

# **AmebaD BLE Stack User Manual**

**V 1.0.1**

**2019/9/17**

## 修订历史 (Revision History)

日期	版本	修改	作者	Reviewer
2018/12/13	V1.0.0	Formal version	Jane	
2019/09/17	V1.0.1	Modify BT features.	Jane	

# 目录

修订历史 (Revision History) .....	2
目录 .....	3
表目录 .....	5
图目录 .....	6
词汇表 .....	8
1 概述 .....	9
1.1 支持的蓝牙特性.....	9
1.2 BLE Profile 架构 .....	9
1.2.1 GAP .....	10
1.2.2 基于 GATT 的 Profile.....	11
2 GAP .....	12
2.1 GAP 结构概述 .....	13
2.1.1 GAP 的位置.....	13
2.1.2 GAP 的功能.....	13
2.1.3 GAP 层设备状态 .....	14
2.1.4 GAP 消息 .....	17
2.1.5 APP 消息流 .....	18
2.2 GAP 的初始化和启动流程 .....	20
2.2.1 GAP 参数的初始化 .....	20
2.2.2 GAP 启动流程.....	25
2.3 BLE GAP 消息.....	26
2.3.1 概述 .....	26
2.3.2 Device 状态消息.....	28
2.3.3 Connection 相关消息.....	29
2.3.4 Authentication 相关消息.....	31
2.4 BLE GAP 回调函数 .....	34
2.4.1 BLE GAP 回调函数消息概述 .....	35

2.5 BLE GAP 用例 .....	38
2.5.1 GAP Service Characteristic 的可写属性 .....	38
2.5.2 本地设备使用 Static Random Address .....	39
2.5.3 Physical (PHY) 设置 .....	41
2.6 GAP 信息存储 .....	44
2.6.1 FTL 简介 .....	44
2.6.2 本地协议栈信息存储 .....	45
2.6.3 绑定信息存储 .....	45
3 GATT Profile .....	53
3.1 BLE Profile Server .....	53
3.1.1 概述 .....	53
3.1.2 支持的 Profile 和 Service .....	54
3.1.3 Profile Server 交互 .....	55
3.1.4 Specific Service 的实现 .....	71
3.2 BLE Profile Client .....	79
3.2.1 概述 .....	79
3.2.2 支持的 Clients .....	80
3.2.3 Profile Client Layer .....	80
4 BLE 示例工程 .....	93
4.1 BLE Peripheral Application .....	93
4.1.1 简介 .....	93
4.1.2 工程概述 .....	93
4.1.3 源代码概述 .....	94
4.1.4 测试步骤 .....	97
参考文献 .....	99

## 表目录

表 1-1 支持的蓝牙特性 .....	9
表 2-1 Advertising 参数设置 .....	23
表 2-2 Authentication 相关消息 .....	31
表 2-3 gap_le.h 相关消息 .....	35
表 2-4 gap_conn_le.h 相关消息 .....	35
表 2-5 gap_bond_le.h 相关消息 .....	37
表 2-6 gap_scan.h 相关消息 .....	37
表 2-7 gap_adv.h 相关消息 .....	37
表 3-1 支持的 Profile 列表 .....	54
表 3-2 支持的 service 列表 .....	55
表 3-3 Flags 的可选值和描述 .....	72
表 3-4 Flags Value 的选择模式 .....	72
表 3-5 Permissions 的可用值 .....	73
表 3-6 Service Table 示例 .....	74
表 3-7 支持的 Clients .....	80
表 3-8 Discovery 状态 .....	83
表 3-9 Discovery 结果 .....	84

## 图目录

图 1-1 蓝牙 Profile.....	10
图 1-2 基于 GATT 的 Profile 层级结构 .....	11
图 2-1 BT Lib .....	12
图 2-2 GAP 头文件 .....	12
图 2-3 GAP 在 SDK 中的位置.....	13
图 2-4 Advertising 状态的状态转换.....	14
图 2-5 Scan 状态的状态转换 .....	15
图 2-6 主动的 Connection 状态转换 .....	16
图 2-7 被动的 Connection 状态转换 .....	17
图 2-8 APP 消息流 .....	19
图 2-9 FTL 布局.....	44
图 2-10 增加一个绑定设备 .....	46
图 2-11 移除一个绑定设备 .....	46
图 2-12 清除所有绑定设备 .....	46
图 2-13 将一个绑定设备设为最高优先级 .....	46
图 2-14 获取最高优先级设备 .....	47
图 2-15 获取最低优先级设备 .....	47
图 2-16 优先级管理示例 .....	47
图 2-17 LE FTL 布局 .....	48
图 3-1 GATT Profile 头文件.....	53
图 3-2 Profile Server 层级.....	53
图 3-3 向 Server 添加 Services.....	56
图 3-4 注册 Service 的流程.....	56
图 3-5 读 Characteristic Value – 由 Attribute Element 提供 Attribute Value .....	59
图 3-6 读 Characteristic Value – 由 APP 提供 Attribute Value 且结果未挂起 .....	59
图 3-7 读 Characteristic Value – 由 APP 提供 Attribute Value 且结果挂起.....	61
图 3-8 Write Characteristic Value – 由 Attribute Element 提供 Attribute Value .....	62
图 3-9 Write Characteristic Value – 由 APP 提供 Attribute Value 且结果未挂起 .....	63
图 3-10 Write Characteristic Value – 由 APP 提供 Attribute Value 且结果挂起.....	64
图 3-11 Write Characteristic Value – 写 CCCD 值.....	65
图 3-12 Write without Response – 由 APP 提供 Attribute Value .....	67
图 3-13 Write Long Characteristic Value – Prepare Write 流程 .....	68

图 3-14 Write Long Characteristic Values– 结果未挂起的 Execute Write .....	68
图 3-15 Write Long Characteristic Values– 结果挂起的 Execute Write .....	69
图 3-16 Characteristic Value Notification .....	69
图 3-17 Characteristic Value Indication.....	70
图 3-18 Profile Client 层级 .....	80
图 3-19 向 Profile Client Layer 添加 Specific Clients.....	82
图 3-20 GATT Discovery 流程 .....	83
图 3-21 Read Characteristic Value by Handle 流程 .....	85
图 3-22 Read Characteristic Value by UUID 流程 .....	86
图 3-23 Write Characteristic Value 流程 .....	87
图 3-24 Write Long Characteristic Value 流程 .....	87
图 3-25 Write Without Response 流程 .....	88
图 3-26 Characteristic Value Notification 流程 .....	88
图 3-27 结果未挂起的 Characteristic Value Indication 流程 .....	89
图 3-28 结果挂起的 Characteristic Value Indication 流程 .....	90
图 4-1 与 iOS 设备测试 .....	98

## 词汇表

缩写	含义
ATT	Attribute protocol
BLE	Bluetooth Low Energy
GAP	Generic Access Profile
GATT	Generic Attribute Profile
L2CAP	Logical Link Control and Adaptation protocol
SDK	Software Development Kit
SMP	Security Manager protocol
SOC	System on Chip



# 1 概述

Realtek SDK (Software Development Kit, SDK)提供的资料包括 Stack/Profiles 介绍文档、示例 profile 和用户示例程序，旨在帮助用户使用 Realtek SOC (System on Chip, SOC) 设备进行产品开发。SDK 可以促进 BLE (Bluetooth Low Energy, BLE) 应用的快速开发。Profile 作为 SDK 组成模块，封装 BLE 协议栈的实现细节，为应用开发提供用户友好和便于使用的接口。

本文综述 BLE 协议栈接口，其中包括基于 GAP (Generic Access Profile, GAP) 的接口和基于 GATT (Generic Attribute Profile, GATT) 的接口。

## 1.1 支持的蓝牙特性

表 1-1 支持的蓝牙特性

Spec Version	BT Feature	AmebaD	Remark
BT4.0	Advertiser	Y	
	Scanner	Y	
	Initiator	Y	
	Master	Y	Maximum number is 3.
	Slave	Y	Maximum number is 1.
BT4.1	Low Duty Cycle Directed Advertising	Y	
	LE L2CAP Connection Oriented Channel	N	
	LE Scatternet	Y	1 Master + 1 Slave
	LE Ping	Y	
BT4.2	LE Data Packet Length Extension	Y	
	LE Secure Connections	Y	
	Link Layer Privacy(Privacy1.2)	N	
	Link Layer Extended Filter Policies	N	
BT5	2 Msym/s PHY for LE	Y	
	LE Long Range	N	
	High Duty Cycle Non-Connectable Advertising	Y	
	LE Advertising Extensions	N	
	LE Channel Selection Algorithm #2	N	

## 1.2 BLE Profile 架构

在蓝牙核心规范 (Bluetooth Core Specification) 中，Profile 的定义不同于 Protocol 的定义。Protocol 被定义为各层协议，例如 Link Layer、Logical Link Control and Adaptation protocol (L2CAP)、Security Manager protocol (SMP) 和 Attribute protocol (ATT)。不同于 Protocol，Profile 从使用蓝牙核心规范中各层协议的角度，定义蓝牙应用互操作性的实现。Profile 定义 Protocol 中的可用特性和功能，以及蓝牙设备互

操作性的实现，使蓝牙协议栈适用于各种场景的应用开发。

在蓝牙核心规范中，Profile 和 Protocol 的关联如图 1-1 所示。

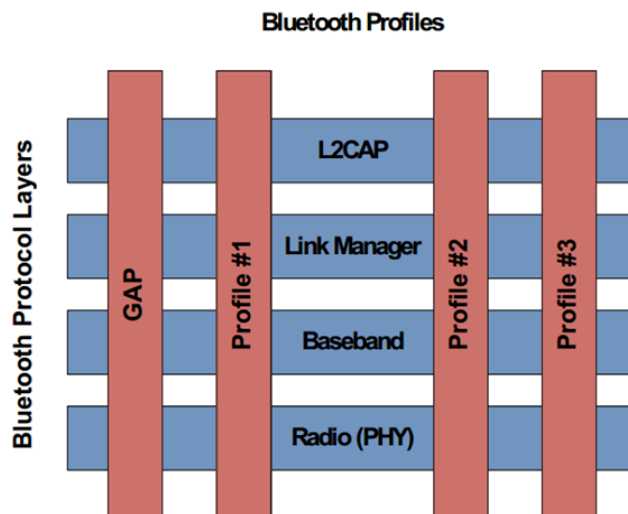


图 1-1 蓝牙 Profile

如图 1-1 所示，Profile 由红色矩形框表示，包括 GAP、Profile #1、Profile #2 和 Profile #3。蓝牙核心规范中的 Profile 分为两种类型：GAP，图中红色矩形框所示的 GAP；基于 GATT 的 Profile，图中红色矩形框所示的 Profile #1、Profile #2 和 Profile #3。

## 1.2.1 GAP

GAP 是所有的蓝牙设备均需实现的 Profile，用于描述 device discovery、connection、security requirement 和 authentication 的行为和方法。GAP 中的 BLE 部分定义四种角色 (Broadcaster、Observer、Peripheral 和 Central)，用于优化各种应用场景。

Broadcaster 用于只通过广播发送数据的应用；Observer 用于接收广播数据的应用；Peripheral 用于通过广播发送数据并且可以建立链路的应用；Central 用于接收广播数据并且建立一条或多条链路的应用。

## 1.2.2 基于 GATT 的 Profile

在蓝牙核心规范中，另一种常用的 Profile 是基于 GATT 的 Profile。GATT 分为 server 和 client 两种角色。Server 用于提供 service 数据。Client 可以访问 service 数据。基于 GATT 的 Profile 是基于 server-client 交互结构，适用于不同应用场景，用于蓝牙设备之间的特定数据交互。如图 1-2 所示，Profile 是以 Service 和 Characteristic 的形式组成的。

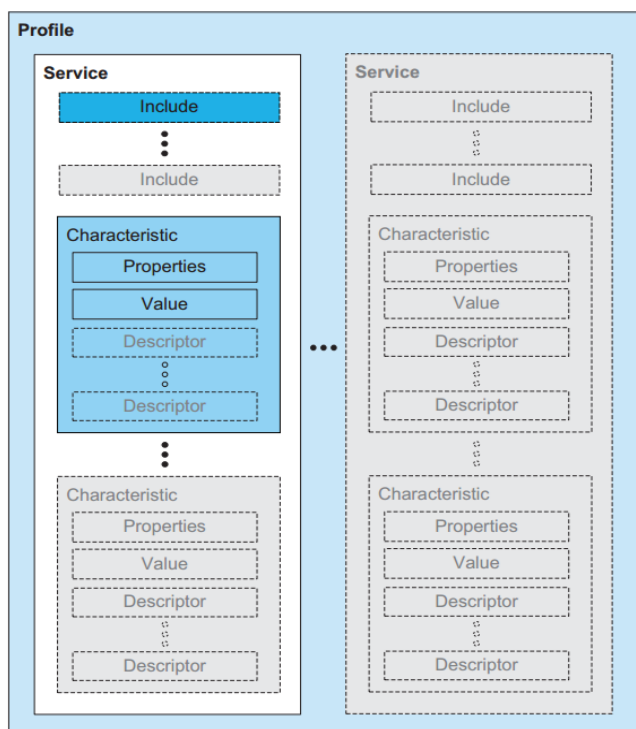


图 1-2 基于 GATT 的 Profile 层级结构

## 2 GAP

GAP 是所有蓝牙设备均需实现的 Profile，用于描述 device discovery、connection、security requirement 和 authentication 的行为和方法。

GAP 层是在 BT Lib 中实现的，提供接口给 application 使用。BT Lib 文件目录为 component\common\bluetooth\realtek\sdk\board\amebad\lib。

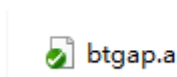


图 2-1 BT Lib

在 SDK 中提供头文件，头文件目录为 component\common\bluetooth\realtek\sdk\board\amebad\inc\bluetooth\gap。

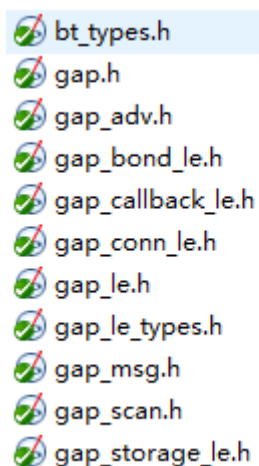


图 2-2 GAP 头文件

GAP 层的内容分为以下几个部分进行介绍：

1. 在章节 [GAP 结构概述](#) 中介绍 GAP 结构。
2. 在章节 [GAP 的初始化和启动流程](#) 中介绍 GAP 参数的配置和 GAP 层内部启动流程。
3. 在章节 [BLE GAP 消息](#) 中介绍 GAP 消息类型的定义和 GAP Message 处理流程。
4. GAP 层使用 GAP 消息回调函数发送消息给 application，在章节 [BLE GAP 回调函数](#) 中介绍关于 GAP 回调函数的内容。
5. 在章节 [BLE GAP 用例](#) 中介绍 GAP 接口的应用示例。
6. 在章节 [GAP 信息存储](#) 中介绍由 GAP 实现的本地信息和设备绑定信息的存储。

## 2.1 GAP 结构概述

### 2.1.1 GAP 的位置

GAP 层作为蓝牙协议层的组成模块，如图 2-3 所示，虚线框内的部分为蓝牙协议层。Application 在蓝牙协议层之上，baseband/RF 位于蓝牙协议层之下。GAP 层给 application 提供访问 Upper Stack 的接口。

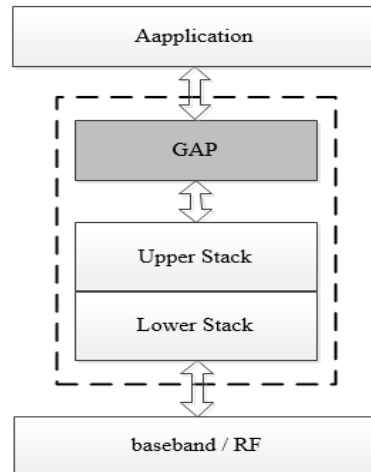


图 2-3 GAP 在 SDK 中的位置

### 2.1.2 GAP 的功能

GAP 层提供的接口的功能如下所示：

1. Advertising
 

设置/获取 advertising 参数，启动/停止 advertising。
2. Scan
 

设置/获取 scan 参数，启动/停止 scan。
3. Connection
 

设置 connection 参数，创建 connection，终止已建立的 connection，更新 connection 参数。
4. 配对
 

设置配对参数，启动配对，使用 passkey entry 方式时输入/显示 passkey，删除绑定设备密钥。
5. 密钥管理
 

根据设备地址和地址类型查找 key entry，保存/加载绑定信息的密钥，解析 random address。
6. 其它
  - 1) 设置 GAP 公共参数，例如 device appearance 和 device name

- 2) 获取支持的最大 BLE 链路数目
- 3) 修改 white list
- 4) 生成/设置本地设备 random address
- 5) 配置本地设备 identity address
- 6) 等等

API 不支持多线程，API 的调用和消息处理必须在同一个 task 中。SDK 中提供的 API 分为同步 API 和异步 API。同步 API 的结果由返回值表示，例如 `le_adv_set_param()`。若 `le_adv_set_param()` 的返回值为 `GAP_CAUSE_SUCCESS`，APP 成功设置一个 GAP advertising 参数。异步 API 的结果是通过 GAP 消息通知的，例如 `le_adv_start()`。若 `le_adv_start()` 的返回值为 `GAP_CAUSE_SUCCESS`，启动 advertising 的请求发送成功，启动 advertising 的结果是通过 GAP 消息 `GAP_MSG_LE_DEV_STATE_CHANGE` 通知 APP 的。

## 2.1.3 GAP 层设备状态

GAP 层设备状态由 advertising 状态、scan 状态和 connection 状态组成。每一个状态都有相应的子状态，本节内容将介绍各子状态。

### 2.1.3.1 Advertising 状态

Advertising 状态有四个子状态，idle 状态、start 状态、advertising 状态和 stop 状态，在 `gap_msg.h` 中定义 Advertising 状态的子状态。

```
/* GAP Advertising State */
#define GAP_ADV_STATE_IDLE      0 // Idle, no advertising
#define GAP_ADV_STATE_START     1 // Start Advertising. A temporary state, haven't received the result.
#define GAP_ADV_STATE_ADVERTISING 2 // Advertising
#define GAP_ADV_STATE_STOP      3 // Stop Advertising. A temporary state, haven't received the result.
```

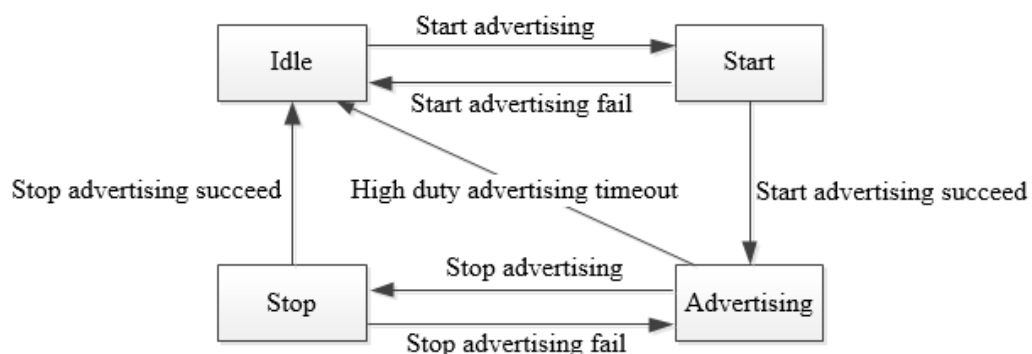


图 2-4 Advertising 状态的状态转换

#### 1. idle 状态

默认状态，不发送 advertisement。

## 2. start 状态

在 idle 状态启动 advertising 之后，启动 advertising 的流程尚未完成。start 状态为临时状态，若成功启动 advertising，则 Advertising 状态进入 advertising 状态；若启动 advertising 失败，则 Advertising 状态回到 idle 状态。

## 3. advertising 状态

成功启动 advertising。在此状态下，设备发送 advertisement。若 advertising 类型是 high duty cycle directed advertising，一旦超时，Advertising 状态进入 idle 状态。

## 4. stop 状态

在 advertising 状态停止 advertising 之后，停止 advertising 的流程尚未完成。stop 状态为临时状态，若成功停止 advertising，则 Advertising 状态进入 idle 状态；若停止 advertising 失败，则 Advertising 状态回到 advertising 状态。

### 2.1.3.2 Scan 状态

Scan 状态有四个子状态，idle 状态、start 状态、scanning 状态和 stop 状态，在 gap\_msg.h 中定义 Scan 状态的子状态。

```
/* GAP Scan State */
```

```
#define GAP_SCAN_STATE_IDLE      0    //Idle, no scanning
#define GAP_SCAN_STATE_START    1    //Start scanning. A temporary state, haven't received the result.
#define GAP_SCAN_STATE_SCANNING 2    //Scanning
#define GAP_SCAN_STATE_STOP     3    //Stop scanning, A temporary state, haven't received the result
```

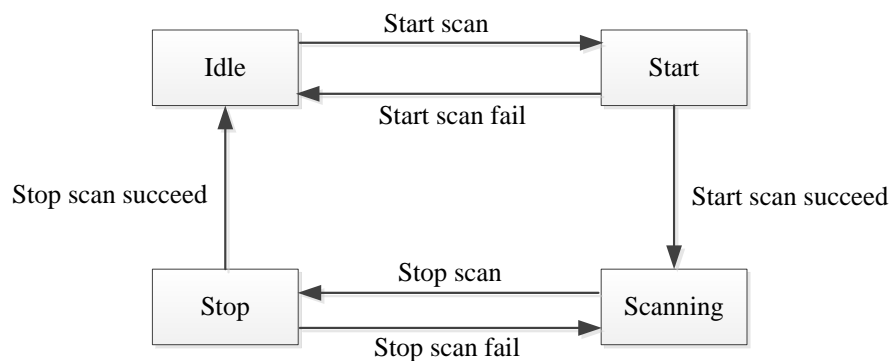


图 2-5 Scan 状态的状态转换

## 1. idle 状态

默认状态，不进行 scan。

## 2. start 状态

在 idle 状态启动 scan 之后，启动 scan 的流程尚未完成。start 状态为临时状态，若成功启动 scan，则 Scan 状态进入 scanning 状态；若启动 scan 失败，则 Scan 状态回到 idle 状态。

## 3. scanning 状态

成功启动 scan。在此状态下，设备进行 scan，接收 advertisement。

#### 4. stop 状态

在 scanning 状态停止 scan 之后，停止 scan 的流程尚未完成。stop 状态为临时状态，若成功停止 scan，则 Scan 状态进入 idle 状态；若停止 scan 失败，则 Scan 状态回到 scanning 状态。

### 2.1.3.3 Connection 状态

由于支持多链路，当 GAP Connection 状态为 idle 状态时，Link 状态是 connected 或 disconnected。因此，Connection 状态的变化需要结合 GAP Connection 状态和 Link 状态。

GAP Connection 状态的子状态包括 idle 状态和 connecting 状态，在 gap\_msg.h 中定义 GAP Connection 状态的子状态。

```
#define GAP_CONN_DEV_STATE_IDLE          0    //!< Idle
#define GAP_CONN_DEV_STATE_INITIATING    1    //!< Initiating Connection
```

注解：当 GAP Connection 状态为 connecting 状态时，application 不能主动创建另一条链路。

Link 状态有四个子状态，disconnected 状态、connecting 状态、connected 状态和 disconnecting 状态，在 gap\_msg.h 中定义 Link 状态的子状态。

```
/* Link Connection State */
typedef enum {
    GAP_CONN_STATE_DISCONNECTED, // Disconnected.
    GAP_CONN_STATE_CONNECTING,   // Connecting.
    GAP_CONN_STATE_CONNECTED,    // Connected.
    GAP_CONN_STATE_DISCONNECTING // Disconnecting.
} T_GAP_CONN_STATE;
```

作为 Master 角色主动创建 connection 时的 Connection 状态变化不同于作为 Slave 角色被动接收 connection indication 时的 Connection 状态变化。以下章节分别介绍这两种场景。

