1

FR8000 SDK 用户手册

Bluetooth Low Energy SOC with SIG Mesh

Version: 1.1

2021.11

FREQCHIP 富芮坤

修订版本

版本	日期	更新内容
V1.0	2021.9.1	首版
V1.1	2021.11.1	修改了格式，添加了部分内容

目录

图目录

1. 概况

本文档是 FR8000 SDK 的应用开发指导。FR8000 SDK 是运行于 FR8000 系列芯片上的软件包，包含了 BLE 5.0 的完整协议栈，芯片的外设驱动以及操作系统抽象层 OSAL 等。

1.1. 简介

FR8000 SDK 的架构如下图所示。SDK 包含了完整的 BLE 5.0 协议栈，包括完整的 controller，host，profile，SIG Mesh 部分。其中蓝牙协议栈的 controller 和 host 部分以及操作系统抽象层 OSAL 都是以库的形式提供，图中为灰色部分。MCU 外设驱动和 profile，以及应用层的例程代码，都是以源码的形式提供，图中为绿色部分。

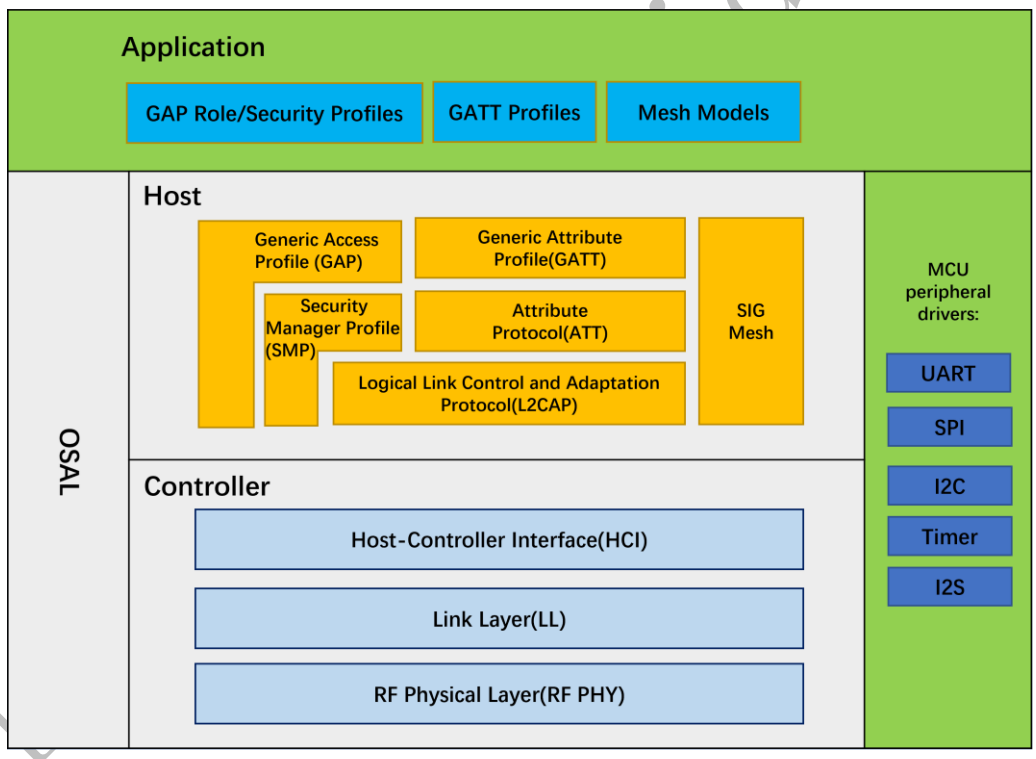

图 1-1 SDK 结构框图

1.2. 程序存储空间

FR8000 的存储空间主要有 ROM、FLASH、RAM 构成，其中 ROM 属于内置不可修改程序，主要实现 bootloader、BLE controller 部分协议栈等功能；FLASH 通过 cache 挂接到

系统 AHB 总线上，支持 XIP（片上运行），用于存储用户程序、不易失变量等；RAM 用于存储程序运行中的临时变量，对响应速度有要求的关键代码（这部分代码在系统启动时从 FLASH 拷贝过来）等。

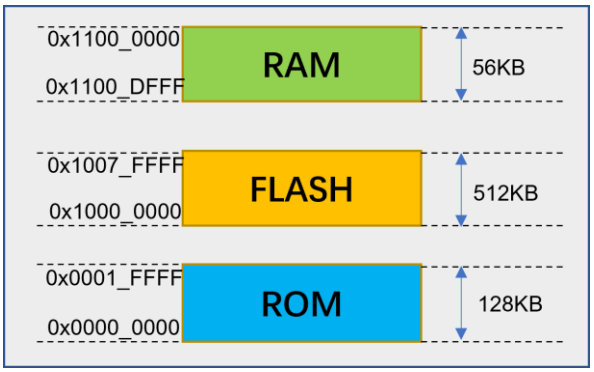

图 1-2 存储空间地址分配

1.3. FLASH 空间分配

FLASH 主要用于存储用户程序和不易失的数据等。因为用户 OTA 的需求，程序空间采用了双备份的方式。不易失的数据包含安全连接需要的 ECDH public-private key pair、IR Key、配对之后的 Link Key 等、做主设备时已绑定从设备的属性、用户数据等。

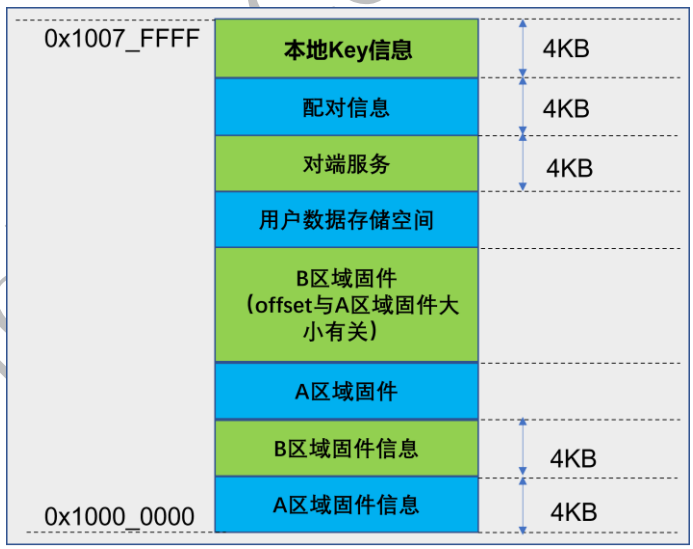

图 1-3 FLASH 空间分配

程序正常运行 A 区域固件，在进行 OTA 升级时将新的固件存储在“B 区域固件”空间、固件信息存储在“B 区域固件信息”空间，传输完成并校验完成之后系统进行重启，由 bootloader 执行固件升级，升级成功之后 B 区域固件信息将会被清除。

1.4. 软件框架

整个软件分为 ROM 固化和用户程序两部分，用户程序存储在 FLASH 中，并且可以直接在 FLASH 上运行。在完成各项初始化之后，系统采用非抢占式的轮询机制进行调度。本节从宏观角度介绍下程序的运行流程。

1.4.1. 启动流程

系统默认复位向量在 ROM 空间，上电之后先运行 ROM 中的 Reset_Handler，然后运行 ROM 中的 bootloader。Bootloader 主要有两个功能：尝试与上位机握手实现烧录（在程序开发阶段、量产烧录阶段均会用到）功能；检测 FLASH 中的存储内容，判断是否需要执行 OTA 最后的拷贝动作。Bootloader 在检测到 FLASH 中的有效固件程序之后，会找到用户程序的入口地址，然后跳转到用户程序。这部分执行的代码参考如下，app_boot 中为 bootloader，app_entry 即为获取的用户程序入口地址。

```
int main(void)
{

    void (*app_entry)(void);

    memcpy((uint8_t *)&__jump_table, (uint8_t *)app_boot(), sizeof(__jump_table));
    __set_MSP((uint32_t)__jump_table.stack_top_address);
    /* do initialization */
    ...
    app_entry = (void (*)(void))(__jump_table.entry);
    app_entry();

}
```

用户程序的入口在 lib 中实现，入口程序中首先进行用户程序的初始化，包括用户程序中 RW 数据段的初始化、ZI 的清零、关键代码拷贝到 RAM 等。将系统异常向量地址重映射到到 RAM 的起始地址后，跳转到应用层代码的入口 user_main。这部分执行的代码参考如下：

```
void app_main(uint8_t index)
{
    uint32_t *dst, *src, *end;
    uint32_t rand_num;

    memset((void *)0x40004000, 0, 8*1024);
    /* init RW section */
    dst = (uint32_t *)&Image$$ER_RW$$RW$$Base;
    src=(uint32_t *)&Load$$ER_RW$$RW$$Base;
    end = (uint32_t *)&Load$$ER_RW$$RW$$Limit;
    for(; (uint32_t)src<(uint32_t)end;)
    {
```

```
        *dst++ = *src++;
    }
    /* init ZI section */
    dst = (uint32_t *)&Image$$ER_ZI$$ZI$$Base;
    end = (uint32_t *)&Image$$ER_ZI$$ZI$$Limit;
    for(; dst < end;)
    {
        *dst++ = 0;
    }
    /* copy RAM code from flash to RAM */
    dst = (uint32_t *)&Image$$USER_RE_RAM$$Base;
    src = (uint32_t *)&Load$$USER_RE_RAM$$Base;
    end = (uint32_t *)&Load$$USER_RE_RAM$$Limit;
    for(; (uint32_t)src<(uint32_t)end;)
    {
        *dst++ = *src++;
    }
    /* copy reset handlers from flash to RAM */
    dst = (uint32_t *)&Image$$ER_BOOT$$Base;
    src = (uint32_t *)&Load$$ER_BOOT$$Base;
    end = (uint32_t *)&Load$$ER_BOOT$$Limit;
    for(; (uint32_t)src<(uint32_t)end;)
    {
        *dst++ = *src++;
    }
    /* some other initialization */
    ...
    /* set exception vector offset to RAM space */
    SCB->VTOR = 0x11000000;
    ...
    /* all interrupt priorities are set to 4 except BLE interrupt is set to 2 */
    intc_init();
    /* initial rand seed with internal TRNG */
    void trng_init(void);
    rand_num = trng_init();
    srand(rand_num);
    /* jump to application code */
    user_main();
}
```

以上两部分分别实现在 ROM 和 lib 中，对用户属于不可见的部分。接下来要运行的 user_main 属于应用程序的一部分，在这里面用户根据实际项目配置外设、配置蓝牙协议栈等，示例如下：

```
void user_main(void)
{
    /* initialize log module */
    log_init();
    /* initialize PMU module at the beginning of this program */
    pmu_sub_init();
    /* start lowpower RC clock calibration */
    NVIC_EnableIRQ(PMU_IRQn);
    pmu_calibration_start(PMU_CALI_SEL_RCLFOSC, LP_RC_CALIB_CNT);
    /* set system clock */
    system_set_clock(SYSTEM_CLOCK_SEL);
   jump_table_set_static_keys_store_offset(JUMP_TABLE_STATIC_KEY_OFFSET);
```

```
    /* initialize ble stack */
    mac_addr_t mac_addr;
    mac_addr.addr[0] = 0xbd;
    mac_addr.addr[1] = 0xad;
    mac_addr.addr[2] = 0x10;
    mac_addr.addr[3] = 0x11;
    mac_addr.addr[4] = 0x20;
    mac_addr.addr[5] = 0x20;
    gap_address_set(&mac_addr, BLE_ADDR_TYPE_PRIVATE);
    ble_stack_configure(false,
                        BLE_STACK_ENABLE_CONNECTIONS,
                        BLE_STACK_RX_BUFFER_CNT,
                        BLE_STACK_RX_BUFFER_SIZE,
                        BLE_STACK_TX_BUFFER_CNT,
                        BLE_STACK_TX_BUFFER_SIZE,
                        BLE_STACK_ADV_BUFFER_SIZE);
    /* interrupt will be generated and should be handled during ble_stack_init */
    GLOBAL_INT_START();
    ble_stack_init();
    gap_dev_name_set("FR8000", strlen("FR8000"));
    gap_bond_manager_init(BLE_BONDING_INFO_SAVE_ADDR, BLE_REMOTE_SERVICE_SAVE_ADDR, 8, true);
    gap_set_cb_func(proj_ble_gap_evt_func);
    proj_init();
    /* enter main loop */
    main_loop();

}
```

1.4.2. 主循环

在完成各项初始化之后系统会进入主循环，主循环采用了非抢占式的轮询调度方法，示例如下：

```
__attribute__((section("ram_code"))) void main_loop(void)
{
    while(1) {
        if(ble_stack_schedule_allow()) {
            /*user code should be add here*/

            /* schedule internal stack event */
            ble_stack_schedule();
        }

        GLOBAL_INT_DISABLE();
        switch(ble_stack_sleep_check()) {
            case 2:
                ble_stack_enter_sleep();
                break;
            default:
                break;
        }
        GLOBAL_INT_RESTORE();

        ble_stack_schedule_backward();
```

```
    }

}
```

ble_stack_schedule 即为调度的入口，在该函数在 ble_stack_schedule_allow 返回 true 时会被调用，用于查询是否有新的事件要处理。用户如果有自己的轮询代码也可以加在这里面，需要注意的是单次执行事件不可过长（几十 ms），以免影响 baseband 调度。

1.4.3. 睡眠与唤醒

上一节中涉及的 ble_stack_sleep_check 函数用来检查当前系统状态是否允许进入睡眠，检查的对象包括有没有需要处理的事件、短时间（几 ms）内有没有要处理的任务、有没有蓝牙的调度任务等，如果发现短时间内没有任务需要处理，那么该函数就会返回可以进入睡眠的结果。在可以进入睡眠时调用 ble_stack_enter_sleep 就可以使得系统进入睡眠状态，待睡眠时间到时或者有异步事件时，系统会被唤醒。在系统唤醒之后 ble_stack_schedule_backward 函数用于完成系统恢复的任务。

在系统睡眠时所有的外设和 MCU core 都处于掉电的状态，因此在唤醒之后对使用的外设需要进行重新初始化；为了防止 GPIO 的漏电，在睡眠的时候需要给 GPIO 设定固定的状态。为此，程序预留了两个接口：user_entry_before_sleep_imp 和 user_entry_after_sleep_imp，分别在睡眠前和唤醒后被调用，用户可以在这两个入口添加必要的程序来实现前面提到的动作。这两个函数的参考示例如下，这里面只是简单实现了低功耗 RC 时钟校准的开启与关闭：

```
__attribute__((section("ram_code"))) void user_entry_before_sleep_imp(void)
{
    pmu_calibration_stop();
    uart_putc_noint_no_wait(UART0, 's');
}

__attribute__((section("ram_code"))) void user_entry_after_sleep_imp(void)
{
    log_init();
    uart_putc_noint_no_wait(UART0, 'w');

    /* reinit IO and peripherals */
    NVIC_EnableIRQ(PMU_IRQn);
    pmu_calibration_start(PMU_CALI_SEL_RCLFOSC, LP_RC_CALIB_CNT);

}
```

1.5. 烧录

对与 FR8000 的烧录分成开发阶段和量产阶段两种烧录场景。开发阶段支持 PC 串口烧录和 J-Link 烧录两种方式，其中 J-Link 烧录支持 keil 和 J-Flash 两种烧录途径。

1.5.1. PC 串口烧录

需要工具：PC 烧录软件、USB 转串口。

在芯片一上电时，内部 boot 程序会尝试通过串口与外部工具进行通信，在握手成功之后就可以进行烧录等后续操作。具体操作如下：

1. 打开 PC 端串口烧录工具，选择正确的串口号，导入 DAT 文件（选择要烧录的 bin 文件），然后打开串口，进入等待连接状态；

图 1-4 烧录工具等待连接

2. 烧录工具与芯片握手有两种方式：

   方式一：将串口工具的 TX 连接到芯片 PA0（芯片端的 RX），RX 连接到芯片的 PA1（芯片端的 TX），然后将电源 VCC 接到芯片 VBAT，最后接地线；

   方式二：将串口工具的 TX、RX、VCC、GND 与芯片连接好，然后按下复位键；

3. 这时芯片与 PC 工具握手成功后在工具端会显示“已经连接（flash）”，然后点击写入所有内容即可将程序烧录到芯片中

图 1-5 烧录工具已经连接

1.5.2. J-Link 烧录

芯片的 J-Link 引脚为 PC6（SWCLK）和 PC7（SWO）。

1.5.2.1. Keil+jlink 烧录

需要工具：keil、J-Link.exe、FR8000.FLM。

用户将文件 FR8000.FLM 存放在 Keil 安装目录下的 ARM\Flash 路径中，然后在 Keil 工程中进行如下配置：

1. 选用 J-Link 作为调试工具

   Keil 工具菜单栏 Flash，下拉选择 Config Flash Tools，如图 1-6 所示，在 Debug 子选项卡，选择 J-LINK/J-TRACE Cortex 选项，如图 1-7 所示。

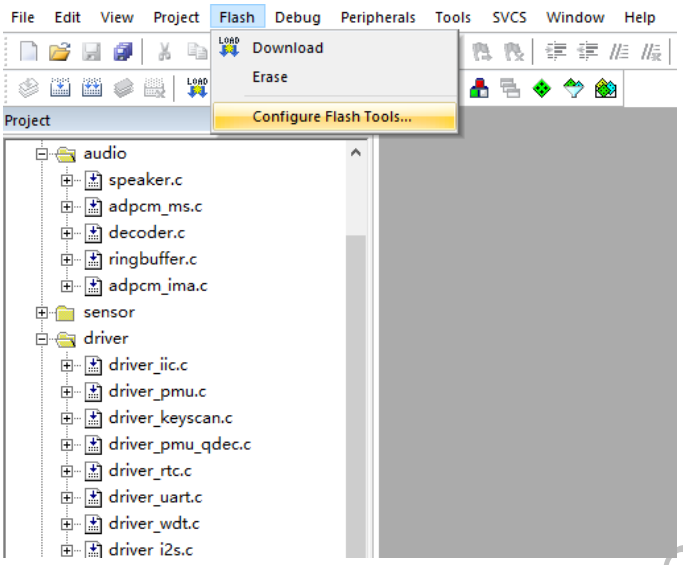

图 1-6 Config Flash Tools 子选项卡

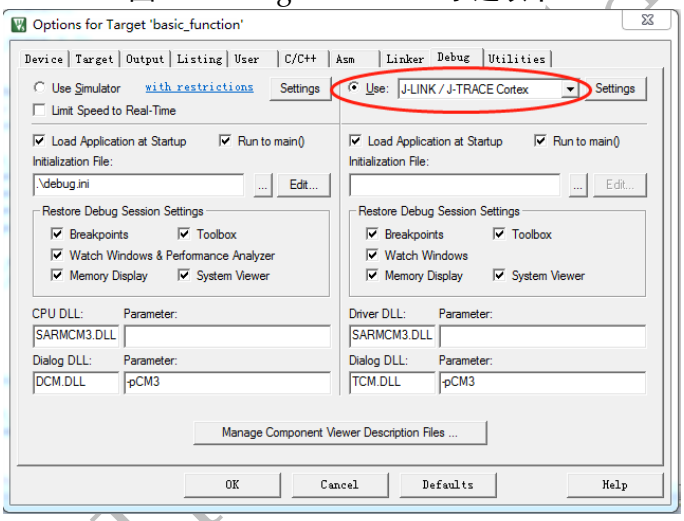

图 1-7 Debug 子选项卡

2. 配置调试方式为 SW

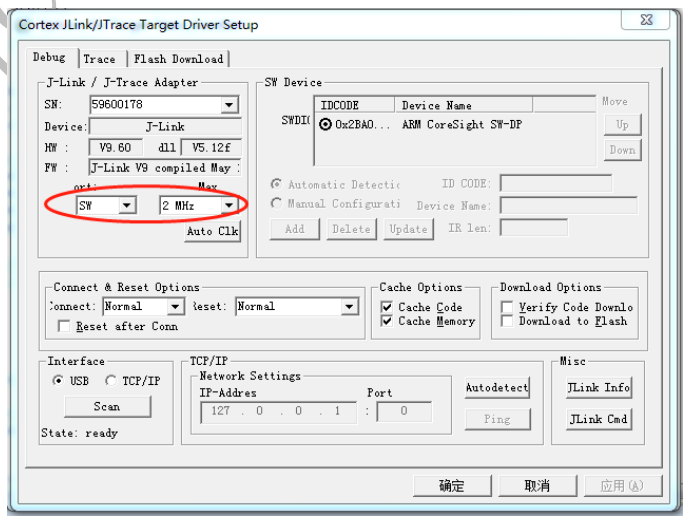

图 1-8 调试方式配置

3. 在 flash download 选项卡中配置下载选项

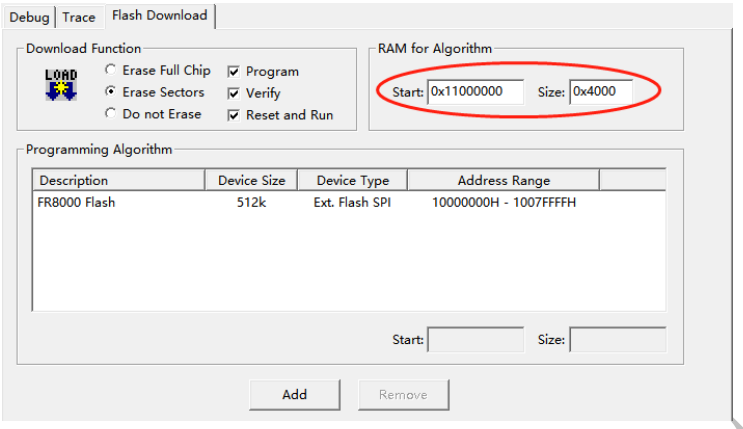

图 1-9 flash download 子选项卡

4. 配置使用 Debug Driver 进行 flash 的烧录

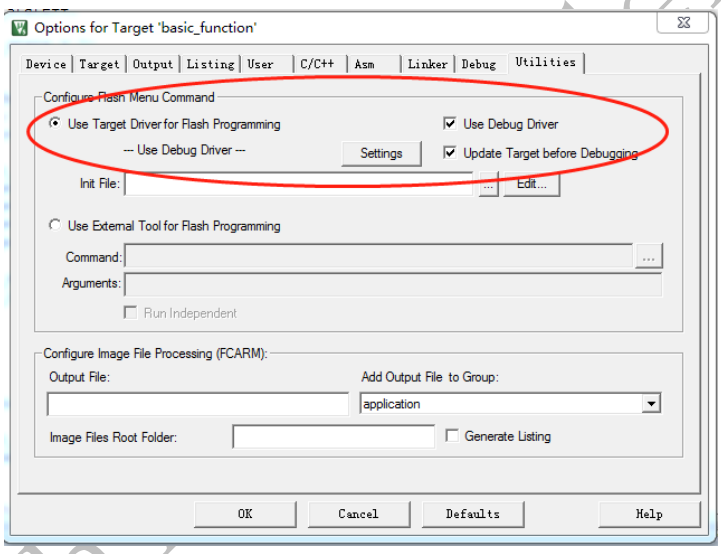

图 1-10 Utilities 子选项卡

通过以上配置就可以实现在 Keil 的 IDE 中进行 flash 的调试和烧录。

1.5.2.2. J-Flash 烧录

需要工具：6.2 以上版本的 J-Link.exe，FR8000.FLM。

1. 在 JLinkDevices.xml 中加入芯片声明：

```
<!--          -->
  <!-- FreqChip -->
  <!--          -->
  <Device>
    <ChipInfo Vendor="FreqChip" Name="FR8000" Core="JLINK_CORE_CORTEX_M3"
WorkRAMAddr="0x11000000" WorkRAMSize="0x4000"/>
        <FlashBankInfo Name="External Flash" BaseAddr="0x10000000" MaxSize="0x80000"
```

```
Loader="Devices/Freqchip/FR8000.FLM" LoaderType="FLASH_ALGO_TYPE_OPEN" AlwaysPresent="1"/>
  </Device>
```

