为 3。该接口在 HILINK_Main 初始化之前调用。

表4-4 网络优化功能接口结构体 WiFiRecoveryApi 描述，在头文件 hilink_network_adapter.h 中，以下接口提供给鸿蒙智联 SDK 调用，由开发者实现并通过 HILINK_RegWiFiRecoveryCallback 函数注册给鸿蒙智联 SDK。

函数名称	是否需要开发者实现	说明
unsigned int (*getWifiRecoveryType)(void);	是	获取网络优化功能类型。开发者通过该接口的返回值，选择开启网络优化的功能类型。 返回 0x00：表示关闭网络优化所有功能； 返回 0x01：表示开启路由器引导设备连接至较优 AP 功能； 返回 0x02：表示开启 SDK 连接 WiFi 逻辑，此时需要设备配合关闭本身 WiFi 重连功能； 返回(0x01 | 0x02）：表示功能全开。 注：使用支持网路优化的 SDK 版本，要求返回 (0x01 | 0x02)，开启所有功能。 该接口由开发者自己实现，在初始化 WiFiRecoveryApi 时使用。
int (*scanAP)(const HILINK_APScanParam *param);	可选	设备配网阶段或断网重连阶段，鸿蒙智联 SDK 调用此接口根据参数扫描周围 AP 信息。 该接口开发者可以自己实现，也可使用鸿蒙智联 SDK 内部提供。如果使用鸿蒙智联 SDK 内部提供，在初始化 WiFiRecoveryApi 时，可直接使用表 1 中的 HILINK_ScanAP 函数。
int (*getAPScanResult)(HILINK_APList *scanList);	可选	scanAP 接口扫描完成后，鸿蒙智联 SDK 通过此接口获取目标 AP 扫描结果。如果环境中有多个同名目标 AP 会全部返回。 该接口开发者可以自己实现，也可使用鸿蒙智联 SDK 内部提供。如果使用鸿蒙智联 SDK 内部提供，在初始化 WiFiRecoveryApi 时，可直接使用表 1 中的 HILINK_GetAPScanResult 函数。
int (*connectWiFiByBssid)(int securityType, const unsigned char *bssid, unsigned int bssidLen);	可选	鸿蒙智联 SDK 在扫描结果中找到信号质量最优的 AP，调用此接口连接到最优 AP。 该接口开发者可以自己实现，也可使用鸿蒙智联 SDK 内部提供。如果使用鸿蒙智联 SDK 内部提供，在初始化 WiFiRecoveryApi 时，可直接使用表 1 中的 HILINK_ConnectWiFiByBssid 函数。
int (*lastConnResult)(int *result);	可选	WiFi 连接过程中如果出现异常，鸿蒙智联 SDK 通过此接口获取连接失败错误码。 该接口开发者可以自己实现，也可使用鸿蒙智联

函数名称	是否需要开发者实现	说明
		SDK 内部提供。如果使用鸿蒙智联 SDK 内部提供，在初始化 WiFiRecoveryApi 时，可直接使用表 1 中的 HILINK_GetLastConnectResult 函数。
int (*restartWiFi)(void);	可选	设备离线一段时间无法恢复，鸿蒙智联 SDK 调用此接口重启 WiFi。该接口只重启 WiFi 的 STA，不重启设备。 该接口开发者可以自己实现，也可使用鸿蒙智联 SDK 内部提供。如果使用鸿蒙智联 SDK 内部提供，在初始化 WiFiRecoveryApi 时，可直接使用表 1 中的 HILINK_RestartWiFi 函数。

表4-5 实现模组重启前的设备操作，在头文件 hilink_device.c 中。

函数名称	是否需要开发者实现	说明
HILINK_ProcessBeforeRestart	是	开启网络优化功能后，设备长时间离线无法恢复，鸿蒙智联 SDK 通过该接口参数 flag=2 的状态通知设备要重启。开发者需实现 flag=2 时模组重启前的操作(如:保存系统状态等)。

操作步骤如下：

步骤 1 设备长时间离线重启通知处理。

需要开发者实现 hilink_device.c 中的 HILINK_ProcessBeforeRestart 函数，对新增的 flag=2 条件分支，实现模组重启前的操作(如:保存系统状态等)。

代码如下：

```
/*
* 功能：实现模组重启前的设备操作
* 参数：flag 入参，触发重启的类型
*          0 表示 HiLink SDK 线程看门狗触发模组重启；
*          1 表示 APP 删除设备触发模组重启；
*          2 表示设备长时间离线无法恢复而重启；
* 返回值：0 表示处理成功，系统可以重启，使用硬重启；
*        1 表示处理成功，系统可以重启，如果通过 HILINK_SetSdkAttr()注册了软重启
(sdkAttr.rebootSoftware)，使用软重启；
```



```
*         负值表示处理失败，系统不能重启
* 注意：(1) 此函数由设备厂商实现；
*       (2) 若 APP 删除设备触发模组重启时，设备操作完务必返回 0，否则会导致删除设备异常；
*       (3) 设备长时间离线无法恢复而重启，应对用户无感，不可影响用户体验，否则不可以重启；
*/
int HILINK_ProcessBeforeRestart(int flag)
{
    /* HiLink SDK 线程看门狗超时触发模组重启 */
    if (flag == 0) {
        /* 实现模组重启前的操作(如:保存系统状态等) */
        return 0;
    }
    /* APP 删除设备触发模组重启 */
    if (flag == 1) {
        /* 实现模组重启前的操作(如:保存系统状态等) */
        return 1;
    }
   /* 设备长时间离线触发模组重启，尝试恢复网络 */
    if (flag == 2) {
       /* 实现模组重启钱的操作（如：保存系统状态等） */
        return 0;
    }
    return -1;
}
```

步骤 2 定义网路优化功能类型选择接口，使用支持网络优化功能的鸿蒙智联 SDK 版本，要求网络优化功能全部打开。

示例代码：

```
int GetWifiRecoveryType(void)
{
    /* (0x01 | 0x02)表示网络优化功能全开，返回 0 则关闭，具体定义详见 WiFiRecoveryApi 结构体 */
    return (0x01 | 0x02);
}
```