#### 2.1.3.3.1 主动的 Connection 状态转换

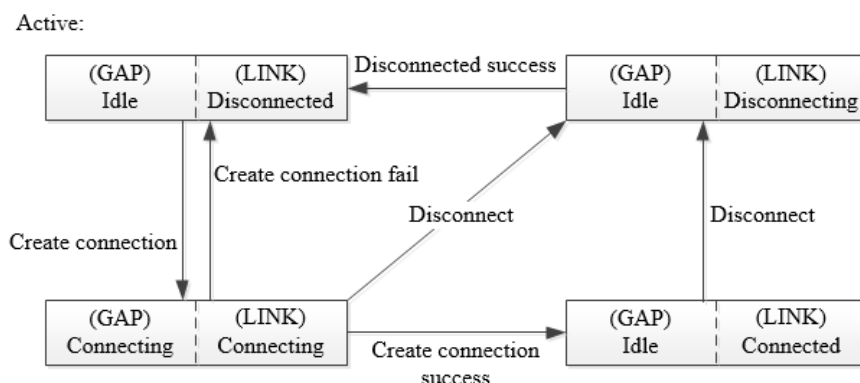


图 2-6 主动的 Connection 状态转换

##### 1. idle 状态 | disconnected 状态



GAP Connection 状态为 idle 状态，Link 状态为 disconnected 状态，未建立 connection。

## 2. connecting 状态 | connecting 状态

Master 创建 connection，创建流程尚未完成。这是临时状态，GAP Connection 状态为 connecting 状态，Link 状态为 connecting 状态。若成功创建 connection，GAP Connection 状态转换为 idle 状态，Link 状态转换为 connected 状态。若创建 connection 失败，GAP Connection 状态回到 idle 状态，Link 状态回到 disconnected 状态。在此状态下，Master 可以断开链路，此时，GAP Connection 状态回到 idle 状态，Link 状态转换为 disconnecting 状态。

## 3. idle 状态 | connected 状态

成功创建 connection，GAP Connection 状态为 idle 状态，Link 状态为 connected 状态。

## 4. idle 状态 | disconnecting 状态

Master 终止 connection，终止流程尚未完成。这是临时状态，GAP Connection 状态为 idle 状态，Link 状态为 disconnecting 状态。若成功终止 connection，Link 状态转换为 disconnected 状态。

### 2.1.3.3.2 被动的 Connection 状态转换

Passive:

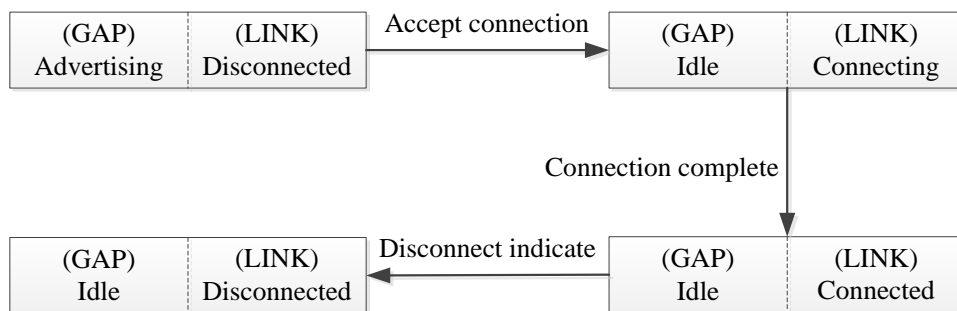


图 2-7 被动的 Connection 状态转换

## 1. Slave 接受 connection

当 Slave 收到 connect indication 后，GAP Advertising 状态将从 advertising 状态转换为 idle 状态，Link 状态将从 disconnected 状态转换为 connecting 状态。当创建 connection 流程完成之后，Link 状态将进入 connected 状态。

## 2. 对端设备断开 connection

当对端设备断开 connection 且本地设备收到 disconnect indication 后，本地设备的 Link 状态将从 connected 状态转换为 disconnected 状态。

## 2.1.4 GAP 消息

GAP 消息包括蓝牙状态消息和 GAP API 消息。蓝牙状态消息用于向 APP 通知蓝牙状态信息，包括 device 状态转换、connection 状态转换以及 bond 状态转换等。GAP API 消息用于向 APP 通知调用 API 后函数的执行状态。每个 API 都有相应的消息，更多关于 GAP 消息的信息参见 [BLE GAP 消息](#)和 [BLE GAP](#)

回调函数。

### 2.1.4.1 蓝牙状态消息

在 gap\_msg.h 中定义蓝牙状态消息。

```
/* BT status message */
#define GAP_MSG_LE_DEV_STATE_CHANGE          0x01 // Device state change msg type.
#define GAP_MSG_LE_CONN_STATE_CHANGE         0x02 // Connection state change msg type.
#define GAP_MSG_LE_CONN_PARAM_UPDATE        0x03 // Connection parameter update changed msg type.
#define GAP_MSG_LE_CONN_MTU_INFO            0x04 // Connection MTU size info msg type.
#define GAP_MSG_LE_AUTHEN_STATE_CHANGE      0x05 // Authentication state change msg type.
#define GAP_MSG_LE_BOND_PASSKEY_DISPLAY     0x06 // Bond passkey display msg type.
#define GAP_MSG_LE_BOND_PASSKEY_INPUT       0x07 // Bond passkey input msg type.
#define GAP_MSG_LE_BOND_OOB_INPUT           0x08 // Bond passkey oob input msg type.
#define GAP_MSG_LE_BOND_USER_CONFIRMATION   0x09 // Bond user confirmation msg type.
#define GAP_MSG_LE_BOND_JUST_WORK           0x0A // Bond user confirmation msg type.
```

### 2.1.4.2 GAP API 消息

在 gap\_callback\_le.h 中定义 GAP API 消息，在每个 API 的注释和示例代码中介绍该 API 对应的消息以及处理方法。

```
/* GAP API message */
#define GAP_MSG_LE_MODIFY_WHITE_LIST         0x01 // response msg type for le_modify_white_list
#define GAP_MSG_LE_SET_RAND_ADDR            0x02 // response msg type for le_set_rand_addr
#define GAP_MSG_LE_SET_HOST_CHANN_CLASSIF   0x03 // response msg type for le_set_host_chann_classif
#define GAP_MSG_LE_WRITE_DEFAULT_DATA_LEN   0x04 // response msg type for le_write_default_data_len
#define GAP_MSG_LE_READ_RSSI                0x10 // response msg type for le_read_rssi
#define GAP_MSG_LE_SET_DATA_LEN             0x13 // response msg type for le_set_data_len
#define GAP_MSG_LE_DATA_LEN_CHANGE_INFO     0x14 // Notification msg type for data length changed
#define GAP_MSG_LE_CONN_UPDATE_IND          0x15 // Indication for le connection parameter update
#define GAP_MSG_LE_CREATE_CONN_IND          0x16 // Indication for create le connection
#define GAP_MSG_LE_PHY_UPDATE_INFO          0x17 // Indication for le physical update information
#define GAP_MSG_LE_REMOTE_FEATS_INFO        0x19 // Information for remote device supported features
#define GAP_MSG_LE_BOND_MODIFY_INFO         0x20 // Notification msg type for bond modify
#define GAP_MSG_LE_SCAN_INFO                0x30 // Notification msg type for le scan
#define GAP_MSG_LE_ADV_UPDATE_PARAM         0x40 // response msg type for le_adv_update_param
```

### 2.1.5 APP 消息流

APP 消息流如图 2-8 所示，实线框内的步骤是必要步骤，虚线框内的步骤是可选步骤。

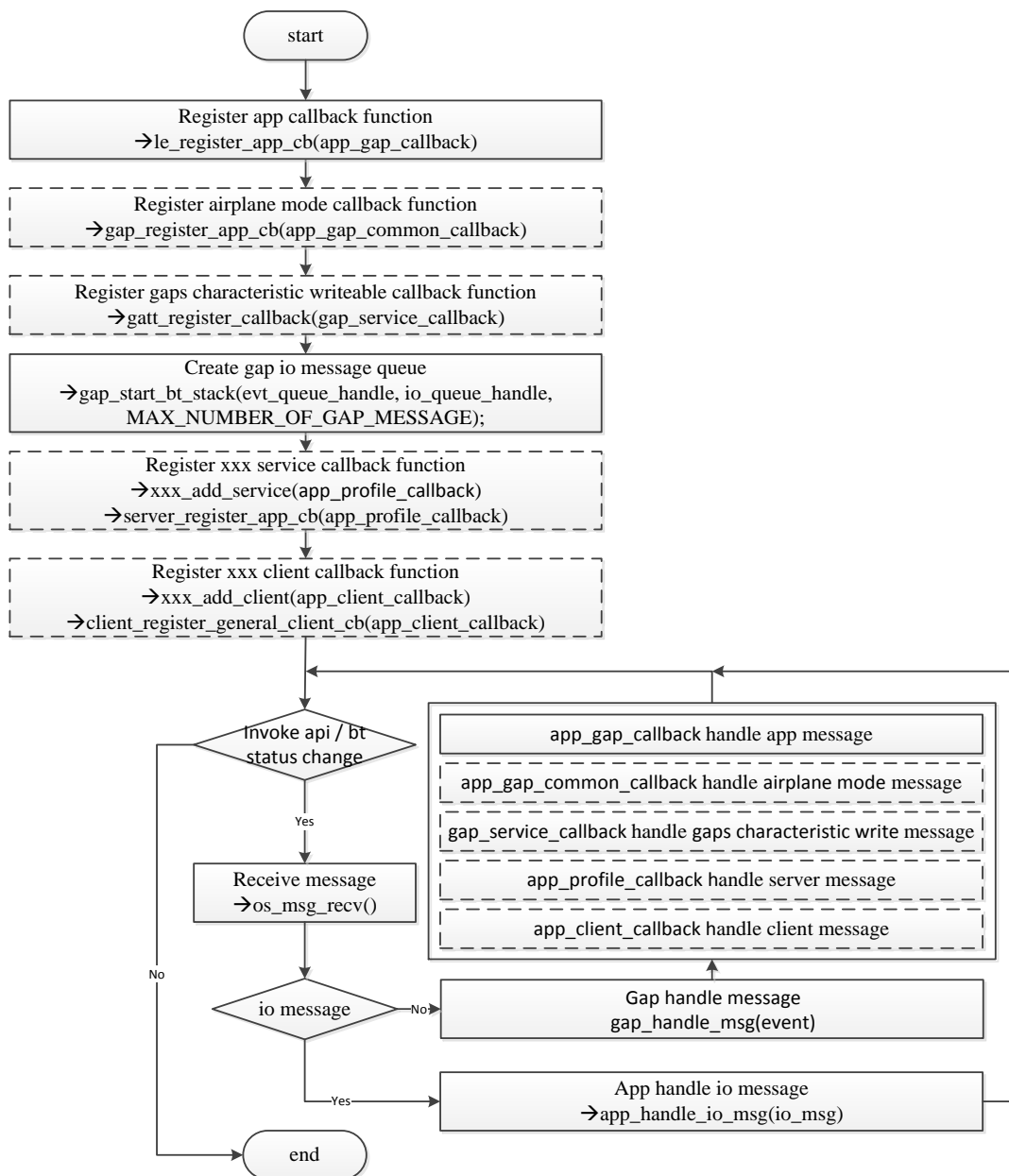


图 2-8 APP 消息流

## 1. 发送消息给 APP 的两种方法

### 1) 回调函数

首先，APP 需要注册回调函数。当上行消息送到 GAP 层之后，GAP 层将调用注册的回调函数以通知 APP 处理该消息。

### 2) 消息队列

首先，APP 需要创建消息队列。当上行消息送到 GAP 层之后，GAP 层将消息发送到消息队列，APP 将循环接收消息队列中的消息。

## 2. 初始化

### 1) 注册回调函数

- (1) 为接收 GAP API 消息, APP 需要通过 `le_register_app_cb()`注册 APP 回调函数。
- (2) 为接收 Vendor Command 消息, APP 需要通过 `gap_register_app_cb()`注册 APP 回调函数。
- (3) 为接收写 GAPS characteristic 的消息, APP 需要通过 `gatt_register_callback()`注册 APP 回调函数。
- (4) 若使用 Peripheral 角色的 APP 需要使用 services, 为接收 server 消息, APP 需要通过 `xxx_add_service()` 和 `server_register_app_cb()`注册 service 回调函数。
- (5) 若使用 Central 角色的 APP 需要使用 clients, 为接收 client 消息, APP 需要通过 `xxx_add_client()` 注册 client 回调函数。

### 2) 创建消息队列

为接收蓝牙状态消息, APP 需要通过 `gap_start_bt_stack()`创建 IO 消息队列。

### 3. 循环接收消息

APP main task 循环接收消息。若收到的事件是发送给 APP 的, APP 收到 IO 消息, 那么 APP 将调用 `app_handle_io_msg()`由 APP 处理该消息。否则, APP 将调用 `gap_handle_msg()`由 GAP 层处理该消息。

### 4. 消息处理

若消息是由回调函数发送的, 则由初始化过程中注册的函数处理该消息。若消息是由消息队列发送的, 则由另一个函数处理该消息。

## 2.2 GAP 的初始化和启动流程

本节介绍如何在 `app_le_gap_init()`中配置 LE GAP 参数以及 GAP 内部启动流程。

### 2.2.1 GAP 参数的初始化

通过修改 `app_le_gap_init()`函数的代码, 在 `main.c` 中实现 GAP 参数的初始化。

#### 2.2.1.1 Device Name 和 Device Appearance 的配置

在 `gap_le.h` 的 `T_GAP_LE_PARAM_TYPE` 中定义参数类型。

##### 2.2.1.1.1 Device Name 的配置

Device Name 的配置用于设置该设备 GAP Service 中 Device Name Characteristic 的值。若在 Advertising 数据中设置 Device Name, 那么 Advertising 数据中的 Device Name 需要与 GAP Service 的 Device Name Characteristic 的值相同, 否则会出现互操作性问题。

```
/** @brief GAP - Advertisement data (max size = 31 bytes, best kept short to conserve power) */
static const uint8_t adv_data[] = {
    /* Flags */
    0x02, /* length */
    GAP_ADTYPE_FLAGS, /* type="Flags" */
}
```

```

GAP_ADTYPE_FLAGS_LIMITED | GAP_ADTYPE_FLAGS_BREDR_NOT_SUPPORTED,
/* Service */
0x03,                /* length */
GAP_ADTYPE_16BIT_COMPLETE,
LO_WORD(GATT_UUID_SIMPLE_PROFILE),
HI_WORD(GATT_UUID_SIMPLE_PROFILE),
/* Local name */
0x0F,                /* length */
GAP_ADTYPE_LOCAL_NAME_COMPLETE,
'B', 'L', 'E', '_', 'P', 'E', 'R', 'I', 'P', 'H', 'E', 'R', 'A', 'L',
};
void app_le_gap_init(void)
{
    /* Device name and device appearance */
    uint8_t device_name[GAP_DEVICE_NAME_LEN] = "BLE_PERIPHERAL";
    .....
    /* Set device name and device appearance */
    le_set_gap_param(GAP_PARAM_DEVICE_NAME, GAP_DEVICE_NAME_LEN, device_name);
    .....
}

```

BT Stack 目前支持的 Device Name 字符串的最大长度是 40 字节（包括结束符）。如果 device name 字符串超过 40 字节，则会出现字符串被截断的情况。

```

#define GAP_DEVICE_NAME_LEN          (39+1) //!< Max length of device name, if device name length
exceeds it, it will be truncated.

```

### 2.2.1.1.2 Device Appearance 的配置

Device Appearance 的配置用于设置该设备 GAP Service 中 Device Appearance Characteristic 的值。若在 Advertising 数据中设置 Device Appearance, 那么 Advertising 数据中的 Device Appearance 需要与 GAP Service 的 Device Appearance Characteristic 的值相同，否则会出现互操作性问题。

Device Appearance 用于描述设备的类型，例如键盘、鼠标、温度计、血压计等。在 gap\_le\_types.h 中定义可以使用的数值。

```

/** @defgroup GAP_LE_APPEARANCE_VALUES GAP Appearance Values
 * @{
 */
#define GAP_GATT_APPEARANCE_UNKNOWN          0
#define GAP_GATT_APPEARANCE_GENERIC_PHONE   64
#define GAP_GATT_APPEARANCE_GENERIC_COMPUTER 128
#define GAP_GATT_APPEARANCE_GENERIC_WATCH   192
#define GAP_GATT_APPEARANCE_WATCH_SPORTS_WATCH 193

```

示例代码如下所示。

```

/** @brief GAP - scan response data (max size = 31 bytes) */
static const uint8_t scan_rsp_data[] = {

```

```

0x03,                                /* length */
GAP_ADTYPE_APPEARANCE,               /* type="Appearance" */
LO_WORD(GAP_GATT_APPEARANCE_UNKNOWN),
HI_WORD(GAP_GATT_APPEARANCE_UNKNOWN),
};
void app_le_gap_init(void)
{
    /* Device name and device appearance */
    uint16_t appearance = GAP_GATT_APPEARANCE_UNKNOWN;
    .....
    /* Set device name and device appearance */
    le_set_gap_param(GAP_PARAM_APPEARANCE, sizeof(appearance), &appearance);
    .....
}

```

### 2.2.1.2 Advertising 参数的配置

在 gap\_adv.h 的 T\_LE\_ADV\_PARAM\_TYPE 中定义 Advertising 参数类型。用户可以配置的 Advertising 参数如下所示：

```

void app_le_gap_init(void)
{
    /* Advertising parameters */
    uint8_t adv_evt_type = GAP_ADTYPE_ADV_IND;
    uint8_t adv_direct_type = GAP_REMOTE_ADDR_LE_PUBLIC;
    uint8_t adv_direct_addr[GAP_BD_ADDR_LEN] = {0};
    uint8_t adv_chann_map = GAP_ADVCHAN_ALL;
    uint8_t adv_filter_policy = GAP_ADV_FILTER_ANY;
    uint16_t adv_int_min = DEFAULT_ADVERTISING_INTERVAL_MIN;
    uint16_t adv_int_max = DEFAULT_ADVERTISING_INTERVAL_MAX;
    .....
    /* Set advertising parameters */
    le_adv_set_param(GAP_PARAM_ADV_EVENT_TYPE, sizeof(adv_evt_type), &adv_evt_type);
    le_adv_set_param(GAP_PARAM_ADV_DIRECT_ADDR_TYPE, sizeof(adv_direct_type), &adv_direct_type);
    le_adv_set_param(GAP_PARAM_ADV_DIRECT_ADDR, sizeof(adv_direct_addr), adv_direct_addr);
    le_adv_set_param(GAP_PARAM_ADV_CHANNEL_MAP, sizeof(adv_chann_map), &adv_chann_map);
    le_adv_set_param(GAP_PARAM_ADV_FILTER_POLICY, sizeof(adv_filter_policy), &adv_filter_policy);
    le_adv_set_param(GAP_PARAM_ADV_INTERVAL_MIN, sizeof(adv_int_min), &adv_int_min);
    le_adv_set_param(GAP_PARAM_ADV_INTERVAL_MAX, sizeof(adv_int_max), &adv_int_max);
    le_adv_set_param(GAP_PARAM_ADV_DATA, sizeof(adv_data), (void *)adv_data);
    le_adv_set_param(GAP_PARAM_SCAN_RSP_DATA, sizeof(scan_rsp_data), (void *)scan_rsp_data);
}

```

adv\_evt\_type 表示 Advertising 类型，不同类型的 advertising 需要不同的参数，如表 2-1 所示。

表 2-1 Advertising 参数设置

adv_evt_type	GAP_ADTYPE_ ADV_IND	GAP_ADTYPE_ ADV_HDC_DIR ECT_IND	GAP_ADTYPE_ ADV_SCAN_IN D	GAP_ADTYPE_ ADV_NONCON N_IND	GAP_ADTYPE_AD V_LDC_DIRECT_I ND
adv_int_min	Y	Ignore	Y	Y	Y
adv_int_max	Y	Ignore	Y	Y	Y
adv_direct_type	Ignore	Y	Ignore	Ignore	Y
adv_direct_addr	Ignore	Y	Ignore	Ignore	Y
adv_chann_map	Y	Y	Y	Y	Y
adv_filter_policy	Y	Ignore	Y	Y	Ignore
allow establish link	Y	Y	N	N	Y

### 2.2.1.3 Scan 参数的配置

在 gap\_scan.h 的 T\_LE\_SCAN\_PARAM\_TYPE 中定义 Scan 参数类型。用户可以配置的 Scan 参数如下所示：

```
void app_le_gap_init(void)
{
    /* Scan parameters */
    uint8_t scan_mode = GAP_SCAN_MODE_ACTIVE;
    uint16_t scan_interval = DEFAULT_SCAN_INTERVAL;
    uint16_t scan_window = DEFAULT_SCAN_WINDOW;
    uint8_t scan_filter_policy = GAP_SCAN_FILTER_ANY;
    uint8_t scan_filter_duplicate = GAP_SCAN_FILTER_DUPLICATE_ENABLE;
    .....
    /* Set scan parameters */
    le_scan_set_param(GAP_PARAM_SCAN_MODE, sizeof(scan_mode), &scan_mode);
    le_scan_set_param(GAP_PARAM_SCAN_INTERVAL, sizeof(scan_interval), &scan_interval);
    le_scan_set_param(GAP_PARAM_SCAN_WINDOW, sizeof(scan_window), &scan_window);
    le_scan_set_param(GAP_PARAM_SCAN_FILTER_POLICY, sizeof(scan_filter_policy),
        &scan_filter_policy);
    le_scan_set_param(GAP_PARAM_SCAN_FILTER_DUPLICATES, sizeof(scan_filter_duplicate),
        &scan_filter_duplicate);
}
```

参数描述：

1. *scan\_mode* - T\_GAP\_SCAN\_MODE
2. *scan\_interval* - scan interval 的取值范围：0x0004 - 0x4000 (单位为 625us)
3. *scan\_window* - scan window 的取值范围：0x0004 - 0x4000 (单位为 625us)
4. *scan\_filter\_policy* - T\_GAP\_SCAN\_FILTER\_POLICY



5. *scan\_filter\_duplicate* - T\_GAP\_SCAN\_FILTER\_DUPLICATE。该参数用于决定是否过滤重复的 Advertising 数据，当 *scan\_filter\_policy* 参数为 GAP\_SCAN\_FILTER\_DUPLICATE\_ENABLE 时，将在协议栈中过滤重复的 Advertising 数据，且不会通知 APP。

### 2.2.1.4 Bond Manager 参数的配置

在 *gap.h* 的 T\_GAP\_PARAM\_TYPE 中和 *gap\_bond\_le.h* 的 T\_LE\_BOND\_PARAM\_TYPE 中定义参数类型。用户可以配置的 Bond Manager 参数如下所示：

```
void app_le_gap_init(void)
{
    /* GAP Bond Manager parameters */
    uint8_t  auth_pair_mode = GAP_PAIRING_MODE_PAIRABLE;
    uint16_t auth_flags = GAP_AUTHEN_BIT_BONDING_FLAG;
    uint8_t  auth_io_cap = GAP_IO_CAP_NO_INPUT_NO_OUTPUT;
    uint8_t  auth_oob = false;
    uint8_t  auth_use_fix_passkey = false;
    uint32_t auth_fix_passkey = 0;
    uint8_t  auth_sec_req_enable = false;
    uint16_t auth_sec_req_flags = GAP_AUTHEN_BIT_BONDING_FLAG;
    .....
    /* Setup the GAP Bond Manager */
    gap_set_param(GAP_PARAM_BOND_PAIRING_MODE, sizeof(auth_pair_mode), &auth_pair_mode);
    gap_set_param(GAP_PARAM_BOND_AUTHEN_REQUIREMENTS_FLAGS, sizeof(auth_flags), &auth_flags);
    gap_set_param(GAP_PARAM_BOND_IO_CAPABILITIES, sizeof(auth_io_cap), &auth_io_cap);
    gap_set_param(GAP_PARAM_BOND_OOB_ENABLED, sizeof(auth_oob), &auth_oob);
    le_bond_set_param(GAP_PARAM_BOND_FIXED_PASSKEY, sizeof(auth_fix_passkey), &auth_fix_passkey);
    le_bond_set_param(GAP_PARAM_BOND_FIXED_PASSKEY_ENABLE, sizeof(auth_use_fix_passkey),
                      &auth_use_fix_passkey);
    le_bond_set_param(GAP_PARAM_BOND_SEC_REQ_ENABLE, sizeof(auth_sec_req_enable),
                      &auth_sec_req_enable);
    le_bond_set_param(GAP_PARAM_BOND_SEC_REQ_REQUIREMENT, sizeof(auth_sec_req_flags),
                      &auth_sec_req_flags);
}
```