2. 在 JLink 安装目录中创建 SEGGER\JLink\Devices\Freqchip 文件夹，并将 FR8000.FLM 文件拷贝到该文件夹下：

图 1-11 FR8000.FLM 存放位置

3. 打开 J-Flash，创建一个新的工程

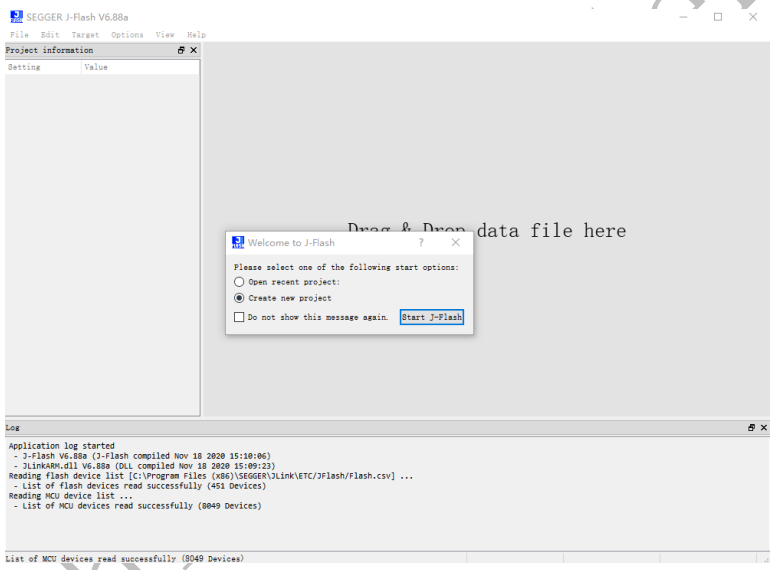

图 1-12 创建新工程图示 1

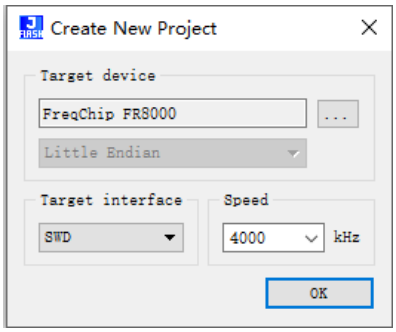

图 1-13 创建新工程图示 2

4. 选择 FR8000 作为 target device

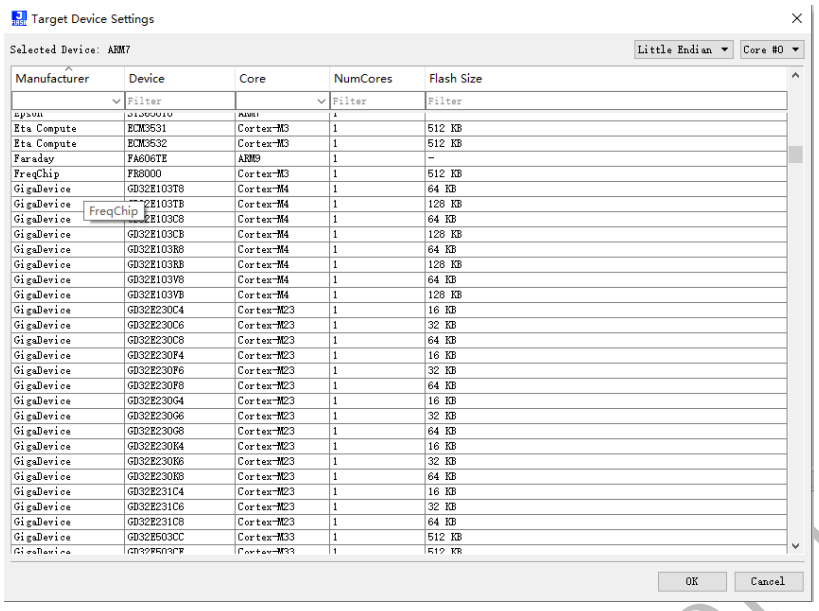

图 1-14 target device 选择

5. 进入如下界面，菜单栏选择 Target->connect 连接待烧录芯片

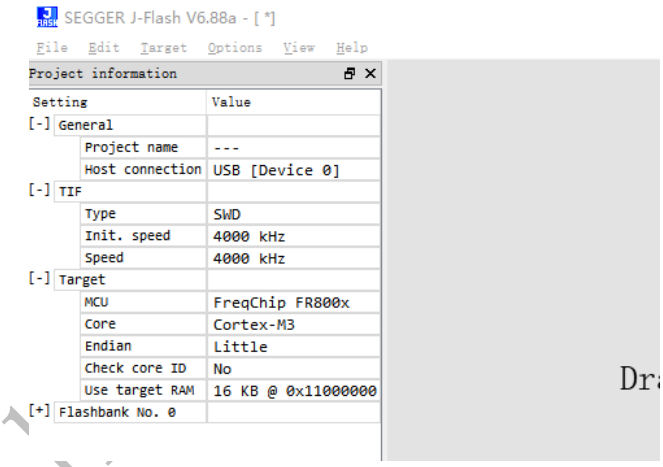

图 1-15 建立连接

6. 菜单栏选择 File->Open data file，并输入起始烧录地址为 0x10000000，打开待烧录文件
7. 菜单栏选择 Target->Production Programming 或者 Target->Maual Programming->Program/Program & Verify 执行文件的烧录。

2. BLE 协议栈

FR8000 SDK 包含了完整的 BLE 协议栈，其中 controller 在 ROM 中实现，host 部分在 lib 中实现，提供了一组丰富的 API 供用户调用来完成蓝牙功能。

2.1. 初始化

2.1.1. 协议栈初始化

在进行蓝牙操作之前需要调用 ble_stack_init 对协议栈进行初始化，因为初始化时涉及到蓝牙地址、蓝牙链路层 buffer 分配等内容，在调用 ble_stack_init 之前需要先调用以下两个函数设置这些内容：

1. 设置蓝牙地址：

```
mac_addr_t mac_addr;
mac_addr.addr[0] = 0xbd;
mac_addr.addr[1] = 0xad;
mac_addr.addr[2] = 0x10;
mac_addr.addr[3] = 0x11;
mac_addr.addr[4] = 0x20;
mac_addr.addr[5] = 0x20;
gap_address_set(&mac_addr, BLE_ADDR_TYPE_PRIVATE);
```

2. 链路层收发 buffer 大小与个数配置：

```
ble_stack_configure(true,
                    BLE_STACK_ENABLE_CONNECTIONS,
                    BLE_STACK_RX_BUFFER_CNT,
                    BLE_STACK_RX_BUFFER_SIZE,
                    BLE_STACK_TX_BUFFER_CNT,
                    BLE_STACK_TX_BUFFER_SIZE,
                    BLE_STACK_ADV_BUFFER_SIZE);
```

蓝牙数据的收发通过软硬件共享的一段存储 RAM 空间，这个空间为 8KB，实际使用的空间与支持的链路个数、收发包的大小与个数等有关，因此在配置这些参数时需要保证不要超过实际的 RAM 大小。

2.1.2. GAP 回调函数

设备连接成功、参数更新、搜到广播包等消息属于 GAP 消息，应用层需要注册接收函数来处理这些消息：gap_set_cb_func(proj_ble_gap_evt_func)。该回调函数示例如下：

```
void proj_ble_gap_evt_func(gap_event_t *event)
{
    switch(event->type){
        case GAP_EVT_ADV_END:
            ...
            break;
        case GAP_EVT_SCAN_END:
            ...
            break;
        case GAP_EVT_ADV_REPORT:
            if(memcmp(event->param.adv_rpt-
>src_addr.addr.addr,"\xfd\x37\xe3\xe1\xfC\x02",6)==0) {
                ...
            }
            break;
        case GAP_EVT_ALL_SVC_ADDED:
            ...
            break;
        case GAP_EVT_MASTER_CONNECT:
            master_link_conidx = (event->param.master_connect.conidx);
            break;
        case GAP_EVT_SLAVE_CONNECT:
            slave_link_conidx = event->param.slave_connect.conidx;
            break;

    ...
```

2.2. 广播

2.2.1. 传统广播

当设备开启传统广播流程时，应用层与 BLE 协议栈之间的交互流程如下所示：

图 2-1 使能传统广播流程

2.2.1.1. 设置广播参数

下面以配置一个可被发现、可被链接、广播间隔为 40ms 的普通广播为例介绍如何设置广播参数：

```
gap_adv_param_t adv_param;
adv_param.adv_mode = GAP_ADV_MODE_UNDIRECT;
adv_param.disc_mode = GAP_ADV_DISC_MODE_GEN_DISC;
adv_param.adv_addr_type = GAP_ADDR_TYPE_STATIC;
adv_param.adv_chnl_map = GAP_ADV_CHAN_ALL;
adv_param.adv_filt_policy = GAP_ADV_ALLOW_SCAN_ANY_CON_ANY;
adv_param.adv_intv_min = 0x40;
adv_param.adv_intv_max = 0x40;
gap_set_advertising_param(&adv_param);
```

对于已经开启的广播，用户需要先暂停当前广播才能重新配置参数，否则将会配置失败。

2.2.1.2. 设置广播内容

当广播参数设置中 adv_mode 字段配置为 GAP_ADV_MODE_DIRECT 或 GAP_ADV_MODE_HDC_DIRECT 时，不需要设置广播数据，其他情况均需要对广播内容

进行设置。传统广播最大数据段为 31 字节，其中 flag 字段由 host 协议栈内部根据广播参数生成，占据三个字节，因此留给用户配置的数据长度为 28 字节。广播的内容需要严格按照蓝牙协议规定的 length，type，data 格式进行填写，否则将会设置失败。

```
uint8_t adv_data[0x1C];
uint8_t *pos;
uint8_t adv_data_len = 0;
pos = &adv_data[0];
uint8_t manufacturer_value[] = {0x00,0x00};
*pos++ = sizeof(manufacturer_value) + 1;
*pos++  = '\xff';
memcpy(pos, manufacturer_value, sizeof(manufacturer_value));
pos += sizeof(manufacturer_value);

uint16_t uuid_value = 0x1812;
*pos++ = sizeof(uuid_value) + 1;
*pos++  = '\x03';
memcpy(pos, (uint8_t *)&uuid_value, sizeof(uuid_value));
pos += sizeof(uuid_value);
adv_data_len = ((uint32_t)pos - (uint32_t)(&adv_data[0]));

gap_set_advertising_data(adv_data,adv_data_len );
```

在需要更新广播内容时，用户无需暂停当前广播，可以直接调用该函数对内容进行更新。

2.2.1.3. 设置扫描回复内容

扫描回复（scan response）内容为设备在广播过程中收到扫描请求（scan request）时做出的回复。当广播参数设置中的 adv_mode 字段为 GAP_ADV_MODE_UNDIRECT 或 GAP_ADV_MODE_NON_CONN_SCAN 时，需要设置扫描响应数据，其他情况不需要设置。在传统广播中，扫描回复内容最长可以配置 31 字节。扫描回复的内容需要严格按照蓝牙协议规定的 length，type，data 格式进行填写，否则将会设置失败。

```
uint8_t scan_rsp_data[0x1F];
uint8_t scan_rsp_data_len = 0;
uint8_t local_name[] = "8000_ADV";
uint8_t local_name_len = sizeof(local_name);
pos = &scan_rsp_data[0];
*pos++ = local_name_len + 1;  //pos len;  (payload + type)
*pos++  = '\x09';   //pos: type
memcpy(pos, local_name, local_name_len);
pos += local_name_len;
scan_rsp_data_len = ((uint32_t)pos - (uint32_t)(&scan_rsp_data[0]));
gap_set_advertising_rsp_data(scan_rsp_data,scan_rsp_data_len );
```

在需要更新扫描回复内容时，用户无需暂停当前广播，可以直接调用该函数对内容进行更新。

2.2.1.4. 开启广播

调用 gap_start_advertising 函数用于开启广播，输入参数是广播持续时间，单位为 10ms，设置为 0 时表示广播不会自动停。

2.2.2. 扩展广播

扩展广播的设置流程与传统广播的设置流程一致，二者的差异主要体现在广播参数的设置上。

2.2.2.1. 设置广播参数

下面以设置一个非定向、可连接、工作在 CODED PHY 上的 40ms 间隔的扩展广播为例，演示如何配置扩展广播的参数：

```
gap_adv_param_t adv_param;
adv_param.adv_mode = GAP_ADV_MODE_EXTEND_CONN_UNDIRECT;
adv_param.disc_mode = GAP_ADV_DISC_MODE_NON_DISC;
adv_param.adv_addr_type = GAP_ADDR_TYPE_STATIC;
adv_param.adv_chnl_map = GAP_ADV_CHAN_ALL;
adv_param.adv_filt_policy = GAP_ADV_ALLOW_SCAN_ANY_CON_ANY;
adv_param.adv_intv_min = 0x40;
adv_param.adv_intv_max = 0x40;
adv_param.phy_mode = GAP_PHY_CODED; //GAP_PHY_2MBPS
adv_param.adv_sid = 0x2;
gap_set_advertising_param(&adv_param);
```

其中：

1. adv_mode 参数来设置广播的类型，具体参见 gap_api.h 中广播类型宏定义；
2. phy_mode 用于设定广播采用的通信速率，如果配置成 GAP_PHY_CODED，那么 primary 和 secondary 都会采用 CODED PHY，如果配置成 GAP_PHY_1MBPS，那么 primary 和 secondary 都会采用 1M PHY，如果配置成 GAP_PHY_2MBPS，那么 primary 将采用 1M PHY，secondary 将采用 2M PHY；
3. adv_sid 参数表示本广播集合的应用层指定的 set_id 号，范围是 0~0xF。

2.2.2.2. 设置广播内容

当 adv_mode 设置为 GAP_ADV_MODE_EXTEND_NON_CONN_SCAN 或 GAP_ADV_MODE_EXTEND_NON_CONN_SCAN_DIRECT 时，需要设置广播数据。其余类型无需设置：

```
uint8_t adv_data[0x1C];
uint8_t local_name[] = "8000_EXT_ADV";
uint8_t local_name_len = sizeof(local_name);
uint8_t *pos;
uint8_t adv_data_len = 0;
pos = &adv_data[0];
*pos++ = local_name_len + 1;  //pos len;  (payload + type)
*pos++  = '\x09';   //pos: type
memcpy(pos, local_name, local_name_len);
pos += local_name_len;
adv_data_len = ((uint32_t)pos - (uint32_t)(&adv_data[0]));
gap_set_advertising_data(adv_data,adv_data_len );
```

协议栈支持的扩展广播最大长度与协议栈初始化中 BLE_STACK_ADV_BUFFER_SIZE 的配置有关。当该值设定小于 251，最大长度与该值一致，如果等于 251，则可以支持到 251*4 字节的长度。

在需要更新广播内容时，需要将当前广播先停掉，以免底层 buffer 不够，造成更新失败。

2.2.2.3. 设置扫描回复内容

当 adv_mode 设置为 GAP_ADV_MODE_EXTEND_NON_CONN_SCAN 或 GAP_ADV_MODE_EXTEND_NON_CONN_SCAN_DIRECT 时，需要设置广播数据。其余类型无需设置：

```
uint8_t scan_rsp_data[0x1F];
uint8_t scan_rsp_data_len = 0;

uint8_t local_name[] = "8000_SCAN_RSP";
uint8_t local_name_len = sizeof(local_name);
pos = &scan_rsp_data[0];
*pos++ = local_name_len + 1;  //pos len;  (payload + type)
*pos++  = '\x09';   //pos: type
memcpy(pos, local_name, local_name_len);
pos += local_name_len;
scan_rsp_data_len = ((uint32_t)pos - (uint32_t)(&scan_rsp_data[0]));
gap_set_advertising_rsp_data(scan_rsp_data,scan_rsp_data_len );
```

协议栈支持的扩展广播最大长度与协议栈初始化中 BLE_STACK_ADV_BUFFER_SIZE 的配置有关。当该值设定小于 251，最大长度与该值一致，如果等于 251，则可以支持到 251*4 字节的长度。

在需要更新广播内容时，需要将当前广播先停掉，以免底层 buffer 不够，造成更新失败。

2.2.2.4. 开启广播

扩展广播的开启方式与传统广播的开启方式一致，可参考 2.2.1.4。

2.2.3. 周期性广播

当设备开启周期性广播流程时，应用层与 BLE 协议栈之间的交互流程如下图所示：

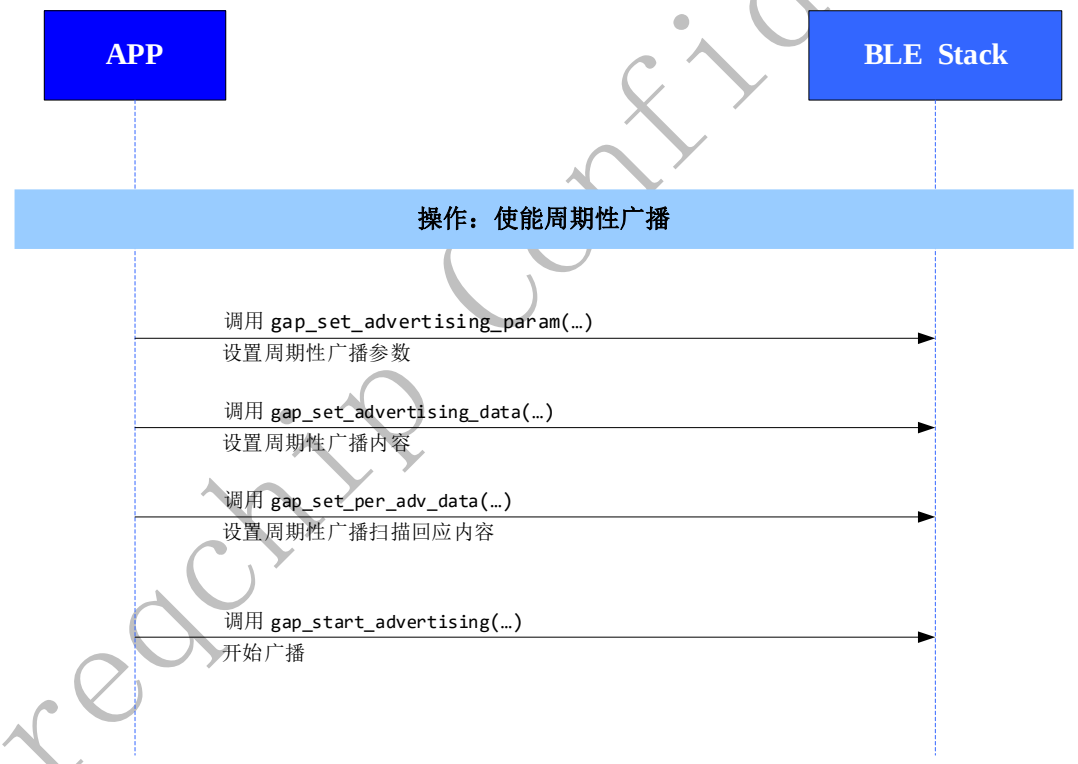

图 2-2 开启周期性广播流程

2.2.3.1. 设置广播参数

```
gap_adv_param_t adv_param;
adv_param.adv_mode = GAP_ADV_MODE_PER_ADV_UNDIRECT;
adv_param.disc_mode = GAP_ADV_DISC_MODE_GEN_DISC;
adv_param.adv_addr_type = GAP_ADDR_TYPE_STATIC;
adv_param.adv_chnl_map = GAP_ADV_CHAN_ALL;
adv_param.adv_filt_policy = GAP_ADV_ALLOW_SCAN_ANY_CON_ANY;
```

```
adv_param.adv_intv_min = 0x40;
adv_param.adv_intv_max = 0x40;
adv_param.phy_mode = GAP_PHY_1MBPS;
adv_param.adv_sid = 9;
adv_param.per_adv_intv_min = 0x100;
adv_param.per_adv_intv_max = 0x100;
gap_set_advertising_param(&adv_param);
```