步骤 3 注册网络优化相关功能函数。

如果使用鸿蒙智联 SDK 内部提供的接口，除 getWifiRecoveryType 需要开发者实现外，scanAP、getAPScanResult、restartWiFi、connectWiFiByBssid、lastConnResult 直接使用 SDK 提供的相关接口初始化即可。

示例代码：

```
/* 网络优化开启 */
WiFiRecoveryApi recApi = {
    .getWifiRecoveryType = GetWifiRecoveryType,
    .scanAP = HILINK_ScanAP,
    .getAPScanResult = HILINK_GetAPScanResult,
    .restartWiFi = HILINK_RestartWiFi,
    .connectWiFiByBssid = HILINK_ConnectWiFiByBssid,
    .lastConnResult = HILINK_GetLastConnectResult,
};
ret = HILINK_RegWiFiRecoveryCallback((const WiFiRecoveryApi *)&recApi,
sizeof(WiFiRecoveryApi));
if (ret != 0) {
    printf("reg wifi recovery api failed\r\n");
```



```
}
/* 启动 Hilink SDK */
if (HILINK_Main() != 0) {
    printf("HILINK_Main start error");
}
```

步骤 4 开启路由引导设备连接指定 AP 功能（GetWifiRecoveryType 返回值包含 0x01）。

修改 wifi 单信道扫描时长为 210ms，信道停留时间修改作用于设备运行整个生命周期。

```
if(GetWifiRecoveryType() != 0) {
    /* 开启网络优化 */
    printf("wifi recovery: scan interval [210]\r\n");
    /* 设置单信道扫描时长 210ms */
    ...
}
```

步骤 5 开启 WiFi 多扫多连并连接至最优 AP 功能。（GetWifiRecoveryType 返回值包含 0x02）。

1. 关闭 WiFi 自动重连，WiFi 自动重连关闭作用于设备运行整个生命周期（注：在鸿蒙智联 SDK 运行前执行该代码）；

```
if((GetWifiRecoveryType() & 0x02) == 0x02) {
    /* 开启功能 2，关闭自动重连功能 */
    printf("wifi recovery: disable autoreconnect\r\n");
    /* 实现关闭自动重连 */
    ...
} else {
    /* 开启自动重连功能 */
    printf("wifi recovery: enable autoreconnect\r\n");
    /* 实现开启自动重连 */
    ...
}
```

2.修改 WiFi 单信道扫描时长为 210ms，信道停留时间修改作用于设备运行整个生命周期。

```
if(GetWifiRecoveryType() != 0) {
    /* 开启网络优化 */
    printf("wifi recovery: scan interval [210]\r\n");
    /* 设置单信道扫描时长 210ms */
    ...
}
```

3.增加 WiFi 连接结果状态码获取，详见 WiFiRecoveryApi 结构体的回调函数 lastConnResult。鸿蒙智联 SDK 通过鸿蒙 RegisterWifiEvent 接口注册的 OnWifiConnectionChanged 回调获取 lastConnResult 状态码。

⚠ 注意

因密码错误导致 WIFI 连接失败，错误码要求返回 14 或 15 或 19 任意一个。其它场景的错误码无约束，根据实际情况传递即可。

为保证 WiFi 连接可靠性，只有确实是密码错误导致的连接失败才可以返回该错误码。

----结束

离线优化不同芯片代码适配示例

6 附录

4.2.6 鸿蒙智联 SDK 快速启动

```
static void HILINK_BT_StateChangeHandler(HILINK_BT_SdkStatus event, const void
*param)
{
    (void)param;
    /* ble sdk 初始化完成后，发送广播让设备被手机发现 */
    if (event == HILINK_BT_SDK_STATUS_SVC_RUNNING) {
        /* 设置蓝牙广播格式，包括靠近发现、碰一碰等，下一次发送广播生效 */
        BLE_SetAdvType(BLE_ADV_NEARBY_V0);

        /* BLE 配网广播控制：参数代表广播时间，0:停止；0xFFFFFFFF:一直广播，其他：广播指定时间后
停止，单位秒 */
        (void)·(BLE_ADV_TIME);
    }
}
static BLE_ConfPara g_isBlePair = {
    .isBlePair = 1,
};
static BLE_InitPara g_bleInitParam = {
    .confPara = &g_isBlePair,
    /* advInfo 为空表示使用 ble sdk 默认广播参数及数据 */
    .advInfo  = NULL,
    .gattList = NULL,
};

static BLE_CfgNetCb g_bleCfgNetCb = {
   .rcvCustomDataCb = BleRcvCustomData,
    .cfgNetProcessCb = CfgNetProcess,
 };

int main(void)
{
    int ret = 0;
    /* 打开星闪功能总开关 */
    HILINK_EnableSle();

    /* 配置端云通道协议类型，根据配网方案设置不同的协议类型*/
    HILINK_SetProtType(DEVICE_PROT_TYPE);

    /* 设备按需设置，例如接入蓝牙网关时，设置广播类型标志及心跳间隔 */
    unsigned char mpp[] = { 0x02, 0x3c, 0x00 };
    ret = BLE_SetAdvNameMpp(mpp, sizeof(mpp));
    if (ret != 0) {
        HILINK_SAL_NOTICE("set adv name mpp failed\r\n");
        return -1;
```



```
    }

    /* 设置配网方式,双联双控选择配网模式为 HILINK_NETCONFIG_OTHER*/
    ret = HILINK_SetNetConfigMode(HILINK_NETCONFIG_OTHER);
    if (ret != 0) {
        printf("SetNetConfigMode failed\r\n");
        return -1;
    }
    /* 注册 SDK 状态接收函数，可在初始化完成后发送广播 */
    ret = HILINK_BT_SetSdkEventCallback(HILINK_BT_StateChangeHandler);
    if (ret != 0) {
        printf("set event callback failed\r\n");
        return -1;
    }
    /* 初始化 ble sdk */
    ret = BLE_CfgNetInit(&g_bleInitParam, &g_bleCfgNetCb);
    if (ret != 0) {
        printf("ble sdk init fail\r\n");
        return -1;
    }
    /* 修改任务属性 */
    HILINK_SdkAttr *sdkAttr = HILINK_GetSdkAttr();
    if (sdkAttr == NULL) {
        printf("sdkAttr is null");
        return -1;
    }
    sdkAttr->monitorTaskStackSize = 0x400;  /* 示例代码 推荐栈大小为 0x400 */
    sdkAttr->rebootHardware = HardReboot;
    sdkAttr->rebootSoftware = HardReboot;
    HILINK_SetSdkAttr(*sdkAttr);

    /* 启动 Hilink SDK */
    if (HILINK_Main() != 0) {
        printf("HILINK_Main start error");
    }
    return 0;
}
```