参数描述：

1. *auth\_pair\_mode* - 决定设备是否处于可配对模式。
  - 1) GAP\_PAIRING\_MODE\_PAIRABLE: 设备处于可配对模式。
  - 2) GAP\_PAIRING\_MODE\_NO\_PAIRING: 设备处于不可配对模式。
2. *auth\_flags* - 表示要求的 security 属性的位域。
  - 1) GAP\_AUTHEN\_BIT\_NONE
  - 2) GAP\_AUTHEN\_BIT\_BONDING\_FLAG
  - 3) GAP\_AUTHEN\_BIT\_MITM\_FLAG



- 4) GAP\_AUTHEN\_BIT\_SC\_FLAG
- 5) GAP\_AUTHEN\_BIT\_FORCE\_BONDING\_FLAG
- 6) GAP\_AUTHEN\_BIT\_SC\_ONLY\_FLAG
3. *auth\_io\_cap* - T\_GAP\_IO\_CAP，表示设备的输入输出能力。
4. *auth\_oob* - 表示是否使能 Out of Band (OOB)。
  - 1) true：设置 OOB 标志位
  - 2) false：未设置 OOB 标志位
5. *auth\_use\_fix\_passkey* - 表示当配对方法为 passkey entry 且本地设备需要生成 passkey 时，是使用随机生成的 passkey 还是固定的 passkey。
  - 1) true：使用固定的 passkey
  - 2) false：使用随机生成的 passkey
6. *auth\_fix\_passkey* - 配对时使用的固定 passkey 的值，当 auth\_use\_fix\_passkey 参数为 true 时 auth\_fix\_passkey 参数是有效的。
7. *auth\_sec\_req\_enable* - 决定在建立 connection 之后，是否发起配对流程。
8. *auth\_sec\_req\_flags* - 表示要求的 security 属性的位域。

## 2.2.1.5 其它参数的配置

### 2.2.1.5.1 GAP\_PARAM\_SLAVE\_INIT\_GATT\_MTU\_REQ 的配置

```
void app_le_gap_init(void)
{
    uint8_t slave_init_mtu_req = false;
    .....
    le_set_gap_param(GAP_PARAM_SLAVE_INIT_GATT_MTU_REQ, sizeof(slave_init_mtu_req),
                    &slave_init_mtu_req);
    .....
}
```

该参数仅适用于 Peripheral 角色，决定在建立 connection 后是否主动发送 exchange MTU request。

## 2.2.2 GAP 启动流程

### 1. 在 main()中初始化 GAP

```
int main(void)
{
    .....
    le_gap_init(APP_MAX_LINKS);
    app_le_gap_init();
    app_le_profile_init();
    .....
}
```

}

- 1) *le\_gap\_init()* - 初始化 GAP 并设置 link 数目
- 2) *app\_le\_gap\_init()* - GAP 参数的初始化
- 3) *app\_le\_profile\_init()* - 初始化基于 GATT 的 Profiles

## 2. 在 app task 中启动蓝牙协议层

```
void app_main_task(void *p_param)
{
    uint8_t event;
    os_msg_queue_create(&io_queue_handle, MAX_NUMBER_OF_IO_MESSAGE, sizeof(T_IO_MSG));
    os_msg_queue_create(&evt_queue_handle, MAX_NUMBER_OF_EVENT_MESSAGE, sizeof(uint8_t));
    gap_start_bt_stack(evt_queue_handle, io_queue_handle, MAX_NUMBER_OF_GAP_MESSAGE);
    .....
}
```

APP 需要调用 *gap\_start\_bt\_stack()* 来启动蓝牙协议层和 GAP 初始化流程。

## 2.3 BLE GAP 消息

### 2.3.1 概述

本节介绍 BLE GAP 消息模块，在 *gap\_msg.h* 中定义 GAP 消息类型和消息数据结构。BLE GAP 消息可以分为以下三种类型：

- *Device 状态消息*
- *Connection 相关消息*
- *Authentication 相关消息*

BLE GAP 消息处理流程如下所示：

1. APP 调用 *gap\_start\_bt\_stack()* 初始化 BLE GAP 消息模块，初始化代码如下所示：

```
void app_main_task(void *p_param)
{
    uint8_t event;
    os_msg_queue_create(&io_queue_handle, MAX_NUMBER_OF_IO_MESSAGE, sizeof(T_IO_MSG));
    os_msg_queue_create(&evt_queue_handle, MAX_NUMBER_OF_EVENT_MESSAGE, sizeof(uint8_t));
    gap_start_bt_stack(evt_queue_handle, io_queue_handle, MAX_NUMBER_OF_GAP_MESSAGE);
    .....
}
```

2. GAP 向 *io\_queue\_handle* 发送 GAP 消息，APP task 接收到 GAP 消息，并调用 *app\_handle\_io\_msg()* 来处理该消息。(事件：EVENT\_IO\_TO\_APP，类型：IO\_MSG\_TYPE\_BT\_STATUS)

```
void app_main_task(void *p_param)
{
    .....
    while (true)
```

```

{
    if (os_msg_rcv(evt_queue_handle, &event, 0xFFFFFFFF) == true)
    {
        if (event == EVENT_IO_TO_APP)
        {
            T_IO_MSG io_msg;
            if (os_msg_rcv(io_queue_handle, &io_msg, 0) == true)
            {
                app_handle_io_msg(io_msg);
            }
        }
        .....
    }
}

```

3. GAP 消息处理函数如下所示:

```

void app_handle_io_msg(T_IO_MSG io_msg)
{
    uint16_t msg_type = io_msg.type;
    switch (msg_type)
    {
        case IO_MSG_TYPE_BT_STATUS:
        {
            app_handle_gap_msg(&io_msg);
        }
        break;
        default:
            break;
    }
}

void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    T_LE_GAP_MSG gap_msg;
    uint8_t conn_id;
    memcpy(&gap_msg, &p_gap_msg->u.param, sizeof(p_gap_msg->u.param));

    APP_PRINT_TRACE1("app_handle_gap_msg: subtype %d", p_gap_msg->subtype);
    switch (p_gap_msg->subtype)
    {
        case GAP_MSG_LE_DEV_STATE_CHANGE:
        {
            app_handle_dev_state_evt(gap_msg.msg_data.gap_dev_state_change.new_state,
                                     gap_msg.msg_data.gap_dev_state_change.cause);
        }
    }
}

```

```

        break;

        .....
    }

```

## 2.3.2 Device 状态消息

### 2.3.2.1 GAP\_MSG\_LE\_DEV\_STATE\_CHANGE

该消息用于通知 GAP Device 状态(T\_GAP\_DEV\_STATE)，GAP Device 状态包括以下五种子状态：

- **gap\_init\_state** : GAP 初始化状态
- **gap\_adv\_state** : GAP Advertising 状态
- **gap\_adv\_sub\_state**: GAP Advertising 子状态，该状态仅适用于 gap\_adv\_state 为 GAP\_ADV\_STATE\_IDLE 的情况。
- **gap\_scan\_state** : GAP Scan 状态
- **gap\_conn\_state** : GAP Connection 状态

消息数据结构为 T\_GAP\_DEV\_STATE\_CHANGE。

```

/** @brief Device State.*/
typedef struct
{
    uint8_t gap_init_state: 1;  //!< @ref GAP_INIT_STATE
    uint8_t gap_adv_sub_state: 1;  //!< @ref GAP_ADV_SUB_STATE
    uint8_t gap_adv_state: 2;  //!< @ref GAP_ADV_STATE
    uint8_t gap_scan_state: 2;  //!< @ref GAP_SCAN_STATE
    uint8_t gap_conn_state: 2;  //!< @ref GAP_CONN_STATE
} T_GAP_DEV_STATE;

/** @brief The msg_data of GAP_MSG_LE_DEV_STATE_CHANGE.*/
typedef struct
{
    T_GAP_DEV_STATE new_state;
    uint16_t cause;
} T_GAP_DEV_STATE_CHANGE;

```

示例代码如下所示：

```

void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    APP_PRINT_INFO4("app_handle_dev_state_evt: init state %d, adv state %d, scan state %d, cause 0x%x",
                    new_state.gap_init_state, new_state.gap_adv_state,
                    new_state.gap_scan_state, cause);
    if (gap_dev_state.gap_init_state != new_state.gap_init_state)
    {
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)

```

```

    {
        APP_PRINT_INFO0("GAP stack ready");
    }
}

if (gap_dev_state.gap_scan_state != new_state.gap_scan_state)
{
    if (new_state.gap_scan_state == GAP_SCAN_STATE_IDLE)
    {
        APP_PRINT_INFO0("GAP scan stop");
    }
    else if (new_state.gap_scan_state == GAP_SCAN_STATE_SCANNING)
    {
        APP_PRINT_INFO0("GAP scan start");
    }
}
if (gap_dev_state.gap_adv_state != new_state.gap_adv_state)
{
    if (new_state.gap_adv_state == GAP_ADV_STATE_IDLE)
    {
        if (new_state.gap_adv_sub_state == GAP_ADV_TO_IDLE_CAUSE_CONN)
        {
            APP_PRINT_INFO0("GAP adv stoped: because connection created");
        }
        else
        {
            APP_PRINT_INFO0("GAP adv stoped");
        }
    }
    else if (new_state.gap_adv_state == GAP_ADV_STATE_ADVERTISING)
    {
        APP_PRINT_INFO0("GAP adv start");
    }
}
gap_dev_state = new_state;
}

```

## 2.3.3 Connection 相关消息

### 2.3.3.1 GAP\_MSG\_LE\_CONN\_STATE\_CHANGE

该消息用于通知 Link 状态 (T\_GAP\_CONN\_STATE), 其数据结构为 T\_GAP\_CONN\_STATE\_CHANGE。

示例代码如下所示:

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    APP_PRINT_INFO4("app_handle_conn_state_evt: conn_id %d old_state %d new_state %d, disc_cause 0x%x",
                    conn_id, gap_conn_state, new_state, disc_cause);
    switch (new_state)
    {
        case GAP_CONN_STATE_DISCONNECTED:
        {
            if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
                && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
            {
                APP_PRINT_ERROR1("app_handle_conn_state_evt: connection lost cause 0x%x", disc_cause);
            }
            le_adv_start();
        }
        break;
        case GAP_CONN_STATE_CONNECTED:
        {
            .....
        }
        break;
        default:
        break;
    }
    gap_conn_state = new_state;
}
```

### 2.3.3.2 GAP\_MSG\_LE\_CONN\_PARAM\_UPDATE

该消息用于通知 connection 参数更新状态，更新状态包括三个子状态：

- **GAP\_CONN\_PARAM\_UPDATE\_STATUS\_PENDING**: 若本地设备调用 le\_update\_conn\_param()更新 Connection 参数，当 Connection 参数更新请求成功但未收到 connection update complete event 时，GAP 层将发送该状态消息。
- **GAP\_CONN\_PARAM\_UPDATE\_STATUS\_SUCCESS**: 更新成功。
- **GAP\_CONN\_PARAM\_UPDATE\_STATUS\_FAIL**: 更新失败，参数 cause 表示失败原因。

消息数据结构为 T\_GAP\_CONN\_PARAM\_UPDATE。示例代码如下所示：

```
void app_handle_conn_param_update_evt(uint8_t conn_id, uint8_t status, uint16_t cause)
{
    switch (status)
    {
        case GAP_CONN_PARAM_UPDATE_STATUS_SUCCESS:
            .....
    }
```

```

        break;
    case GAP_CONN_PARAM_UPDATE_STATUS_FAIL:
        .....
        break;
    case GAP_CONN_PARAM_UPDATE_STATUS_PENDING:
        .....
        break;
    }
}

```

### 2.3.3.3 GAP\_MSG\_LE\_CONN\_MTU\_INFO

该消息用于通知 exchange MTU (Maximum Transmission Unit) procedure 已完成。exchange MTU procedure 的目的是更新 client 和 server 之间交互数据包的最大长度，即更新 ATT\_MTU。消息数据结构为 T\_GAP\_CONN\_MTU\_INFO，示例代码如下所示：

```

void app_handle_conn_mtu_info_evt(uint8_t conn_id, uint16_t mtu_size)
{
    APP_PRINT_INFO2("app_handle_conn_mtu_info_evt: conn_id %d, mtu_size %d", conn_id, mtu_size);
}

```

## 2.3.4 Authentication 相关消息

配对方法与 Authentication 消息的对应关系如表 2-2 所示：

表 2-2 Authentication 相关消息

Pairing Method	Message
Just Works	GAP_MSG_LE_BOND_JUST_WORK
Numeric Comparison	GAP_MSG_LE_BOND_USER_CONFIRMATION
Passkey Entry	GAP_MSG_LE_BOND_PASSKEY_INPUT GAP_MSG_LE_BOND_PASSKEY_DISPLAY

### 2.3.4.1 GAP\_MSG\_LE\_AUTHEN\_STATE\_CHANGE

该消息表示新的 Authentication 状态。

- **GAP\_AUTHEN\_STATE\_STARTED** : Authentication 流程已开始。
- **GAP\_AUTHEN\_STATE\_COMPLETE**: Authentication 流程已完成，参数 cause 表示 Authentication 的结果。

消息数据结构为 T\_GAP\_AUTHEN\_STATE，示例代码如下所示：

```

void app_handle_authen_state_evt(uint8_t conn_id, uint8_t new_state, uint16_t cause)
{
    APP_PRINT_INFO2("app_handle_authen_state_evt:conn_id %d, cause 0x%x", conn_id, cause);
}

```

```

switch (new_state)
{
case GAP_AUTHEN_STATE_STARTED:
{
    APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_STARTED");
}
    break;
case GAP_AUTHEN_STATE_COMPLETE:
{
    if (cause == GAP_SUCCESS)
    {
        APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE pair
            success");
    }
    else
    {
        APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE pair
            failed");
    }
}
    break;
default:
    break;
}
}

```

### 2.3.4.2 GAP\_MSG\_LE\_BOND\_PASSKEY\_DISPLAY

该消息用于表示配对方法为 Passkey Entry，且本地设备需要显示 Passkey。

在本地设备显示 Passkey，且对端设备需要输入相同的 Passkey。一旦收到该消息，APP 可以在用户终端界面显示 Passkey（处理 Passkey 的方法取决于 APP），此外 APP 需要调用 le\_bond\_passkey\_display\_confirm() 确认是否与对端设备配对。

消息数据结构为 T\_GAP\_BOND\_PASSKEY\_DISPLAY。示例代码如下所示：

```

void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    .....
    case GAP_MSG_LE_BOND_PASSKEY_DISPLAY:
    {
        uint32_t display_value = 0;
        conn_id = gap_msg.msg_data.gap_bond_passkey_display.conn_id;
        le_bond_get_display_key(conn_id, &display_value);
        APP_PRINT_INFO1("GAP_MSG_LE_BOND_PASSKEY_DISPLAY:passkey %d", display_value);
        le_bond_passkey_display_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
    }
}

```



```

    }
    break;
}

```

### 2.3.4.3 GAP\_MSG\_LE\_BOND\_PASSKEY\_INPUT

该消息用于表示配对方法为 Passkey Entry，且本地设备需要输入 Passkey。

对端设备会显示 Passkey，且本地设备需要输入相同的 Passkey。一旦收到该消息，APP 需要调用 `le_bond_passkey_input_confirm()` 确认是否与对端设备配对。

消息数据结构为 `T_GAP_BOND_PASSKEY_INPUT`。示例代码如下所示：

```

void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    .....
    case GAP_MSG_LE_BOND_PASSKEY_INPUT:
    {
        uint32_t passkey = 888888;
        conn_id = gap_msg.msg_data.gap_bond_passkey_input.conn_id;
        APP_PRINT_INFO1("GAP_MSG_LE_BOND_PASSKEY_INPUT: conn_id %d", conn_id);
        le_bond_passkey_input_confirm(conn_id, passkey, GAP_CFM_CAUSE_ACCEPT);
    }
    break;
}

```

### 2.3.4.4 GAP\_MSG\_LE\_BOND\_USER\_CONFIRMATION

该消息用于表示配对方法为 Numeric Comparison。

在本地设备和对端设备均显示需要校验的数值，用户需要确认该数值是否相同。APP 需要调用 `le_bond_user_confirm()` 确认是否与对端设备配对。

消息数据结构为 `T_GAP_BOND_USER_CONF`。示例代码如下所示：

```

void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    .....
    case GAP_MSG_LE_BOND_USER_CONFIRMATION:
    {
        uint32_t display_value = 0;
        conn_id = gap_msg.msg_data.gap_bond_user_conf.conn_id;
        le_bond_get_display_key(conn_id, &display_value);
        APP_PRINT_INFO1("GAP_MSG_LE_BOND_USER_CONFIRMATION: passkey %d",
                        display_value);
        le_bond_user_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
    }
    break;
}

```

}

### 2.3.4.5 GAP\_MSG\_LE\_BOND\_JUST\_WORK

该消息用于表示配对方法为 Just Work。APP 需要调用 le\_bond\_just\_work\_confirm() 确认是否与对端设备配对。

消息数据结构为 T\_GAP\_BOND\_JUST\_WORK\_CONF。示例代码如下所示：

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    .....
    case GAP_MSG_LE_BOND_JUST_WORK:
    {
        conn_id = gap_msg.msg_data.gap_bond_just_work_conf.conn_id;
        le_bond_just_work_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
        APP_PRINT_INFO0("GAP_MSG_LE_BOND_JUST_WORK");
    }
    break;
}
```

## 2.4 BLE GAP 回调函数

本节介绍 BLE GAP 回调函数，GAP 层使用注册的回调函数发送消息给 APP。

不同于 BLE GAP 消息，回调函数是直接被 GAP 层调用的。因此，不建议在回调函数内执行耗时操作，耗时操作会使底层处理流程延缓和暂停，在某些情况下会引发异常。若 APP 在收到 GAP 层发送的消息后确实需要执行耗时操作，在 APP 处理该消息时，可以通过 APP 回调函数将该消息发送到 APP 的队列，APP 回调函数将在把消息发送到队列之后结束，因此该操作不会延缓底层处理流程。

BLE GAP 回调函数的使用方法如下所示：

#### 1. 注册回调函数

```
void app_le_gap_init(void)
{
    .....
    le_register_app_cb(app_gap_callback);
}
```

#### 2. 处理 GAP 回调函数消息

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
        case GAP_MSG_LE_DATA_LEN_CHANGE_INFO:
```

```

APP_PRINT_INFO3("GAP_MSG_LE_DATA_LEN_CHANGE_INFO: conn_id %d, tx octets 0x%x,
                max_tx_time 0x%x",
                p_data->p_le_data_len_change_info->conn_id,
                p_data->p_le_data_len_change_info->max_tx_octets,
                p_data->p_le_data_len_change_info->max_tx_time);

        break;
        .....
    }
}

```

## 2.4.1 BLE GAP 回调函数消息概述

本节介绍 GAP 回调函数消息，在 gap\_callback\_le.h 中定义 GAP 回调函数消息类型和消息数据。由于 GAP 层提供的大部分接口都是异步的，因此 GAP 层使用回调函数发送响应信息给 APP。例如，APP 调用 le\_read\_rssi() 读取 Received Signal Strength Indication (RSSI)，若返回值为 GAP\_CAUSE\_SUCCESS，则表示成功发送请求。此时，APP 需要等待 GAP\_MSG\_LE\_READ\_RSSI 消息以获取结果。

BLE GAP 回调函数消息的详细信息如下所示：

### 1. gap\_le.h 相关消息

表 2-3 gap\_le.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_MODIFY_WHITE_LIST	T_LE_MODIFY_WHITE_LIST_RSP *p_le_modify_white_list_rsp;	le_modify_white_list
GAP_MSG_LE_SET_RAND_ADDR	T_LE_SET_RAND_ADDR_RSP *p_le_set_rand_addr_rsp;	le_set_rand_addr
GAP_MSG_LE_SET_HOST_CHANN_CLAS SIF	T_LE_SET_HOST_CHANN_CLASSIF_ RSP *p_le_set_host_chann_classif_rsp;	le_set_host_chann_clas sif

### 2. gap\_conn\_le.h 相关消息

表 2-4 gap\_conn\_le.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_READ_RSSI	T_LE_READ_RSSI_RSP *p_le_read_rssi_rsp;	le_read_rssi
GAP_MSG_LE_SET_DATA_LEN	T_LE_SET_DATA_LEN_RSP *p_le_set_data_len_rsp;	le_set_data_len
GAP_MSG_LE_DATA_LEN_CHANGE_INFO	T_LE_DATA_LEN_CHANGE_INFO *p_le_data_len_change_info;	
GAP_MSG_LE_CONN_UPDATE_IND	T_LE_CONN_UPDATE_IND *p_le_conn_update_ind;	

GAP_MSG_LE_CREATE_CONN_IND	T_LE_CREATE_CONN_IND *p_le_create_conn_ind;
GAP_MSG_LE_PHY_UPDATE_INFO	T_LE_PHY_UPDATE_INFO *p_le_phy_update_info;
GAP_MSG_LE_REMOTE_FEATS_INFO	T_LE_REMOTE_FEATS_INFO *p_le_remote_feats_info;

### 1) GAP\_MSG\_LE\_DATA\_LEN\_CHANGE\_INFO

该消息用于向 APP 通知在 Link Layer 的发送或接收方向其最大 Payload 长度或数据包的最大传输时间的变化。

### 2) GAP\_MSG\_LE\_CONN\_UPDATE\_IND

该消息仅适用于 Central 角色。当对端设备请求更新 Connection 参数时，GAP 层将通过回调函数发送该消息并检查回调函数的返回值。因此，APP 可以返回 APP\_RESULT\_ACCEPT 以接受参数更新，或者返回 APP\_RESULT\_REJECT 以拒绝参数更新。

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    .....
    case GAP_MSG_LE_CONN_UPDATE_IND:
        APP_PRINT_INFO5("GAP_MSG_LE_CONN_UPDATE_IND: conn_id %d, conn_interval_max 0x%x,
            conn_interval_min 0x%x, conn_latency 0x%x, supervision_timeout 0x%x",
            p_data->p_le_conn_update_ind->conn_id,
            p_data->p_le_conn_update_ind->conn_interval_max,
            p_data->p_le_conn_update_ind->conn_interval_min,
            p_data->p_le_conn_update_ind->conn_latency,
            p_data->p_le_conn_update_ind->supervision_timeout);
        /* if reject the proposed connection parameter from peer device, use APP_RESULT_REJECT. */
        result = APP_RESULT_ACCEPT;
        break;
}
```

### 3) GAP\_MSG\_LE\_CREATE\_CONN\_IND

该消息仅适用于 Peripheral 角色。由 APP 决定是否建立 connection。当 central 设备发起 connection 时，默认情况下，GAP 层不会发送该消息并直接接受 connection。若 APP 希望使用该功能，需要将 GAP\_PARAM\_HANDLE\_CREATE\_CONN\_IND 设为 true。示例代码如下所示：

```
void app_le_gap_init(void)
{
    .....
    uint8_t handle_conn_ind = true;
    le_set_gap_param(GAP_PARAM_HANDLE_CREATE_CONN_IND, sizeof(handle_conn_ind),
        &handle_conn_ind);
}

T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
```

```
{
    .....
    case GAP_MSG_LE_CREATE_CONN_IND:
        /* if reject the connection from peer device, use APP_RESULT_REJECT. */
        result = APP_RESULT_ACCEPT;
        break;
}
```

#### 4) GAP\_MSG\_LE\_PHY\_UPDATE\_INFO

该消息表示 Controller 已经切换正在使用的发射机 PHY 或接收机 PHY。

#### 5) GAP\_MSG\_LE\_REMOTE\_FEATS\_INFO

在 connection 建立成功后，controller 会主动读取对端设备的 feature。读取结果会通过该消息通知给 application。

### 3. gap\_bond\_le.h 相关消息

表 2-5 gap\_bond\_le.h 相关消息

Callback type(cb_type)	Callback data (p_cb_data)	Reference API
GAP_MSG_LE_BOND_MODIFY_INFO	T_LE_BOND_MODIFY_INFO *p_le_bond_modify_info;	

#### 1) GAP\_MSG\_LE\_BOND\_MODIFY\_INFO

该消息用于向 APP 通知绑定信息已变更，更多信息参见 [LE 密钥管理](#)。

### 4. gap\_scan.h 相关消息

表 2-6 gap\_scan.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_SCAN_INFO	T_LE_SCAN_INFO *p_le_scan_info;	le_scan_start

#### 1) GAP\_MSG\_LE\_SCAN\_INFO

Scan 状态为 GAP\_SCAN\_STATE\_SCANNING，当蓝牙协议层收到 advertising 数据或 scan response 数据时，GAP 将发送该消息以通知 APP。