其中：

1. per_adv_intv_min 和 per_adv_intv_max 表示周期性广播建立后，周期性广播内容的出现的最小最大时间间隔，单位：1.25ms。
2. 其他参数含义与 2.2.2.1 节中描述一致。

2.2.3.2. 设置广播内容

广播内容设置方法与 2.2.2.2 中描述一致。

2.2.3.3. 设置周期性广播数据

周期性广播的数据采用下面的方式进行设置：

```
uint8_t per_adv_data1[]="\x09\x09\x11\x12\x13\x14\x15\x16\x17\x18";
gap_set_per_adv_data(per_adv_data1, sizeof(per_adv_data1));
```

2.2.3.4. 开启广播

扩展广播的开启方式与传统广播的开启方式一致，可参考 2.2.1.4。

2.2.4. 相关 GAP 事件

与广播相关的 GAP 事件主要由以下两种：

1. GAP_EVT_ADV_END，该事件在主动调用 gap_stop_advertising、广播超时、设备被链接三种情况下会产生。
2. GAP_EVT_SLAVE_CONNECT，正在广播可连接数据的设备在被链接上之后，引用层 GAP 回调函数会接收到该事件。

2.3. 扫描

2.3.1. 传统扫描

当设备开启扫描流程时，应用层与 BLE 协议栈之间的交互流程如下：

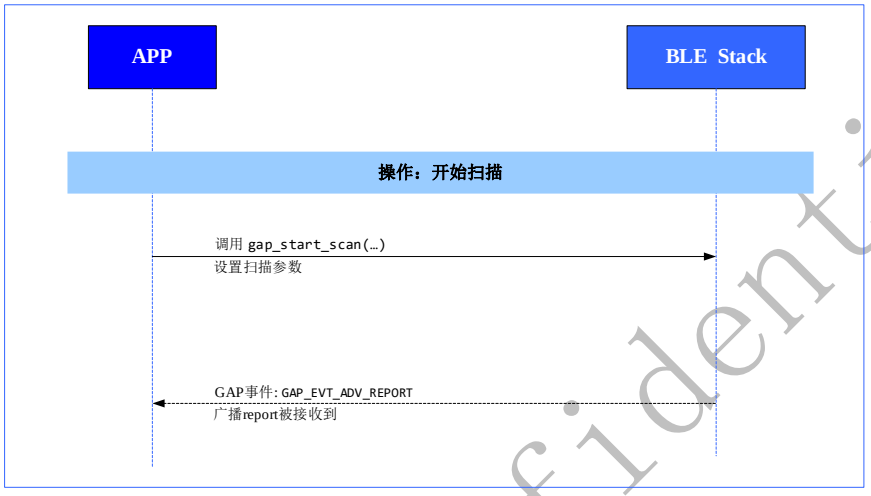

图 2-3 传统扫描流程

以下示例代码演示如何开启普通主动扫描：

```
gap_scan_param_t scan_param;
scan_param.scan_mode = GAP_SCAN_MODE_GEN_DISC;
scan_param.dup_filt_pol = 0;
scan_param.scan_intv = 32;  //scan event on-going time
scan_param.scan_window = 20;
scan_param.duration = 1500;     //15s
scan_param.phy_mode = GAP_PHY_1MBPS;
gap_start_scan(&scan_param);
```

其中：

1. scan_mode 指示扫描的模式：GAP_SCAN_MODE_GEN_DISC 表示主动扫描，这种模式会根据 ADV 类型选择发送 scan request 数据包用来获取 scan response；GAP_SCAN_MODE_OBSERVER 表示被动扫描，这种模式只会处于被动监听状态。
2. dup_filt_pol 表示对扫描到的内容上报前是否进行重复性过滤检查。
3. scan_window 和 scan_intv 分别表示扫描窗口和扫描间隔，单位为 625us。
4. duration 表示整个扫描动作持续时间，时间到后自动停止，单位 10ms。如果设置为 0，表示扫描动作不会自动停止。
5. 对于传统扫描 phy_mode 设置为 GAP_PHY_1MBPS。

2.3.2. 扩展扫描

扩展广播的启动流程和调用方式与 2.3.1 中描述的传统扫描一致，当扫描 primary 采用 CODEC_PHY 的扩展广播时，需要将参数中的 phy_mode 设置为 GAP_PHY_CODED。

2.3.3. 同步扫描

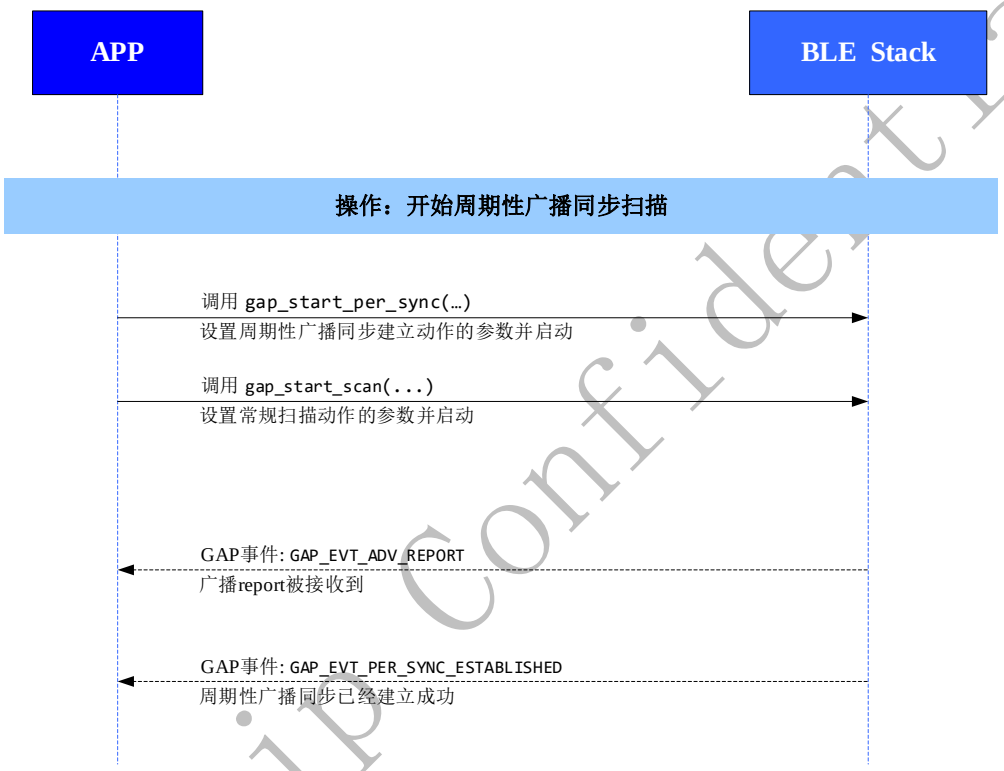

图 2-4 周期性广播同步扫描流程

2.3.3.1. 设置同步建立参数

采用如下代码可以用来设定同步参数：

```
gap_per_sync_param_t per_sync_param;
per_sync_param.sup_to = 600;
per_sync_param.adv_sid = 9;
per_sync_param.adv_dev_addr.addr_type = 0;
memcpy(per_sync_param.adv_dev_addr.addr,"\x1F\x19\x07\x09\x17\x20",MAC_ADDR_LEN);
gap_start_per_sync(&per_sync_param);
```

其中：

1. sup_to 表示周期性广播同步建立之后，连续一段时间内没有收到广播数据，就会产生握手超时停止同步动作，单位为 10ms。
2. adv_sid 参数表示同步动作针对的周期性广播的 set_id 号。
3. adv_dev_addr 表示同步对象的地址和地址类型。

2.3.3.2. 开启扫描

同步扫描启动方式与 2.3.2 节中描述一致。扫描到满足参数要求的 primary 通道广播包时，会产生周期性广播同步建立事件 GAP_EVT_PER_SYNC_ESTABLISHED，之后的扩展广播包和周期性广播数据都通过 GAP_EVT_ADV_REPORT 上传。

2.3.4. 相关 GAP 事件

与扫描相关的 GAP 事件主要有以下几种：

1. GAP_EVT_ADV_REPORT，该事件用于上报扫描到的广播数据
2. GAP_EVT_SCAN_END，该事件在调用停止扫描函数 gap_stop_advertising，或者扫描时间到时会产生。
3. GAP_EVT_PER_SYNC_ESTABLISHED，该事件表示同步扫描建立成功。
4. GAP_EVT_PER_SYNC_END，该事件表示因为超时导致同步结束。

2.4. 连接

2.4.1. 连接传统广播

当设备发起传统广播连接流程时，应用层与 BLE 协议栈之间的交互流程下图所示。该主动连接的动作只能连接可被连接的传统广播，广播类型为：GAP_ADV_MODE_UNDIRECT 、GAP_ADV_MODE_DIRECT 和 GAP_ADV_MODE_HDC_DIRECT。

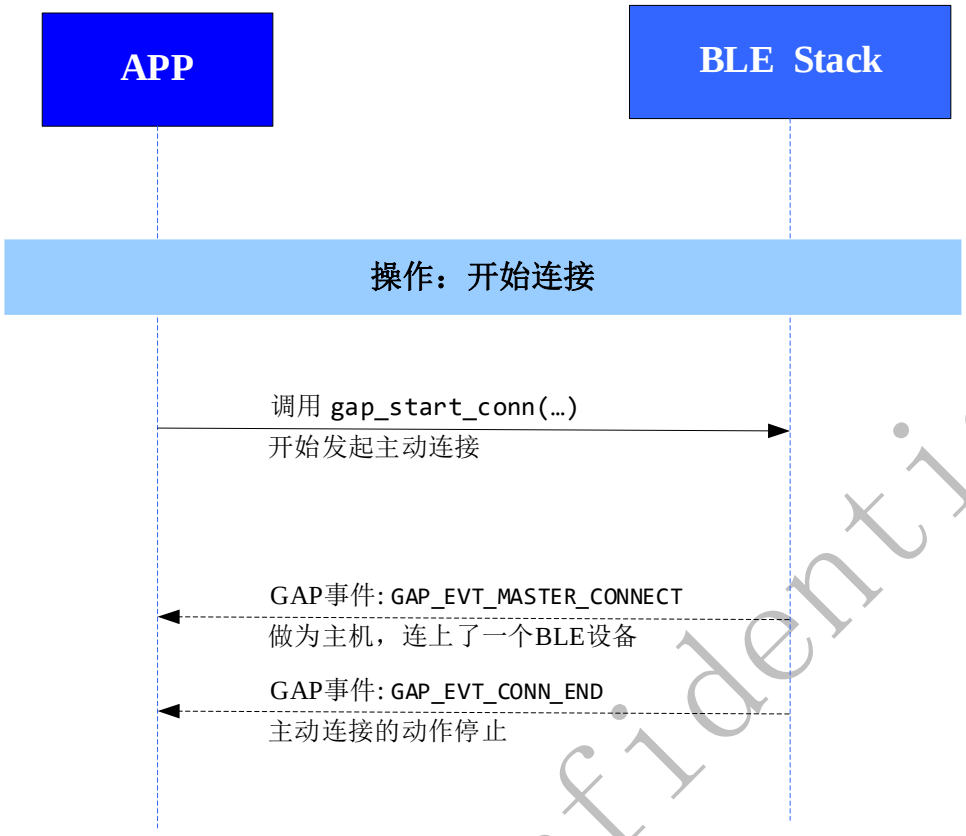

图 2-5 连接建立流程

以下代码展示如何发起一个传统连接：

```
void gap_start_conn(mac_addr_t *addr, uint8_t addr_type, uint16_t min_itvl, uint16_t max_itvl,
uint16_t slv_latency, uint16_t timeout);
addr_t mac = {{0x1F, 0x19, 0x07, 0x09, 0x17, 0x20}};
gap_start_conn(&mac, 0, 9, 9, 0, 400);
```

其中：

1. addr 表示要连接的 BLE 设备的 mac 地址。
2. addr_type 表示要连接的 BLE 设备的 mac 地址类型。0 表示 public addr；1 表示 private addr。
3. min_itvl 和 max_itvl 表示连接建立后采用的握手基础间隔时间，单位：1.25ms。
4. slv_latency 表示连接建立后 slave 设备采用的 latency 参数。
5. timeout 表示连接建立后 slave 设备采用的超时断开时间。单位：10ms。

2.4.2. 连接扩展广播

当设备发起扩展广播连接流程时，应用层与 BLE 协议栈之间的交互流程与连接传统广播一致。该主动连接动作只能连接可连接类型的扩展广播。广播类型为：GAP_ADV_MODE_EXTEND_CONN_UNDIRECT 和 GAP_ADV_MODE_EXTEND_CONN_DIRECT。

1. Primary 信道采用 1 MBPS 或者 2 MBPS

当扩展广播的 primary 信道采用 1 MBPS 或者 2 MBPS 时， 交互流程与 2.4.1 中完全一致。

2. Primary 信道采用 CODED PHY

当扩展广播的 primary 信道采用 CODED PHY 时，系统提供函数

```
void gap_start_conn_long_range(mac_addr_t *addr, uint8_t addr_type, uint16_t min_itvl, uint16_t max_itvl, uint16_t slv_latency, uint16_t timeout);
```

来建立连接，它采用的参数含义与 gap_start_conn 一致。

2.4.3. 其他

1. 断开连接

采用如下函数可以将已经建立的连接断开。该函数对于工作在主机或者从机模式下均适用。

```
void gap_disconnect_req(uint8_t conidx);
```

2. 取消当前连接

在连接建立过程中，用户可以调用下面的函数来取消本次连接。

```
void gap_stop_conn(void);
```

2.4.4. 相关 GAP 事件

与连接相关的 GAP 事件主要由以下几个：

1. GAP_EVT_MASTER_CONNECT，在发起主动连接传统广播或者发起主动连接扩展广播，连上对端 BLE 设备后，协议栈会产生主动连接成功建立事件。

2. GAP_EVT_CONN_END，在以下两种情况时：调用停止主动连接函数或者连接建立后，会产生主动连接动作停止事件。

2.5. Profile 定义

通用属性规范 GATT（Generic Attribute Profile）用于两个连接设备之间的数据通信。在 BLE GATT 层中，数据以特征（Characteristic）的形式进行传输和存储。

2.5.1. GATT 角色

从 GATT 的角度来看，当两个设备处于连接状态时，一个设备作为 GATT 服务端，另一个设备作为 GATT 客户端。这种角色的分配与 GAP 层设备角色（外围设备、中央设备）无关。一个外围设备既可以充当 GATT 客户端，也可以充当 GATT 服务端；一个中央设备同样既可以充当 GATT 客户端，也可以充当 GATT 服务端。

- GATT 客户端：设备发起命令、请求并接收响应、通知和指示。
- GATT 服务端：设备接收命令、请求并发出响应、通知和指示。

2.5.2. GATT Profile 层级

一个 profile 中可包含一个或多个服务；一个服务可包含一个或者多个特征；一个特征至少包含两个属性，包括特征声明和特征值。下面章节将具体描述属性、特征和服务。

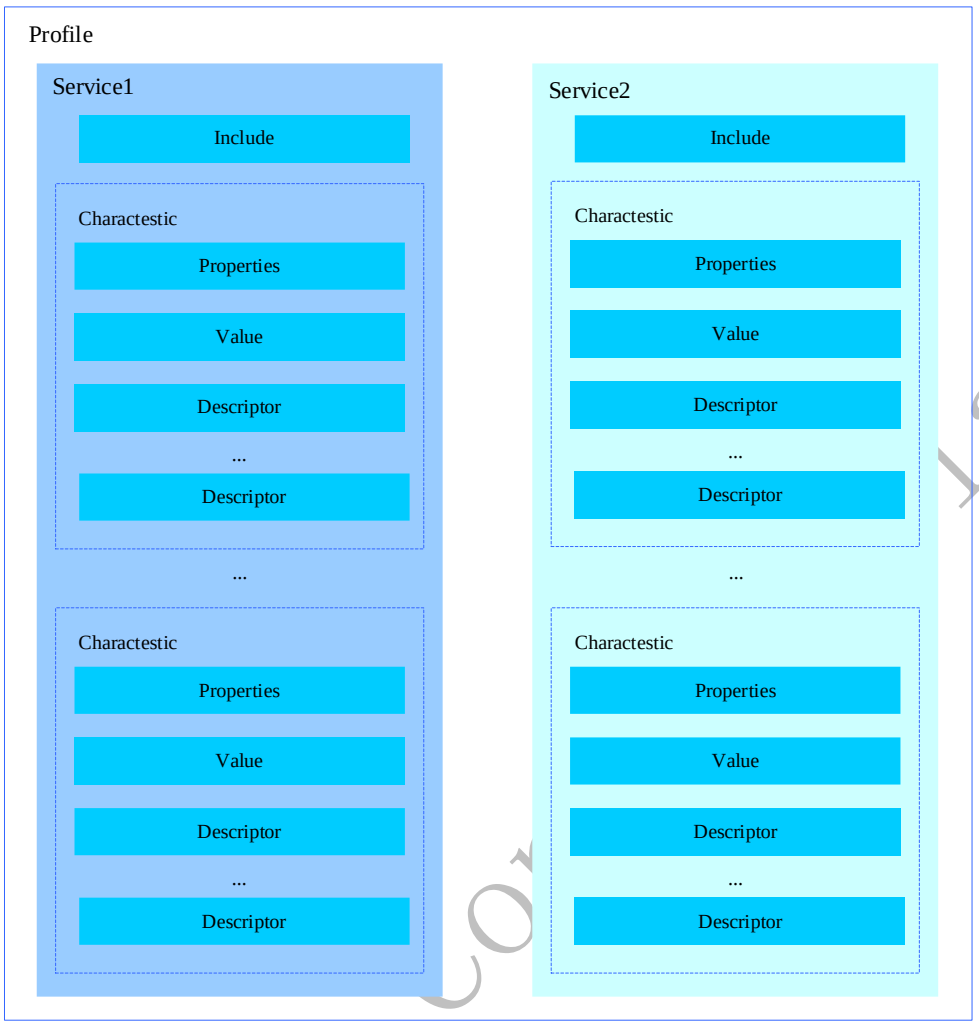

图 2-6 服务、特征、属性层级关系

2.5.2.1. 属性（Attribute）

在低功耗蓝牙中，特征可以看作是一组属性信息的集合，包括特征声明、特征值以及特征描述符，数据以属性的形式进行交互。属性主要包括以下几个部分：

- 属性句柄：句柄是属性在 GATT 属性表中的索引，每个属性都有唯一的句柄。
- 属性类型：表示数据代表的事物，通常是 Bluetooth SIG 规定或由用户自定义的 UUID（通用唯一标识符）。
- 属性值：属性的数据值。
- 属性权限：规定了 GATT 客户端设备对属性的访问权限，包括是否能访问和怎样访问。