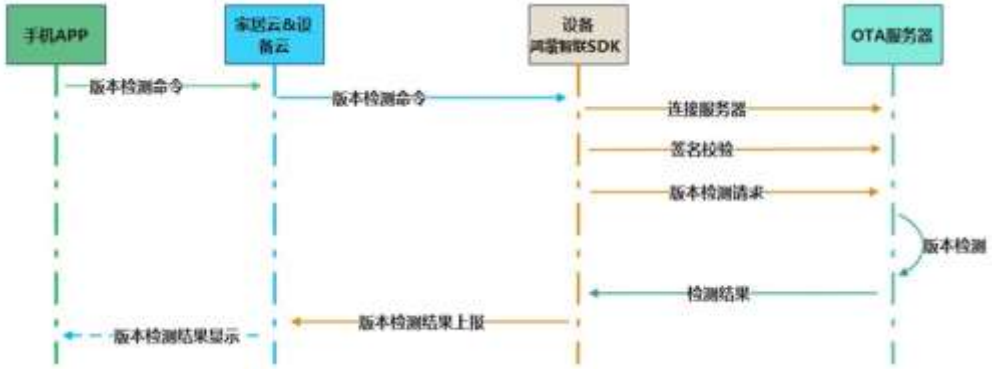
4.2.7 设备 OTA 升级

目前支持鸿蒙智联 SDK 的模组都具备支持华为 OTA 升级的能力，开发者可选择华为 OTA 云升级或者非华为 OTA 升级；

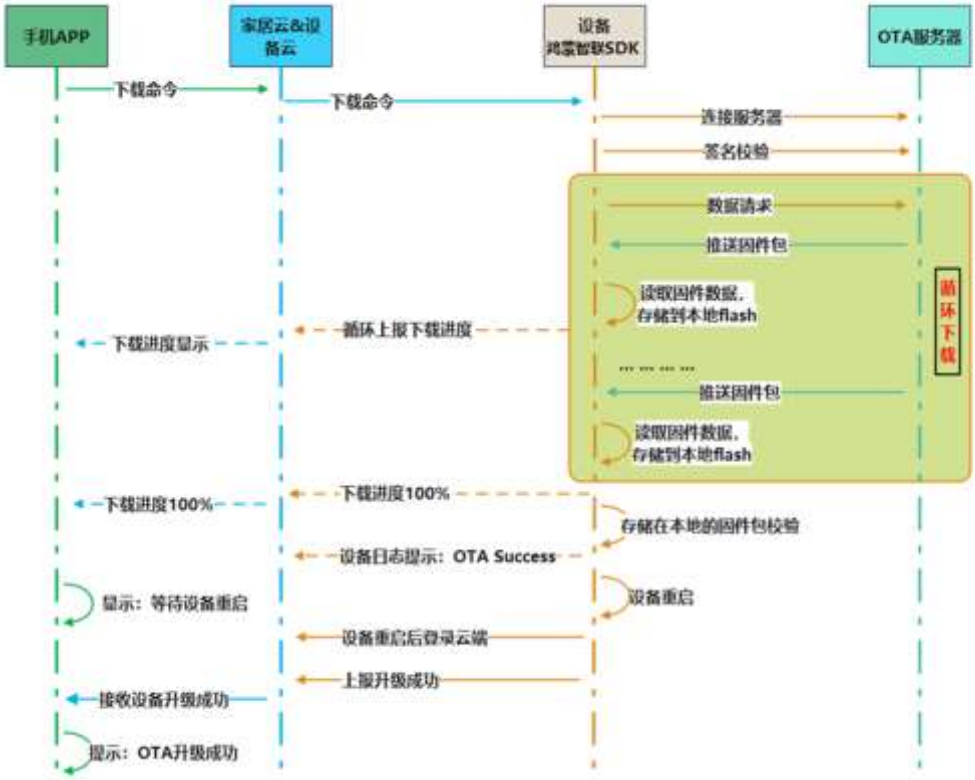
选择华为 OTA 升级，开发者适配完成后进行固件编译，编译会生成 ota 升级固件，可以将此固件升级包通过 Device Partner 平台部署。

- 升级包部署网址：https://devicepartner.huawei.com/console/ota#/　升级分为模组升级和 MCU 升级。
- 鸿蒙智联设备升级通过华为智慧生活 App 触发或者用户打开自动升级功能自动触发。

  OTA 检测流程如下图所示：



OTA 升级流程如下图所示：



同时，双联双控方案支持蓝牙 OTA 升级，智慧生活 App 访问 hota 服务器，通过蓝牙通道将升级包传递给 SDK 完成升级。

华为 OTA 升级开发者需适配 OTA 相关接口：hilink_open_ota_adapter.c, hilink_open_ota_mcu_adapter.c