### 5. gap\_adv.h 相关消息

表 2-7 gap\_adv.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_ADV_UPDATE_PARAM	T_LE_ADV_UPDATE_PARAM_RSP *p_le_adv_update_param_rsp;	le_adv_update_param

## 2.5 BLE GAP 用例

本节介绍如何使用 BLE GAP 接口，以下为一些典型用例。

### 2.5.1 GAP Service Characteristic 的可写属性

GAP service 的 Device Name characteristic 和 Device Appearance characteristic 具有可选的可写属性，该可写属性默认是关闭的。APP 可以通过调用 `gaps_set_parameter()` 设置 `GAPS_PARAM_APPEARANCE_PROPERTY` 和 `GAPS_PARAM_DEVICE_NAME_PROPERTY` 来配置可写属性。

#### 1. 可写属性的配置

```
void app_le_gap_init(void)
{
    uint8_t appearance_prop = GAPS_PROPERTY_WRITE_ENABLE;
    uint8_t device_name_prop = GAPS_PROPERTY_WRITE_ENABLE;
    T_LOCAL_APPEARANCE appearance_local;
    T_LOCAL_NAME local_device_name;
    if (flash_load_local_appearance(&appearance_local) == 0)
    {
        gaps_set_parameter(GAPS_PARAM_APPEARANCE, sizeof(uint16_t),
                           &appearance_local.local_appearance);
    }
    if (flash_load_local_name(&local_device_name) == 0)
    {
        gaps_set_parameter(GAPS_PARAM_DEVICE_NAME, GAP_DEVICE_NAME_LEN,
                           local_device_name.local_name);
    }
    gaps_set_parameter(GAPS_PARAM_APPEARANCE_PROPERTY, sizeof(appearance_prop),
                       &appearance_prop);
    gaps_set_parameter(GAPS_PARAM_DEVICE_NAME_PROPERTY, sizeof(device_name_prop),
                       &device_name_prop);
    gatt_register_callback(gap_service_callback);
}
```

#### 2. GAP Service 回调函数消息处理

APP 需要调用 `gatt_register_callback()` 以注册回调函数，该回调函数用于处理 GAP service 消息。

```
T_APP_RESULT gap_service_callback(T_SERVER_ID service_id, void *p_para)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_GAPS_CALLBACK_DATA *p_gap_data = (T_GAPS_CALLBACK_DATA *)p_para;
    APP_PRINT_INFO2("gap_service_callback conn_id = %d msg_type = %d\n", p_gap_data->conn_id,
                    p_gap_data->msg_type);
}
```

```

if (p_gap_data->msg_type == SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE)
{
    switch (p_gap_data->msg_data.opcode)
    {
        case GAPS_WRITE_DEVICE_NAME:
        {
            T_LOCAL_NAME device_name;
            memcpy(device_name.local_name, p_gap_data->msg_data.p_value,
                p_gap_data->msg_data.len);
            device_name.local_name[p_gap_data->msg_data.len] = 0;
            flash_save_local_name(&device_name);
        }
        break;
        case GAPS_WRITE_APPEARANCE:
        {
            uint16_t appearance_val;
            T_LOCAL_APPEARANCE appearance;
            LE_ARRAY_TO_UINT16(appearance_val, p_gap_data->msg_data.p_value);
            appearance.local_appearance = appearance_val;
            flash_save_local_appearance(&appearance);
        }
        break;
        default:
            break;
    }
}
return result;
}

```

APP 需要将 device name 和 device appearance 保存到 Flash ， 具体内容参见 [本地协议栈信息存储](#)。

## 2.5.2 本地设备使用 Static Random Address

在 advertising、scanning 和 connection 时，默认使用的 local address type 是 Public Address，可以配置为 Static Random Address。

### 1. Random address 的生成和存储

APP 第一次可以调用 `le_gen_rand_addr()` 生成 static random address。然后将生成的地址存储到 Flash。如果 Flash 已经存储过 random 地址，则可以直接使用。然后调用 `le_set_gap_param()` 的 `GAP_PARAM_RANDOM_ADDR` 来设置 random address。

### 2. 设置 Identity Address

Stack 默认使用 public address 作为 Identity Address。APP 需要调用 `le_cfg_local_identity_address()` 将 Identity Address 修改为 static random address。不正确的 Identity Address 设置会导致配对后无法重连。

### 3. 设置 local address type

Peripheral 角色或 Broadcaster 角色调用 `le_adv_set_param()` 设置 local address type 以使用本地设备的 Static Random Address。Central 角色或 Observer 角色调用 `le_scan_set_param()` 设置 local address type 以使用本地设备的 Static Random Address。示例代码如下所示：

```
void app_le_gap_init(void)
{
    .....
    T_APP_STATIC_RANDOM_ADDR random_addr;
    bool gen_addr = true;
    uint8_t local_bd_type = GAP_LOCAL_ADDR_LE_RANDOM;
    if (app_load_static_random_address(&random_addr) == 0)
    {
        if (random_addr.is_exist == true)
        {
            gen_addr = false;
        }
    }
    if (gen_addr)
    {
        if (le_gen_rand_addr(GAP_RANDOM_ADDR_STATIC, random_addr.bd_addr) == GAP_CAUSE_SUCCESS)
        {
            random_addr.is_exist = true;
            app_save_static_random_address(&random_addr);
        }
    }
    le_cfg_local_identity_address(random_addr.bd_addr, GAP_IDENT_ADDR_RANDOM);
    le_set_gap_param(GAP_PARAM_RANDOM_ADDR, 6, random_addr.bd_addr);
    //only for peripheral, broadcaster
    le_adv_set_param(GAP_PARAM_ADV_LOCAL_ADDR_TYPE, sizeof(local_bd_type), &local_bd_type);
    //only for central, observer
    le_scan_set_param(GAP_PARAM_SCAN_LOCAL_ADDR_TYPE, sizeof(local_bd_type), &local_bd_type);
    .....
}
```

Central 角色调用 `le_connect()` 设置 local address type 以使用本地设备的 Static Random Address。示例代码如下所示：

```
static T_USER_CMD_PARSE_RESULT cmd_condev(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    .....
    T_GAP_LOCAL_ADDR_TYPE local_addr_type = GAP_LOCAL_ADDR_LE_RANDOM;
    .....
    cause = le_connect(GAP_PHYS_CONN_INIT_1M_BIT,
                      dev_list[dev_idx].bd_addr,
```



```

        (T_GAP_REMOTE_ADDR_TYPE)dev_list[dev_idx].bd_type,
        local_addr_type,
        1000);
    .....
}

```

## 2.5.3 Physical (PHY) 设置

LE 中必须实现符号速率为 1 mega symbol per second (Msym/s)，一个 symbol 表示一个 bit，支持的比特率为 1 megabit per second (Mb/s)，即 **LE 1M PHY**。若支持可选符号速率 2 Msym/s，比特率为 2 Mb/s，即 **LE 2M PHY**。2 Msym/s 符号速率只支持未编码的数据，LE 1M PHY 和 LE 2M PHY 统称为 LE Uncoded PHYs<sup>[1]</sup>。

### 1. 设置 Default PHY

APP 可以指定其发射机 PHY 和接收机 PHY 的优先值，用于随后所有建立在 LE transport 上的 connection。

```

void app_le_gap_init(void)
{
    uint8_t phys_prefer = GAP_PHYS_PREFER_ALL;
    uint8_t tx_phys_prefer = GAP_PHYS_PREFER_1M_BIT | GAP_PHYS_PREFER_2M_BIT;
    uint8_t rx_phys_prefer = GAP_PHYS_PREFER_1M_BIT | GAP_PHYS_PREFER_2M_BIT;
    le_set_gap_param(GAP_PARAM_DEFAULT_PHYS_PREFER, sizeof(phys_prefer), &phys_prefer);
    le_set_gap_param(GAP_PARAM_DEFAULT_TX_PHYS_PREFER, sizeof(tx_phys_prefer), &tx_phys_prefer);
    le_set_gap_param(GAP_PARAM_DEFAULT_RX_PHYS_PREFER, sizeof(rx_phys_prefer), &rx_phys_prefer);
}

```

### 2. 读取 connection 的 PHY 类型

成功建立 connection 后，APP 可以调用 le\_get\_conn\_param() 以读取 TX PHY 和 RX PHY 类型。

```

void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    .....
    switch (new_state)
    {
        case GAP_CONN_STATE_CONNECTED:
        {
            .....
            data_uart_print("Connected success conn_id %d\r\n", conn_id);
#ifdef F_BT_LE_5_0_SET_PHY_SUPPORT
            uint8_t tx_phy;
            uint8_t rx_phy;
            le_get_conn_param(GAP_PARAM_CONN_RX_PHY_TYPE, &rx_phy, conn_id);
            le_get_conn_param(GAP_PARAM_CONN_TX_PHY_TYPE, &tx_phy, conn_id);
            APP_PRINT_INFO2("GAP_CONN_STATE_CONNECTED: tx_phy %d, rx_phy %d", tx_phy,
                           rx_phy);
#endif
        }
    }
}

```

```

    }
    break;
}

```

### 3. 检查对端设备的 Features

成功建立 connection 后，蓝牙协议层会读取对端设备的 Features。GAP 层将通过 GAP\_MSG\_LE\_REMOTE\_FEATS\_INFO 向 APP 通知对端设备的 Features，APP 可以检查对端设备是否支持 LE 2M PHY。

```

T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
#ifdef F_BT_LE_5_0_SET_PHY_SUPPORT
        case GAP_MSG_LE_REMOTE_FEATS_INFO:
        {
            uint8_t remote_feats[8];
            APP_PRINT_INFO3("GAP_MSG_LE_REMOTE_FEATS_INFO: conn id %d, cause 0x%x,
                           remote_feats %b",
                           p_data->p_le_remote_feats_info->conn_id,
                           p_data->p_le_remote_feats_info->cause,
                           TRACE_BINARY(8, p_data->p_le_remote_feats_info->remote_feats));
            if (p_data->p_le_remote_feats_info->cause == GAP_SUCCESS)
            {
                memcpy(remote_feats, p_data->p_le_remote_feats_info->remote_feats, 8);
                if (remote_feats[LE_SUPPORT_FEATURES_MASK_ARRAY_INDEX1] &
                    LE_SUPPORT_FEATURES_LE_2M_MASK_BIT)
                {
                    APP_PRINT_INFO0("GAP_MSG_LE_REMOTE_FEATS_INFO: support 2M");
                }
            }
        }
        break;
#endif
    }
}

```

### 4. 切换 PHY

le\_set\_phy()用于设置 connection (由 conn\_id 确定) 的 PHY preferences，而 Controller 有可能不能成功切换 PHY (例如对端设备不支持请求的 PHY)或者认为当前 PHY 更好。

```

static T_USER_CMD_PARSE_RESULT cmd_setphy(T_USER_CMD_PARSED_VALUE *p_parse_value)
{

```

```

uint8_t conn_id = p_parse_value->dw_param[0];
uint8_t all_phys;
uint8_t tx_phys;
uint8_t rx_phys;
T_GAP_PHYS_OPTIONS phy_options = GAP_PHYS_OPTIONS_CODED_PREFER_S8;
T_GAP_CAUSE cause;
if (p_parse_value->dw_param[1] == 0)
{
    all_phys = GAP_PHYS_PREFER_ALL;
    tx_phys = GAP_PHYS_PREFER_1M_BIT;
    rx_phys = GAP_PHYS_PREFER_1M_BIT;
}
else if (p_parse_value->dw_param[1] == 1)
{
    all_phys = GAP_PHYS_PREFER_ALL;
    tx_phys = GAP_PHYS_PREFER_2M_BIT;
    rx_phys = GAP_PHYS_PREFER_2M_BIT;
}
.....
cause = le_set_phy(conn_id, all_phys, tx_phys, rx_phys, phy_options);
return (T_USER_CMD_PARSE_RESULT)cause;
}

```

## 5. PHY 的更新

GAP\_MSG\_LE\_PHY\_UPDATE\_INFO 用于向 APP 通知 Controller 使用的发射机 PHY 或接收机 PHY 的更新结果。

```

T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
        #if F_BT_LE_5_0_SET_PHY_SUPPORT
        case GAP_MSG_LE_PHY_UPDATE_INFO:
            APP_PRINT_INFO4("GAP_MSG_LE_PHY_UPDATE_INFO:conn_id %d, cause 0x%x, rx_phy %d,
                            tx_phy %d",
                            p_data->p_le_phy_update_info->conn_id,
                            p_data->p_le_phy_update_info->cause,
                            p_data->p_le_phy_update_info->rx_phy,
                            p_data->p_le_phy_update_info->tx_phy);

            break;
        #endif
    }
}

```

## 2.6 GAP 信息存储

在 `gap_storage_le.h` 中定义常量和函数原型。本地协议栈信息和绑定信息保存在 FTL 中，更多关于 FTL 的信息参见 [FTL 简介](#)。

### 2.6.1 FTL 简介

BT stack 和 user application 使用 FTL 作为抽象层保存或载入 flash 中的数据。

#### 2.6.1.1 FTL 布局

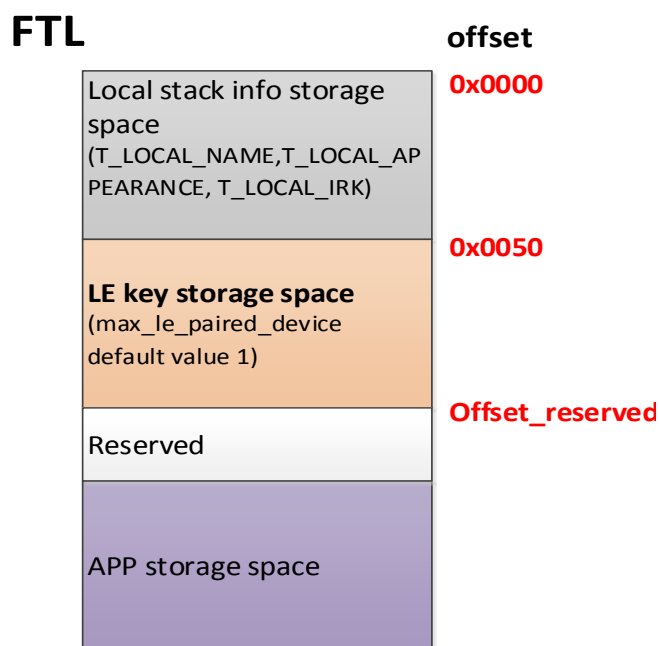


图 2-9 FTL 布局

FTL 可以分为以下四个区域：

1. Local stack information storage space
  - 1) 地址范围: 0x0000 - 0x004F
  - 2) 该区域用于存储本地协议栈信息，包括 device name、device appearance 和本地设备 IRK。更多信息参见 [本地协议栈信息存储](#)。
2. LE key storage space
  - 1) 地址范围: 0x0050 - (Offset\_reserved - 1)
  - 2) 该区域用于存储 LE 密钥信息，更多信息参见 [绑定信息存储](#)。
3. APP storage space
  - 1) APP 可以使用该区域存储信息。

## 2.6.2 本地协议栈信息存储

### 2.6.2.1 Device Name 存储

GAP 层目前支持的 device name 字符串的最大长度是 40 字节（包括结束符）。

flash\_save\_local\_name()用于保存 device name 到 FTL。

flash\_load\_local\_name()用于从 FTL 载入 device name。

若 GAP service 的 Device Name characteristic 是可写的，APP 可以调用该函数保存 device name。示例代码参见 [GAP Service Characteristic 的可写属性](#)。

### 2.6.2.2 Device Appearance 存储

Device Appearance 用于描述设备的类型，例如键盘、鼠标、温度计、血压计等。

flash\_save\_local\_appearance()用于保存 device appearance 到 FTL。

flash\_load\_local\_appearance()用于从 FTL 载入 device appearance。

若 GAP service 的 Device Appearance characteristic 是可写的，APP 可以调用该函数保存 device appearance。示例代码参见 [GAP Service Characteristic 的可写属性](#)。

## 2.6.3 绑定信息存储

### 2.6.3.1 绑定设备优先级管理

GAP 层实现绑定设备优先级管理机制，其中优先级控制模块被保存在 FTL 中。LE 设备有存储空间和优先级控制模块。

优先级控制模块包括以下两部分：

- **bond\_num**: 已保存绑定设备的数目
- **bond\_idx** 数组: 已保存绑定设备的索引数组。GAP 层可以根据绑定设备索引查找到其在 FTL 中的起始偏移。

优先级管理包括如下操作：

#### 1. 添加一个绑定设备

GAP LE API: 不对外提供，仅供内部使用。

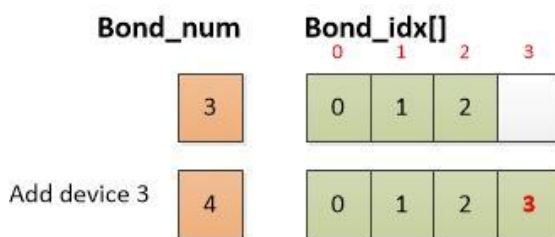


图 2-10 增加一个绑定设备

## 2. 移除一个绑定设备

GAP LE API: `le_bond_delete_by_idx()` 或 `le_bond_delete_by_bd()`

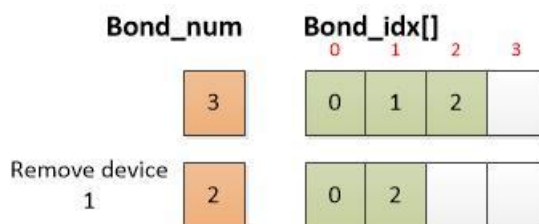


图 2-11 移除一个绑定设备

## 3. 清除所有绑定设备

GAP LE API: `le_bond_clear_all_keys()`

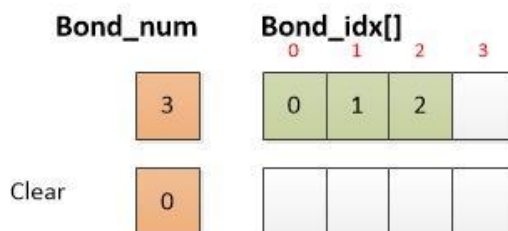


图 2-12 清除所有绑定设备

## 4. 将一个绑定设备设为最高优先级

GAP LE API: `le_set_high_priority_bond()`

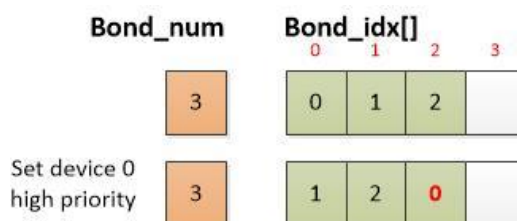


图 2-13 将一个绑定设备设为最高优先级

## 5. 获取最高优先级设备

最高优先级设备为 `bond_idx[bond_num - 1]`。

GAP LE API: `le_get_high_priority_bond()`

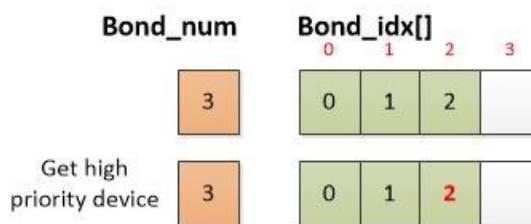


图 2-14 获取最高优先级设备

## 6. 获取最低优先级设备

最低优先级设备为 `bond_idx[0]`。

GAP LE API: `le_get_low_priority_bond()`

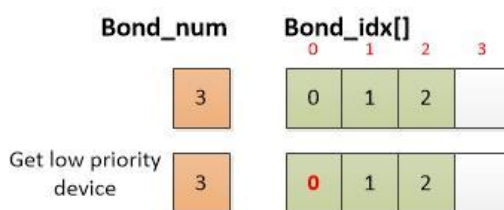


图 2-15 获取最低优先级设备

优先级管理示例如图 2-16 所示。

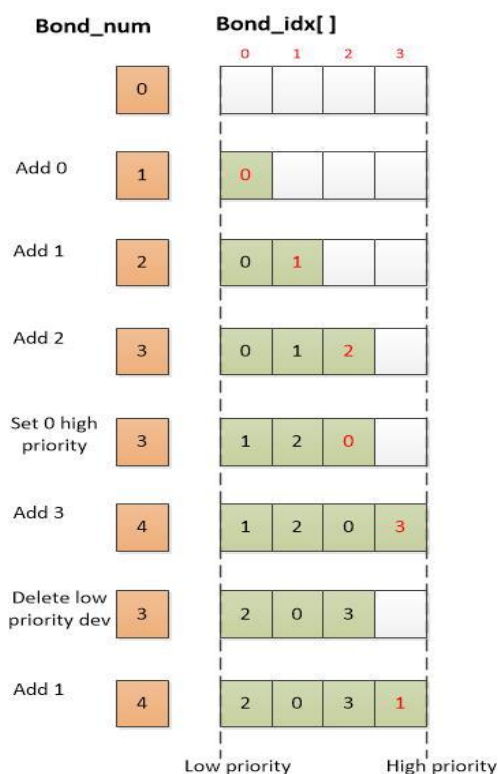


图 2-16 优先级管理示例

## 2.6.3.2 BLE 密钥存储

BLE 密钥信息存储在 LE key storage space，LE FTL 布局如图 2-17 所示。

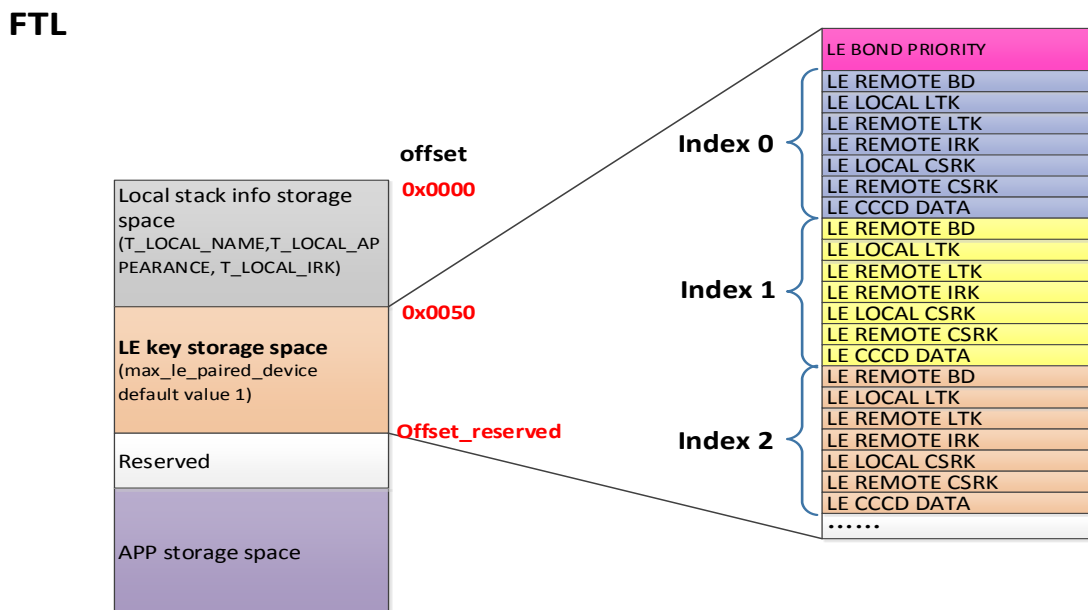


图 2-17 LE FTL 布局

LE key storage space 可以分为以下两部分：

1. **LE BOND PRIORITY:** LE 优先级控制模块，更多信息参见[绑定设备优先级管理](#)。
2. **Bonded device keys storage block:** 设备索引 0、索引 1 等等。
  - 1) LE REMOTE BD: 保存对端设备地址
  - 2) LE LOCAL LTK: 保存本地设备 Long Term Key (LTK)
  - 3) LE REMOTE LTK: 保存对端设备 LTK
  - 4) LE REMOTE IRK: 保存对端设备 IRK
  - 5) LE LOCAL CSRK: 保存本地设备 Connection Signature Resolving Key (CSRK)
  - 6) LE REMOTE CSRK: 保存对端设备 CSRK
  - 7) LE CCCD DATA: 保存 Client Characteristic Configuration declaration (CCCD)数据