2.5.2.2. 特征（Characteristic）

一个典型的特征由以下属性组成：

- 特征声明：描述特征的性质、存储位置（句柄）、类型。
- 特征值：特征的数据值。
- 特征描述符：描述特征的附加信息或配置。

2.5.2.3. 服务（Service）

一个 profile 包含一个或多个服务，GATT 服务是一系列特征的集合，例如，心率服务包括心率测量特征和身体位置特征等。常见的服务有：

- 通用访问服务（Generic Access Service）：该服务包括设备及访问信息等，比如设备名称、供应商标志、产品标志等。该服务定义的特征有：设备名称（Device Name）、外观（Appearance）、外设优先连接参数（Peripheral Preferred Connection Parameters）等。
- 通用属性服务（Generic Attribute Service）：该服务主要用于服务端通知所连接的客户端其提供的服务发生了变化。该服务包含 Service changed 特征。

上述两个服务在 BLE 协议栈初始化时，默认添加到服务端的数据库中。但用户可以在协议栈初始化后可以调用 gatt_delete_svc 函数删除这两个默认的服务，然后在引用 SDK folder\components\ble\profiles\ble_gap & ble_gatt 里定义的 gap 和 gatt 服务。这样用户可以自由更改这两个服务的定义。

2.5.3. 属性表

低功耗蓝牙的通信就是通过对属性进行读写等操作来实现的，本节介绍如果定义这些属性来构成一个完整的服务。

2.5.3.1. 属性表定义

每个 Service 必须定义一个属性表并通过 profile 的初始化函数将其传递给协议栈。以一个 batt 服务的实例来解释属性表的定义。该服务代码在 SDK_Folder\components\profiles\ble_batt\batt_service.c。其定义的属性表数组如下：

```
enum
{
    IDX_BATT_SERVICE,
    IDX_BATT_LEVEL_CHAR_DECLARATION,
    IDX_BATT_LEVEL_CHAR_VALUE,
    IDX_BATT_LEVEL_CCCD,
    IDX_BATT_NB,
};
typedef struct
{
    uint8_t size;               //!< Length of UUID
    uint8_t p_uuid[16];         //!< Pointer to uuid, could be 2 or 16 bytes array.
} gatt_uuid_t;
typedef struct
{
    gatt_uuid_t     uuid;       //!< Attribute UUID
    uint16_t        prop;       //!< Attribute properties, see @GATT_PROP_BITMAPS_DEFINES
    uint16_t        max_size;   //!< Attribute data maximum size
    uint8_t         *p_data;    //!< Attribute data pointer
} gatt_attribute_t;
static const uint8_t batt_svc_uuid[UUID_SIZE_2] = UUID16_ARR(BATT_SERV_UUID);
static const gatt_attribute_t batt_att_table[] =
{
    [IDX_BATT_SERVICE] = { { UUID_SIZE_2, UUID16_ARR(GATT_PRIMARY_SERVICE_UUID)},
        GATT_PROP_READ,  UUID_SIZE_2, (uint8_t *)batt_svc_uuid,
    },
    [IDX_BATT_LEVEL_CHAR_DECLARATION] = { { UUID_SIZE_2, UUID16_ARR(GATT_CHARACTER_UUID)},
GATT_PROP_READ,0, NULL,
    },
    [IDX_BATT_LEVEL_CHAR_VALUE] = { { UUID_SIZE_2, UUID16_ARR(BATT_LEVEL_UUID)},
        GATT_PROP_READ | GATT_PROP_NOTI, sizeof(uint8_t),NULL,
    },
    [IDX_BATT_LEVEL_CCCD] = { {UUID_SIZE_2, UUID16_ARR(GATT_CLIENT_CHAR_CFG_UUID)},
        GATT_PROP_READ | GATT_PROP_WRITE,sizeof(uint16_t),NULL,
    },

};
```

说明：

- 属性表 batt_att_table[] 是一个数组。数组变量类型为结构体类型 gatt_attribute_t。数组里每一个变量构成属性表里的一个属性。
- 结构体类型 gatt_attribute_t 里各元素解释如下：

a) uuid 表示这条属性的 UUID。uuid.size 是 UUID 的长度，值可以是 16bits 或者是 128bits，用宏 UUID_SIZE_2 和 UUID_SIZE_16 表示。uuid.p_uuid 是 UUID 的内容。

b) prop 表示这条属性的读写权限。profile 属性权限宏定义参见 gatt_api.h 宏定义组 GATT_PROP_BITMAPS_DEFINES 里的定义，如下：

```
#define GATT_PROP_BROADCAST       (1<<0)  //!< Attribute is able to broadcast
#define GATT_PROP_READ            (1<<1)  //!< Attribute is Readable
#define GATT_PROP_WRITE_CMD       (1<<2)  //!< Attribute supports write with no response
#define GATT_PROP_WRITE_REQ       (1<<3)  //!< Attribute supports write request
#define GATT_PROP_NOTI            (1<<4)  //!< Attribute is able to send notification
#define GATT_PROP_INDI            (1<<5)  //!< Attribute is able to send indication
#define GATT_PROP_AUTH_SIG_WRTIE (1<<6)  //!< Attribute supports authenticated signed write
#define GATT_PROP_EXTEND_PROP     (1<<7)  //!< Attribute supports extended properities
#define GATT_PROP_WRITE           (1<<8)  //!< Attribute is Writable (write_req and write_cmd)
#define GATT_PROP_AUTHEN_READ     (1<<9)  //!< Read requires Authentication
#define GATT_PROP_AUTHEN_WRITE    (1<<10) //!< Write requires Authentication
```

常用的读写权限是第 2~6 行，和第 9~11 行。可以通过或的方式设置属性支持多条权限。如：GATT_PROP_READ丨GATT_PROP_NOTI 表示属性同时支持 ntf 和 read

c) max_size 表示该属性支持的最大数据长度。

d) p_data 表示指向该属性要处理的数据的指针。

- 在给属性表数组变量赋值的时候，max_size 和 p_data 有以下几种组合，它们的含义如下。

a) max_size>0， p_data != NULL。 创建 profile 时内部会分配内存，并拷贝 p_data 的数据到内部。

b) max_size>0，p_data == NULL。创建 profile 时内部不会分配内存，对端如果读取该属性的值，会产生读操作事件回调。

c) max_size==0，p_data == NULL。该属性不能被读，或者读该属性会返回 Null。

d) max_size==0，p_data != NULL。 该情况不能出现。

- 给 UUID 为 GATT_PRIMARY_SERVICE_UUID 的属性变量赋值时，max_size 和 p_data 分别表示服务组 UUID 的长度和 UUID 的内容。示例属性表中，该属性元素 max_size 为 UUID_SIZE_2，p_data 为(uint8_t *)batt_svc_uuid。分别表示这个服务组的 UUID 长度是 16bits，UUID 的值是 batt_svc_uuid 的值。

2.5.3.2. 建立属性表

当设备上电或者重启时，应用层需在初始化阶段构建自己的服务列表。每个服务都包含一些属性，并根据自己的功能需求定义服务的回调函数。属性表和回调函数在调用添加服务函数时传递给协议栈。例如，在应用层初始化函数中，依次调用 svcA_gatt_add_service 以及 svcB_gatt_add_service 函数添加两个自定义的服务，执行流程如下图所示。

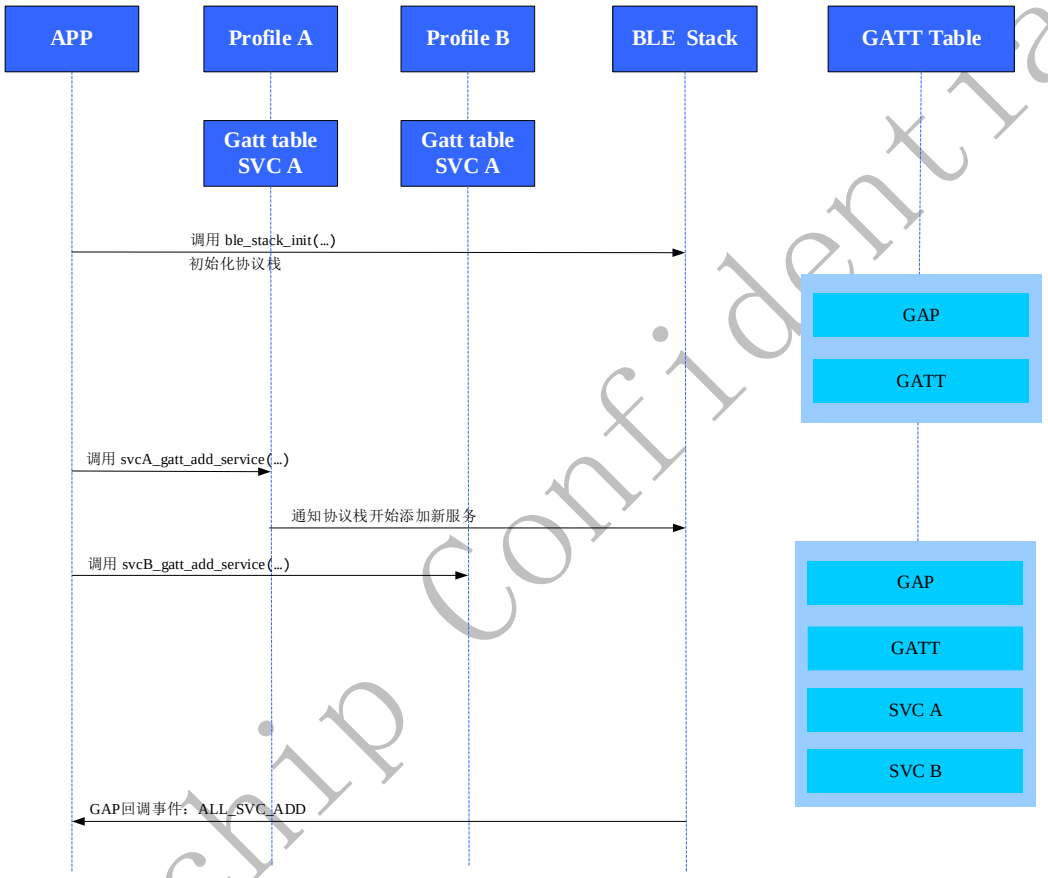

图 2-7 建立属性表流程

2.5.4. GATT 客户端基础流程

2.5.4.1. 客户端

GATT 客户端主要向服务端发起读写命令和请求并接收服务端的响应、指示和通知。

GATT 客户端没有属性表，它主要用于获取属性信息而不是提供属性信息和服务。对属性表属性的读写操作通过使用 gatt_api.h 定义的的 API 接口完成。大部分 GATT 客户端的 API 被调用之后，能通过客户端事件回调函数返回结果，包括读取的属性值、写是否完成状态、指

示等。关于 API 的详细描述，请参考 gatt_api.h 内函数定义说明。客户端 GATT 基础流程如下图所示。

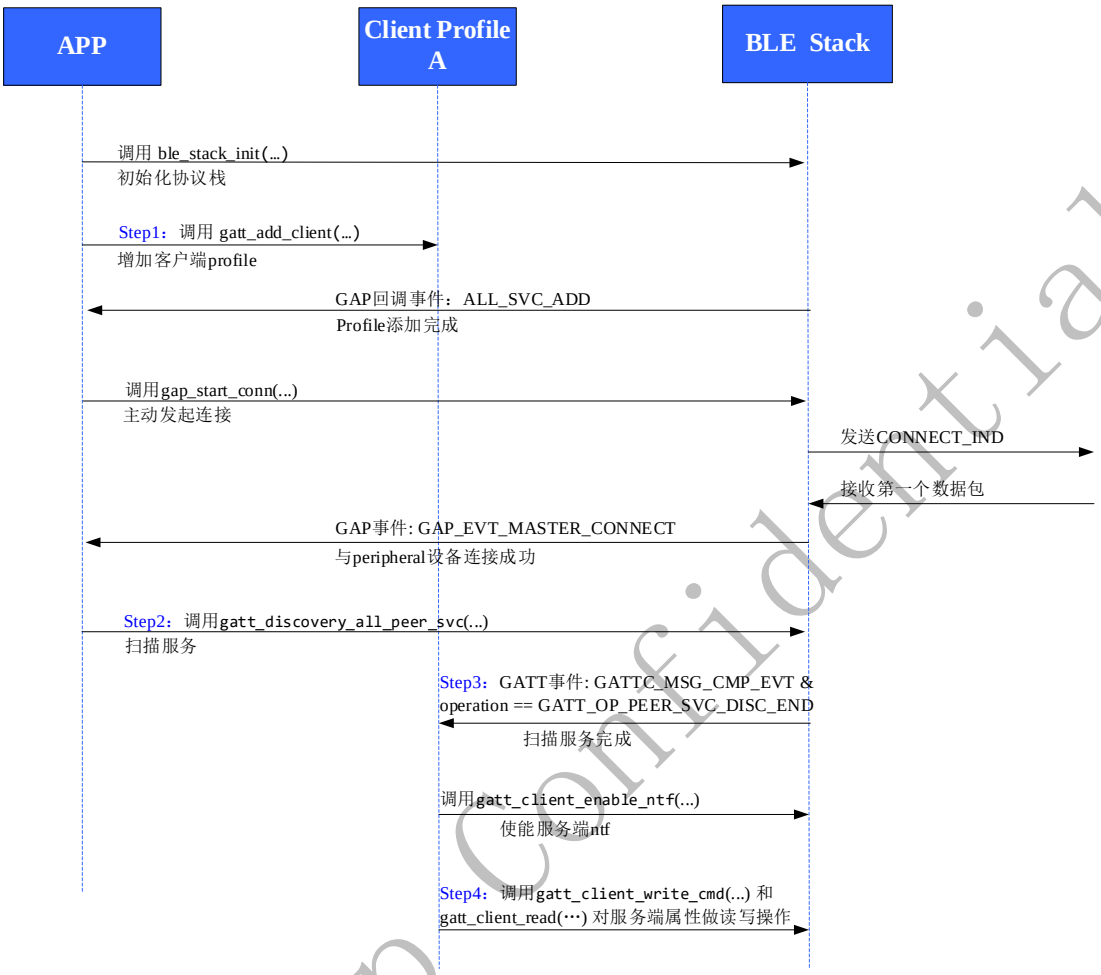

图 2-8 SDK 中客户端工作流程

2.5.4.2. 增加客户端 profile

客户端虽然没有属性表，但是在 FR8000 的协议栈实现中，为了简化应用层的服务发现流程，用户需要向协议栈注册一个应用层关心的服务端 uuid 数组和用于接收消息的回调函数。这个 uuid 数组包含了客户端可能会发起读、写、接收通知等功能会涉及到的属性的 uuid。在连接建立好之后，用户可以调用 gatt_discovery_all_peer_svc 函数，控制协议栈去发现服务端数据库是否包含在初始化时注册的 uuid。下面以一个客户端 profile 注册的实例来解释该过程。该客户端 profile 代码在 SDK_Folder\examples\none_evm\ ble_simple_central\code\ ble_simple_central.c。

```
#define SP_CHAR1_UUID          0xFFF1
#define SP_CHAR2_UUID          0xFFF2
#define SP_CHAR3_UUID          0xFFF4
```

```
const gatt_uuid_t client_att_tb[] =
{
    [0]  ={ UUID_SIZE_2, UUID16_ARR(SP_CHAR1_UUID)},
    [1]  ={ UUID_SIZE_2, UUID16_ARR(SP_CHAR2_UUID)},
    [2]  ={ UUID_SIZE_2, UUID16_ARR(SP_CHAR3_UUID)},

};
uint8_t client_id;
static uint16_t simple_central_msg_handler(gatt_msg_t *p_msg)
{
    co_printf("CCC:%x\r\n",p_msg->msg_evt);
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_NTF_REQ:
        {
            co_printf("GATTC_MSG_NTF_REQ\r\n");
        }
        break;
    }
}
void user_main(void)
{
...
    gatt_client_t client;
    client.p_att_tb = client_att_tb;
    client.att_nb = 3;
    client.gatt_msg_handler = simple_central_msg_handler;
    client_id = gatt_add_client(&client);

}
```

说明：

- client_att_tb[]是一个 uuid 的数组，记录用户真正需要使用的属性的 uuid。调用扫描服务函数后，协议栈能上传这些 uuid 属性对应的 handle 号。
- 第 11 行，client_id 是定义的记录这个客户端 profile 的 id 号，后续对服务端进行读写操作时需要用到这个数值。
- 第 12 行，simple_central_msg_handler，定义的是客户端 profile 对应的事件回调函数，协议栈需要上报 profile 对应的消息时，会调用该函数。
- 第 13~23 行，是回调函数处理协议栈上传消息的代码。
- 第 27~31 行，定义了客户端 profile 的变量，将 uuid 数组和回调函数赋值给该变量后，调用 gatt_add_cleint 函数将该客户端 profile 添加到协议栈内部去。
- 客户端 profile 添加成功后，协议栈会上传 GAP 事件：GAP_EVT_ALL_SVC_ADDED。

- Profile 的添加只能在初始化的时候进行，一旦 GAP 事件 GAP_EVT_ALL_SVC_ADDED 上传后，用户不能在通过调用 gatt_add_cleint 函数，增加新的 profile。

2.5.4.3. 扫描服务

链接建立后，添加了客户端 profile 的设备，需要扫描服务端来获取需要的 uuid 对应的 handle 号。如果能扫描到客户端 profile 注册的 uuid 对应的 handle 号，说明对端服务包含这个 uuid 对应的属性。可以对该属性进行各种 gatt 读写操作。见如下示例代码：

```
static void app_gap_evt_cb(gap_event_t *p_event)
{
    switch(p_event->type)
    {
        case GAP_EVT_MASTER_CONNECT:
           extern uint8_t client_id;
           gatt_discovery_all_peer_svc(client_id,p_event->param.master_encrypt_conidx);
           break;
    }
}
```

说明：

- 链接建立后，通过判断 GAP 事件类型为 GAP_EVT_MASTER_CONNECT 来获取主机链接已建立的事件处理入口。
- 在该事件下，调用函数 gatt_discovery_all_peer_svc 来扫描对端设备的所有服务。扫描结束时，BLE 协议栈会将扫描结果和扫描结束的状态上传到客户端 profile 注册的 GATT 事件回调函数内，上传的事件是 GATT 事件：GATT_MSG_CMP_EVT，operation = GATT_OP_PEER_SVC_REGISTERED。
- 客户端 profile 对应的 GATT 事件回调处理参见下一节。

2.5.4.4. 客户端 profile 回调函数

为了处理 profile 属性的读写操作，profile 需要定义自己的读写回调函数。回调函数接收协议栈上传的关于这个 profile 的所有消息。以一个客户端 profile 的回调函数实例来解释。该回调函数代码在 SDK_Folder\examples\none_evm\ ble_simple_central\code\ ble_simple_central.c。

```
static uint16_t simple_central_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_NTF_REQ:
            show_reg(p_msg->param.msg.p_msg_data,p_msg->param.msg.msg_len,1);
        break;
        case GATTC_MSG_READ_REQ:
           if(p_msg->att_idx == 2)
            {
                memcpy(p_msg->param.msg.p_msg_data, "\x7\x8\x9", 3);
                return 3;
            }
        break;
        case GATTC_MSG_READ_IND:
            show_reg(p_msg->param.msg.p_msg_data,p_msg->param.msg.msg_len,1);
        break;
        case GATTC_MSG_CMP_EVT:
        {
            if(p_msg->param.op.operation == GATT_OP_PEER_SVC_REGISTERED)
            {
                uint16_t att_handles[3];
                memcpy(att_handles,p_msg->param.op.arg,6);
                gatt_client_enable_ntf_t ntf_enable;
                ntf_enable.conidx = p_msg->conn_idx;
                ntf_enable.client_id = client_id;
                ntf_enable.att_idx = 2; //NF
                gatt_client_enable_ntf(ntf_enable);
            }
        }
        break;
    }
    return 0;

}
```

说明：

- Profile 的 GATT 事件回调函数会传入一个消息变量。该变量包含了消息的类型、链接号、消息对应的 uuid 数组的序号、消息对应的 handle 号、消息的内容。
- GATT 的事件回调函数根据上传的消息的类型分别处理，函数以 return 0 结尾。
- 消息是 GATTC_MSG_NTF_REQ 表示接收到服务端设备发过来的 notification 消息，示例中做了打印处理。
- 消息是 GATTC_MSG_READ_REQ 表示接收到服务端设备发过来的读消息，示例中做了回复，注意：回复完毕之后，需要 return 回复数据的长度，而不是 return 0。
- 消息是 GATTC_MSG_READ_IND 表示对服务端设备进行 read 操作后，接收到服务端设备发过来的 read response 数据，示例中做了打印处理。

- 消息是 GATTC_MSG_CMP_EVT 表示针对客户端 profile 的某个操作完成。p_msg->param.op.operation 代表操作的类型，类型一共有以下几种，参见在 gatt_api.h 内宏定义组 GATT_OPERATION_NAME。

```
#define GATT_OP_NOTIFY                  0x01    //!< GATT notification operation
#define GATT_OP_INDICA                  0x02    //!< GATT indication operation
#define GATT_OP_PEER_SVC_REGISTERED     0x03    //!< Used with GATTC_CMP_EVT, GATT peer device
service registered
#define GATT_OP_WRITE_REQ               0x05    //!< GATT wirte request operation
#define GATT_OP_WRITE_CMD               0x06    //!< GATT wirte command operation, write
without response
#define GATT_OP_READ                    0x07    //!< GATT read operation
#define GATT_OP_PEER_SVC_DISC_END       0x08    //!< Used with GATTC_CMP_EVT, GATT peer device
service discovery is ended
```