4.2.8 三方库适配

Device SDK 依赖以下相关库进行 json 字符串处理以及加解密，相关版本可由产品自行选定，通过 adapter 适配层进行适配。

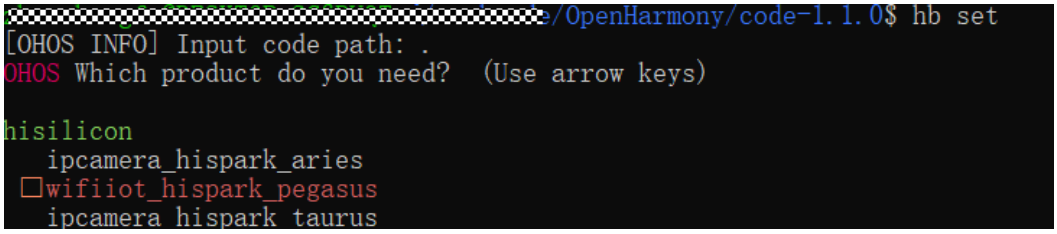
库名称	版本号	说明
mbedtls	建议使用 3.4.1 及以上	供嵌入式设备使用的一个 TLS 协议/加解密算法的轻量级实现库。

库名称	版本号	说明
cJSON	建议使用 1.7.17 及以上	基于 C 语言的 JSON 解析库，支持 JSON 的解析和构建。
SECUREC		securec 是一个华为内部开源的 C 语言安全库。 开源地址：https://codehub-y.huawei.com/hwsecurec_group/huawei_secure_c/files?ref=master。

4.3 编译固件

步骤 1 进入工程根目录，执行“hb set”、“.”，并选择需要构建的产品。

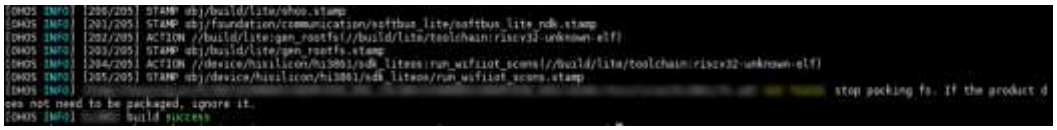
图4-4 构建设置示例



步骤 2 在工程根目录，使用“hb build xx”命令进行版本构建。

- 在执行 XTS 测试时，需要使用 debug 版本进行构建，即执行构建命令“hb build -f”。
- 在提交认证预约时，需要使用 release 版本进行构建，即执行构建命令“hb build -f -b release"。

返回“xxxx build success”，表明构建成功。



----结束

4.4 烧录固件

步骤 1 从模组商处获取串口驱动和烧录工具。

不同芯片使用的驱动和烧录工具均不同，建议联系模组商获取支撑。

步骤 2 按照模组商提供的指导文档安装驱动。

步骤 3　使用烧录工具烧写固件到模组上。

----结束

5 功能验证

5.1 测试配网和设备控制

5.1 测试配网和设备控制

5.1.1 配置调测环境

步骤 1 配置测试帐号。

----结束

- 方式一：申请测试权限。

须知

申请权限操作仅对当前帐号生效，不会对团队的其他帐号生效。申请权限的华为帐号，必须与手机调测使用的华为帐号保持一致。

1. 登录 Device Partner 平台，进入管理中心。
2. 选择“帐号管理 > 基本资料”，单击右上角的“申请测试权限”。
3. 单击“立刻申请测试权限”申请测试权限。

图5-1 申请测试权限

- 方式二：下载智慧生活 App Debug 版本。
  a. 登录华为智能硬件合作伙伴平台，单击右上角的“管理中心”。
  b. 进入“产品开发 > 集成开发”页面，下载智慧生活 App Debug 版本。通过手机浏览器扫描二维码，或者在手机浏览器中输入链接地址下载即可。

图5-2 智慧生活 App 下载方式

1. 配置智慧生活 App 测试环境。

   Android 版智慧生活 App：打开智慧生活 App 后，进入“我的 > 设置 > 关于 > 环境设置”，选择“认证沙箱”环境。

   IOS 版智慧生活 App：我的-设置-关于-切换服务器-选择的环境。

5.1.2 测试设备配网与设备控制功能

步骤 1 打开智慧生活 App，点击右上角“+”，选择“添加设备”，智慧生活 App 会扫描附近所有处于待配网状态的设备。

图5-3 智慧生活 App 首界面

步骤 2 选择需要配网的设备，点击“连接”开始配网。

步骤 3 配网成功后，设置设备位置信息（如卧室、阳台等）。

步骤 4 打开设备卡片，进入设备控制界面。

设备控制界面为交互设计环节部署的 H5 界面，展示了设备状态和功能控制服务等。

调测阶段，因为产品还没有提交认证，所以会有警告窗口，点击“继续”即可。

步骤 5 点击设备控制按钮，如开关等，设备侧会收到相关指令。

----结束

5.1.3 添加设备失败问题分析

本节对添加设备过程进行拆解和说明，帮助伙伴了解添加设备的主要过程，以达成快速对问题定位定界的目的。

从执行顺序上看，添加设备过程依次经历以下两个过程阶段。

表5-1 添加设备过程介绍

阶段	作用	开始标志	结束标志	成功日志	失败日志
配网阶段	设备连接到 Wi-Fi 热点，具备联网能力	wait STA join AP	connect success	-	-
注册阶段	注册设备信息，建立设备和帐号的关联关系	set dev status [2]	set dev status [4]	set dev status [4]	未打印“set dev status [4]”，或者打印“set dev status [6]”。

6 附录

6.1 3861 网络优化工程修改示例

6.1.1 工程更新

工程请联系海思 FAE 获取，否则编译报错，“hi_wifi_scan_strategy_stru”结构体和“hi_wifi_set_scan_strategy”接口未定义。

```
“hi_wifi_scan_strategy_stru”：扫描策略结构体；
“hi_wifi_set_scan_strategy”：设置扫描策略。
```