### 2.6.3.2.1 配置

LE key storage space 的大小与以下两个参数有关：

1. LE 绑定设备数目的最大值
  - 1) 默认值为 1
2. CCCD 数目的最大值
  - 1) 默认值为 16



### 2.6.3.2.2 E Key Entry 结构体

GAP 层使用结构体 T\_LE\_KEY\_ENTRY 管理绑定设备。

```
#define LE_KEY_STORE_REMOTE_BD_BIT    0x01
#define LE_KEY_STORE_LOCAL_LTK_BIT    0x02
#define LE_KEY_STORE_REMOTE_LTK_BIT   0x04
#define LE_KEY_STORE_REMOTE_IRK_BIT   0x08
#define LE_KEY_STORE_LOCAL_CSRK_BIT   0x10
#define LE_KEY_STORE_REMOTE_CSRK_BIT  0x20
#define LE_KEY_STORE_CCCD_DATA_BIT    0x40
#define LE_KEY_STORE_LOCAL_IRK_BIT    0x80

/** @brief LE key entry */
typedef struct
{
    bool is_used;
    uint8_t idx;
    uint16_t flags;
    uint8_t local_bd_type;
    uint8_t app_data;
    uint8_t reserved[2];
    T_LE_REMOTE_BD remote_bd;
    T_LE_REMOTE_BD resolved_remote_bd;
} T_LE_KEY_ENTRY;
```

参数描述:

- **is\_used** - 是否使用
- **idx** - 设备索引，GAP 使用 idx 查找绑定设备信息在 FTL 的存储位置
- **flags** - LE Key Storage Bits，表示密钥是否存在的位域
- **local\_bd\_type** - 配对过程中使用的 local address type，T\_GAP\_LOCAL\_ADDR\_TYPE
- **remote\_bd** - 对端设备地址
- **resolved\_remote\_bd** - 对端设备的 identity address

### 2.6.3.2.3 LE 密钥管理

当本地设备与对端设备配对或本地设备与绑定设备加密时，GAP 层将发送 GAP\_MSG\_LE\_AUTHEN\_STATE\_CHANGE 消息向 APP 通知 authentication 状态的变化。

```
void app_handle_authen_state_evt(uint8_t conn_id, uint8_t new_state, uint16_t
                                cause)
{
    APP_PRINT_INFO2("app_handle_authen_state_evt:conn_id %d, cause 0x%x", conn_id, cause);
    switch (new_state)
    {
        case GAP_AUTHEN_STATE_STARTED:
        {
```

```

        APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_STARTED");
    }
    break;
case GAP_AUTHEN_STATE_COMPLETE:
    {
        if (cause == GAP_SUCCESS)
        {
            APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE
                             pair success");
        }
        else
        {
            APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE
                             pair failed");
        }
    }
    break;
.....
}
}

```

GAP\_MSG\_LE\_BOND\_MODIFY\_INFO 用于向 APP 通知绑定信息已变更。

```

typedef struct
{
    T_LE_BOND_MODIFY_TYPE type;
    P_LE_KEY_ENTRY          p_entry;
} T_LE_BOND_MODIFY_INFO;

T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
        case GAP_MSG_LE_BOND_MODIFY_INFO:
            APP_PRINT_INFO1("GAP_MSG_LE_BOND_MODIFY_INFO: type 0x%x",
                             p_data->p_le_bond_modify_info->type);

            break;
        .....
    }
}

```

绑定信息变更类型的定义如下所示：

```

typedef enum {
    LE_BOND_DELETE,

```

```
LE_BOND_ADD,
LE_BOND_CLEAR,
LE_BOND_FULL,
LE_BOND_KEY_MISSING,
} T_LE_BOND_MODIFY_TYPE;
```

#### 1. LE\_BOND\_DELETE

LE\_BOND\_DELETE 表示绑定信息已被删除，当满足以下条件时会发送该消息：

- 1) 调用 le\_bond\_delete\_by\_idx()
- 2) 调用 le\_bond\_delete\_by\_bd()
- 3) 链路加密失败
- 4) 密钥存储空间已满，因此最低优先级的绑定信息会被删除。

#### 2. LE\_BOND\_ADD

LE\_BOND\_ADD 表示新增绑定设备，仅在第一次与对端设备配对时发送该消息。

#### 3. LE\_BOND\_CLEAR

LE\_BOND\_CLEAR 表示所有绑定消息已被删除，在调用 le\_bond\_clear\_all\_keys()后会发送该消息。

#### 4. LE\_BOND\_FULL

LE\_BOND\_FULL 表示密钥存储空间已满，当参数 GAP\_PARAM\_BOND\_KEY\_MANAGER 设为 true 时发送该消息。此时，GAP 层不会自动删除绑定信息。设为 false 时不会发送该消息，GAP 层将删除最低优先级的绑定信息，存储当前的绑定信息，然后发送 LE\_BOND\_DELETE 消息。

#### 5. LE\_BOND\_KEY\_MISSING

LE\_BOND\_KEY\_MISSING 表示链路加密失败且密钥失效，当参数 GAP\_PARAM\_BOND\_KEY\_MANAGER 设为 true 时发送该消息。此时，GAP 层不会自动删除绑定信息。设为 false 时不会发送该消息，GAP 层将删除绑定信息，发送 LE\_BOND\_DELETE 消息。

### 2.6.3.2.4 GAP 层内部 BLE 设备优先级管理

#### 1. 与新设备配对

- 1) 密钥存储空间未满
  - (1) GAP 层将在优先级控制模块添加绑定设备，发送 LE\_BOND\_ADD 消息给 APP。该新增设备具有最高优先级。
- 2) 密钥存储空间已满
  - (1) 当 GAP\_PARAM\_BOND\_KEY\_MANAGER 为 true 时，GAP 层发送消息 LE\_BOND\_FULL 给 APP。
  - (2) 当 GAP\_PARAM\_BOND\_KEY\_MANAGER 为 false 时，GAP 层将从优先级控制模块移除最低优先级的绑定设备，发送 LE\_BOND\_DELETE 消息。GAP 层将在优先级控制模块添加绑定设备，发送 LE\_BOND\_ADD 消息给 APP。该新增设备具有最高优先级。

#### 2. 与绑定设备加密成功

GAP 层将该绑定设备设为最高优先级。

### 3. 与绑定设备加密失败

- 1) 当 GAP\_PARAM\_BOND\_KEY\_MANAGER 为 true 时，GAP 层将发送 LE\_BOND\_KEY\_MISSING 给 APP。
- 2) 当 GAP\_PARAM\_BOND\_KEY\_MANAGER 为 false 时，GAP 层将从优先级控制模块移除该绑定设备，发送 LE\_BOND\_DELETE 给 APP。

#### 2.6.3.2.5 APIs

```
/* gap_storage_le.h */
P_LE_KEY_ENTRY le_find_key_entry(uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE bd_type);
P_LE_KEY_ENTRY le_find_key_entry_by_idx(uint8_t idx);
uint8_t le_get_bond_dev_num(void);
P_LE_KEY_ENTRY le_get_low_priority_bond(void);
P_LE_KEY_ENTRY le_get_high_priority_bond(void);
bool le_set_high_priority_bond(uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE bd_type);
bool le_resolve_random_address(uint8_t *unresolved_addr, uint8_t *resolved_addr,
                               T_GAP_IDENT_ADDR_TYPE *resolved_addr_type);
bool le_get_cccd_data(T_LE_KEY_ENTRY *p_entry, T_LE_CCCD *p_data);
/* gap_bond_le.h */
void le_bond_clear_all_keys(void);
T_GAP_CAUSE le_bond_delete_by_idx(uint8_t idx);
T_GAP_CAUSE le_bond_delete_by_bd(uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE bd_type);
```

## 3 GATT Profile

在 SDK 中提供基于 GATT specification 的 GATT Profile APIs。基于 GATT 的 Profile 的实现分为两种类型：Profile-Server 和 Profile-Client。

Profile-Server 是基于 GATT 的 Profile 在 server 端的实现的公共接口，更多信息参见 [BLE Profile Server](#)。

Profile-Client 是基于 GATT 的 Profile 在 client 端的实现的公共接口，更多信息参见 [BLE Profile Client](#)。

GATT Profile Layer 已经在 BT Lib 中实现，在 SDK 中通过头文件提供接口给 APP 使用。GATT Profile 头文件路径为 component\common\bluetooth\realtek\sdk\board\amebad\inc\bluetooth\profile。

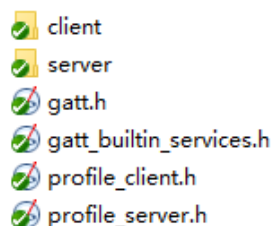


图 3-1 GATT Profile 头文件

### 3.1 BLE Profile Server

#### 3.1.1 概述

Server 是可以接收来自 client 的 command 和 request 且发送 response、indication 和 notification 给 client 的设备。GATT Profile 定义作为 GATT server 和 GATT client 的 BLE 设备的交互方式。Profile 可能包含一个或多个 GATT services，service 是一组 characteristics 的集合，因而 GATT server 展示的是其 characteristics。

用户可以使用 Profile Server 导出的 APIs 实现 specific service。Profile server 层级如图 3-2 所示。Profile 包括 profile server layer 和 specific service。位于 protocol stack 之上的 profile server layer 封装供 specific service 访问 protocol stack 的接口，因此针对 specific service 的开发不涉及 protocol stack 的细节，使开发变得更简单和清晰。基于 profile server layer 的 specific service 是由 application layer 实现的，specific service 由 attribute value 组成并提供接口供 APP 发送数据。

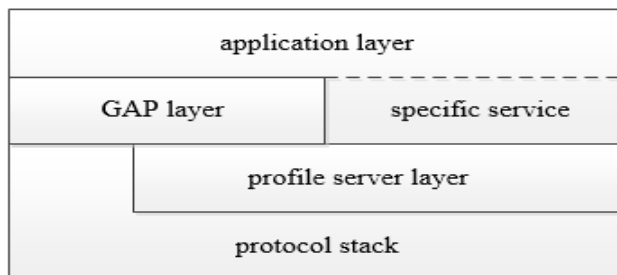


图 3-2 Profile Server 层级

### 3.1.2 支持的 Profile 和 Service

支持的 Profile 列表如表 3-1 所示。

表 3-1 支持的 Profile 列表

Abbr.	Definition	GATT server	GATT client
GAP	Generic Access Profile	Server role shall support GAS(M)	client role has no claim
PXP	Proximity Profile	Proximity Reporter role shall support LLS(M), IAS(O), TPS(O)	Proximity Monitor role has no claim
ScPP	Scan Parameters Profile	Scan Server role shall support ScPS(M)	Scan Client role has no claim
HTP	Health Thermometer Profile	Thermometer role shall support HTS(M), DIS(M)	Collector role has no claim
HRP	Heart Rate Profile	Heart Rate Sensor role shall support HRS(M), DIS(M)	Collector role has no claim
LNP	Location and Navigation Profile	LN Sensor role shall support LNS(M), DIS(O), BAS(O)	Collector role has no claim
WSP	Weight Scale Profile	Weight Scale role shall support WSS(M), DIS(M), BAS(O)	Weight Scale role shall support WSS(M), DIS(M), BAS(O)
GLP	Glucose Profile	Glucose Sensor role shall support GLS(M), DIS(M)	Collector role has no claim
FMP	Fine Me Profile	Find Me Target role shall support IAS(M)	Find Me Locator role has no claim
HOGP	HID over GATT Profile	HID Device shall support HIDS(M), BAS(M), DIS(O), ScPS(O)	Boot Host has no claim
RSCP	Running Speed and Cadence Profile	RSC Sensor role shall support RSCS(M), DIS(M)	Collector role has no claim
CSCP	Cycling Speed and Cadence Profile	CSC Sensor role shall support CSCS(M), DIS(M)	Collector role has no claim
IPSP	Internet Protocol Support Profile	Node role shall support IPSS(M)	Router role has no claim
<b>NOTE:</b> M: mandatory O: optional			

支持的 service 列表如表 3-2 所示。

表 3-2 支持的 service 列表

Abbr.	Definition	Files
GATTS	Generic Attribute Service	<code>gatt_builtin_services.h</code>
GAS	Generic Access Service	<code>gatt_builtin_services.h</code>
BAS	Battery Service	<code>bas.c</code> , <code>bas.h</code> <code>bas_config.h</code>
DIS	Device Information Service	<code>dis.c</code> , <code>dis.h</code> <code>dis_config.h</code>
HIDS	Human Interface Device Service	<code>hids.c</code> , <code>hids.h</code> <code>hids_kb.c</code> , <code>hids_kb.h</code>

### 3.1.3 Profile Server 交互

Profile server layer 处理与 protocol stack layer 的交互, 并提供接口用于设计 specific service。Profile Server 交互包括向 server 添加 service、读取 characteristic value、写入 characteristic value、characteristic value notification 和 characteristic value indication。

#### 3.1.3.1 添加 Service

Protocol stack 维护通过 profile server layer 添加的所有 services 的信息。首先, 需要通过 `server_init()` 接口初始化 service attribute table 的总数目。

```
void app_le_profile_init(void)
{
    server_init(2);
    simp_srv_id = simp_ble_service_add_service(app_profile_callback);
    bas_srv_id = bas_add_service(app_profile_callback);
    server_register_app_cb(app_profile_callback);
    ...
}
```

Profile server layer 提供 `server_add_service()` 接口用于向 profile server layer 添加 service。

```
T_SERVER_ID simp_ble_service_add_service(void *p_func)
{
    if (false == server_add_service(&simp_service_id,
                                    (uint8_t *)simple_ble_service_tbl,
                                    sizeof(simple_ble_service_tbl),
                                    simp_ble_service_cbs))
    {
        APP_PRINT_ERROR0("simp_ble_service_add_service: fail");
        simp_service_id = 0xff;
    }
}
```

```

return simp_service_id;
}
pfn_simp_ble_service_cb = (P_FUN_SERVER_GENERAL_CB)p_func;
return simp_service_id;
}

```

图 3-3 表示一个 server 包含多个 service tables，向该 server 添加 service 的情况。

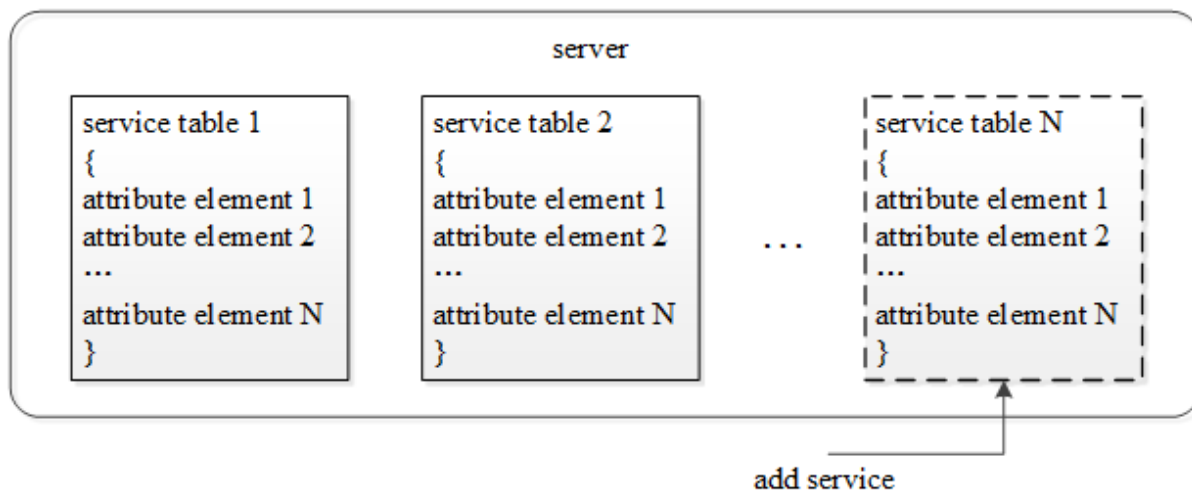


图 3-3 向 Server 添加 Services

当 service 添加到 profile server layer 后，GAP 初始化流程会注册所有 services，一旦完成注册流程，GAP 层会发送 PROFILE\_EVT\_SRV\_REG\_COMPLETE 消息。

注册 service 的流程如图 3-4 所示。

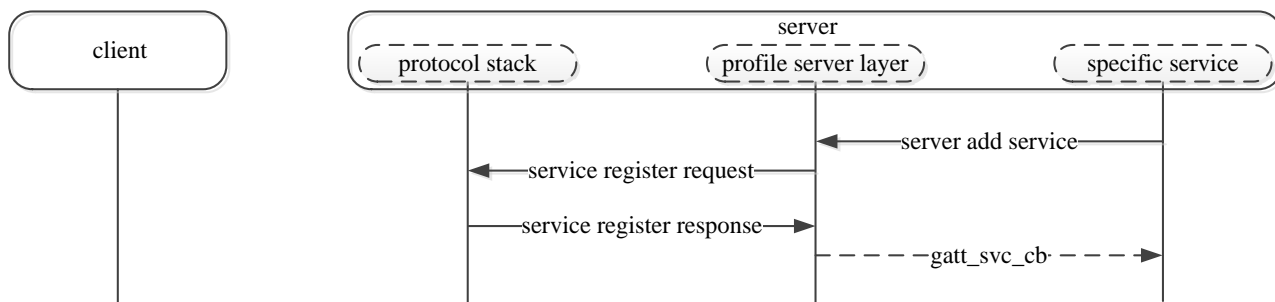


图 3-4 注册 Service 的流程

在 GAP 初始化过程中，通过向 protocol stack 发送 service 注册请求以启动 service 的注册流程，然后注册所有已添加 services。若 server 通用回调函数不为 NULL，一旦最后一个服务被成功注册，profile server layer 将通过注册的回调函数 app\_profile\_callback() 向 APP 发送 PROFILE\_EVT\_SRV\_REG\_COMPLETE 消息。

```

T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{

```



```

T_APP_RESULT app_result = APP_RESULT_SUCCESS;
if (service_id == SERVICE_PROFILE_GENERAL_ID)
{
    T_SERVER_APP_CB_DATA *p_param = (T_SERVER_APP_CB_DATA *)p_data;
    switch (p_param->eventId)
    {
        case PROFILE_EVT_SRV_REG_COMPLETE:// srv register result event.
            APP_PRINT_INFO1("PROFILE_EVT_SRV_REG_COMPLETE: result %d",
                           p_param->event_data.service_reg_result);
            break;
        ...
    }
}

```

### 3.1.3.2 Service 的回调函数

#### 3.1.3.2.1 Server 通用回调函数

Server 通用回调函数用于向 APP 发送事件，其中包含 service 注册完成事件和通过 characteristic value notification 或 indication 发送数据完成事件。通过 server\_register\_app\_cb() 初始化该回调函数。在 profile\_server.h 中定义 server 通用回调函数。

#### 3.1.3.2.2 Specific Service 回调函数

为访问由 specific service 提供的 attribute value，需要在 specific service 中实现回调函数，该回调函数用于处理来自 client 的 read/write attribute value 和更新 CCCD 数值的流程。通过 server\_add\_service() 初始化该回调函数。在 profile\_server.h 中定义回调函数的结构体。

```

/* service related callback functions struct */
typedef struct {
    P_FUN_GATT_READ_ATTR_CB read_attr_cb;           // Read callback function pointer
    P_FUN_GATT_WRITE_ATTR_CB write_attr_cb;         // Write callback function pointer
    P_FUN_GATT_CCCD_UPDATE_CB cccd_update_cb;       // update cccd callback function pointer
} T_FUN_GATT_SERVICE_CBS;

```

**read\_attr\_cb:** 读 attribute 回调函数，当 client 发送 attribute read request 时，该回调函数用于获取 specific service 提供的 attribute value。

**write\_attr\_cb:** 写 attribute 回调函数，当 client 发送 attribute write request 时，该回调函数用于写入 specific service 提供的 attribute value。

**cccd\_update\_cb:** 更新 CCCD 数值回调函数，用于通知 specific service，service 中相应 CCCD 数值已被 client 写入。

```

const T_FUN_GATT_SERVICE_CBS simp_ble_service_cbs =
{
    simp_ble_service_attr_read_cb, // Read callback function pointer
    simp_ble_service_attr_write_cb, // Write callback function pointer
}

```

```
simp_ble_service_cccd_update_cb // CCCD update callback function pointer
};
```

### 3.1.3.2.3 Write Indication Post Procedure 回调函数

Write indication post procedure 回调函数用于在处理 client 的 write request 之后执行一些后续流程。该回调函数是在 write attribute 回调函数中被初始化的。若无后续流程需要执行，write attribute 回调函数中的 p\_write\_post\_proc 指针必须为 NULL。在 profile\_server.h 中定义 Write indication post procedure 回调函数。

### 3.1.3.3 Characteristic Value Read

该流程用于从 server 读取 characteristic value。有四个子流程可以用于读取 characteristic value, 包括 read characteristic value、read using characteristic UUID、read long characteristic values 和 read multiple characteristic values。一个可读的 attribute 必须配置可读 permission。根据不同的 attribute flag，可以从 service 或 APP 读取 attribute value。

#### 3.1.3.3.1 由 Attribute Element 提供 Attribute Value

flag 为 ATTRIB\_FLAG\_VALUE\_INCL 的 attribute 会涉及该流程。

```
{
    ATTRIB_FLAG_VALUE_INCL,                /* flags */
    {                                       /* type_value */
        LO_WORD(0x2A04),
        HI_WORD(0x2A04),
        100,
        200,
        0,
        LO_WORD(2000),
        HI_WORD(2000)
    },
    5,                                       /* bValueLen */
    NULL,
    GATT_PERM_READ                          /* permissions */
},
```

该流程中各层之间的交互如图 3-5 所示, protocol stack layer 将从 attribute element 中读取 attribute value, 并直接在 read response 中响应该 attribute value。

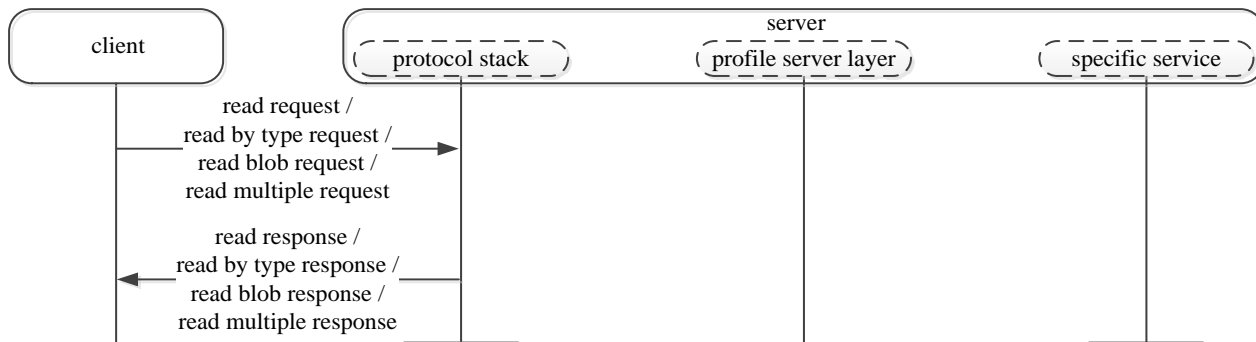


图 3-5 读 Characteristic Value – 由 Attribute Element 提供 Attribute Value

### 3.1.3.3.2 由 APP 提供 Attribute Value 且结果未挂起

flag 为 ATTRIB\_FLAG\_VALUE\_APPL 的 attribute 会涉及该流程。

```

{
    ATTRIB_FLAG_VALUE_APPL,
    {
        LO_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ),
        HI_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ)
    },
    0,
    NULL,
    GATT_PERM_READ
},

```

该流程中各层之间的交互如图 3-6 所示。当本地设备收到 read request 时，protocol stack 将发送 read indication 给 profile server layer，profile server layer 将调用 read attribute 回调函数获取 specific service 中的 attribute value。然后，profile server layer 通过 read confirmation 将数据传递给 protocol stack。