- 对客户端 profile 来说，能用到的操作有，0x03、0x05、0x06、0x07、0x08
  a) 0x03 代表，扫描服务步骤中扫描对端服务动作完成，并将扫描到的服务 handle 号注册到协议栈内部。此时，op.arg 指向的是客户端 uuid 数组中的 uuid 对应在服务端的 handle 号。如果 handle 号为 0，表示在服务端没有这个 uuid 的属性。为非 0，表示服务端有这个 uuid 的属性，只能对 handle 号为非 0 的属性进行读写操作。
  b) 0x05 表示，对服务端属性进行写操作（需要 response）完成。
  c) 0x06 表示，对服务端属性进行写操作（不需要 response）完成。
  d) 0x07 表示，对服务端属性进行读操作完成。
  e) 0x08 表示，注册需要的对端服务属性到协议栈内部完成。
- p_msg->param.op.operation 等于 0x03，表示扫描对端服务完成，做为客户端此时一般需要调用 gatt_client_enable_ntf 函数对含 notification 权限的属性进行使能 ntf 的操作。示例中，服务扫描完成后，对 uuid 数组的第 2 个 uuid 对应的属性发送 ntf_enable 包。

客户端 profile 回调函数参数的类型是 gatt_msg_t。解释如下

```
typedef struct
{
    gatt_msg_evt_t  msg_evt;    //!< The event that message come with
    uint8_t         conn_idx;   //!< Connection index
    uint8_t         svc_id;     //!< service id of this message
    uint16_t        att_idx;    //!< Attribute index of in the service table
    uint16_t        handle;     //!< Attribute handle number in peer service
    union
    {
        gatt_msg_hdl_t  msg;    //!< GATT message, length, data pointer
```

```
        gatt_op_cmp_t   op;     //!< GATT operation, read, write, notification, indication
    } param;
} gatt_msg_t;
```

说明：

- msg_evt 元素是消息的类型。参见 gatt_msg_evt_t 的定义。一共有以下几种
  a) GATTC_MSG_READ_REQ，服务端 profile 接收到对端的读操作。
  b) GATTC_MSG_WRITE_REQ，服务端 profile 接收到对端的写操作（write_req 和 write_cmd）。
  c) GATTC_MSG_ATT_INFO_REQ，服务端 profile 接收到对端的读属性 value 长度操作。
  d) GATTC_MSG_NTF_REQ，客户端接收到对端发送的 ntf 数据。
  e) GATTC_MSG_IND_REQ，客户端接收到对端发送的 indication 数据。
  f) GATTC_MSG_READ_IND，客户端接收到对端发送的 read 操作回复的数据。
  g) GATTC_MSG_CMP_EVT，profile 接收到某个事件完成的消息。
  h) GATTC_MSG_LINK_CREATE，profile 被通知有一个链接已经建立。
  i) GATTC_MSG_LINK_LOST，profile 被通知有一个链接已经断开。
  j) GATTC_MSG_SVC_REPORT，客户端接收到扫描到的服务信息。
- conn_idx 元素表示本消息来自于哪一个链接的链接号。
- svc_id，表示本消息对应的 profile 的 id 号，由 profile 添加函数返回。
- att_idx，表示本消息针对哪条属性，对客户端表示 uuid 数组的组号。对服务端表示 att 属性表的组号。
- handle，表示本消息针对的属性在服务端对应的 handle 号。
- param.msg，表示在有数据相关的消息（除了 GATTC_MSG_CMP_EVT、GATTC_MSG_LINK_CREATE、GATTC_MSG_LINK_LOST）上传时的数据 buffer 指针和长度。
- param.op，表示消息类型为 GATTC_MSG_CMP_EVT 时，对应的操作号和完成的状态。注：对应操作为 GATT_OP_PEER_SVC_REGISTERED，param.op.arg 表示扫描到的属性的 handle 号，其他操作情况，无意义。

2.5.4.5. 向服务端进行写操作

以代码在 SDK_Folder\examples\none_evm\ ble_simple_central\code\ ble_simple_central.c 内的客户端 profile 为例，客户端进行写操作的交互流程如下图所示。

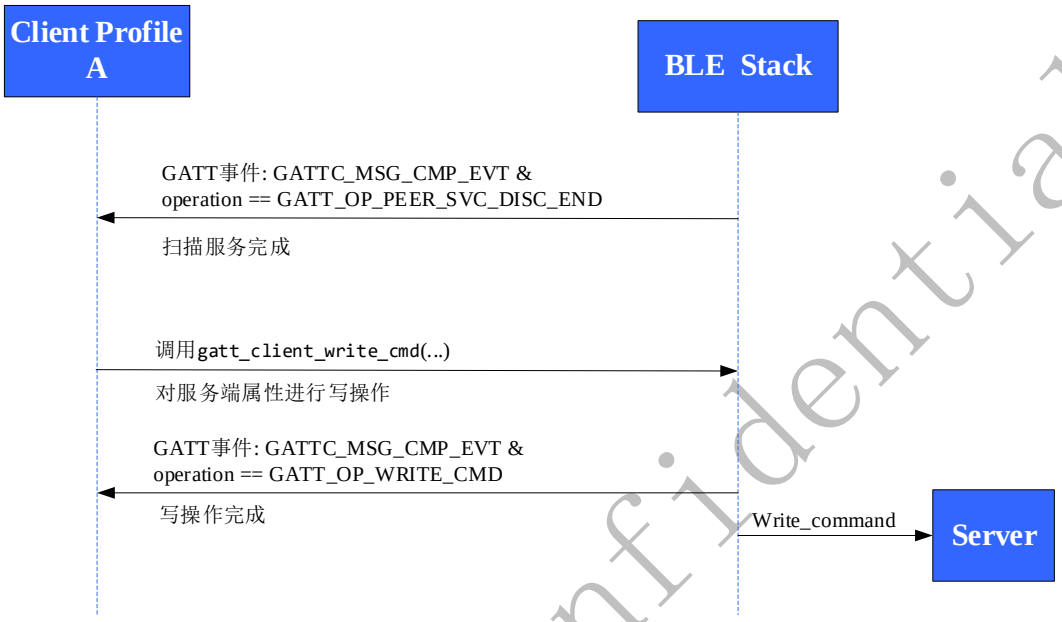

图 2-9 客户端 write_command 处理流程

说明：

- 在收到客户端 profile 上传的 对端服务扫描完成的回调消息后，按上一节 GATT 回调函数内容，需要判断客户端 uuid 数组对应的 handle 号为非 0。只有对非 0 的 uuid 对应的属性才能进行写操作。
- 步骤 1 后，应用层可以调用 GATT 写函数对可写属性进行写操作。写函数定义在 gatt_api.h 内，一共有三种如下：
  a) gatt_client_write_req 表示写操作，需要服务端回复 response。
  b) gatt_client_write_cmd 表示写操作，不需要服务端回复 response。
  c) gatt_client_enable_ntf 表示使能服务端某个属性的 ntf 功能。
- 调用 write_cmd 操作发送数据到协议栈，协议栈会分配一段内存缓存应用层发送的数据，然后立即上传写完成消息到客户端回调函数，消息是 GATTC_MSG_CMP_EVT & operation == GATT_OP_WRITE_REQ 或 GATT_OP_WRITE_CMD。用户可以在回调函数内判断此消息做为 write_cmd 数据发送完成处理。

- 应用层的数据在用户调用 write_req 和 enable_ntf 时，操作完成的消息需要等到对端接收成功后才会上传。所以应用层在上一次操作完成前，对同一个属性只能调用一次 write_req 和 enable_ntf 操作，否则后面的 write_req 和 enable_ntf 操作会无效。
- 应用层的数据在用户调用 write_cmd 发送数据(非 write_req)时，协议栈只是缓存数据到内部，在链路层可以发送的时候将缓存的数据发到对端 BLE 设备。所以应用层在往协议栈发送数据的时候，可以反复调用多次写操作，但要注意系统内存的消耗，一般建议设一个能往协议栈发送数据包的最个数，用一个计数器记录当前发包的个数，每发送完成一次计数器加一，每调写函数往下发一次，就减 1。示例代码如下：

```
uint8_t nb_report = 0;
uint16_t client_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_CMP_EVT:
        {
            if(p_msg->param.op.operation == GATT_OP_WRITE_CMD )
            {
                nb_report--;
                write_att_cmd(p_msg->conidx, client_id,p_data, data_len );
            }
        }
        break;
    }
}
void write_att_cmd(uint8_t conidx, uint8_t client_id,uint8_t *p_data, uint16_t data_len )
{
   if(nb_report < 20)
   {
      gatt_client_write_t write;
      write.conidx = conidx;
      write.client_id = client_id;
      write.att_idx =1; //RX
      write.p_data = p_data;
      write.data_len =data_len;
      gatt_client_write_cmd(write);
      nb_report++;
    }
}
```

2.5.4.6. 向服务端进行读操作

以代码在 SDK_Folder\examples\none_evm\ ble_simple_central\code\ ble_simple_central.c 内的客户端 profile 为例，客户端进行读操作的交互流程如下图所示。

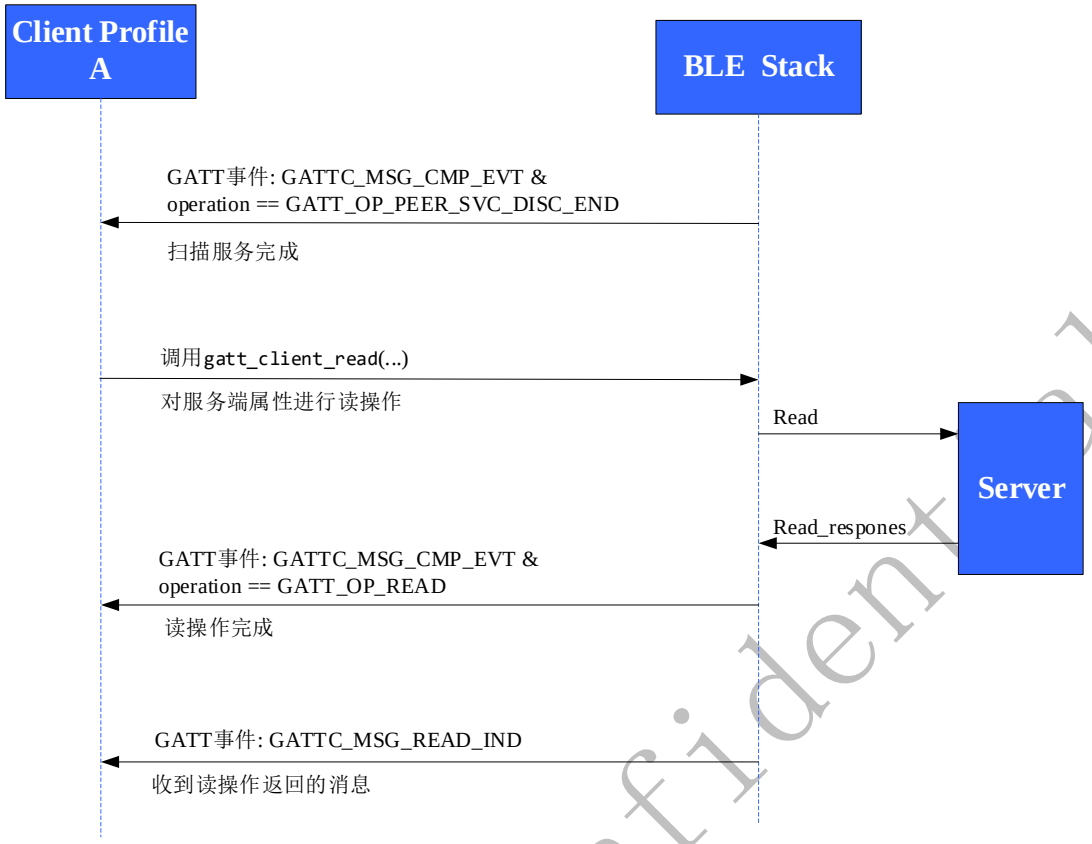

图 2-10 客户端 read 处理流程

说明：

- 在收到客户端 profile 上传的 对端服务扫描完成的回调消息后，按上一节 GATT 回调函数内容，需要判断客户端 uuid 数组对应的 handle 号为非 0。只有对非 0 的 uuid 对应的属性才能进行读操作。
- 步骤 1 后，应用层可以调用 GATT 写函数对可写属性进行写操作。读函数为 gatt_client_read，定义在 gatt_api.h 内。
- 调用 read 函数发起读操作后，操作完成的消息需要等到对端接收成功，然后回复 read_response 后才会上传。消息是 GATTC_MSG_CMP_EVT & operation == GATT_OP_ READ。用户可以在回调函数内判断此消息做为 read 操作完成。
- Read 操作完成的消息上报之后，协议栈会再次上传 GATT 消息 GATTC_MSG_READ_IND，该消息表示接收到对端回复的数据。

2.5.5. GATT 服务端基础流程

2.5.5.1. 服务端

GATT 服务端负责接收对端设备 GATT 客户端发送的命令以及请求，并且根据收到的命令以及请求回复对端设备响应，或者向对端设备发送通知或指示。GATT 服务端的功能需要和各个 profile 紧密配合才能够完成。

下面章节将描述如何在 GATT 服务端添加 profile 和注册 profile 回调函数，以及 GATT 服务端如何处理客户端的读/写请求。关于 API 的详细描述，请参考 gatt_api.h 内函数定义说明。服务端 GATT 基础流程如下图所示。

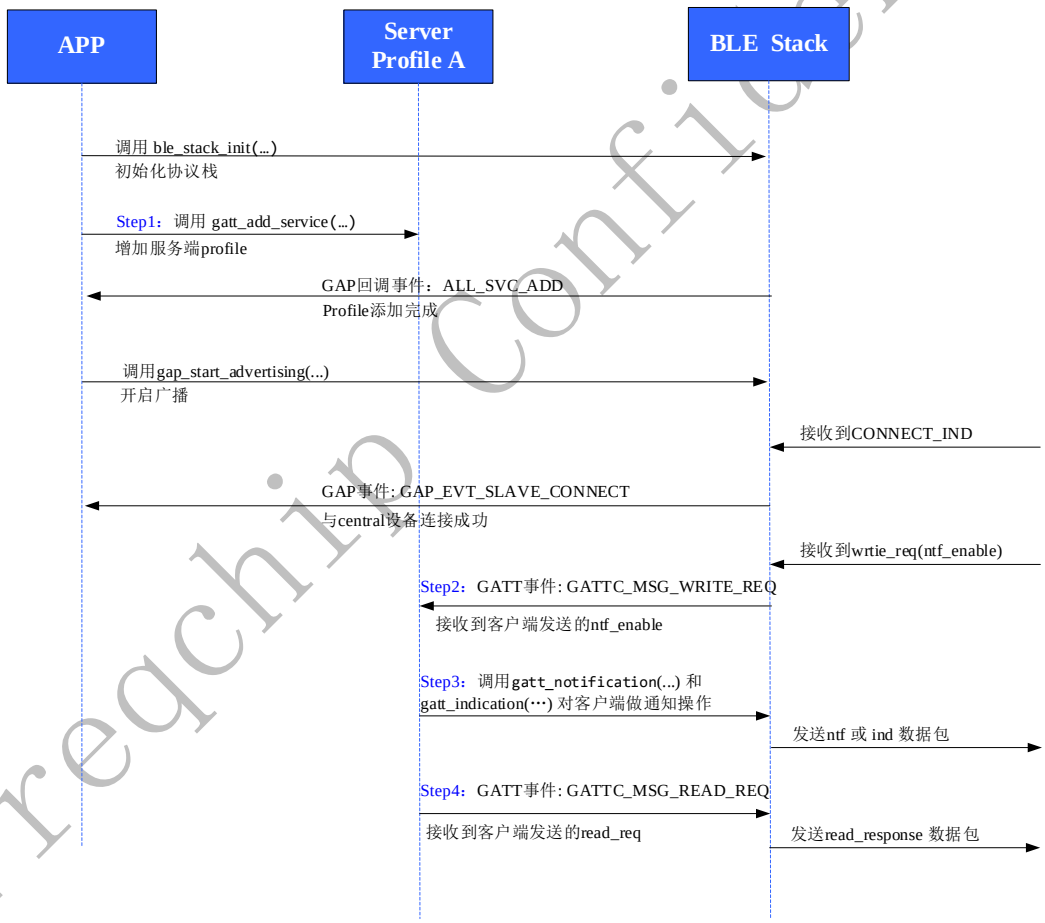

图 2-11 服务端 GATT 基础流程

2.5.5.2. 增加服务端 profile

针对服务端 profile，用户需要给出一个属性数组 att_table 和 profile 的回调函数，注册一个服务端的 profile，便于协议栈上传消息，和侦听特定的属性的消息。以一个服务端 profile 注册的实例来解释该过程。

```
uint8_t svc_id = 0;
const uint8_t svc_group_uuid[] =
"\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
const gatt_attribute_t att_db[] =
{
    [0] = { {UUID_SIZE_2,UUID16_ARR(GATT_PRIMARY_SERVICE_UUID)}
        ,GATT_PROP_READ,UUID_SIZE_16,(uint8_t *)svc_group_uuid
    },

    [1] = { {UUID_SIZE_2,UUID16_ARR(GATT_CHARACTER_UUID)}
        ,GATT_PROP_READ,0,NULL
    },
    [2] = { {UUID_SIZE_16,  {0x00, 0x00, 0x48, 0x43, 0x45, 0x54, 0x43, 0x49, 0x47, 0x4F, 0x4C,
0x49, 0x04, 0x18, 0x00, 0x00,}}
        ,GATT_PROP_READ/GATT_PROP_NOTI/GATT_PROP_INDI,20,NULL
    },
    [3] = { {UUID_SIZE_2,UUID16_ARR(GATT_CLIENT_CHAR_CFG_UUID)}
        ,GATT_PROP_READ/GATT_PROP_WRITE,0,NULL
    },
    [4] = { {UUID_SIZE_2,UUID16_ARR(GATT_CHAR_USER_DESC_UUID)}
        ,GATT_PROP_READ/GATT_PROP_WRITE,0xC,NULL
    },
};
uint16_t svc_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_READ_REQ:
        break;
     }
}
void user_main(void)
{
    gatt_service_t service;
    service.p_att_tb = att_db;
    service.att_nb = 5;
    service.gatt_msg_handler = svc_msg_handler;
    svc_id=gatt_add_service(&service);
}
```

说明：

- att_tb[]是一个服务端 profile 属性数组，属性表定义参见 2.5.3.1。
- 第 1 行，svc_id 变量记录这个服务端 profile 的 id 号，后续对客户端进行 ntf/ind 操作时需要用到这个数值。

- 第 22 行，svc_msg_handler，定义的是服务端 profile 对应的事件回调函数，协议栈需要上报 profile 对应的消息时，会调用该函数。
- 第 23~27 行，是回调函数处理协议栈上传消息的代码。
- 第 32~36 行，定义了服务端 profile 的变量，并将属性表数组和回调函数赋值给该变量后，调用 gatt_add_service 函数将该服务端 profile 添加到协议栈内部去。
- 用户可以在初始化时，通过 gatt_add_service 函数反复添加多个服务端 profile 到协议栈内部。
- 所有的 profile（包括客户端和服务端）添加成功后，协议栈会上传 GAP 事件：GAP_EVT_ALL_SVC_ADDED。
- Profile 的添加只能在初始化的时候进行，一旦 GAP 事件 GAP_EVT_ALL_SVC_ADDED 上传后，用户不能在通过调用 gatt_add_service 函数，增加新的 profile。

2.5.5.3. 等待客户端发送 ntf/ind 使能

链接建立后，添加了服务端 profile 的设备，在收到客户端发送的 notification/indication enable 的消息后，就可以向客户端发送 ntf/ind 的消息。见如下示例代码：

```
uint16_t svc_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_WRITE_REQ:
            if(p_msg->att_idx == 3)
            {
                if( *(uint8_t *)(p_msg->param.msg.p_msg_data) & 0x1 )
                    ntf_enable[p_msg->conn_idx] = true;
                if( *(uint8_t *)(p_msg->param.msg.p_msg_data) & 0x2)
                    ind_enable[p_msg->conn_idx] = true;
            }
         break;
    }
}
```

说明：

- 链接建立后，通过判断 GATT 事件类型为 GATTC_MSG_WRITE_REQ 来获取客户端写数据的事件处理入口。通过判断写入数据的第一个字节的置位判断 ntf/ind 使能。
- 服务端 profile 对应的 GATT 事件回调处理参见下一节。