6.1.2 三方工程 Wi-Fi 参数修改 demo 示例

1. 关闭 Wifi 自动重连。

   Wifi 自动重连关闭作用于设备运行整个生命周期。

```
if((GetWifiRecoveryType() & 0x02) == 0x02) {
    /* 开启功能 2，关闭自动重连功能 */
    printf("wifi recovery: disable autoreconnect\r\n");
    /* 实现关闭自动重连 */
    ...
} else {
    /* 开启自动重连功能 */
    printf("wifi recovery: enable autoreconnect\r\n");
    /* 实现开启自动重连 */
    ...
}
```


2. 修改扫描信道停留时间。

   信道停留时间修改作用于设备运行整个生命周期。

```
if(GetWifiRecoveryType() != 0) {
    /* 开启网络优化 */
    printf("wifi recovery: scan interval [210]\r\n");
    /* 设置单信道扫描时长 210ms */
    ...
}
```

6.1.3 三方工程 Wi-Fi 参数修改示例

1. 关闭 Wifi 自动重连。

   接口 WifiErrorCode EnableWifi(void)，在 WiFi 使能时增加修改：

```
/* 定义 int 型变量，接收接口返回值 */
int hiRet;
/* 如果开启了网络优化功能 2 */
if ((GetWifiRecoveryType() & 0x02) == 0x02) {
    printf("wifi recovery: disable autoreconnect\r\n");
    /* 关闭 WiFi 自动重连 */
hiRet = hi_wifi_sta_set_reconnect_policy(WIFI_RECONN_POLICY_DISABLE,
WIFI_RECONN_POLICY_TIMEOUT, WIFI_RECONN_POLICY_PERIOD,
WIFI_RECONN_POLICY_MAX_TRY_COUNT);
} else {
    printf("wifi recovery: enable autoreconnect\r\n");
    /* 如果没有开启网络优化功能 2，则开启 WiFi 自动重连 */
hiRet = hi_wifi_sta_set_reconnect_policy(WIFI_RECONN_POLICY_ENABLE,
WIFI_RECONN_POLICY_TIMEOUT, WIFI_RECONN_POLICY_PERIOD,
WIFI_RECONN_POLICY_MAX_TRY_COUNT);
}
/* 接口返回异常处理：以下为复制的代码段，以实际为准 */
if (hiRet != HISI_OK) {
    printf("[wifi_service]:EnableWifi set reconn policy fail\n");
    if (UnlockWifiGlobalLock() != WIFI_SUCCESS) {
        return ERROR_WIFI_UNKNOWN;
    }
    return ERROR_WIFI_UNKNOWN;
}
```

2. 修改扫描信道停留时间。

   接口 WifiErrorCode EnableWifi(void)，在 Wifi 使能时增加修改：

```
#define WIFI_SCAN_CNT 7 /* wifi 扫描时长，取值范围 1~10，单位 30ms。7 表示 210ms */
/* 如果开启了网络优化 */
if (GetWifiRecoveryType() != 0) {
    printf("wifi recovery: scan interval [%d]\r\n", (30 * WIFI_SCAN_CNT));
    /* 设置 wifi 扫描策略，关闭 STA/AP 再启动 STA/AP 恢复默认值 2 */
    hi_wifi_scan_strategy_stru scanStrategy;
scanStrategy.scan_cnt = WIFI_SCAN_CNT;
/* 调用"设置扫描策略"接口修改 WiFi 扫描策略，修改单个信道停留时长为 210ms，保证环境中的 AP 尽可能扫全 */
    int ret = hi_wifi_set_scan_strategy(ifName, &scanStrategy);
    /* 接口返回异常处理：以下为复制的代码段，以实际为准 */
if (ret != HISI_OK) {
        printf("set wifi scan strategy fail [%d]\r\n", ret);
```

```
        return ERROR_WIFI_UNKNOWN;
    }
}
```

3. BUG 修复：WiFi 扫描结果中 rssi 的值被放大 100 倍。

   rssi 正常取值范围在(-100, 0)之间。WiFi 扫描结果中 rssi 返回值为正常值 100 倍，在接口 WifiErrorCode GetScanInfoList(WifiScanInfo* result, unsigned int* size)修改如下：

```
/* WiFi 信号指令 rssi 范围(-99, 0)dB，如果获取到的值被放大 100 倍需除 100 */
result[i].rssi = pstResults[i].rssi / 100;
```

6.2 AIW4211 网络优化工程修改实例

6.2.1 工程更新

请联系爱旗模组厂商获取支持网络优化工程，否则编译报错，
“ext_wifi_scan_strategy_stru”结构体和“aich_wifi_set_scan_strategy”接口未定义。

6.2.2 三方工程 Wi-Fi 参数修改 demo 示例

1. 关闭 Wifi 自动重连。

   Wifi 自动重连关闭作用于设备运行整个生命周期。

```
if((GetWifiRecoveryType() & 0x02) == 0x02) {
    /* 开启功能 2，关闭自动重连功能 */
    printf("wifi recovery: disable autoreconnect\r\n");
    /* 实现关闭自动重连 */
    ...
} else {
    /* 开启自动重连功能 */
    printf("wifi recovery: enable autoreconnect\r\n");
    /* 实现开启自动重连 */
    ...
}
```

2. 修改扫描信道停留时间。

   信道停留时间修改作用于设备运行整个生命周期。

```
if(GetWifiRecoveryType() != 0) {
    /* 开启网络优化 */
    printf("wifi recovery: scan interval [210]\r\n");
    /* 设置单信道扫描时长 210ms */
    ...
}
```