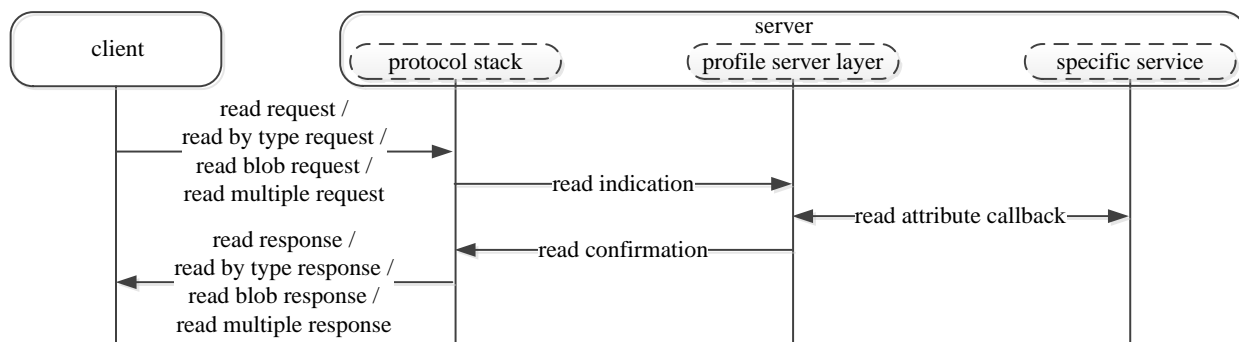


图 3-6 读 Characteristic Value – 由 APP 提供 Attribute Value 且结果未挂起

示例代码如下所示，app\_profile\_callback()的返回结果必须为 APP\_RESULT\_SUCCESS。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ...
    else if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE:
            {
                if (p_simp_cb_data->msg_data.read_value_index == SIMP_READ_V1)
                {
                    uint8_t value[2] = {0x01, 0x02};
                    APP_PRINT_INFO0("SIMP_READ_V1");
                    simp_ble_service_set_parameter(
                        SIMPLE_BLE_SERVICE_PARAM_V1_READ_CHAR_VAL, 2, &value);
                }
            }
            break;
        }
        ...
        return app_result;
    }
}
```

### 3.1.3.3.3 由 APP 提供 Attribute Value 且结果挂起

flag 为 ATTRIB\_FLAG\_VALUE\_APPL 的 attribute 会涉及该流程。

由于 APP 提供的 attribute value 不能立即被读取，specific service 需要调用 server\_attr\_read\_confirm() 传递 attribute value。该流程中各层之间的交互如图 3-7 所示。

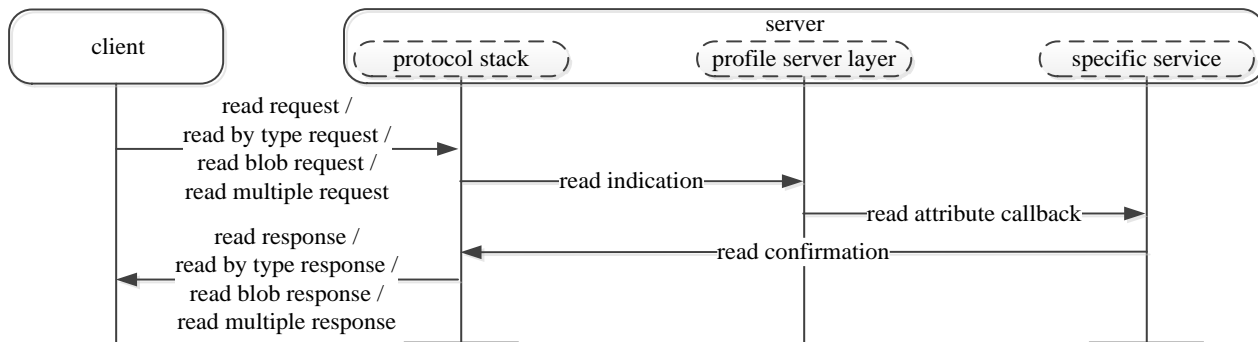


图 3-7 读 Characteristic Value - 由 APP 提供 Attribute Value 且结果挂起

示例代码如下所示，app\_profile\_callback()的返回结果必须为 APP\_RESULT\_PENDING。

```

T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_PENDING;
    ...
    else if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE:
            {
                if (p_simp_cb_data->msg_data.read_value_index == SIMP_READ_V1)
                {
                    uint8_t value[2] = {0x01, 0x02};
                    APP_PRINT_INFO("SIMP_READ_V1");
                    simp_ble_service_set_parameter(
                        SIMPLE_BLE_SERVICE_PARAM_V1_READ_CHAR_VAL, 2, &value);
                }
            }
            break;
        }
    }
    ...
    return app_result;
}
    
```

### 3.1.3.4 Characteristic Value Write

该流程用于向 server 写入 characteristic value。有四个子流程可以用于写入 characteristic value，包括 write without response、signed write without response、write characteristic value 和 write long characteristic values。

### 3.1.3.4.1 Write Characteristic Value

#### 1. 由 Attribute Element 提供 Attribute Value

flag 为 ATTRIB\_FLAG\_VOID 的 attribute 会涉及该流程。

```
uint8_t cha_val_v8_011[1] = {0x08};
const T_ATTRIB_APPL gatt_dfindme_profile[] = {
    .....
    /* handle = 0x000e Characteristic value -- Value V8 */
    {
        ATTRIB_FLAG_VOID,                /* flags */
        {                                /* type_value */
            LO_WORD(0xB008),
            HI_WORD(0xB008),
        },
        1,                               /* bValueLen */
        (void *)cha_val_v8_011,
        GATT_PERM_READ | GATT_PERM_WRITE /* permissions */
    },
    .....
}
```

该流程中各层之间的交互如图 3-8 所示，write request 用于请求 server 写 attribute value，且在写操作结束之后直接发送 write response。

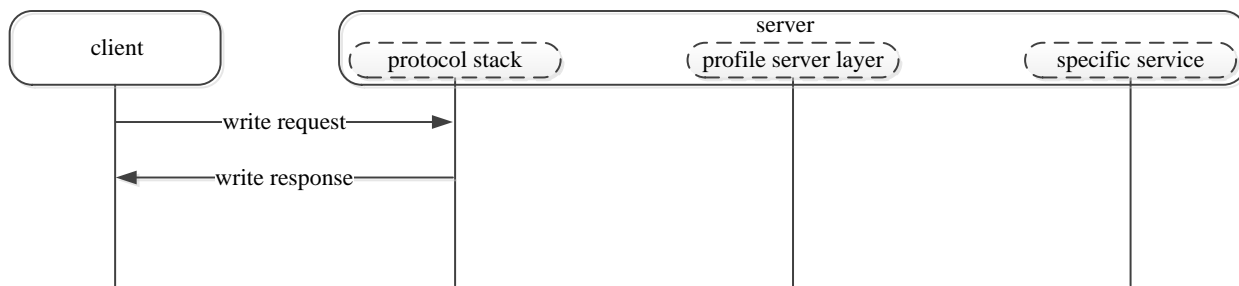


图 3-8 Write Characteristic Value – 由 Attribute Element 提供 Attribute Value

#### 2. 由 APP 提供 Attribute Value 且结果未挂起

flag 为 ATTRIB\_FLAG\_VALUE\_APPL 的 attribute 会涉及该流程。

该流程中各层之间的交互如图 3-9 所示，当本地设备收到 write request，protocol stack 将发送 write request indication 给 profile server layer，profile server layer 将调用 write attribute callback 写入 specific service 中的 attribute value。Profile server layer 将通过 write request confirmation 返回写入结果。

若在 profile server layer 返回 write confirmation 之后 server 需要执行后续流程，回调函数指针 write\_ind\_post\_proc()不为 NULL 时，将调用该回调函数。

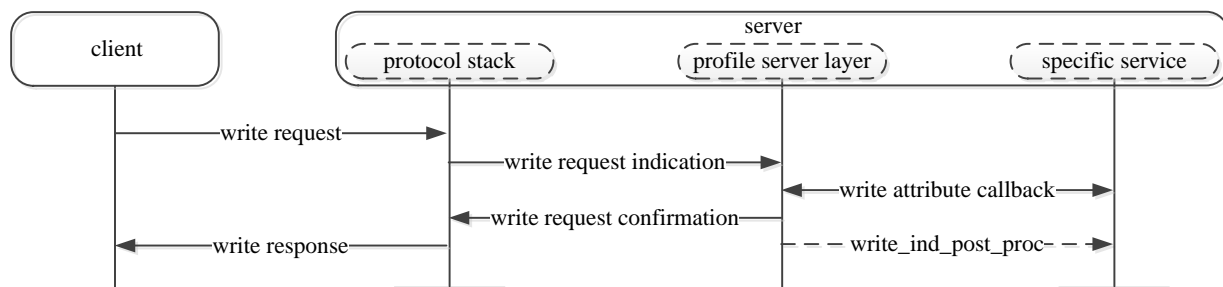


图 3-9 Write Characteristic Value - 由 APP 提供 Attribute Value 且结果未挂起

由 server\_add\_service()注册的 srv\_cbs 回调函数通知 APP， write\_type 为 WRITE\_REQUEST。示例代码如下所示，app\_profile\_callback()的返回结果必须为 APP\_RESULT\_SUCCESS。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ...
    else if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE:
            {
                switch (p_simp_cb_data->msg_data.write.opcode)
                {
                    case SIMP_WRITE_V2:
                    {
                        APP_PRINT_INFO2("SIMP_WRITE_V2: write type %d, len %d",
                                      p_simp_cb_data->msg_data.write.write_type,
                                      p_simp_cb_data->msg_data.write.len);
                    }
                }
            }
            ...
        }
        return app_result;
    }
}
```

### 3. 由 APP 提供 Attribute Value 且结果挂起

flag 为 ATTRIB\_FLAG\_VALUE\_APPL 的 attribute 会涉及该流程。

若写 attribute value 流程不能立即结束，specific service 将会调用 server\_attr\_write\_confirm()。该流程中各层之间的交互如图 3-10 所示。Write indication post procedure 为可选流程。

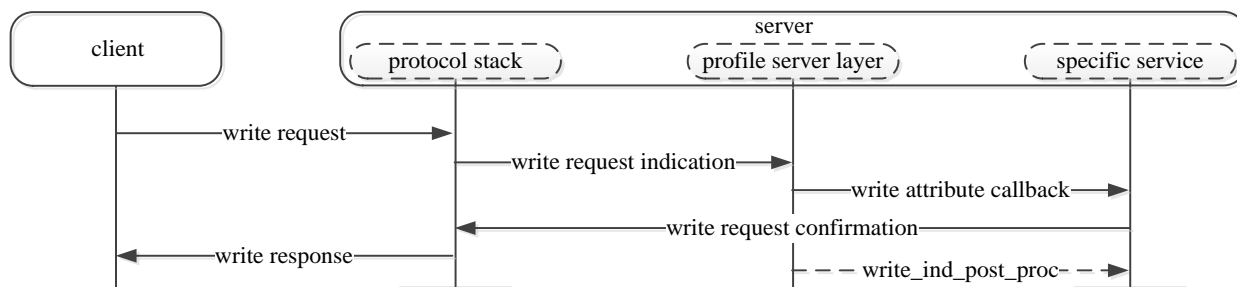


图 3-10 Write Characteristic Value – 由 APP 提供 Attribute Value 且结果挂起

由 server\_add\_service()注册的 srv\_cbs 回调函数通知 APP， write\_type 为 WRITE\_REQUEST。示例代码如下所示，app\_profile\_callback()的返回结果必须为 APP\_RESULT\_PENDING。

```

T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_PENDING;
    ...
    else if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE:
            {
                switch (p_simp_cb_data->msg_data.write.opcode)
                {
                    case SIMP_WRITE_V2:
                    {
                        APP_PRINT_INFO2("SIMP_WRITE_V2: write type %d, len %d",
                                        p_simp_cb_data->msg_data.write.write_type,
                                        p_simp_cb_data->msg_data.write.len);
                    }
                }
            }
            ...
        }
        return app_result;
    }
}

```

#### 4. 写 CCCD 值

若本地设备收到 client 的 write request 以写 CCCD, protocol stack 将更新 CCCD 信息, profile server layer 通过更新 CCCD 回调函数向 APP 通知 CCCD 信息已更新。该流程中各层之间的交互如图 3-11 所示。



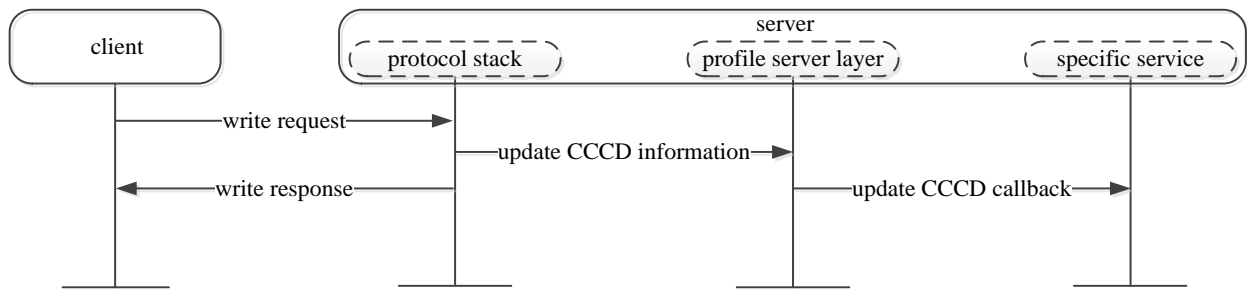


图 3-11 Write Characteristic Value – 写 CCCD 值

```

void simp_ble_service_cccd_update_cb(uint8_t conn_id, T_SERVER_ID service_id, uint16_t index,
                                     uint16_t cccbits)
{
    TSIMP_CALLBACK_DATA callback_data;
    bool is_handled = false;
    callback_data.conn_id = conn_id;
    callback_data.msg_type = SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION;
    APP_PRINT_INFO2("simp_ble_service_cccd_update_cb: index = %d, cccbits 0x%x", index, cccbits);
    switch (index)
    {
        case SIMPLE_BLE_SERVICE_CHAR_NOTIFY_CCCD_INDEX:
        {
            if (cccbits & GATT_CLIENT_CHAR_CONFIG_NOTIFY)
            {
                // Enable Notification
                callback_data.msg_data.notification_indification_index = SIMP_NOTIFY_INDICATE_V3_ENABLE;
            }
            else
            {
                // Disable Notification
                callback_data.msg_data.notification_indification_index = SIMP_NOTIFY_INDICATE_V3_DISABLE;
            }
            is_handled = true;
        }
        break;
        case SIMPLE_BLE_SERVICE_CHAR_INDICATE_CCCD_INDEX:
        {
            if (cccbits & GATT_CLIENT_CHAR_CONFIG_INDICATE)
            {
                // Enable Indication
                callback_data.msg_data.notification_indification_index = SIMP_NOTIFY_INDICATE_V4_ENABLE;
            }
        }
    }
}

```

```

        else
        {
            // Disable Indication
            callback_data.msg_data.notification_indification_index = SIMP_NOTIFY_INDICATE_V4_DISABLE;
        }
        is_handled = true;
    }
    break;
default:
    break;
}
/* Notify Application. */
if (pfn_simp_ble_service_cb && (is_handled == true))
{
    pfn_simp_ble_service_cb(service_id, (void *)&callback_data);
}
}

```

由 server\_add\_service() 注册的 srv\_cbs 回调函数通知 APP，msg\_type 为 SERVICE\_CALLBACK\_TYPE\_INDIFICATION\_NOTIFICATION。

```

T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ...
    else if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            switch (p_simp_cb_data->msg_type)
            {
            case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:
                {
                    switch (p_simp_cb_data->msg_data.notification_indification_index)
                    {
                        case SIMP_NOTIFY_INDICATE_V3_ENABLE:
                            {
                                APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V3_ENABLE");
                            }
                            break;
                        case SIMP_NOTIFY_INDICATE_V3_DISABLE:
                            {
                                APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V3_DISABLE");
                            }
                    }
                }
            }
        }
    }
}

```

```

        break;
    case SIMP_NOTIFY_INDICATE_V4_ENABLE:
    {
        APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V4_ENABLE");
    }
    break;
    case SIMP_NOTIFY_INDICATE_V4_DISABLE:
    {
        APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V4_DISABLE");
    }
    break;
    default:
        break;
    }
}
break;
}
...
return app_result;
}

```

### 3.1.3.4.2 Write without Response

Write without Response 和 write characteristic value 流程的区别在于 server 不会发送写入结果给 client。

#### 1. 由 APP 提供 Attribute Value

flag 为 ATTRIB\_FLAG\_VALUE\_APPL 的 attribute 会涉及该流程。

该流程中各层之间的交互如图 3-12 所示。当本地设备收到 write command，由 server\_add\_service()注册的回调函数 write\_attr\_cb()将会被调用。

由 server\_add\_service() 注册的 srv\_cbs 回调函数将会通知 APP， write\_type 为 WRITE\_WITHOUT\_RESPONSE。

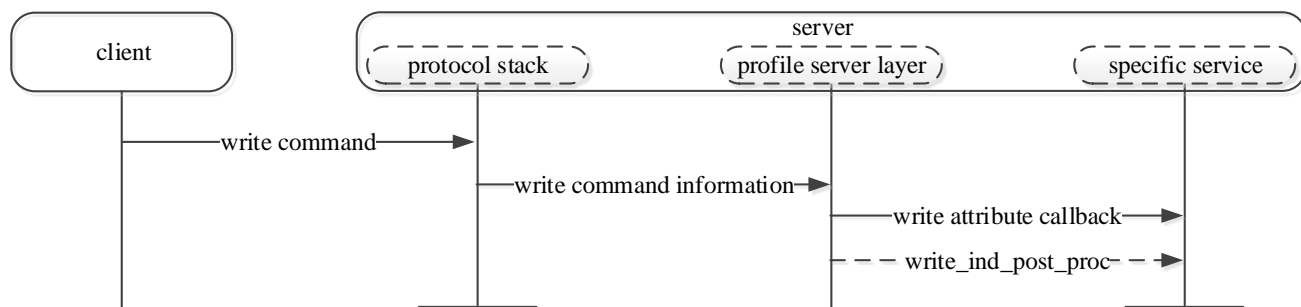


图 3-12 Write without Response - 由 APP 提供 Attribute Value

### 3.1.3.4.3 Write Long Characteristic Values

#### 1. Prepare Write

若 characteristic value 的长度大于 write request 支持的 characteristic value 的最大长度(ATT\_MTU - 3), client 将使用 prepare write request。需要被写入的值将先存储在 profile server layer, 然后 profile server layer 将处理 prepare write request indication, 并返回 prepare write confirmation。该流程中各层之间的交互如图 3-13 所示。

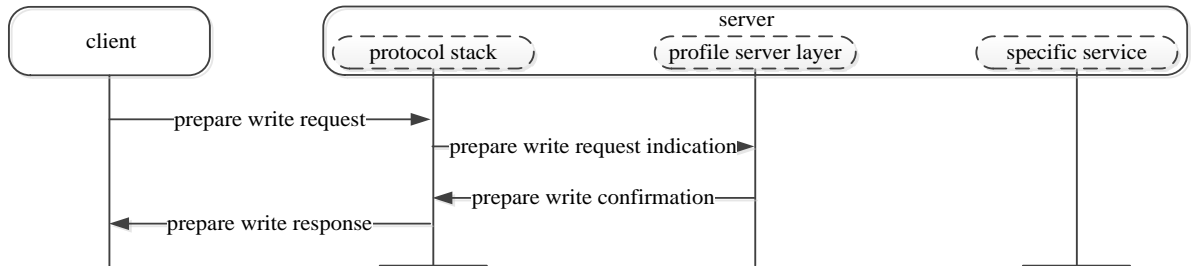


图 3-13 Write Long Characteristic Value - Prepare Write 流程

#### 2. 结果未挂起的 Execute Write

在发送 prepare write request 之后, execute write request 用于完成写入 attribute value 的流程。由 server\_add\_service()注册的 srv\_cbs 回调函数通知 APP, write\_type 为 WRITE\_LONG。Write indication post procedure 为可选流程。该流程中各层之间的交互如图 3-14 所示。

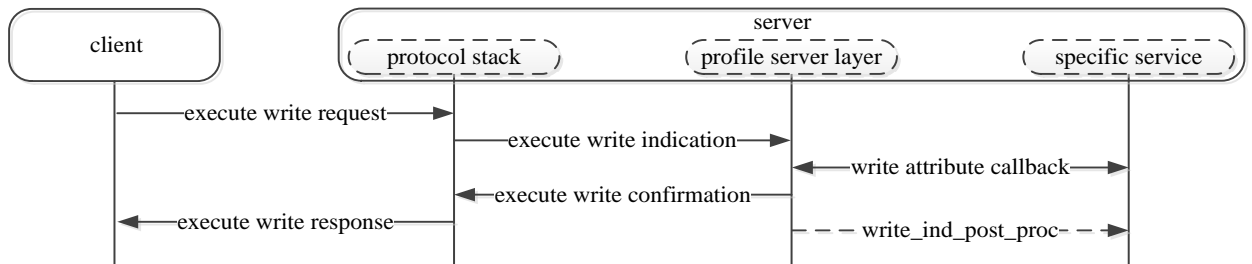


图 3-14 Write Long Characteristic Values- 结果未挂起的 Execute Write

#### 3. 结果挂起的 Execute Write

若写入操作不能立即完成, specific service 将调用 server\_exec\_write\_confirm(). Write indication post procedure 为可选流程。该流程中各层之间的交互如图 3-15 所示。

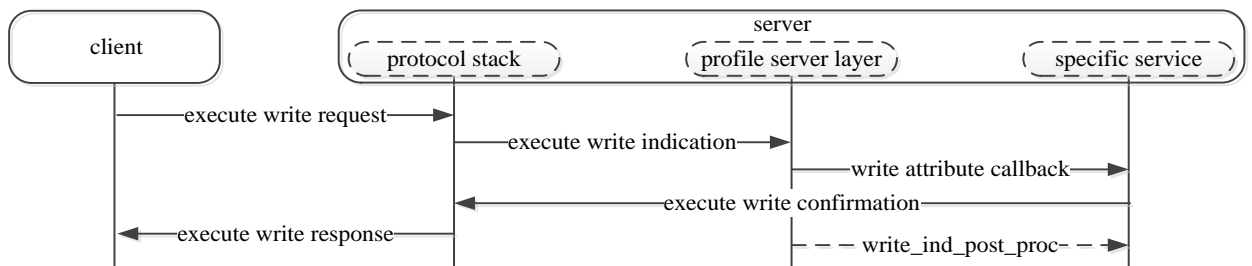


图 3-15 Write Long Characteristic Values- 结果挂起的 Execute Write

### 3.1.3.5 Characteristic Value Notification

Server 用该流程通知 client 一个 characteristic value。Server 主动调用 server\_send\_data()发送数据，在发送流程完成之后，会通过 server 通用回调函数通知 APP。该流程中各层之间的交互如图 3-16 所示。

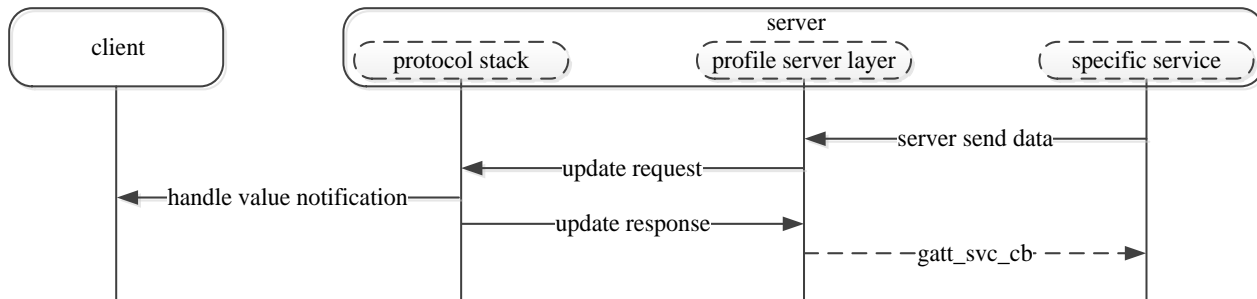


图 3-16 Characteristic Value Notification

```

bool simp_ble_service_send_v3_notify(uint8_t conn_id, T_SERVER_ID service_id, void *p_value,
                                     uint16_t length)
{
    APP_PRINT_INFO0("simp_ble_service_send_v3_notify");
    // send notification to client
    return server_send_data(conn_id, service_id, SIMPLE_BLE_SERVICE_CHAR_V3_NOTIFY_INDEX, p_value,
                           length, GATT_PDU_TYPE_ANY);
}
    
```

### 3.1.3.6 Characteristic Value Indication

Server 用该流程给 client 指示一个 characteristic value。一旦收到 indication，client 必须用 confirmation 响应。在 server 收到 handle value confirmation 之后，会通过 server 通用回调函数通知 APP。该流程中各层之间的交互如图 3-17 所示。