2.5.5.4. 服务端 profile 回调函数

为了处理 profile 属性的读写操作，profile 需要定义自己的读写回调函数。回调函数接收协议栈上传的关于这个 profile 的所有消息。以一个服务端 profile 的回调函数实例来解释。

```
uint16_t svc_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_READ_REQ:
            if(p_msg->att_idx == 4)
            {
                memcpy(p_msg->param.msg.p_msg_data, "\x7\x8\x9", 3);
                return 3;
            }
            break;
        case GATTC_MSG_WRITE_REQ:
            if(p_msg->att_idx == 4)
                show_reg(p_msg->param.msg.p_msg_data,p_msg->param.msg.msg_len,1);
            break;
        case GATTC_MSG_CMP_EVT:
            if(p_msg->att_idx == GATT_OP_NOTIFY)
                co_printf("ntf done\r\n");
            break;
        case GATTC_MSG_LINK_CREATE:
            co_printf("ntf done\r\n");
            break;
        case GATTC_MSG_LINK_LOST:
            co_printf("ntf done\r\n");
            break;
        default:
            break;
    }
    return 0;
};
```

说明：

- Profile 的 GATT 事件回调函数会传入一个消息变量。该变量包含了消息的类型、链接号、消息对应的 uuid 数组的序号、消息对应的 handle 号、消息的内容。
- GATT 的事件回调函数根据上传的消息的类型分别处理，函数以 return 0 结尾。
- 消息是 GATTC_MSG_READ_REQ 表示接收到服务端设备发过来的读消息，示例中做了回复，注意：回复完毕之后，需要 return 回复数据的长度，而不是 return 0。
- 消息是 GATTC_MSG_WRITE_REQ 表示接收到服务端设备发过来的写消息，示例中做了打印处理。

- 消息是 GATTC_MSG_CMP_EVT 表示针对服务端 profile 的某个操作完成。p_msg->param.op.operation 代表操作的类型，类型一共有以下几种，参见在 gatt_api.h 内宏定义组 GATT_OPERATION_NAME。

```
#define GATT_OP_NOTIFY                  0x01    //!< GATT notification operation
#define GATT_OP_INDICA                  0x02    //!< GATT indication operation
#define GATT_OP_PEER_SVC_REGISTERED     0x03    //!< Used with GATTC_CMP_EVT, GATT peer device
service registered
#define GATT_OP_WRITE_REQ               0x05    //!< GATT wirte request operation
#define GATT_OP_WRITE_CMD               0x06    //!< GATT wirte command operation, write
without response
#define GATT_OP_READ                    0x07    //!< GATT read operation
#define GATT_OP_PEER_SVC_DISC_END       0x08    //!< Used with GATTC_CMP_EVT, GATT peer device
service discovery is ended
```

对服务端 profile 来说，能用到的操作有，0x01、0x02。

- 0x01 表示，对客户端属性进行 notification 操作（不需要 response）完成。
- 0x02 表示，对客户端属性进行 indication 操作（需要 response）完成。

服务端 profile 回调函数参数的类型是 gatt_msg_t。与客户端 profile 输入参数的解释相同。参见章节 2.5.4.4。

2.5.5.5. 向客户端进行 ntf/ind 操作

向客户端 profile 进行 ntf/ind 操作的交互流程如下图 2-5-8 所示。

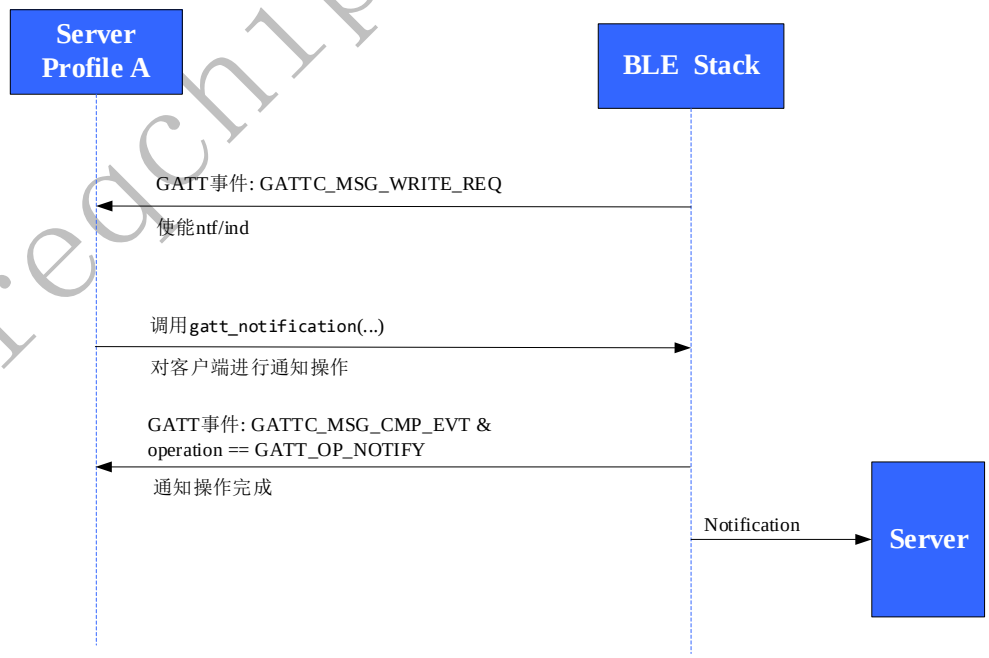

图 2-12 服务端 notification 处理流程

说明：

- 在收到服务端 profile 上传的 GATT 消息 GATTC_MSG_WRITE_REQ 后，判断写入数据的第一个字节 bit 0 置位，确定 ntf 使能。判断写入数据的第一个字节 bit 1 置位，确定 ind 使能。
- Ntf 使能后，应用层可以调用 GATT 写函数对被使能 ntf 的属性进行通知操作。通知函数定义在 gatt_api.h 内，一共有两种如下：
  a) gatt_notification 表示通知操作，不需要服务端回复 response。
  b) gatt_indication 表示指示操作，需要服务端回复 response。
- 调用 notification 操作发送数据到协议栈，协议栈会分配一段内存缓存应用层发送的数据，然后立即上传 ntf 完成消息到服务端回调函数，消息是 GATTC_MSG_CMP_EVT & operation == GATT_OP_NOTIFY。用户可以在回调函数内判断此消息做为 notification 数据发送完成处理。
- 调用 indication 操作发送数据到协议栈，操作完成的消息需要等到对端接收成功后才会上传。所以应用层在上一次操作完成前，对同一个属性只能调用一次 indication 操作，否则后面的 indication 操作会无效。
- 应用层的数据在用户调用 notification 发送数据时，协议栈只是缓存数据到内部，在链路层可以发送的时候将缓存的数据发到对端 BLE 设备。所以应用层在往协议栈发送数据的时候，可以反复调用多次写操作，但要注意系统内存的消耗，一般建议设一个能往协议栈发送数据包的最个数，用一个计数器记录当前发包的个数，每发送完成一次计数器加一，每调写函数往下发一次，就减 1。示例代码如下：

```
uint8_t nb_report = 0;
uint16_t svc_msg_handler (gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_CMP_EVT:
        {
            if(p_msg->param.op.operation == GATT_OP_NOTIFY )
            {
                nb_report--;
                att_ntf(p_msg-> conn_idx,svc_id,p_data, data_len );
            }
        }
        break;
    }
}
void att_ntf( uint8_t conidx, uint8_t svc_id,uint8_t *p_data, uint16_t data_len )
{
```

```
    if(nb_report < 20 && ntf_enable[condix])
    {
        gatt_ntf_t ntf_att;
        ntf_att.conidx = conidx;
        ntf_att.svc_id = svc_id;
        ntf_att.att_idx = 2;
        ntf_att.p_data = p_data;
        ntf_att.data_len =data_len;
        gatt_notification (ntf_att);
        nb_report++;
    }
}
```

2.5.5.6. 向客户端回应读操作

服务端接收到读请求后，向客户端回复数据的交互流程如下图 2-5-9 所示。

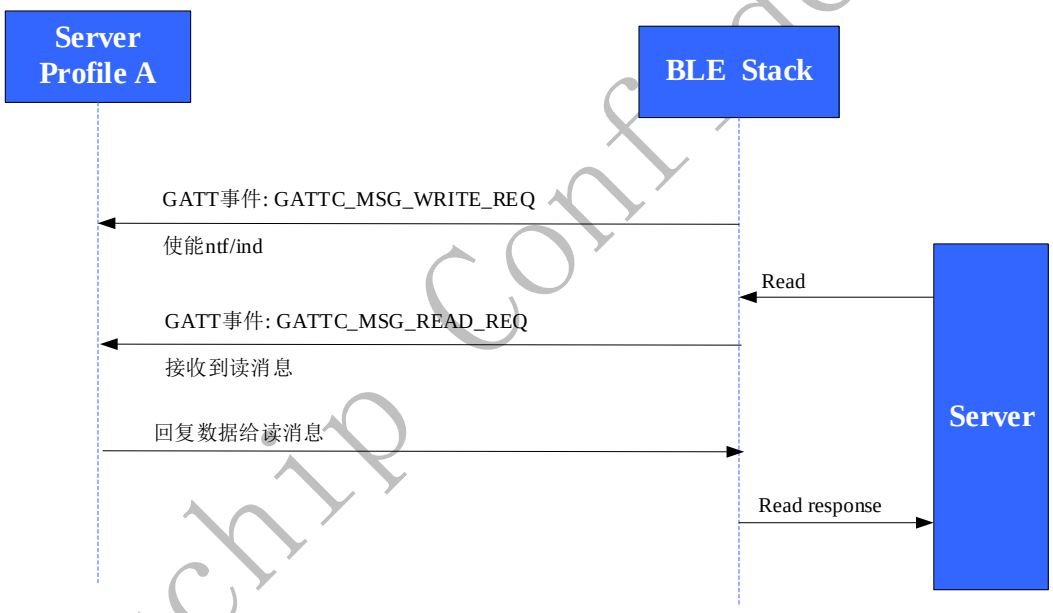

图 2-13 服务端 read response 处理流程

说明：

- 在收到服务端 profile 上传的 GATT 消息 GATTC_MSG_WRITE_REQ 后，并判断 ntf/ind 使能后，确定客户端已经正确的扫描到了服务端的属性 handle 号。
- 收到服务端 profile 上传的 GATT 消息 GATTC_MSG_READ_REQ，表明客户端需要读某个属性的数据，服务端需要对这个读操作进行回复，如下代码。

```
uint16_t svc_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_READ_REQ:
            if(p_msg->att_idx == 2)
```

```
            {
                memcpy(p_msg->param.msg.p_msg_data, "\x7\x8\x9", 3);
                return 3;
            }
        break;
    }
}
```

首先判断读消息针对的属性在属性表中的序号是 2，然后将要回复的读数据直接拷贝到回调函数参数的数据指针指向的 buffer，最后返回拷贝数据的长度即可。回调函数返回后，协议栈会发送 read reponse 包给客户端。

2.6. 安全管理

2.6.1. 配对

配对过程，即生成和分发密钥的过程，可以分为 3 个阶段：

1. 交换配对信息。
2. 生成链路加密密钥。
3. 在加密链路上分发其它指定的密钥信息，并根据设备是否支持绑定决定是否需要在安全数据库中存储分发的密钥信息。

配对流程图如下所示，配对方式分成 LE Legacy Pairing 和 LE Secure Connection（从 core 4.2 版本添加进来）两种，二者区别主要在于 step2-生成密钥的过程，另外在该步骤，LE Legacy Pairing 生成 STK，LE Secure Connection 生成 LTK。

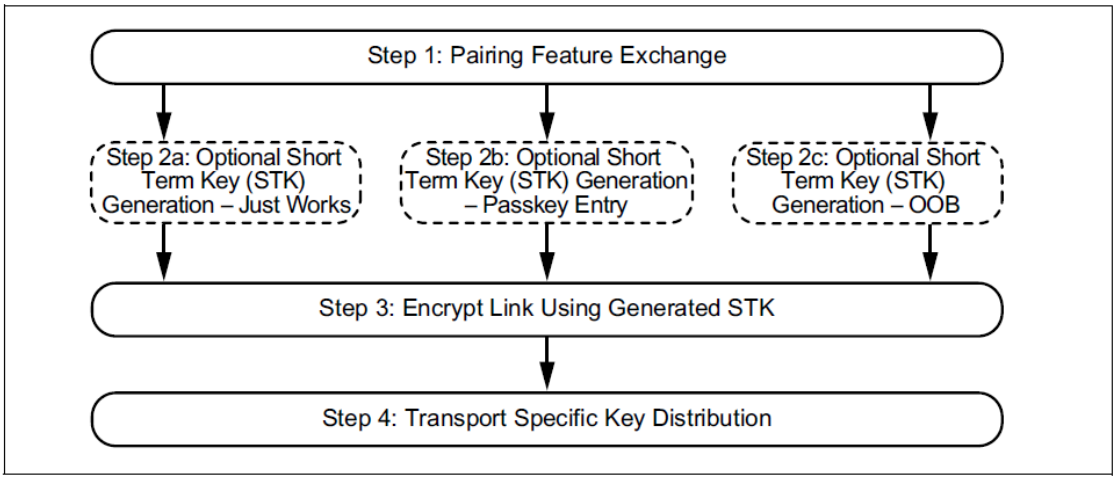

图 2-14 BLE 配对过程

LE Secure Connection 中引入了 Elliptic Curve Diffie-Hellman 加密算法，使得安全性得到了很大的提升。根据设备所支持的安全特性，配对方法可分为如下四种：

- Just Works（Secure Connections or Legacy），适合于配对双方没有输入输出 io 设备（比如：键盘输入或显示器输出）的情况，由于在 Just Works 配对过程中无需 MITM 认证，因此也就无法抵御 MITM 攻击。Just Works 配对既可以用于 Legacy Pairing，也可以用于 Secure Connection。
- Passkey Entry（Secure Connections or Legacy），配对流程支持 MITM 认证，既可用于 Legacy Pairing，也可以用于 Secure Connection。配对过程中需要用户输入 6 位十进制数的密码，作为配对过程所需的输入参数。
- Numeric Comparison（Secure Connections），如果双方设备都支持 Secure Connection，IO 能力都具有显示和输入功能，且需带 MITM 认证，则可以使用 Numeric Comparison 配对流程。
- Out of Band（Secure Connections or Legacy），该方式通过 NFC 等带外方式交互配对过程需要的中间值。目前版本的 SDK 中未支持该配对方式。

2.6.1.1. JustWorks 配对

如果双方设备都无需带中间人（Man-in-the-middle，MITM）认证，那么就可以使用 Just Works 配对流程。由于在 Just Works 配对过程中无需 MITM 认证，因此也就无法抵御 MITM 攻击。Just Works 配对既可以是 Legacy Pairing 也可以是 Secure Connection，用户只需在配对流程开始之前配置好安全参数，配对流程启动后，配对过程中无需用户的任何交互。

下面以 Legacy Pairing、Justworks 配对方式为例，介绍参数如何配置：

```
gap_security_param_t param =
{
    .mitm = false,
    .ble_secure_conn = false,
    .io_cap = GAP_IO_CAP_NO_INPUT_NO_OUTPUT,
    .pair_init_mode = GAP_PAIRING_MODE_WAIT_FOR_REQ,
    .bond_auth = true,
    .password = 0,
};
gap_security_param_init(& param);
```

说明：

- mitm 参数表示是否启动带中间人的认证，设为 false 表示不启用。
- ble_secure_conn 参数表示是否启动 security connection。设为 false 表示不启用。
- io_cap 表示本设备的 io 输出能力，是否有显示或键盘输入的能力。本示例中没有 io 能力
- pair_init_mode 参数表示是否使能配对。示例中使能配对。
- bond_auth 参数表示是否启用绑定，也就是配对完成之后是不是执行密钥分发，用于下次的连接。设为 true 表示启用绑定。
- password 参数设置用于 mitim 中间人认证的认证码，需要对端设备输入认证码用于认证。只有在 io_cap 含输出能力且 mitm 使能时，才有用。

2.6.1.2. Passkey Entry 配对

Passkey Entry 配对流程支持 MITM 认证，配对既可以是 Legacy Pairing 也可以是 Secure Connection。配对过程中拥有 display 输出能力的一方显示 6 位十进制密码，拥有 keyboard 输入能力的一方输入该密码来作为 temp key，进行后续的配对流程。

下面以具有输出能力的一种配置介绍 Passkey entry 配对模式的参数设置：

```
gap_security_param_t param =
{
    .mitm = true,
    .ble_secure_conn = false,
    .io_cap = GAP_IO_CAP_DISPLAY_ONLY,
    .pair_init_mode = GAP_PAIRING_MODE_WAIT_FOR_REQ,
    .bond_auth = true,
    .password = 123456,
};
gap_security_param_init(& param);
```

说明：

- mitm 参数表示是否启动带中间人的认证，设为 true 表示启用。
- ble_secure_conn 参数表示是否启动 security connection。设为 false 表示不启用。
- io_cap 表示本设备的 io 输出能力，是否有显示或键盘输入的能力。本示例中有 io 显示能力，能提示需要对端输入认证码。
- pair_init_mode 参数表示是否使能配对。示例中使能配对。
- bond_auth 参数表示是否启用绑定。设为 true 表示启用绑定。

- password 参数设置用于 mitim 中间人认证的认证码，需要对端设备输入认证码用于认证。

对于具有输入能力的设备，Passkey entry 配对模式参数设置如下：

```
gap_security_param_t param =
{
    .mitm = true,
    .ble_secure_conn = false,
    .io_cap = GAP_IO_CAP_KEYBOARD_ONLY,
    .pair_init_mode = GAP_PAIRING_MODE_WAIT_FOR_REQ,
    .bond_auth = true,
    .password = 123456,
};
gap_security_param_init(& param);
```

在对端显示出来待输入的密码时，可以调用如下函数来传递该密码给协议栈进行后续的配对流程：

```
void gap_security_send_pairing_password(uint32_t conidx,uint32_t value)
```

说明：

- conidx 为该连接的索引。
- value 为该密码。

2.6.1.3. Security Connections 配对

如果双方设备都支持 Secure Connection 配对，可以使用 Secure Connection 配对流程。具体的安全参数配置可参考如下：

```
gap_security_param_t param =
{
    .mitm = false,
    .ble_secure_conn = true,
    .io_cap = GAP_IO_CAP_NO_INPUT_NO_OUTPUT,
    .pair_init_mode = GAP_PAIRING_MODE_WAIT_FOR_REQ,
    .bond_auth = true,
    .password = 0,
};
gap_security_param_init(& param);
```

说明：

- mitm 参数表示是否启动带中间人的认证，设为 false 表示不启用。
- ble_secure_conn 参数表示是否启动 security connection。设为 true 表示启用。

- io_cap 表示本设备 io 能力，是否有显示或键盘输入的能力。本示例中没有 io 能力。
- pair_init_mode 参数表示是否使能配对。示例中使能配对。
- bond_auth 参数表示是否启用认证。设为 true 表示启用认证。
- password 参数设置用于 mitim 中间人认证的认证码，需要对端设备输入认证码用于认证。只有在 io_cap 含输出能力且 mitm 使能时，才有用。

2.6.1.4. 禁止配对

配对禁止的参数设置如下

```
gap_security_param_t param =
{
    .mitm = false,
    .ble_secure_conn = false,
    .io_cap = GAP_IO_CAP_NO_INPUT_NO_OUTPUT,
    .pair_init_mode = GAP_PAIRING_MODE_NO_PAIRING,
    .bond_auth = false,
    .password = 0,
};
gap_security_param_init(& param);
```

说明：

- pair_init_mode 参数表示是否使能配对。示例中设为 no paring，禁止配对。
- bond_auth 参数表示是否启用绑定。设为 false 表示不启用绑定。

2.6.1.5. 配对 key 的保存

如果用户程序需要保存绑定信息、对端服务信息等内容，在协议栈运行之前需要先调用绑定管理初始化函数。

```
void gap_bond_manager_init(uint32_t flash_addr,uint32_t svc_flash_addr,uint8_t max_dev_num,bool enable);
```

说明：

- 函数输入参数 flash_addr 表示用于存储配对产生的 key 的 flash 地址。需要是 0x1000 的整数倍，SDK 中预留一个 sector（4KB）用于存储相关内容。
- svc_flash_addr 表示该设备作为客户端时，扫描到服务端 profile 的 flash 存储地址。需要是 0x1000 的整数倍，SDK 中预留一个 sector（4KB）用于存储相关内容。

- max_dev_num 表示绑定管理功能支持的最大的绑定设备个数。取址范围[1,20]。不同 mac 地址的设备占用一个设备数，如果存储的设备数超过该值，会从 0 开始覆盖第 0 个设备的存储信息。
- enable 参数表示是否存储配对信息。设为 true，协议栈会自动存储配对产生的 key 到 flash。设为 false，协议栈不会存储配对产生的 key。
- 在整个工程初始化时调用本函数，会从输入参数指定的 flash 地址读取所有存储的 key 和服务信息到内部变量。