6.2.3 三方工程 Wi-Fi 参数修改实例示例

1. 关闭 Wifi 自动重连。

   接口 WifiErrorCode EnableWifi(void)，在 Wifi 使能是增加修改。

```
/* 定义 int 型变量，接收接口返回值 */
int ret;
```



```
/* 如果开启了网络优化功能 2 */
if ((GetWifiRecoveryType() & 0x02) == 0x02) {
    printf("wifi recovery: autoconnect disable\r\n");
    /* 关闭 WiFi 自动重连 */
ret = aich_wifi_sta_set_reconnect_policy(WIFI_RECONN_POLICY_DISABLE,
WIFI_RECONN_POLICY_TIMEOUT, WIFI_RECONN_POLICY_PERIOD,
WIFI_RECONN_POLICY_MAX_TRY_COUNT);
} else {
    printf("wifi recovery: autoconnect enable\r\n");
    /* 如果没有开启网络优化功能 2，则开启 WiFi 自动重连 */
ret = aich_wifi_sta_set_reconnect_policy(WIFI_RECONN_POLICY_ENABLE,
WIFI_RECONN_POLICY_TIMEOUT, WIFI_RECONN_POLICY_PERIOD,
WIFI_RECONN_POLICY_MAX_TRY_COUNT);
}
/* 接口返回异常处理：以下为复制的代码段，以实际为准 */
if (ret != SOC_OK) {
    printf("[wifi_service]:EnableWifi set reconn policy fail\n");
    if (UnlockWifiGlobalLock() != WIFI_SUCCESS) {
        return ERROR_WIFI_UNKNOWN;
    }
    return ERROR_WIFI_UNKNOWN;
}
```

2. 修改扫描信道停留时间。

   接口 WifiErrorCode EnableWifi(void)，在 Wifi 使能时增加修改。

```
#define WIFI_SCAN_CNT 7 /* wifi 扫描时长，取值范围 1~10，单位 30ms。7 表示 210ms */

/* 如果开启了网络优化 */
if (GetWifiRecoveryType() != 0) {
    printf("wifi recovery: scan interval time [%d]\r\n", (30 * WIFI_SCAN_CNT));
    ext_wifi_scan_strategy_stru scanStrategy;
    scanStrategy.scan_cnt = WIFI_SCAN_CNT;
    /* 调用“aich_wifi_set_scan_strategy”接口修改 WiFi 扫描策略，修改单个信道停留时长为
210ms，保证环境中的 AP 尽可能扫全 */
int ret = aich_wifi_set_scan_strategy(ifName, &scanStrategy);
    /* 接口返回异常处理：以下为复制的代码段，以实际为准 */
if (ret != SOC_OK) {
        printf("set wifi scan strategy fail [%d]\r\n", ret);
        return ERROR_WIFI_UNKNOWN;
    }
}
```

3. BUG 修复：Wifi 扫描结果中 rssi 的值被放大 100 倍。

   rssi 正常取值范围在(-100, 0)之间。WiFi 扫描结果中 rssi 返回值为正常值 100 倍，在接口 WifiErrorCode GetScanInfoList(WifiScanInfo* result, unsigned int* size)修改如下：

```
/* WiFi 信号指令 rssi 范围(-99, 0)dB，如果获取到的值被放大 100 倍需除 100 */
result[i].rssi = pstResults[i].rssi / 100;
```


6.3 8720 网络优化工程修改示例

6.3.1 三方工程 Wi-Fi 参数修改 demo 示例

1. 关闭 Wifi 自动重连。

   Wifi 自动重连关闭作用于设备运行整个生命周期。

```
if((GetWifiRecoveryType() & 0x02) == 0x02) {
    /* 开启功能 2，关闭自动重连功能 */
    printf("wifi recovery: disable autoreconnect\r\n");
    /* 实现关闭自动重连 */
    ...
} else {
    /* 开启自动重连功能 */
    printf("wifi recovery: enable autoreconnect\r\n");
    /* 实现开启自动重连 */
    ...
}
```

2. 修改扫描信道停留时间。

   信道停留时间修改作用于设备运行整个生命周期。

```
if(GetWifiRecoveryType() != 0) {
    /* 开启网络优化 */
    printf("wifi recovery: scan interval [210]\r\n");
    /* 设置单信道扫描时长 210ms */
    ...
}
```

6.3.2 三方工程 Wi-Fi 参数修改示例

1. 关闭 WiFi 自动重连

   接口 WiFiErrorCode ConnectToNoLock(int networkId)，在 WiFi 连接之前修改两处：

   - 修改在 WiFi 连接前：

```
int connect_result, max_retry = 1; /* max_retry 改之前为 5。减少重连次数，
ConnectToNoLock 是阻塞的，连接次数太多，影响其它业务流程 */
/* 注：8720 在 WiFi 连接前会先调用 wifi_set_autoreconnect(0);禁用重连模式，所以网络
优化功能开启后，保证不会打开 WiFi 重连模式即可 */
WIFI_CNT:/* 截取的代码片段，以实际为准 */
    /* 首次连接 WiFi 且网络优化功能 2 没有开启 */
    if((max_retry == 1) && ((GetWifiRecoveryType() & 0x02) != 0x02)) {
        printf("wifi recovery: enable autoreconnect\r\n");
        /* 启用无限重连模式，设置 2 */
wifi_set_autoreconnect(2);
    }
```

   - 修改在 Wifi 连接后：

```
   /* 网络优化功能 2 没有开启 */
if((GetWifiRecoveryType() & 0x02) != 0x02) {
        printf("wifi recovery: ebable autoreconnect\r\n");
        /* 启用无限重连模式，设置 2 */
wifi_set_autoreconnect(2);
}
/* 以下为截取的代码片段，以实际为准 */
```

```
    if(config->ipType == DHCP)
    {
        LwIP_DHCP(0, DHCP_START);
    }
```

2. 修改扫描信道停留时间。

   接口 WifiErrorCode AdvanceScanNoLock(WifiScanParams *params)，在 WiFi 扫描之前增加修改：

```
/* 网络优化功能开启 */
if (GetWifiRecoveryType() != 0) {
    printf("wifi recovery: scan interval [210]\r\n");
    /* 网络优化功能开启，修改 WiFi 单信道停留时长为 210ms，保证环境中的 AP 尽可能扫全 */
wifi_set_partial_scan_chan_interval(210);
}
```