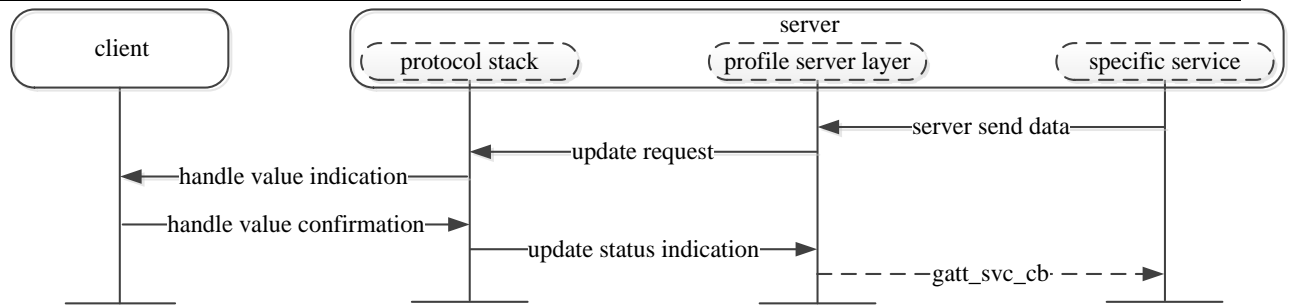


图 3-17 Characteristic Value Indication

```

bool simp_ble_service_send_v4_indicate(uint8_t conn_id, T_SERVER_ID service_id, void *p_value,
                                        uint16_t length)
{
    APP_PRINT_INFO0("simp_ble_service_send_v4_indicate");
    // send indication to client
    return server_send_data(conn_id, service_id, SIMPLE_BLE_SERVICE_CHAR_V4_INDICATE_INDEX,
                           p_value, length, GATT_PDU_TYPE_ANY);
}
    
```

在收到 handle value confirmation 后，将调用 app\_profile\_callback()。

```

T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    if (service_id == SERVICE_PROFILE_GENERAL_ID)
    {
        T_SERVER_APP_CB_DATA *p_param = (T_SERVER_APP_CB_DATA *)p_data;
        switch (p_param->eventId)
        {
            ...
            case PROFILE_EVT_SEND_DATA_COMPLETE:
                APP_PRINT_INFO5("PROFILE_EVT_SEND_DATA_COMPLETE: conn_id %d, cause 0x%x,
                                service_id %d, attrib_idx 0x%x, credits %d",
                                p_param->event_data.send_data_result.conn_id,
                                p_param->event_data.send_data_result.cause,
                                p_param->event_data.send_data_result.service_id,
                                p_param->event_data.send_data_result.attrib_idx,
                                p_param->event_data.send_data_result.credits);
                if (p_param->event_data.send_data_result.cause == GAP_SUCCESS)
                {
                    APP_PRINT_INFO0("PROFILE_EVT_SEND_DATA_COMPLETE success");
                }
                else
                {
                    ...
                }
            }
        }
    }
}
    
```

```

        APP_PRINT_ERROR0("PROFILE_EVT_SEND_DATA_COMPLETE failed");
    }
    break;
default:
    break;
}
}

```

### 3.1.4 Specific Service 的实现

为实现一个用例，一个 Profile 需要包含一个或多个 services，而 service 由 characteristics 组成，每个 characteristic 包括必选的 characteristic value 和可选的 characteristic descriptor。因而，service、characteristic 以及 characteristic 的组成成分（例如，值和 descriptors）包含 Profile 数据，并存储于 server 的 attributes 中。

以下为开发 specific service 的步骤：

1. 定义 Service 和 Profile Spec
2. 定义 Service Attribute Table
3. 定义 Service 与 APP 之间的接口
4. 定义 `xxx_add_service()`, `xxx_set_parameter()`, `xxx_notify()`, `xxx_indicate()` 等 API
5. 用 `T_FUN_GATT_SERVICE_CBS` 实现回调函数 `xxx_ble_service_cbs`

本节内容以 simple BLE service 为例，简单介绍如何实现 specific service。具体细节参见 `simple_ble_service.c` 和 `simple_ble_service.h` 中的源代码。

#### 3.1.4.1 定义 Service 和 Profile Spec

为实现 specific service，定义 Service 和 Profile Spec。

#### 3.1.4.2 定义 Service Attribute Table

由 attribute elements 组成的 service 是通过 service table 定义的，一个 service table 可以由多个 services 组成。

##### 3.1.4.2.1 Attribute Element

Attribute Element 是 service 的基本单元，其结构体定义在 `gatt.h` 中。

```

typedef struct {
    uint16_t    flags;                /**< Attribute flags @ref GATT_ATTRIBUTE_FLAG */
    uint8_t     type_value[2 + 14]; /**< 16 bit UUID + included value or 128 bit UUID */
    uint16_t    value_len;           /**< Length of value */
    void        *p_value_context;    /**< Pointer to value if @ref ATTRIB_FLAG_VALUE_INCL
    and @ref ATTRIB_FLAG_VALUE_APPL not set */
    uint32_t    permissions;         /**< Attribute permission @ref GATT_ATTRIBUTE_PERMISSIONS */
}

```

```
} T_ATTRIB_APPL;
```

## 1. Flags

Flags 的可选值和描述见表 3-3。

表 3-3 Flags 的可选值和描述

Option Values	Description
ATTRIB_FLAG_LE	Used only for primary service declaration attributes if GATT over BLE is supported
ATTRIB_FLAG_VOID	Attribute value is neither supplied by application nor included following 16bit UUID. Attribute value is pointed by p_value_context and value_len shall be set to the length of attribute value.
ATTRIB_FLAG_VALUE_INCL	Attribute value is included following 16 bit UUID
ATTRIB_FLAG_VALUE_APPL	Application has to supply attribute value
ATTRIB_FLAG_UUID_128BIT	Attribute uses 128 bit UUID
ATTRIB_FLAG_ASCII_Z	Attribute value is ASCII_Z string
ATTRIB_FLAG_CCCD_APPL	Application will be informed if CCCD value is changed
ATTRIB_FLAG_CCCD_NO_FILTER	Application will be informed about CCCD value when CCCD is written by client, no matter it is changed or not

注：

**ATTRIB\_FLAG\_LE** 仅适用于 type 为 primary service declaration 的 attribute，表示 primary service 允许通过 LE 链路访问。

**ATTRIB\_FLAG\_VOID**、**ATTRIB\_FLAG\_VALUE\_INCL** 和 **ATTRIB\_FLAG\_VALUE\_APPL** 三者之一必须在 attribute element 中使用。

**ATTRIB\_FLAG\_VALUE\_INCL** 表示 attribute value 被置于 type\_value 的最后 14 字节 (type\_value 的前 2 个字节用于保存 UUID)，且 value\_len 是放入最后 14 字节区域的字节数目。由于 type\_value 提供 attribute value，p\_value\_context 指针为 NULL。

**ATTRIB\_FLAG\_VALUE\_APPL** 表示由 APP 提供 attribute value。只要协议栈涉及对该 attribute value 的操作，协议栈将与 APP 进行交互以完成相应处理流程。由于 attribute value 是由 APP 提供的，type\_value 仅保存 UUID，value\_len 为 0，且 p\_value\_context 指针为 NULL。

**ATTRIB\_FLAG\_VOID** 表示 attribute value 既不放置于 type\_value 的最后 14 个字节，也不由 APP 提供。此时，type\_value 仅保存 UUID，p\_value\_context 指针指向 attribute value，value\_len 表示 attribute value 的长度。

表 3-4 展示 flags value 与 read attribute 流程使用的 actual value 之间的关联。

表 3-4 Flags Value 的选择模式

	APPL	APPL ASCII_Z	INCL	INCL ASCII_Z	VOID	VOID ASCII_Z
If set	value_len	Any(NULL)	Any(NULL)	Strlen(value)	Strlen(value)	Strlen(value)



	<b>type_value+2</b>	Any(NULL)	Any(NULL)	value	value	Any(NULL)	Any(NULL)
	<b>p_value_context</b>	Any(NULL)	Any(NULL)	Any(NULL)	Any(NULL)	value	value
Actual get by read attribute process	<b>Actual length</b>	Reply by application	Reply by application	Strlen(value)	Strlen(value)+1	Strlen(value)	Strlen(value)+1
	<b>Actual value</b>	Reply by application	Reply by application	value	Value + '\0'	Value	Value + '\0'

APPL: ATTRIB\_FLAG\_VALUE\_APPL

VOID: ATTRIB\_FLAG\_VOID

INCL: ATTRIB\_FLAG\_VALUE\_INCL

ASCII\_Z: ATTRIB\_FLAG\_ASCII\_Z

## 2. Permissions

Attribute 的 Permissions 指定 read 或 write 访问需要的安全级别，同样包括 notification 或 indication。Permissions 的值表示该 attribute 的 permission。Attribute permissions 是 access permissions、encryption permissions、authentication permissions 和 authorization permissions 的组合，其可用值如表 3-5 所示。

表 3-5 Permissions 的可用值

Types	Permissions
Read Permissions	GATT_PERM_READ
	GATT_PERM_READ_AUTHEN_REQ
	GATT_PERM_READ_AUTHEN_MITM_REQ
	GATT_PERM_READ_AUTHOR_REQ
	GATT_PERM_READ_ENCRYPTED_REQ
	GATT_PERM_READ_AUTHEN_SC_REQ
Write Permissions	GATT_PERM_WRITE
	GATT_PERM_WRITE_AUTHEN_REQ
	GATT_PERM_WRITE_AUTHEN_MITM_REQ
	GATT_PERM_WRITE_AUTHOR_REQ
	GATT_PERM_WRITE_ENCRYPTED_REQ
	GATT_PERM_WRITE_AUTHEN_SC_REQ
Notify/Indicate Permissions	GATT_PERM_NOTIF_IND
	GATT_PERM_NOTIF_IND_AUTHEN_REQ
	GATT_PERM_NOTIF_IND_AUTHEN_MITM_REQ
	GATT_PERM_NOTIF_IND_AUTHOR_REQ
	GATT_PERM_NOTIF_IND_ENCRYPTED_REQ
	GATT_PERM_NOTIF_IND_AUTHEN_SC_REQ

### 3.1.4.2.2 Service Table

Service 包含一组 attributes，其被称之为 service table。一个 service table 包含各种类型的 attributes，例如 service declaration、characteristic declaration、characteristic value 和 characteristic descriptor declaration。

Service table 的示例如表 3-6 所示，在 ble\_peripheral 示例工程的 simple\_ble\_service.c 中实现。

表 3-6 Service Table 示例

Flags	Attribute Type	Attribute Value	Permission
INCL   LE	<<primary service declaration>>	<<simple profile UUID – 0xA00A>>	read
INCL	<<characteristic declaration>>	Property(read)	read
APPL	<<characteristic value>>	UUID(0xB001), Value not defined here	read
VOID   ASCII_Z	<<Characteristic User Description>>	UUID(0x2901) Value defined in p_value_context	read
INCL	<<characteristic declaration>>	Property(write   write without response)	read
APPL	<<characteristic value>>	UUID(0xB002), Value not defined here	write
INCL	<<characteristic declaration>>	Property(notify)	read
APPL	<<characteristic value>>	UUID(0xB003), Value not defined here	none
CCCD_APPL	<<client characteristic configuration descriptor>>	Default CCCD value	read   write
INCL	<<characteristic declaration>>	Property(indicate)	read
APPL	<<characteristic value>>	UUID(0xB004), Value not defined here	none
CCCD_APPL	<<client characteristic configuration descriptor>>	Default CCCD value	read   write

注：

引号中的参数为 UUID 的值，其定义在蓝牙核心规范中或由用户自定义。

LE 为 ATTRIB\_FLAG\_LE 的缩写。

INCL 为 ATTRIB\_FLAG\_VALUE\_INCL 的缩写。

APPL 为 ATTRIB\_FLAG\_VALUE\_APPL 的缩写。

Service table 的示例代码如下所示：

```
const T_ATTRIB_APPL simple_ble_service_tbl[] =
{
    /* <<Primary Service>>, .. */
    {
        (ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_LE), /* flags */
        { /* type_value */
```

```

        LO_WORD(GATT_UUID_PRIMARY_SERVICE),
        HI_WORD(GATT_UUID_PRIMARY_SERVICE),
        LO_WORD(GATT_UUID_SIMPLE_PROFILE),          /* service UUID */
        HI_WORD(GATT_UUID_SIMPLE_PROFILE)
    },
    UUID_16BIT_SIZE,                                /* bValueLen */
    NULL,                                           /* p_value_context */
    GATT_PERM_READ                                  /* permissions */
},
/* <<Characteristic>> demo for read */
{
    ATTRIB_FLAG_VALUE_INCL,                        /* flags */
    {                                              /* type_value */
        LO_WORD(GATT_UUID_CHARACTERISTIC),
        HI_WORD(GATT_UUID_CHARACTERISTIC),
        GATT_CHAR_PROP_READ                        /* characteristic properties */
        /* characteristic UUID not needed here, is UUID of next attrib. */
    },
    1,                                              /* bValueLen */
    NULL,
    GATT_PERM_READ                                  /* permissions */
},
{
    ATTRIB_FLAG_VALUE_APPL,                        /* flags */
    {                                              /* type_value */
        LO_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ),
        HI_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ)
    },
    0,                                              /* bValueLen */
    NULL,
    GATT_PERM_READ                                  /* permissions */
},
{
    ATTRIB_FLAG_VOID | ATTRIB_FLAG_ASCII_Z,        /* flags */
    {                                              /* type_value */
        LO_WORD(GATT_UUID_CHAR_USER_DESCR),
        HI_WORD(GATT_UUID_CHAR_USER_DESCR),
    },
    (sizeof(v1_user_descr) - 1),                    /* bValueLen */
    (void *)v1_user_descr,
    GATT_PERM_READ                                  /* permissions */
},
/* <<Characteristic>> demo for write */
{

```

```

        ATTRIB_FLAG_VALUE_INCL,                                /* flags */
        {                                                       /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            (GATT_CHAR_PROP_WRITE | GATT_CHAR_PROP_WRITE_NO_RSP) /* characteristic properties */
        },
        1,                                                       /* bValueLen */
        NULL,
        GATT_PERM_READ                                           /* permissions */
    },
    {
        ATTRIB_FLAG_VALUE_APPL,                                /* flags */
        {                                                       /* type_value */
            LO_WORD(GATT_UUID_CHAR_SIMPLE_V2_WRITE),
            HI_WORD(GATT_UUID_CHAR_SIMPLE_V2_WRITE)
        },
        0,                                                       /* bValueLen */
        NULL,
        GATT_PERM_WRITE                                           /* permissions */
    },
    /* <<Characteristic>>, demo for notify */
    {
        ATTRIB_FLAG_VALUE_INCL,                                /* flags */
        {                                                       /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            (GATT_CHAR_PROP_NOTIFY)                             /* characteristic properties */
        },
        1,                                                       /* bValueLen */
        NULL,
        GATT_PERM_READ                                           /* permissions */
    },
    {
        ATTRIB_FLAG_VALUE_APPL,                                /* flags */
        {                                                       /* type_value */
            LO_WORD(GATT_UUID_CHAR_SIMPLE_V3_NOTIFY),
            HI_WORD(GATT_UUID_CHAR_SIMPLE_V3_NOTIFY)
        },
        0,                                                       /* bValueLen */
        NULL,
        GATT_PERM_NONE                                           /* permissions */
    }

```

```

},
/* client characteristic configuration */
{
    ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL,          /* flags */
    {
        /* type_value */
        LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
        HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
        /* NOTE: this value has an instantiation for each client, a write to */
        /* this attribute does not modify this default value: */
        LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT), /* client char. config. bit field */
        HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
    },
    2, /* bValueLen */
    NULL,
    (GATT_PERM_READ | GATT_PERM_WRITE) /* permissions */
},
/* <<Characteristic>> demo for indicate */
{
    ATTRIB_FLAG_VALUE_INCL, /* flags */
    {
        /* type_value */
        LO_WORD(GATT_UUID_CHARACTERISTIC),
        HI_WORD(GATT_UUID_CHARACTERISTIC),
        (GATT_CHAR_PROP_INDICATE) /* characteristic properties */
        /* characteristic UUID not needed here, is UUID of next attrib. */
    },
    1, /* bValueLen */
    NULL,
    GATT_PERM_READ /* permissions */
},
{
    ATTRIB_FLAG_VALUE_APPL, /* flags */
    {
        /* type_value */
        LO_WORD(GATT_UUID_CHAR_SIMPLE_V4_INDICATE),
        HI_WORD(GATT_UUID_CHAR_SIMPLE_V4_INDICATE)
    },
    0, /* bValueLen */
    NULL,
    GATT_PERM_NONE /* permissions */
},
/* client characteristic configuration */
{
    ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL,          /* flags */
    {
        /* type_value */
        LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),

```

```

        HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
        /* NOTE: this value has an instantiation for each client, a write to */
        /* this attribute does not modify this default value: */
        LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT), /* client char. config. bit field */
        HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
    },
    2, /* bValueLen */
    NULL,
    (GATT_PERM_READ | GATT_PERM_WRITE) /* permissions */
},
};

```

### 3.1.4.3 定义 Service 与 APP 之间的接口

当 service 的 attribute value 被读取或写入时，将通过 APP 注册的回调函数通知 APP。以 simple BLE service 为例，定义类型为 TSIMP\_CALLBACK\_DATA 的数据结构以保存需要通知的结果。

```

typedef struct {
    uint8_t          conn_id;
    T_SERVICE_CALLBACK_TYPE msg_type;
    TSIMP_UPSTREAM_MSG_DATA msg_data;
} TSIMP_CALLBACK_DATA;

```

**msg\_type** 表示操作类型是读操作、写操作或更新 CCCD 操作。

```

typedef enum {
    SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION = 1,
    SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE = 2,
    SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE = 3,
} T_SERVICE_CALLBACK_TYPE;

```

**msg\_data** 保存读操作、写操作或更新 CCCD 操作的数据。

### 3.1.4.4 定义 xxx\_add\_service(), xxx\_set\_parameter(), xxx\_notify(), xxx\_indicate() 等 API

**xxx\_add\_service()** 用于向 profile server layer 添加 service table，注册针对 attribute 的读操作、写操作或更新 CCCD 操作的回调函数。

**xxx\_set\_parameter()** 用于供 APP 设置 service 相关数据。

**xxx\_notify()** 用于发送 notification 数据。

**xxx\_indicate()** 用于发送 indication 数据。

### 3.1.4.5 用 T\_FUN\_GATT\_SERVICE\_CBS 实现回调函数 xxx\_ble\_service\_cbs

xxx\_ble\_service\_cbs 用于处理 client 的读操作、写操作或更新 CCCD 操作。

```
const T_FUN_GATT_SERVICE_CBS simp_ble_service_cbs = {
    simp_ble_service_attr_read_cb, // Read callback function pointer
    simp_ble_service_attr_write_cb, // Write callback function pointer
    simp_ble_service_cccd_update_cb // CCCD update callback function pointer
};
```

在 xxx\_ble\_service\_add\_service() 中调用 server\_add\_service() 以注册该回调函数。

```
T_SERVER_ID simp_ble_service_add_service(void *p_func)
{
    if (false == server_add_service(&simp_service_id,
                                     (uint8_t *)simple_ble_service_tbl,
                                     sizeof(simple_ble_service_tbl),
                                     simp_ble_service_cbs))
    {
        APP_PRINT_ERROR0("simp_ble_service_add_service: fail");
        simp_service_id = 0xff;
        return simp_service_id;
    }
    pfn_simp_ble_service_cb = (P_FUN_SERVER_GENERAL_CB)p_func;
    return simp_service_id;
}
```

## 3.2 BLE Profile Client

### 3.2.1 概述

Profile 的 client 接口给开发者提供 discover services、接收和处理 indication 和 notification、给 GATT Server 发送 read/write request 的功能。

图 3-18 展示 Profile client 层级。Profile 包括 profile client layer 和 specific profile client。位于 protocol stack 之上的 profile client layer 封装供 specific client 访问 protocol stack 的接口，因此针对 specific client 的开发不涉及 protocol stack 的细节，使开发变得更简单和清晰。基于 profile client layer 的 specific client 是由 application layer 实现的，其实现不同于 specific server 的实现。profile client 不涉及 attribute table，提供收集和获取信息的功能，而不是提供 service 和信息。

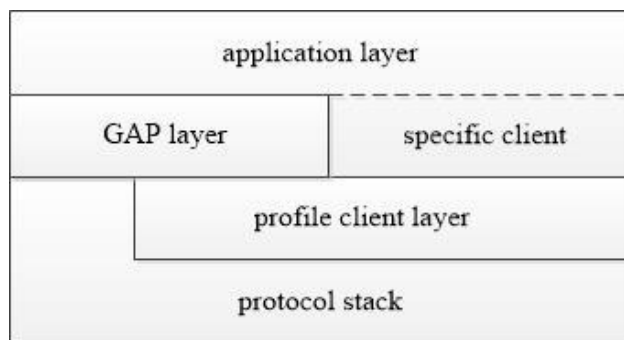


图 3-18 Profile Client 层级

## 3.2.2 支持的 Clients

支持的 clients 如表 3-7 所示。

表 3-7 支持的 Clients

Terms	Definitions	Files
GAP Client	Attribute Service Client	gaps_client.c gaps_client.h
BAS Client	Battery Service Client	bas_client.c bas_client.h
ANCS Client	Apple Notification Center Service Client	ancs_client.c ancs_client.h
SIMP Client	Simple BLE Service Client	simple_ble_client.c simple_ble_client.h

## 3.2.3 Profile Client Layer

Profile Client Layer 处理与 protocol stack layer 的交互，提供接口用于设计 specific client。Client 将对 server 进行 discover services 和 discover characteristics、读取和写入 attribute、接收和处理 notifications 和 indications 相关操作。

### 3.2.3.1 Client 通用回调函数

Client 通用回调函数用于向 APP 发送 client\_all\_primary\_srv\_discovery() 的结果，client\_id 为 CLIENT\_PROFILE\_GENERAL\_ID。该回调函数由 client\_register\_general\_client\_cb() 初始化。

```

void app_le_profile_init(void)
{
    client_init(3);
    .....
    client_register_general_client_cb(app_client_callback);
}

```



```

}

static T_USER_CMD_PARSE_RESULT cmd_srvdis(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    uint8_t conn_id = p_parse_value->dw_param[0];
    T_GAP_CAUSE cause;
    cause = client_all_primary_srv_discovery(conn_id, CLIENT_PROFILE_GENERAL_ID);
    return (T_USER_CMD_PARSE_RESULT)cause;
}

T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void *p_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    APP_PRINT_INFO2("app_client_callback: client_id %d, conn_id %d",
                    client_id, conn_id);
    if (client_id == CLIENT_PROFILE_GENERAL_ID)
    {
        T_CLIENT_APP_CB_DATA *p_client_app_cb_data = (T_CLIENT_APP_CB_DATA *)p_data;
        switch (p_client_app_cb_data->cb_type)
        {
            case CLIENT_APP_CB_TYPE_DISC_STATE:
                .....
        }
    }
}

```

若 APP 不使用 client\_id 为 CLIENT\_PROFILE\_GENERAL\_ID 的 client\_all\_primary\_srv\_discovery(), APP 不需要注册该通用回调函数。

### 3.2.3.2 Specific Client 回调函数

#### 3.2.3.2.1 添加 Client

Protocol client layer 维护所有添加的 specific clients 的信息。首先，初始化需要添加的 client table 的总数目，profile client layer 提供 client\_init()接口用于初始化 client table 数目。

Protocol client layer 提供 client\_register\_spec\_client\_cb()接口以注册 specific client 回调函数。图 3-19 展示 client layer 包含多个 specific client tables，添加 specific client 的情况。