2.6.1.6. 主机发起配对

主机发起配对操作后，应用层与 BLE 协议栈的交互过程如下图所示。

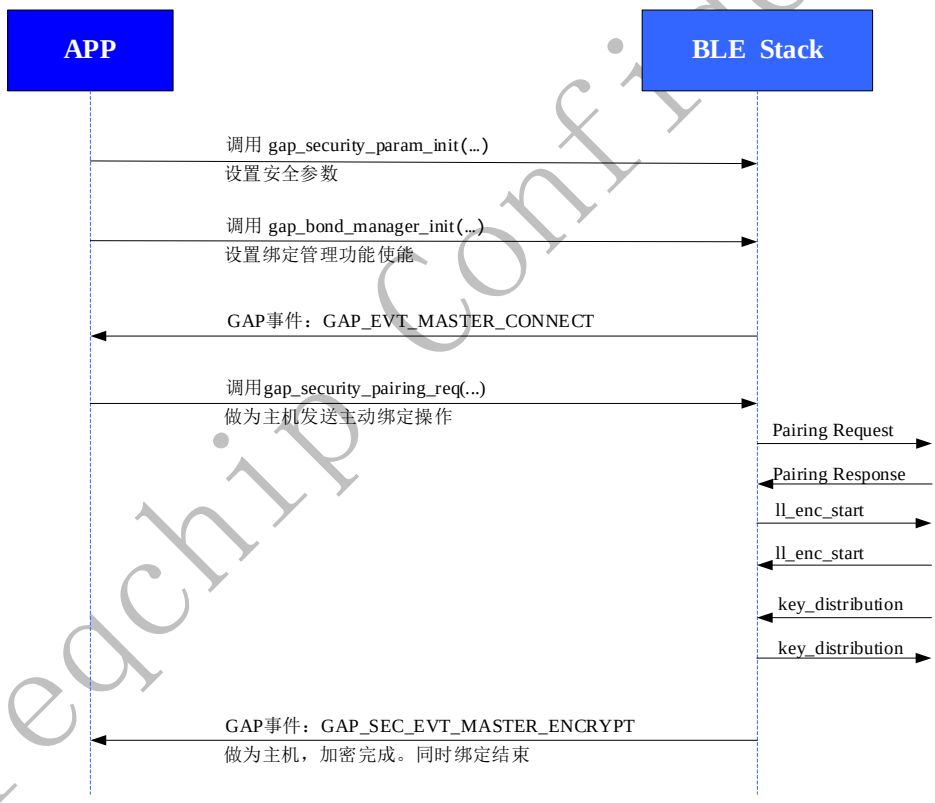

图 2-15 主机发起配对操作

主机开启配对操作流程的具体步骤如下：

```
void proj_ble_gap_evt_func(gap_event_t *event)
{
    switch(event->type)
    {
        case GAP_EVT_MASTER_CONNECT:
        {
            if (gap_security_get_bond_status())
                gap_security_enc_req(p_event->param.master_connect.conidx);
```

```
            else
                gap_security_pairing_req(p_event->param.master_connect.conidx);
        }
        break;
        case GAP_SEC_EVT_MASTER_ENCRYPT:
            co_printf("master[%d]_encrypted\r\n",event->param.master_encrypt_conidx);
        break;
    }
}
```

说明：

- 在 GAP 事件回调函数内，通过判断事件类型为 GAP_EVT_MASTER_CONNECT 来获取主机连接事件处理的入口。在该事件下面调用 gap_security_get_bond_status() 判断当前链接的对端设备是否是已绑定设备，如果是未绑定，调用 gap_security_pairing_req()函数发起绑定动作。
- 绑定之后会自动对链接进行加密，加密成功后，BLE 协议栈会上传 GAP 事件 GAP_SEC_EVT_MASTER_ENCRYPT。

2.6.1.7. 从机进行配对

从机需要主动发起配对请求时，应用层与 BLE 协议栈的交互过程如下图所示

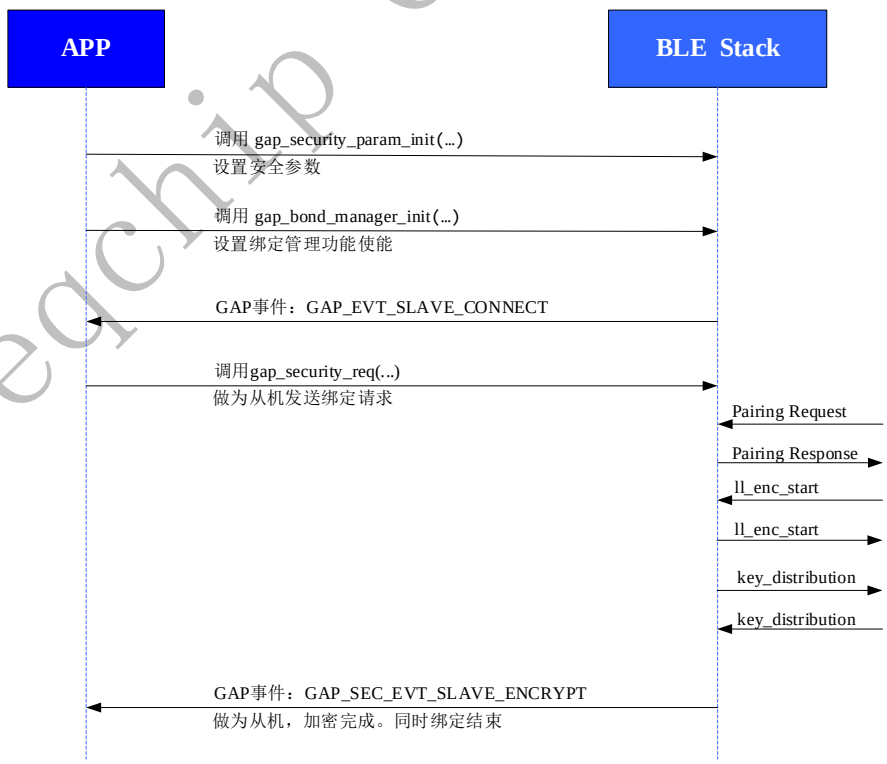

图 2-16 从机发起配对操作

从机开启配对请求流程的具体步骤如下：

```
void proj_ble_gap_evt_func(gap_event_t *event)
{
    switch(event->type)
    {
        case GAP_EVT_SLAVE_CONNECT:
        {
            gap_security_req(event->param.slave_connect.conidx);
        }
        break;
        case GAP_SEC_EVT_SLAVE_ENCRYPT:
            co_printf("slave[%d]_encrypted\r\n",event->param.slave_encrypt_conidx);
            break;
    }
}
```

说明：

- 在 GAP 事件回调函数内，通过判断事件类型为 GAP_EVT_SLAVE_CONNECT 来获取从机连接事件处理的入口。在该事件下面调用 gap_security_req 函数请求主机发起绑定动作。
- 绑定之后会自动对链接进行加密，加密成功后，BLE 协议栈会上传 GAP 事件 GAP_SEC_EVT_SLAVE_ENCRYPT。

2.6.2. 加密

设备进行做配对之后，只要开启了绑定管理功能。可以直接调用加密函数对已建立的链接进行加密操作。加密操作是主机在链接建立后发起的。

2.6.2.1. 主机发起加密

```
void proj_ble_gap_evt_func(gap_event_t *event)
{
    switch(event->type)
    {
        case GAP_EVT_MASTER_CONNECT:
        {
            if (gap_security_get_bond_status())
                gap_security_enc_req(p_event->param.master_connect.conidx);
            else
                gap_security_pairing_req(p_event->param.master_connect.conidx);
        }
        break;
        case GAP_SEC_EVT_MASTER_ENCRYPT:
            co_printf("master[%d]_encrypted\r\n",event->param.master_encrypt_conidx);
```

```
            break;
        }
    }
```

说明：

- 在 GAP 事件回调函数内，通过判断事件类型为 GAP_EVT_MASTER_CONNECT 来获取主机连接事件处理的入口。在该事件下面调用 gap_security_get_bond_status()判断当前链接的对端设备是否是已绑定设备，如果是绑定设备，调用 gap_security_enc_req()函数发起加密动作。
- 加密成功后，BLE 协议栈会上传 GAP 事件 GAP_SEC_EVT_MASTER_ENCRYPT。

2.6.2.2. 从机响应加密

```
void proj_ble_gap_evt_func(gap_event_t *event)
{
    switch(event->type)
    {
        case GAP_EVT_SLAVE_CONNECT:
        {

        }
        break;
        case GAP_SEC_EVT_SLAVE_ENCRYPT:
            co_printf("slave[%d]_encrypted\r\n",event->param.slave_encrypt_conidx);
            break;
    }
}
```

说明：

- 在 GAP 事件回调函数内，通过判断事件类型为 GAP_EVT_SLAVE_CONNECT 来获取从机连接事件处理的入口。
- 加密成功后，BLE 协议栈会上传 GAP 事件 GAP_SEC_EVT_SLAVE_ENCRYPT。

2.6.3. 隐私管理

BLE 中的隐私管理，是指已认证设备可以跟踪识别目标设备，而其他非认证设备无法跟踪识别目标设备。隐私管理使得已认证的设备可以正常地与目标设备进行连接和通讯，同时防止其他非认证设备、恶意破坏设备对目标设备的跟踪。

2.6.3.1. 开启隐私管理功能

系统初始化时可以设置白名单，可以在广播和主动链接时过滤掉白名单以外的设备。开启白名单管理功能的具体步骤如下：

1. 主设备与从设备进行连接并绑定，绑定过程中需要交换 IRK 以及身份地址信息，绑定之后主设备与从设备断开连接。绑定的具体流程请参考 2.6.1 节。
2. 白名单配置。用户可以在系统初始化过程中，发送广播、发起主动连接之前设置白名单。

```
gap_bond_manager_init(BLE_BONDING_INFO_SAVE_ADDR,BLE_REMOTE_SERVICE_SAVE_ADDR,1,true);
gap_bond_info_t info;
gap_bond_manager_get_info(0,&info);
if(info.bond_flag)
{
    gap_mac_addr_t mac_set[1];
    memcpy(&mac_set[0],&info.peer_addr,sizeof(gap_mac_addr_t));
    gap_set_wl(&mac_set[0],1);
    gap_ral_t ral_set[1];
    memcpy(&ral_set[0].addr,&info.peer_addr,sizeof(gap_mac_addr_t));
    memcpy(ral_set[0].peer_irk,info.peer_irk,16);
    gap_set_ral(&ral_set[0],1);
}
```

说明：

- 第 1 行，调用绑定管理初始化函数 gap_bond_manager_init 使能绑定管理功能。
- 第 2、3 行，调用 gap_bond_manager_get_info 获取序号为 0 的绑定设备的绑定信息，包含 mac 地址和 IRK。
- 第 4 行，判断绑定设备的绑定状态为 true 时，将其设置到白名单中
- 第 6、7、8 行，拷贝该绑定设备的 mac 地址和类型，并调用 gap_set_wl 设置该设备 mac 地址和类型到 whitelist 中。

- 第 9 至 12 行，拷贝该绑定设备的 IRK 值，并调用 gap_set_ral 函数设置该设备的 IRK 值到 ral(resolved address list)中。
- 如果绑定的设备地址是非 resolved address 类型，即 public 或 private 固定地址，则不需要执行 9 至 12 行内容设置 ral(resolved address list)。

3. 广播设备使用白名单，广播可以通过设置广播参数中的过滤政策来使能白名单过滤功能。

```
gap_adv_param_t adv_param;
adv_param.adv_filt_policy = GAP_ADV_ALLOW_SCAN_WLST_CON_WLST;
adv_param.adv_mode = GAP_ADV_MODE_UNDIRECT;
adv_param.disc_mode = GAP_ADV_DISC_MODE_GEN_DISC;
adv_param.adv_addr_type = GAP_ADDR_TYPE_PUBLIC;
adv_param.adv_chnl_map = GAP_ADV_CHAN_ALL;
adv_param.adv_intv_min = 0x40;
adv_param.adv_intv_max = 0x40;
gap_set_advertising_param(&adv_param);
```

说明：

- 具体的广播发起流程参见 2.2 章节。
- 示例代码中第 2 行设置了广播的过滤选项为扫描和连接只响应白名单内的设备。设置之后，只有步骤 2 白名单内的设备能扫描和连接本广播。

4. 主动连接使用白名单，用户可以调用针对白名单设备的发起主动连接的函数，只连接白名单内的设备。

```
mac_addr_t peer_addr;
memcpy(peer_addr.addr,"\x00\x00\x00\x00\x00\x00",6);
gap_start_conn_whitelist(&peer_addr,0,6,6,0,400);
```

说明：

- 调用函数 gap_start_conn_whitelist 去自动连接白名单内的设备，这个连接动作不会自动停止，即使连上了某个 BLE 设备。除非用户调用 gap_stop_conn 函数停止该动作。
- 使用白名单连接函数之前，必须要执行第 2 步设置白名单才行。

2.6.3.2. 地址配置说明

当开启广播、扫描、建立连接时，controller 会使用用户设置的本机地址发送空口数据包。本机地址在初始化时进行设置一次即可。广播、扫描、主动连接时均采用初始化时设置的本机地址做为设备地址。

1. BLE mac 地址类型，本机地址一共有 4 种类型，参见 ble_stack.h 中的定义

```
enum ble_addr_type
{
//addr is set by user,value is fixed forever
    BLE_ADDR_TYPE_PUBLIC,
//addr is set by user,value is fixed during power on lifetime
    BLE_ADDR_TYPE_PRIVATE,
//addr is generated by stack with IRK
    BLE_ADDR_TYPE_RANDOM_RESOVABLE,
//addr is generated by stack and is randomly
    BLE_ADDR_TYPE_RANDOM_NONE_RESOVABLE,
};
```

说明：

- PUBLIC 类型，表示设备的 mac 地址会一直不变，该类型地址需要向 IEEE 申请。
- PRIVATE 类型地址可以用户自行指定，在一个上电周期类保持不变或者一直不变，该类型地址的最高 2bit 要设为 11b。
- RANDOM_RESOVABLE 类型地址通过一个随机数和 IRK（identity resolving key ）来生成，采用这种类型地址的设备可以防止被未知设备扫描和追踪。
- RANDOM_NONE_RESOVABLE 类型地址与 PRIVATE 类型地址类似，但是 RANDOM_NONE_RESOVABLE 类型地址会定时更新，更新周期由 GAP 来设定。

2. 设置设备本地地址，在 user_main()初始化时，需要给 mac 地址赋值的同时，指定本地 mac 地址的类型，本地 mac 地址类型一旦设定后续不可修改，扫描，广播，和主动发起连接的动作，均使用此时设定的本地地址类型进行操作。

```
/* set local BLE address */
mac_addr_t mac_addr;
mac_addr.addr[0] = 0xbd;
mac_addr.addr[1] = 0xad;
mac_addr.addr[2] = 0x10;
mac_addr.addr[3] = 0x11;
mac_addr.addr[4] = 0x20;
mac_addr.addr[5] = 0x20;
gap_address_set(&mac_addr, BLE_ADDR_TYPE_PRIVATE);
```

3. OSAL API

3.1. Task

该 task 的调度依赖于主循环的非抢占式轮询调度机制，这里面涉及到的事件按照先进先出的顺序被执行。任务的创建需要在 BLE 协议栈初始化后执行，消息的推送需要在任务创建之后执行。

3.1.1. 创建 task

当前 SDK 最多支持用户创建 20 个 task，os_task_create 用于创建一个新的 task，参数为该 task 的事件回调函数，由于 task 调度依赖于主循环，因此单个事件的处理时间不可太长（几十 ms）。以下代码为创建一个 task 的示例：

```
static int user_task_func(os_event_t *param)
{
    switch(param->event_id)
    {
        case USER_EVT_AT_COMMAND:
            app_at_cmd_recv_handler(param->param, param->param_len);
            break;
    }

    return EVT_CONSUMED;
}

void user_task_init(void)
{
    user_task_id = os_task_create(user_task_func);
}
```

3.1.2. 向 task 中推送消息

在 task 创建之后，可以调用 os_msg_post 函数向该 task 推送消息，用于异步处理，示例代码如下：

```
os_event_t at_cmd_event;
at_cmd_event.event_id = USER_EVT_AT_COMMAND;
at_cmd_event.param = at_recv_buffer;
at_cmd_event.param_len = at_recv_index;
os_msg_post(user_task_id, &at_cmd_event);
```

os_msg_post 函数的第一个参数为接收消息任务的 id，该 id 为任务创建时的返回值。消息结构体中的 event_id 用于标识消息类型，供事件回调函数 user_task_func 识别。

3.2. Timer

该 timer 属于软件定时器，依赖于 baseband 的运行，精度为 10ms，当前 SDK 支持最多创建 40 个 timer。定时器的回调函数依赖于主循环的非抢占式调度，因此回调函数中的执行时间不可太长（几十 ms）。

创建并开启 timer，表述 timer 的结构体 test_timer 需要常驻内存，因此不可使用局部变量，调用 os_timer_start 之前，需要先调用 os_timer_init 进行初始化。其中 os_timer_start 函数中的第三个参数表示这个 timer 是否需要循环触发：

```
os_timer_init(&test_timer, test_timer_handler, NULL);
os_timer_start(&test_timer, 500, true);
```

在初始化时注册的回调函数中可以执行预定任务回调函数的参数为调用 os_timer_init 时传递的第三个参数：

```
void test_timer_handler(void *param)
{
    LOG_INFO(app_tag, "test timer\r\n");
}
```

调用 os_timer_stop 可以停止一个正在运行的 timer：

```
os_timer_stop(&test_timer);
```

3.3. Memory

SDK 中将所有未分配的 RAM 预留给了 Heap 空间，并提供了一组函数供用户调用：

1. 分配指定大小的内存

```
void *os_malloc(uint32_t size)
```

2. 重新分配指定大小的内存，并保留之前已有的数据

```
void  *os_realloc(void  *ptr,uint32_t  new_size)
```

3. 释放已分配的内存

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

4. 获取当前可用的 heap 大小

```
uint16_t  os_get_free_heap_size(void)
```

4. 外设驱动

4.1. GPIO

FR8000 系列芯片不同的型号封装出不同数量的 GPIO，但是 GPIO 的使用方法都是一致的，每个 GPIO 都可以复用成不同的功能，并且具有外部中断、唤醒等功能。GPIO 的控制权可以交由两个模块：普通控制单元和低功耗控制单元。在系统处于正常工作模式下时，这两个控制单元都可以用来控制 GPIO；当系统处于低功耗模式下时，普通控制单元处于掉电状态，这时只有低功耗控制单元可以控制 GPIO。

在普通控制单元控制 GPIO 时，GPIO 可以用作通用输入输出口、外部中断口、串口、SPI 口等常用外部接口功能。在低功耗单元控制 GPIO 时，GPIO 可以用作输入输出口、外部电平变化检测、Keyscan 等功能。SDK 提供了两个接口来设置 GPIO 的控制权：

设置 GPIO 控制权给普通控制单元：

```
void  pmu_set_pin_to_CPU(enum  system_port_t  port,  uint8_t  bits);
```

设置 GPIO 控制权给低功耗控制单元：

```
void  pmu_set_pin_to_PMU(enum  system_port_t  port,  uint8_t  bits);
```

4.1.1. GPIO 控制（普通控制单元）

GPIO 在普通控制单元控制时可以复用成不同外设的功能，需要注意在进入睡眠状态之后普通控制单元会掉电，导致这些配置丢失，因此从睡眠中唤醒之后，这些配置需要重新调用。

4.1.1.1. 输入

下面的示例展示如何将 PA0 成普通控制单元控制下的输入功能：

```
/* configure PA0 IO MUX */
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_0, PORTA0_FUNC_A0);
/* set PA0 function to GPIO */
gpio_set_func(GPIO_PORT_A, GPIO_BIT_0, GPIO_FUNC_GPIO);
/* set PA0 to input mode */
gpio_set_dir(GPIO_PORT_A, GPIO_BIT_0, GPIO_DIR_IN);
/* get PA0 value */
```

```
gpio_get_pin_value(GPIO_PORT_A, GPIO_BIT_0);
/* get PORTA value */
gpio_porta_read();
```

4.1.1.2. 输出

下面的示例展示如何将 PA1 成普通控制单元控制下的输出功能：

```
/* configure PA1 IO MUX */
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_1, PORTA1_FUNC_A1);
/* set PA1 function to GPIO */
gpio_set_func(GPIO_PORT_A, GPIO_BIT_1, GPIO_FUNC_GPIO);
/* set PA1 to input mode */
gpio_set_dir(GPIO_PORT_A, GPIO_BIT_1, GPIO_DIR_OUT);
/* get PA1 value */
gpio_set_pin_value(GPIO_PORT_A, GPIO_BIT_1, 1);
/* get PORTA value */
gpio_porta_write(gpio_porta_read() | 0x02);
```

4.1.1.3. 上下拉

下面的示例展示如何配置上下拉和取消上下拉：

```
/* set PA2 pull down */
system_set_port_pull(GPIO_PA2, GPIO_PULL_DOWN, true);
/* set PA3 pull up */
system_set_port_pull(GPIO_PA3, GPIO_PULL_UP, true);
/* remove PA4 pull down and pull up */
system_set_port_pull(GPIO_PA4, GPIO_PULL_NONE, false);
```