3. BUG 修复：Wifi 断开状态码返回。

   Wifi 连接失败错误码通过回调接口“void (*OnWifiConnectionChanged)(int state, WifiLinkedInfo* info);”传递给 SDK，WiFi 连接错误码在第二个参数“WifiLinkedInfo* info”中携带。“WifiLinkedInfo* info”通过接口“WifiErrorCode GetLinkedInfo(WifiLinkedInfo* result)”获取时，对 WiFi 连接失败场景，没有传递连接失败错误码。下述修改是 WiFi 连接失败场景 WiFi 错误码获取方法。

   接口 WifiErrorCode GetLinkedInfoNoLock(WifiLinkedInfo* result)，增加 WiFi 断开时错误码获取如下：

```
/* WiFi 连接成功 */
if ((wifi_is_connected_to_ap() == RTW_SUCCESS) && (wifi_get_setting(WLAN0_NAME,
&setting) == RTW_SUCCESS))
{
    /* WiFi 连接成功代码逻辑，已实际为准 */
    ...
}
/* WiFi 连接失败：以下代码为新增 */
else
{
    /* 获取 WiFi 连接失败错误码 */
result->disconnectedReason = wifi_get_last_error();
/* 获取 WiFi 连接状态 */
    result->connState = WIFI_DISCONNECTED;
}
```

6.4 ASR 网络优化工程修改实例

6.4.1 三方工程 Wi-Fi 参数修改 demo 示例

1. 关闭 Wifi 自动重连。

```
WiFi 自动重连关闭作用于设备运行整个生命周期。
if((GetWifiRecoveryType() & 0x02) == 0x02) {
    /* 开启功能 2，关闭自动重连功能 */
    printf("wifi recovery: disable autoreconnect\r\n");
    /* 实现关闭自动重连 */
```

```
    ...
} else {
    /* 开启自动重连功能 */
    printf("wifi recovery: enable autoreconnect\r\n");
    /* 实现开启自动重连 */
    ...
}
```

2. 修改扫描信道停留时间。

- ASR 如果不支持 WiFi 停留时间修改。为了尽可能把环境中的 AP 扫全，建议通过接口修改扫描参数，增加扫描次数达到扫全目的：

接口：int HILINK_SetWiFiRecoveryTimesParam(unsigned int scanTimes, unsigned int connectTimes);

scanTimes：扫描次数，默认 3 次，可以把该参数改大以提高 AP 扫全的概率。增大扫描次数会增加配网时长，如果产品有配网时长规格，建议扫描时长维持在 8s 左右，但要平衡好配网时长和 AP 扫全的关系，可以通过多次测试来确定扫描次数。

connectTimes：连接次数，默认 3 次，如果使用默认连接次数可以直接传入 3。

使用：HILINK_SetWiFiRecoveryTimesParam 在 HILINK_Main 前调用即可。

- 信道停留时间修改作用于设备运行整个生命周期。

```
if(GetWifiRecoveryType() != 0) {
    /* 开启网络优化 */
    printf("wifi recovery: scan interval [210]\r\n");
    /* 设置单信道扫描时长 210ms */
    ...
}
```

6.4.2 三方工程 Wi-Fi 参数修改示例

1. 关闭 Wifi 自动重连。

   接口 WifiErrorCode ConnectTo(int networkId)，在 WiFi 连接之前增加修改如下：

```
/* 如果开启网络优化功能 2 */
if ((GetWifiRecoveryType() & 0x02) == 0x02) {
    printf("wifi recovery: disable autoconnect\r\n");
    /* 关闭 WiFi 自动重连 */
lega_wlan_set_sta_autoconnect(0); // 1 开 0 关
}
```

6.5 BK7231M 网络优化工程修改示例

6.5.1 三方工程 Wi-Fi 参数修改 demo 示例

1. 关闭 Wifi 自动重连。

   Wifi 自动重连关闭作用于设备运行整个生命周期。

```
if((GetWifiRecoveryType() & 0x02) == 0x02) {
    /* 开启功能 2，关闭自动重连功能 */
    printf("wifi recovery: disable autoreconnect\r\n");
```



```
    /* 实现关闭自动重连 */
    ...
} else {
    /* 开启自动重连功能 */
    printf("wifi recovery: enable autoreconnect\r\n");
    /* 实现开启自动重连 */
    ...
}
```

2. 修改扫描信道停留时间。

   信道停留时间修改作用于设备运行整个生命周期。

```
if(GetWifiRecoveryType() != 0) {
    /* 开启网络优化 */
    printf("wifi recovery: scan interval [210]\r\n");
    /* 设置单信道扫描时长 210ms */
    ...
}
```

6.5.2 三方工程 Wi-Fi 参数修改示例

1. 支持 BSSID 连接。

   接口 WifiErrorCode ConnectTo(int networkId)，Wifi 连接时新增修改如下：

```
/* 如果开启网络优化功能 2 */
if ((GetWifiRecoveryType() & 0x02) == 0x02)
{
    /* WiFi 连接时指定 BSSID 连接 */
    user_bssid_app_init(g_wifiConfigs[networkId].bssid,
g_wifiConfigs[networkId].preSharedKey);
}
/* 如果没有开启网络优化功能 2 */
else
{
    /* 使用默认连接方式 */
    los_wlan_start_sta(&wNetConfig,
g_wifiConfigs[networkId].preSharedKey,64,chan);
}
```

2. 关闭 WiFi 自动重连。

   修改扫描信道停留时间　接口 WifiErrorCode AdvanceScan(WifiScanParams *params)，在 Wifi 断开时增加修改如下：

```
/* 如果开启网络优化功能 */
if(GetWifiRecoveryType() != 0)
{
    /* 释放 hw_list_ssid 内存 */
    if(hw_list_ssid)
    {
        free(hw_list_ssid);
}
/* 给 hw_list_ssid 重新申请内存 */
    hw_list_ssid = malloc(WIFI_MAX_SSID_LEN);
    if(hw_list_ssid == NULL)
    {
        printf("hw_list_ssid malloc fail\r\n");
    }
```

```
    else
    {
   /* hw_list_ssid 初始化 */
os_memset(hw_list_ssid, 0, WIFI_MAX_SSID_LEN);
}
/* 设置扫描类型 */
    scan_type = NORMAL_SCAN_TYPE;
}
/* 如果开启网络优化功能 2 */
if ((GetWifiRecoveryType() & 0x02) == 0x02)
{
    extern void user_set_auto_reconnect_switch(int value);
    /* 1 自动重连关，0 自动重连开 */
    user_set_auto_reconnect_switch(1);
    printf("wifi recovery: autoconnect disable\r\n");
}
if(scan_type== NORMAL_SCAN_TYPE)
{
    /* 外部声明 */
    extern void user_set_chan_time(uint32_t scan_time);
    /* 调用“user_set_chan_time”接口修改 WiFi 扫描策略，修改单个信道停留时长为 210ms，保证环境中的 AP 尽可能扫全 */
user_set_chan_time(210*1024);
    printf("wifi recovery: scan interval time [210]\r\n");
    if(hw_list_ssid != NULL)
{
    /* ssid 拷贝 */
        strcpy(hw_list_ssid, params->ssid);
        /* 1 底层过滤指定 SSID 的 AP 存入缓存；0 不过滤存放所有扫描结果 */
        user_set_scan_ssid(hw_list_ssid, 1);
}
mhdr_scanu_reg_cb(los_scan_cb, 0);
/* 启动扫描 */
    bk_wlan_start_scan();
}
else
{
    /* 设置扫描参数，并启动扫描 */
    ssid_array = params->ssid;
    mhdr_scanu_reg_cb(los_scan_cb, 0);
    bk_wlan_start_assign_scan(&ssid_array, 1);
}
```

3. WiFi 连接状态码返回。

   接口 WifiErrorCode ConnectTo(int networkId)，在 Wifi 连接返回时修改如下：

```
/* 原代码段，以实际为准 */
while(time_out_cnt --)
{
    /* 原代码段，以实际为准 */
    osDelay(500);//4//1s
    /* 原代码段，以实际为准 */
GetLinkedInfo(&info);
/* 如果 WiFi 已连接或 WiFi 连接完成并返回了错误码 */
    if((info.connState == WIFI_CONNECTED) || hw_get_wifi_disc_reason_code())
    {
```



```
        ret = WIFI_SUCCESS;
        break;
}
/* 如果开启网络优化功能 2 且 WiFi 连接完成并返回了错误码 */
    if (((GetWifiRecoveryType() & 0x02) == 0x02) &&
(hw_get_wifi_disc_reason_code()))
    {
        ret = WIFI_SUCCESS;
        break;
    }
}
return ret;
```

6.6 BL602C 网络优化工程修改示例

6.6.1 三方工程 Wi-Fi 参数修改 demo 示例

1. 关闭 Wifi 自动重连。

   Wifi 自动重连关闭作用于设备运行整个生命周期。

```
if((GetWifiRecoveryType() & 0x02) == 0x02) {
    /* 开启功能 2，关闭自动重连功能 */
    printf("wifi recovery: disable autoreconnect\r\n");
    /* 实现关闭自动重连 */
    ...
} else {
    /* 开启自动重连功能 */
    printf("wifi recovery: enable autoreconnect\r\n");
    /* 实现开启自动重连 */
    ...
}
```

2. 修改扫描信道停留时间。

   信道停留时间修改作用于设备运行整个生命周期。

```
if(GetWifiRecoveryType() != 0) {
    /* 开启网络优化 */
    printf("wifi recovery: scan interval [210]\r\n");
    /* 设置单信道扫描时长 210ms */
    ...
}
```

6.6.2 三方工程 Wi-Fi 参数修改示例

1. 关闭 Wifi 自动重连 。

   接口 WifiErrorCode ConnectTo(int networkId)，在 Wifi 连接之前增加修改：

```
/* 开启网络优化功能 2 */
if ((GetWifiRecoveryType() & 0x02) == 0x02) {
    printf("wifi recovery: autoconnect disable\r\n");
    /* 关闭 WiFi 自动重连 */
wifi_mgmr_sta_autoconnect_disable();
}
```


2. 修改扫描信道停留时间 。

   接口 WifiErrorCode AdvanceScan(WifiScanParams *params)，在 Wifi 扫描之前增加修改：

```
/* 网络优化功能开启 */
if (GetWifiRecoveryType() != 0) {
    printf("wifi recovery: scan interval time [210]\r\n");
    /* 调用“WifiSetScanIntervalTime”接口修改 WiFi 扫描策略，修改单个信道停留时长为 210ms，保证环境中的 AP 尽可能扫全 */
WifiSetScanIntervalTime(210);
}
```

3. BUG 修复。

   WiFi 断开后 sta 状态不是空闲，无法扫描到 AP。在 WiFi 断开时调用“wifi_mgmr_sta_disable(NULL);”，让 WiFi 状态变为空闲。接口 static void turbox_event_cb_wifi_event(input_event_t *event, void *private_data)，修改如下：

```
/* 原代码段，已实际代码为准 */
case CODE_WIFI_ON_DISCONNECT:
{
    /* 原代码段，已实际代码为准 */
    TURBOX_PRINTF("[APP] [EVT] disconnect %lld, Reason: %s\r\n", aos_now_ms(), wifi_mgmr_status_code_str(event->value));
/* 原代码段，已实际代码为准 */
hf_gpio_nlink(0);
/* 原代码段，已实际代码为准 */
hfwifi_sta_set_connected_state(0);
/* 如果网络优化功能 2 开启 */
    if ((GetWifiRecoveryType() & 0x02) == 0x02) {
        printf("wifi recovery: sta disable\r\n");
        /* sta disable */
wifi_mgmr_sta_disable(NULL);
}
/* 以下为原代码段，已实际代码为准 */
    extern void WifiConnectionChangedCallback(int state, unsigned short disreason);
    WifiConnectionChangedCallback(0, event->value);
}
break;
```

7 参考

华为智能硬件合作伙伴 > 常见问题