4.1.1.4. 外部中断

下面示例展示如何配置 PA1 低电平中断，防抖时间为 10ms：

```
/* configure PA1 IO MUX */
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_1, PORTA1_FUNC_A1);
/* set PA1 function to GPIO */
gpio_set_func(GPIO_PORT_A, GPIO_BIT_1, GPIO_FUNC_EXTI);
/* set PA1 to input mode */
gpio_set_dir(GPIO_PORT_A, GPIO_BIT_1, GPIO_DIR_IN);
/* set PA1 pull up */
system_set_port_pull(GPIO_PA1, GPIO_PULL_UP, true);
/* set anti-shake time */
exti_set_control(EXTI_GPIOA_1, 1000, 10);
/* set trigger type to low level */
exti_set_type(EXTI_GPIOA_1, EXTI_TYPE_LOW);
```

```
/* enable PA1 external interrupt */
exti_enable(GPIO_PA1);
/* enable NVIC GPIO interrupt */
NVIC_EnableIRQ(GPIO_IRQn);
```

外部中断处理函数示例如下：

```
void exti_isr(void)
{
    uint32_t status;

    /* get and clear exti interrupt status */
    status = exti_get_status();
    exti_clear(status);

    if(status & GPIO_PA1) {
        /* reset trigger type to high level */
        exti_set_type(EXTI_GPIOA_1, EXTI_TYPE_HIGH);
    }
}
```

4.1.1.5. 配置 IO 为其他功能

以下为采用 I2C 配置为例展示如何将 IO 配置为其他外设功能引脚：

```
system_set_port_pull(GPIO_PC4|GPIO_PC5|GPIO_PC2|GPIO_PC3, GPIO_PULL_UP, true);
system_set_port_mux(GPIO_PORT_C, GPIO_BIT_4, PORTC4_FUNC_I2C0_CLK);
system_set_port_mux(GPIO_PORT_C, GPIO_BIT_5, PORTC5_FUNC_I2C0_DAT);
system_set_port_mux(GPIO_PORT_C, GPIO_BIT_2, PORTC2_FUNC_I2C1_CLK);
system_set_port_mux(GPIO_PORT_C, GPIO_BIT_3, PORTC3_FUNC_I2C1_DAT);
```

4.1.2. GPIO 控制（低功耗控制单元）

当 GPIO 由低功耗控制单元控制时，所配置的功能在普通工作模式下或者低功耗模式下均可以正常工作。在睡眠模式下，IO 可能存在不定态，这时可以将 IO 交由低功耗控制单元控制，从而避免不定态造成的漏电。

4.1.2.1. 输入

下面的示例展示如何将 PB0 成低功耗控制单元控制下的输入功能：

```
/* config PB0 IO mux */
pmu_set_pin_mux(GPIO_PORT_B, GPIO_BIT_0, PMU_PIN_FUNC_GPIO);
/* set PB0 to input mode */
```

```
pmu_set_pin_dir(GPIO_PORT_B, (1<<GPIO_BIT_0), GPIO_DIR_IN);
/* get PORTB value */
pmu_portb_read();
```

4.1.2.2. 输出

下面的示例展示如何将 PB0 成低功耗控制单元控制下的输出功能：

```
/* config PB0 IO mux */
pmu_set_pin_mux(GPIO_PORT_B, GPIO_BIT_0, PMU_PIN_FUNC_GPIO);
/* set PB0 to input mode */
pmu_set_pin_dir(GPIO_PORT_B, (1<<GPIO_BIT_0), GPIO_DIR_OUT);
/* set PORTB value */
pmu_portb_write(pmu_portb_read() | 0x01);
```

4.1.2.3. 上下拉

下面的示例展示如何配置上下拉和取消上下拉：

```
/* set PA2 pull down */
pmu_set_pin_pull(GPIO_PORT_A, (1<<GPIO_BIT_2), GPIO_PULL_DOWN);
/* set PA3 pull up */
pmu_set_pin_pull(GPIO_PORT_A, (1<<GPIO_BIT_3), GPIO_PULL_UP);
/* remove PA4 pull down and pull up */
pmu_set_pin_pull(GPIO_PORT_A, (1<<GPIO_BIT_4), GPIO_PULL_NONE);
```

4.1.2.4. GPIO 电平监测

在低功耗控制单元中有一个 GPIO 电平监测模块，它的实现机制类似异或操作，用于监测关注引脚的电平变化，当监测对象引脚的电平与设置的初始值不一致时就会产生 PMU 中断。PMU 中断可以将系统从睡眠中唤醒，因此 GPIO 的唤醒功能就通过这种方式实现。

SDK 中提供了 API 用于配置监测对象：

```
void pmu_port_wakeup_func_set(enum system_port_t port, uint8_t bits)
{
    uint8_t last_status_reg = PMU_REG_PORTA_LAST_STATUS;

    last_status_reg += port;

    /* PMU control */
    pmu_set_pin_to_PMU(port, bits);

    /* Config input、Pull up */
```

```
    pmu_set_pin_pull(port, bits, GPIO_PULL_UP);
    pmu_set_pin_dir(port, bits, GPIO_DIR_IN);

    /* Read the current value and write */
    ool_write(last_status_reg, ool_read(last_status_reg));

    /* XOR Enable */
    pmu_set_pin_xor_en(port, bits, true);

    /* XOR interrupt enable */
    pmu_enable_isr(PMU_GPIO_XOR_INT_EN);
}
```

以使能 PD0、PD1 为例，下面演示如何调用上述函数实现监测：

```
pmu_port_wakeup_func_set(GPIO_PORT_D,  (1<<GPIO_BIT_0)  |  (1<<GPIO_BIT_1));
```

当 PD0 或者 PD1 电平发生变化时，会产生 PMU 中断：

```
__attribute__((section("ram_code"))) void pmu_isr(void)
{
    uint16_t state = pmu_get_isr_state();

    pmu_clear_isr_state(state);

    if(state & PMU_GPIO_XOR_INT_STATUS) {
        uint32_t port_value = ool_read32(PMU_REG_PORTA_LAST_STATUS);
        LOG_INFO(NULL, "current port state is 0x%08x\r\n", port_value);
        ool_write32(PMU_REG_PORTA_LAST_STATUS, port_value);
    }
}
```

4.1.3. GPIO 控制（新增驱动方式）

在 FR8000 SDK 中新增了 driver_gpio.c 驱动代码。老版的 GPIO 驱动依旧保持，新增的 GPIO 驱动更加灵活，更加简易。

以输出为例：

```
GPIO_Handle.Pin        = GPIO_PIN_0|GPIO_PIN_1;
GPIO_Handle.Mode       = GPIO_MODE_OUTPUT_PP;
gpio_init(GPIO_B, &GOIO_Handle);
gpio_write_pin(GPIO_B, GPIO_PIN_0|GPIO_PIN_1, GPIO_PIN_SET);
gpio_write_pin(GPIO_B, GPIO_PIN_0|GPIO_PIN_1, GPIO_PIN_CLEAR);
gpio_write_group(GPIO_B, 0x03);
gpio_write_group(GPIO_B, 0x00);
```

初始化需要选择 PIN 脚，一次性可传入 8 个 PIN，选择 GPIO 功能。调用初始化函数即可。输出可控制单个或多个 PIN，还可以控制整租，在这里提供了两个函数供用户使用。

以输入为例：

```
GPIO_Handle.Pin        = GPIO_PIN_2|GPIO_PIN_3;
GPIO_Handle.Mode       = GPIO_MODE_INPUT;
GPIO_Handle.Pull       = GPIO_PULLDOWN;
gpio_init(GPIO_B, &GOIO_Handle);
gpio_read_group(GPIO_B);
gpio_read_pin(GPIO_B, GPIO_PIN_2);
gpio_read_pin(GPIO_B, GPIO_PIN_3);
```

初始化需要选择 PIN 脚，一次性可传入 8 个 PIN，选择 GPIO 功能，并配置上下拉电阻。调用初始化函数即可。读取 PIN 脚状态时可以读取单个 PIN 脚状态，或读取整组状态。

以 EXTI 为例：

```
GPIO_Handle.Pin        = GPIO_PIN_0|GPIO_PIN_1;
GPIO_Handle.Mode       = GPIO_MODE_EXTI_IT_RISING;
GPIO_Handle.Pull       = GPIO_PULLDOWN;
gpio_init(GPIO_B, &GOIO_Handle);
while (1)
{
    if (exti_get_LineStatus(8))
    {
        exti_clear_LineStatus(8);
    }
    if (exti_get_LineStatus(9))
    {
        exti_clear_LineStatus(9);
    }
}
```

初始化需要选择 PIN 脚，一次性可传入 8 个 PIN，选择 GPIO 功能，这里可选上升沿、下降沿、高电平、低电平触发中断，并配置上下拉电阻。调用初始化函数即可。查询方式可使用中断方式或循环查询方式，读取 EXTI 线状态。

复用外设为例：

```
GPIO_Handle.Pin        = GPIO_PIN_0|GPIO_PIN_1;
GPIO_Handle.Mode       = GPIO_MODE_AF_PP;
GPIO_Handle.Pull       = GPIO_PULLDOWN;
GOIO_Handle.Alternate = GPIO_FUNCTION_4;
gpio_init(GPIO_B, &GOIO_Handle);
```

当使用 UART、SPI、I2C 等外设功能时，需要配置 GPIO 为复用功能。用户可参考 FR8000 芯片手册的附录 I，来决定复用功能。

4.2. ADC

ADC 驱动中提供了初始化；ADC 通道；测量内部温度；测量电池电压；获得转换结果；中断使能，查询，清除等功能。

以 ADC 通道为例：

```
ADC_InitParam.ADC_CLK_DIV          = 5;
ADC_InitParam.ADC_SetupDelay       = 80;
ADC_InitParam.ADC_Reference        = ADC_REF_LDOIO;
ADC_InitParam.FIFO_Enable          = FIFO_ENABLE;
ADC_InitParam.FIFO_AlmostFullLevel = 8;
adc_init(ADC_InitParam);
adc_Channel_ConvertConfig(ADC_CHANNEL_0|ADC_CHANNEL_1|ADC_CHANNEL_2|ADC_CHANNEL_3);
adc_convert_enable();
```

初始化参数从上到下分别是，ADC 采样时钟分频，通道采集建立时间延时，参考电压，FIFO 使能控制，FIFO 近满阈值。若使能 FIFO，则只能通过 adc_get_data()函数，读取 ADC 的 DATA 寄存器获得结果值。另外，测量内部温度，电池电压时，也只能通过 adc_get_data()函数来获得结果值。

不使用 FIFO 时，可以通过 adc_get_channel_data()函数，读取 ADC 的 DATA0~DATA7 寄存器获得通道转换的结果值。

当只需测量单通道时，推荐开启 FIFO，若需测量多通道，则不要开启 FIFO。

注意：通道转换与测量内部信号如温度，电池电压不能同时进行。

4.3. DMA

DMA 的使用场景，大多数为内存到外设；外设到内存这一类。这里我们以 I2S 的发送为例子，参考 DMA 的配置使用情况。

当目标或地址为外设时，需要用户手动分配 DMA 的请求编号。SDK 中已经做好了封装。在 driver_system.h 中可以看到可供配置的外设，如下所示：

```
#define __DMA_REQ_ID_UART0_RX(__REQ_ID__)
#define __DMA_REQ_ID_UART0_TX(__REQ_ID__)
#define __DMA_REQ_ID_SPI0_MASTER_RX(__REQ_ID__)
#define __DMA_REQ_ID_SPI0_MASTER_TX(__REQ_ID__)
#define __DMA_REQ_ID_I2C1_RX(__REQ_ID__)
#define __DMA_REQ_ID_I2C1_TX(__REQ_ID__)
```

以 I2S 发送为例

```
__DMA_REQ_ID_I2S_TX(1);

DMA_Chan0.Channel               = DMA_Channel0;
DMA_Chan0.Init.Data_Flow        = DMA_M2P_DMAC;
DMA_Chan0.Init.Request_ID       = 1;
DMA_Chan0.Init.Source_Inc       = DMA_ADDR_INC_INC;
DMA_Chan0.Init.Desination_Inc   = DMA_ADDR_INC_NO_CHANGE;
DMA_Chan0.Init.Source_Width     = DMA_TRANSFER_WIDTH_32;
DMA_Chan0.Init.Desination_Width = DMA_TRANSFER_WIDTH_32;
dma_init(&DMA_Chan0);
dma_start(&DMA_Chan0, (uint32_t)TxBuffer32, (uint32_t)&I2S->DATA, 256, DMA_BURST_LEN_4,
DMA_BURST_LEN_4);
while(!dma_get_tfr_Status(DMA_Channel0));
dma_clear_tfr_Status(DMA_Channel0);
```

手动分配 I2S_TX 的请求号为‘1’。

配置 DMA 参数。从上到下分别为。所使用的通道，搬运方向，请求号，源地址是否递增，目标地址是否递增，搬运数据的位宽。配置好参数后，调用 dma_init()函数完成通道初始化。

调用 dma_start()或 dma_start_IT()使能搬运。其中形参中有 DestBurstLen，此形参的含义为，一次搬运多少个数据。比如此时数据位宽 `Source_Width` 与 `Desination_Width` 配置为 `DMA_TRANSFER_WIDTH_32`。若选择了 `DMA_BURST_LEN_4` 则表示一次搬运四个 `Source_Width` 位宽的数据。也就是 4 * 4 = 16byte；使用 BurstLength 需要考虑目标地址或源地址是否有 FIFO 空间一次容纳多个字节的数据。搬运的总数据量由形参 Size 决定。（Size 以字节为单位）

最后可通过查询方式，或中断方式来判断 DMA 搬运是否完成。

除此之外 DMA 还支持链表传输，每个链表中包含着每次传输的 1.目标地址，2.源地址，3.下一次传输的链表基地址（如本次传输是最后一次，则赋值为 NULL），4.控制寄存器。如下所示：

```
typedef struct DMA_NextLink
{
    uint32_t SrcAddr;               /* source address */
    uint32_t DstAddr;               /* desination address */
    struct DMA_NextLink *Next;      /* Next Link */
    REG_CTL1_t  CTL1;               /* Control Register for Channel */
    REG_CTL2_t  CTL2;               /* Control Register for Channel */
}DMA_LLI_InitTypeDef;
```

在第一个链表传输完成后，将查询第一个链表的 *Nxet 值，如果不为 NULL，则进行下一个链表的传输 ••••••

4.4. I2C

I2C 驱动中提供了初始化；主机阻塞式收发数据；主机中断式收发数据；EEPROM 读写功能。

以读写 EEPROM 为例：

```
I2C1_Handle.I2Cx = I2C1;
I2C1_Handle.Init.I2C_Mode = I2C_MODE_MASTER_7BIT;
I2C1_Handle.Init.SCL_HCNT = 500;
I2C1_Handle.Init.SCL_LCNT = 500;
I2C1_Handle.Init.Slave_Address = 0xAE;
i2c_init(&I2C1_Handle);
NVIC_EnableIRQ(I2C1_IRQn);
// EEPROM write/read
i2c_memory_write(&I2C1_Handle, 0xA0, 32*0, &TxBufferI2C[32*0], 32);
i2c_memory_read(&I2C1_Handle, 0xA0, 0, RxBufferI2C, 256);
```

首先初始化参数。

I2Cx。选择使用那一个 I2C。

I2C_Mode。选择主从模式，7bit 或 10bit 地址模式。

SCL_HCNT。配置 SCL 高电平保持时间。单位为 I2C 外设时钟源。

SCL_LCNT。配置 SCL 低电平保持时间。单位为 I2C 外设时钟源。

Slave_Address。若选用从机模式，则配置从机自身的地址。

配置好初始化参数后，调用 i2c_init()初始化 I2C。再调用读写函数便可。

4.5. 其他

其他外设可参考相关驱动程序和 FR8000 specification。

5. 系统函数

5.1. 系统时钟

当前系统可以在 48Mhz 和 96MHz 两种主频下运行，可以通过函数 system_set_clock 和 system_get_clock 来设置和获取当前的系统时钟。由于总线时钟与系统时钟保持一致，因此在切换系统时钟之后，一些外设和总线时钟也会随之改变。

5.2. 系统睡眠

SDK 中提供 system_sleep_enable 和 systerm_sleep_disable 来开启和关闭睡眠功能。需要注意 system_sleep_enable 调用之后并不会立即进入睡眠状态，而是使能是否可以睡眠的监测功能，能否进入睡眠由 1.4.3 节中描述的内容决定。

5.3. 其他

5.3.1. 程序运行时间

SDK 提供了函数 system_get_curr_time 用来获取从上电到调用时刻系统运行了多长时间，该函数单位为 ms，计时到 83886079 后会重新从 0 开始计数。

5.3.2. SDK 编译时间

用户程序可以通过函数 get_SDK_compile_date_time 获取 SDK 的编译时间， 用于跟踪版本。

6. OTA

FR8000 采用双备份的方法实现 OTA 升级，正在运行的固件存储在 A 区域，待升级的固件存储在 B 区域，bootloader 通过固件信息区域内容来判断两个固件存储区域的有效性。

图 6-1 固件信息区域数据格式

在系统启动时 Bootloader 首先检查“A 区域固件信息”中的“配置信息有效标识”，检查通过后即认为 A 区域中存储有有效固件；之后再检查“B 区域固件信息”中的“B 区域有效标识”，如果检查通过则认为 B 区域中存储有待升级的固件，如果没有通过则运行检查通过的 A 区域固件。在检测到有待升级固件时，bootloader 会将 B 区域固件升级到 A 区域。

6.1. 固件的生成

工程编译生成的 bin 文件可以供 PC 烧录工具、量产烧录工具直接使用，其基本格式如下：

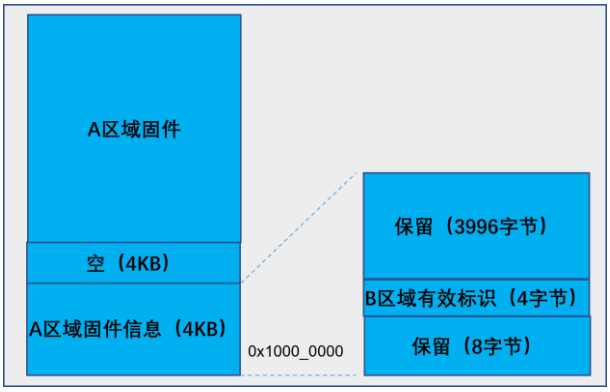

图 6-2 编译出 bin 文件基本格式

该文件若用作 OTA 升级，则需要在固件结尾填充 0xFF，使得固件总大小为 4KB 对齐，然后在固件信息字段添加部分内容，橙色字段为依照实际信息计算并填写，绿色字段填充为 0xFFFFFFFF：

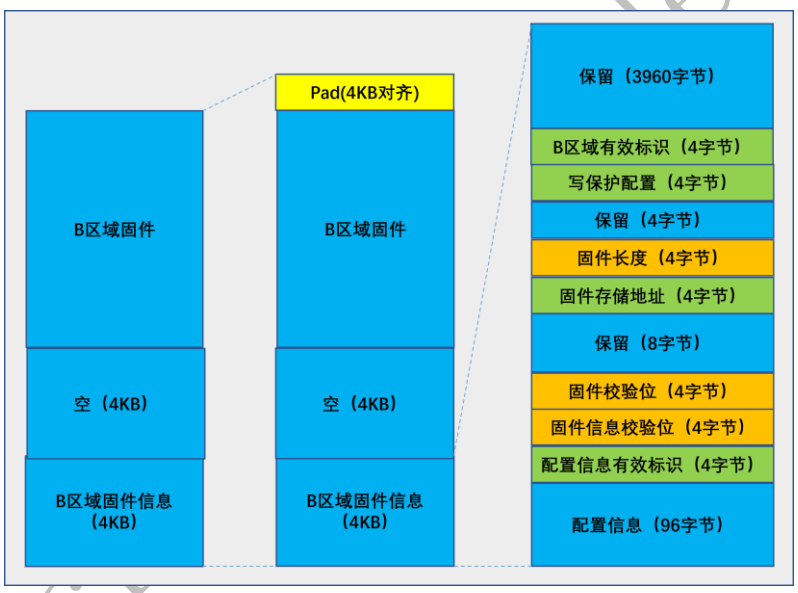

图 6-3 待升级固件信息添加字段

6.2. 固件写入

固件写入分成两部分：一个是固件信息区域；一个是固件区域。在进行 OTA 固件写入时建议采用如下步骤：

1. 将固件信息区域写入“B 区域固件信息”位置；
2. 获取 A 固件区域的结束位置，并向上按照 4KB 对齐作为固件区域的写入起始位置 B_offset；
3. 将 OTA 固件的固件区域存储到从 B_offset 开始的 FLASH 空间；

4. 存储结束后程序检查“固件信息校验位”和“配置信息校验位”是否与实际匹配。如果检查通过则填充“固件存储地址”（B_offset）、“B 区域有效标识”和写保护配置；如果检查未通过，则 OTA 升级失败；
5. 在 4 通过的情况下系统重启，bootloader 负责进行固件的实际升级。

联系方式

欢迎大家针对富芮坤产品和文档提出建议。

反馈：doc@freqchip.com.

网站：www.freqchip.com

销售：sales@freqchip.com

电话：+86-21-5027-0080