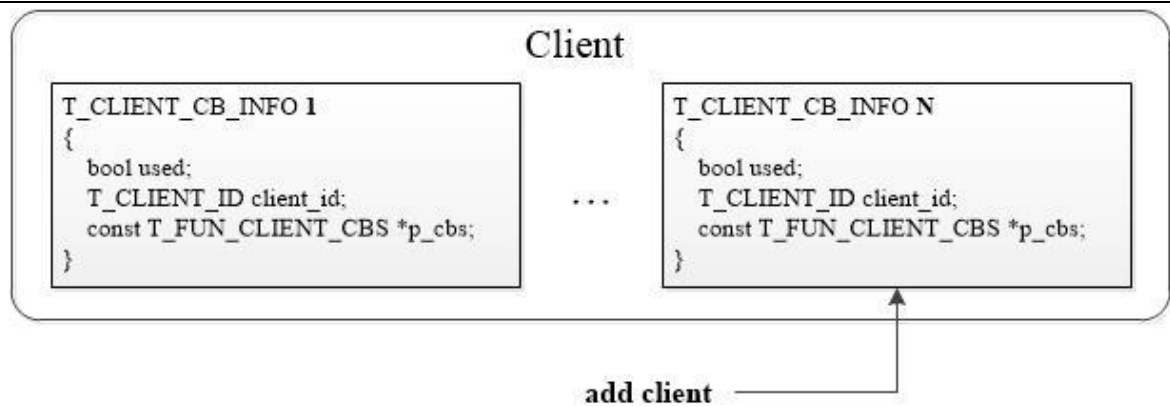


图 3-19 向 Profile Client Layer 添加 Specific Clients

APP 向 profile client layer 添加 specific client 之后，APP 将记录每个已添加 specific client 对应返回的 client id，以实现后续的数据交互流程。

### 3.2.3.2.2 回调函数

需要在 specific client 模块实现 specific client 回调函数，在 `profile_client.h` 中定义 specific client 回调函数数据结构。

```

typedef struct {
    P_FUN_DISCOVER_STATE_CB    discover_state_cb;    //!< Discovery state callback function pointer
    P_FUN_DISCOVER_RESULT_CB   discover_result_cb;   //!< Discovery result callback function pointer
    P_FUN_READ_RESULT_CB       read_result_cb;       //!< Read response callback function pointer
    P_FUN_WRITE_RESULT_CB      write_result_cb;      //!< Write result callback function pointer
    P_FUN_NOTIFY_IND_RESULT_CB  notify_ind_result_cb; //!< Notify Indication callback function pointer
    P_FUN_DISCONNECT_CB        disconnect_cb;        //!< Disconnection callback function pointer
} T_FUN_CLIENT_CBS;
    
```

**discover\_state\_cb:** discovery 状态回调函数，用于向 specific client 模块通知 client\_xxx\_discovery 的 discover 状态。

**discover\_result\_cb:** discovery 结果回调函数，用于向 specific client 模块通知 client\_xxx\_discovery 的 discover 结果。

**read\_result\_cb:** read 结果回调函数，用于向 specific client 模块通知 client\_attr\_read() 或 client\_attr\_read\_using\_uuid() 的读取结果。

**write\_result\_cb:** write 结果回调函数，用于向 specific client 模块通知 client\_attr\_write() 的写入结果。

**notify\_ind\_result\_cb:** notification 或 indication 回调函数，用于向 specific client 模块通知收到来自 server 的 notification 或 indication 数据。

**disconnect\_cb:** disconnection 回调函数，用于向 specific client 模块通知一条 LE 链路已断开。

### 3.2.3.3 Discovery 流程

若本地设备不保存 server 的 handle 信息，那么在与 server 建立 connection 后，client 通常会执行 discovery

流程。Specific client 需要调用 client\_xxx\_discovery() 启动 discovery 流程，然后 specific client 需要处理回调函数 discover\_state\_cb() 中的 discovery 状态以及回调函数 discover\_result\_cb() 中的 discovery 结果。

该流程中各层中间的交互如图 3-20 所示。

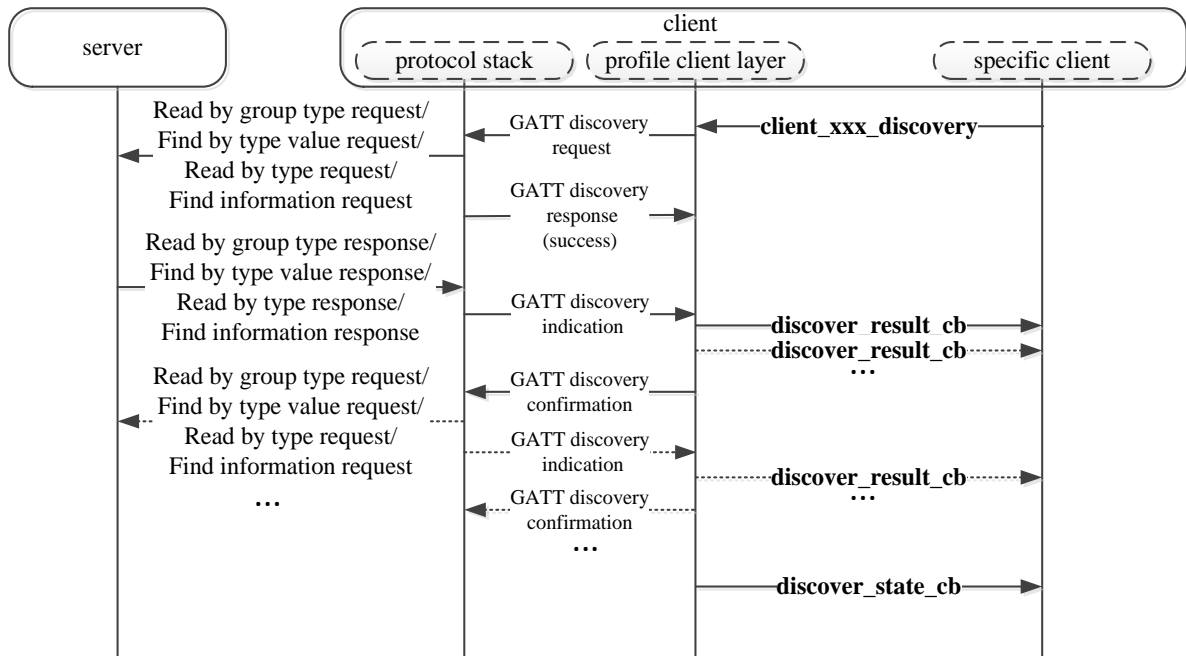


图 3-20 GATT Discovery 流程

### 3.2.3.3.1 Discovery 状态

表 3-8 Discovery 状态

Reference API	T_DISCOVERY_STATE
client_all_primary_srv_discovery()	DISC_STATE_SRV_DONE, DISC_STATE_FAILED
client_by_uuid_srv_discovery()	DISC_STATE_SRV_DONE DISC_STATE_FAILED
client_by_uuid128_srv_discovery()	DISC_STATE_SRV_DONE DISC_STATE_FAILED
client_all_char_discovery()	DISC_STATE_CHAR_DONE DISC_STATE_FAILED
client_all_char_descriptor_discovery()	DISC_STATE_CHAR_DESCRIPTOR_DONE DISC_STATE_FAILED
client_relationship_discovery()	DISC_STATE_RELATION_DONE DISC_STATE_FAILED
client_by_uuid_char_discovery()	DISC_STATE_CHAR_UUID16_DONE DISC_STATE_FAILED

client\_by\_uuid128\_char\_discovery()

DISC\_STATE\_CHAR\_UUID128\_DONE  
DISC\_STATE\_FAILED

### 3.2.3.3.2 Discovery 结果

表 3-9 Discovery 结果

Reference API	T_DISCOVERY_RESULT_TYPE	T_DISCOVERY_RESULT_DATA
client_all_primary_srv_discovery()	DISC_RESULT_ALL_SRV_UUID16	T_GATT_SERVICE_ELEM16 *p_srv_uuid16_disc_data;
client_all_primary_srv_discovery()	DISC_RESULT_ALL_SRV_UUID128	T_GATT_SERVICE_ELEM128 *p_srv_uuid128_disc_data;
client_by_uuid_srv_discovery(), client_by_uuid128_srv_discovery()	DISC_RESULT_SRV_DATA	T_GATT_SERVICE_BY_UUID_ELEM *p_srv_disc_data;
client_all_char_discovery()	DISC_RESULT_CHAR_UUID16	T_GATT_CHARACTER_ELEM16 *p_char_uuid16_disc_data;
client_all_char_discovery()	DISC_RESULT_CHAR_UUID128	T_GATT_CHARACTER_ELEM128 *p_char_uuid128_disc_data;
client_all_char_descriptor_discovery()	DISC_RESULT_CHAR_DESC_UUID16	T_GATT_CHARACTER_DESC_ELEM16 *p_char_desc_uuid16_disc_data;
client_all_char_descriptor_discovery()	DISC_RESULT_CHAR_DESC_UUID128	T_GATT_CHARACTER_DESC_ELEM128 *p_char_desc_uuid128_disc_data;
client_relationship_discovery()	DISC_RESULT_RELATION_UUID16	T_GATT_RELATION_ELEM16 *p_relation_uuid16_disc_data;
client_relationship_discovery()	DISC_RESULT_RELATION_UUID128	T_GATT_RELATION_ELEM128 *p_relation_uuid128_disc_data;
client_by_uuid_char_discovery()	DISC_RESULT_BY_UUID16_CHAR	T_GATT_CHARACTER_ELEM16 *p_char_uuid16_disc_data;
client_by_uuid_char_discovery()	DISC_RESULT_BY_UUID128_CHAR	T_GATT_CHARACTER_ELEM128 *p_char_uuid128_disc_data;

### 3.2.3.4 Characteristic Value Read

该流程用于读取 server 的 characteristic value。在 profile client layer 有两个子流程可用于读取 characteristic value: Read Characteristic Value by Handle 和 Read Characteristic Value by UUID。

#### 3.2.3.4.1 Read Characteristic Value by Handle

当 client 已知 Characteristic Value Handle 时, 该流程可用于读取 server 的 characteristic value。Read Characteristic Value by Handle 流程包含三个阶段, Phase 2 为可选阶段:

1. Phase 1: 调用 `client_attr_read()` 以读取 characteristic value。
2. Phase 2: 可选阶段。若 characteristic value 的长度大于  $(ATT\_MTU - 1)$  字节，Read Response 仅包含 characteristic value 的前  $(ATT\_MTU - 1)$  字节，之后则使用 Read Long Characteristic Value 流程读取 characteristic value。

3. Phase 3: Profile client layer 调用 `read_result_cb()` 以返回读取结果。

该流程中各层之间的交互如图 3-21 所示。

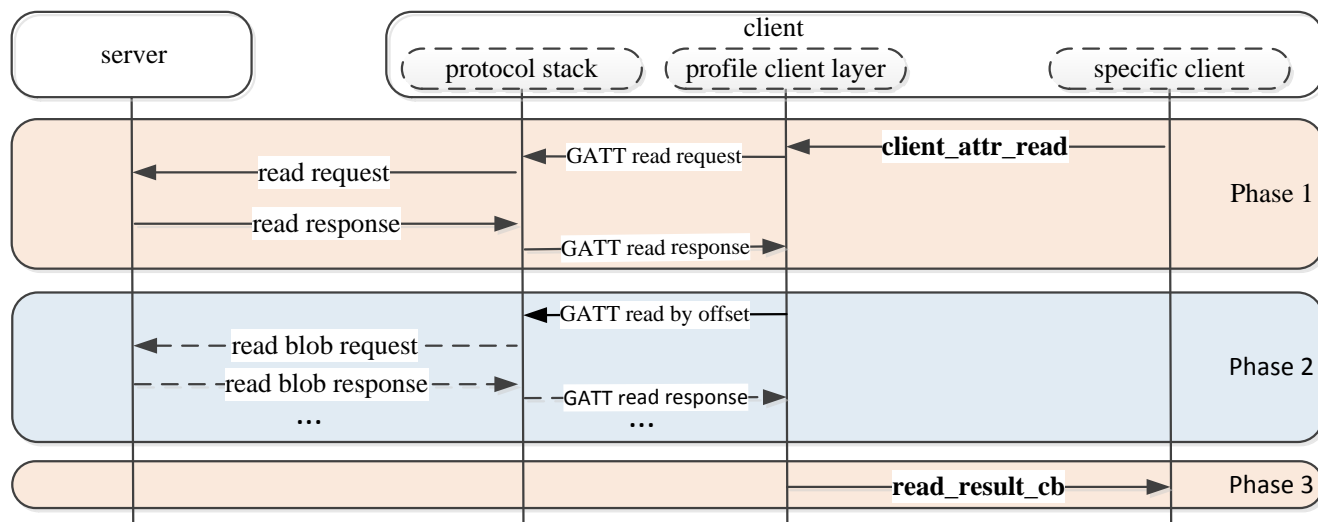


图 3-21 Read Characteristic Value by Handle 流程

### 3.2.3.4.2 Read Characteristic Value by UUID

当 client 仅已知 UUID 时，该流程可用于读取 server 的 characteristic value。Read Characteristic Value by UUID 流程包含三个阶段，Phase 2 为可选阶段：

1. Phase 1: 调用 `client_attr_read_using_uuid()` 以读取 characteristic value。
2. Phase 2: 可选阶段。若 characteristic value 的长度大于  $(ATT\_MTU - 4)$  字节，Read by Type Response 仅包含 characteristic value 的前  $(ATT\_MTU - 4)$  字节，之后则使用 Read Long Characteristic Value 流程读取 characteristic value。

3. Phase 3: Profile client layer 调用 `read_result_cb()` 以返回读取结果。

该流程中各层之间的交互如图 3-22 所示。

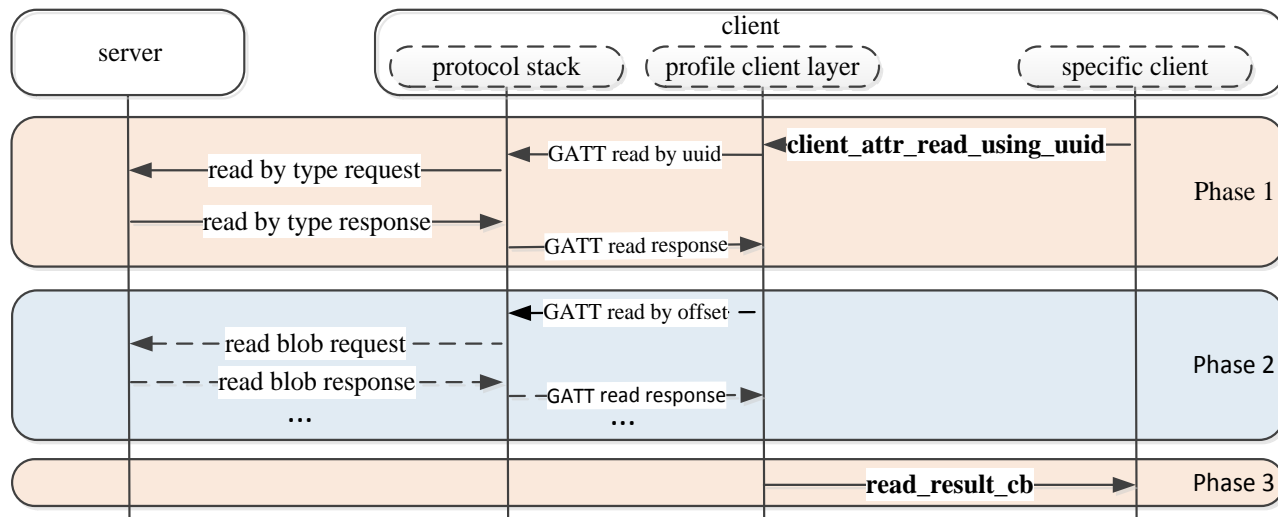


图 3-22 Read Characteristic Value by UUID 流程

### 3.2.3.5 Characteristic Value Write

该流程用于写入 server 的 characteristic value。在 profile client layer 有四个子流程可用于写入 characteristic value: Write without Response、Signed Write without Response、Write Characteristic Value 和 Write Long Characteristic Values。

#### 3.2.3.5.1 Write Characteristic Value

当 client 已知 Characteristic Value Handle 时, 该流程可用于写入 server 的 characteristic value。当 characteristic value 的长度小于或等于 (ATT\_MTU - 3)字节, 将使用该流程。否则, 将使用 Write Long Characteristic Values 流程。

该流程中各层之间的交互如图 3-23 所示。

Value length <= mtu\_size - 3

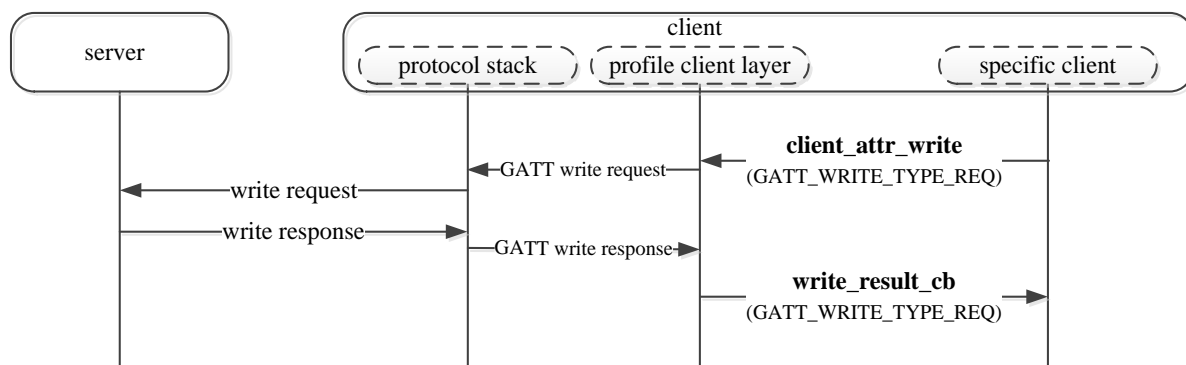


图 3-23 Write Characteristic Value 流程

### 3.2.3.5.2 Write Long Characteristic Values

当 client 已知 Characteristic Value Handle，且 characteristic value 的长度大于 (ATT\_MTU - 3) 字节时，该流程可用于写入 server 的 characteristic value。

该流程中各层之间的交互如图 3-24 所示。

Value length > mtu\_size - 3  
Value length <= 512

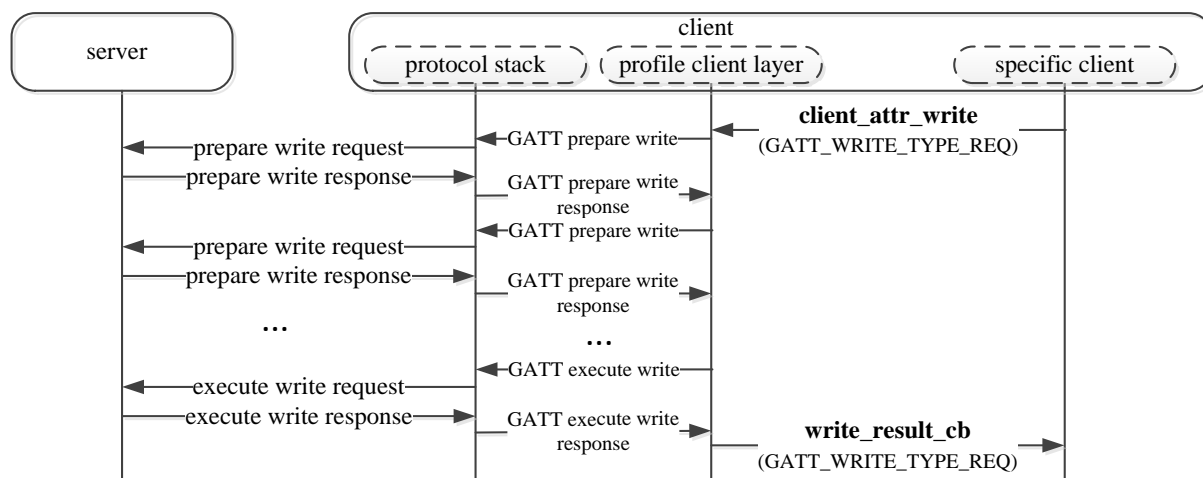


图 3-24 Write Long Characteristic Value 流程

### 3.2.3.5.3 Write Without Response

当 client 已知 Characteristic Value Handle，且 client 不需要写入操作成功执行的应答时，该流程可用于写入 server 的 characteristic value。characteristic value 的长度小于或等于 (ATT\_MTU - 3) 字节。

该流程中各层之间的交互如图 3-25 所示。

Value length <= mtu\_size - 3

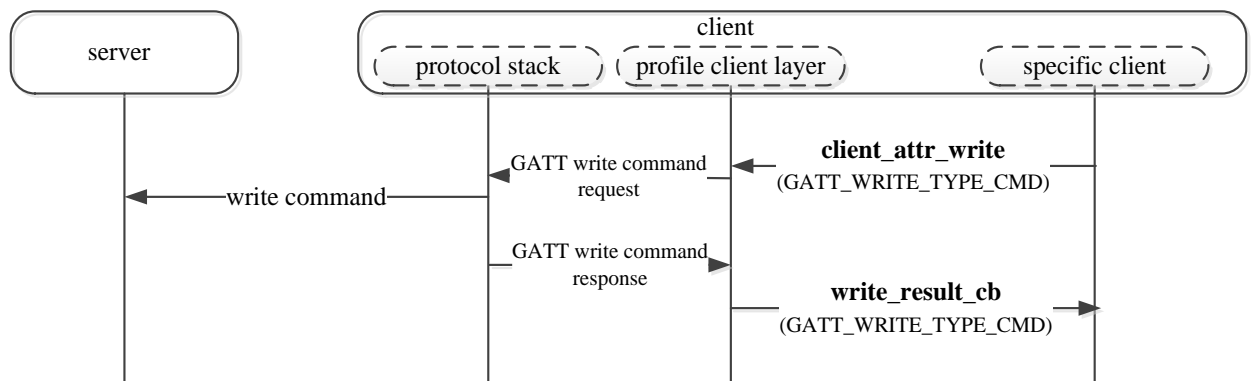


图 3-25 Write Without Response 流程

### 3.2.3.6 Characteristic Value Notification

该流程适用于 server 已被配置为向 client 通知 characteristic value，且不需要成功接收 notification 的应答的情况。

该流程中各层之间的交互如图 3-26 所示。

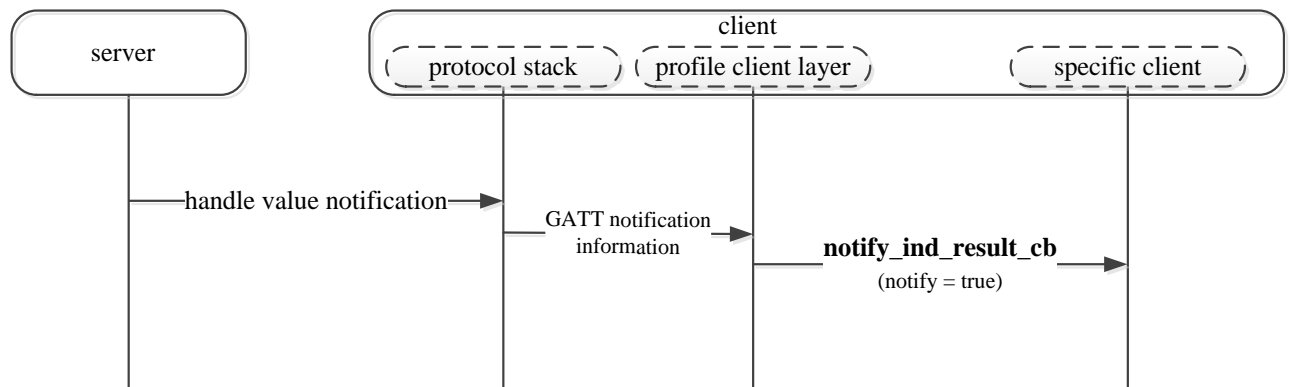


图 3-26 Characteristic Value Notification 流程

profile client layer 未存储 service handle 信息，因此 profile client layer 无法确定发送该 notification 的 specific client。profile client layer 将调用所有注册的 specific clients 回调函数，因此 specific client 需要检查是否需要处理该 notification。

示例代码如下所示：

```

static T_APP_RESULT bas_client_notify_ind_cb(uint8_t conn_id, bool notify, uint16_t handle,
    uint16_t value_size, uint8_t *p_value)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
}
    
```



```

T_BAS_CLIENT_CB_DATA cb_data;
uint16_t *hdl_cache;
hdl_cache = bas_table[conn_id].hdl_cache;
cb_data.cb_type = BAS_CLIENT_CB_TYPE_NOTIF_IND_RESULT;

if (handle == hdl_cache[HDL_BAS_BATTERY_LEVEL])
{
    cb_data.cb_content.notify_data.battery_level = *p_value;
}
else
{
    return APP_RESULT_SUCCESS;
}
if (bas_client_cb)
{
    app_result = (*bas_client_cb)(bas_client, conn_id, &cb_data);
}
return app_result;
}
    
```

### 3.2.3.7 Characteristic Value Indication

该流程适用于 server 已被配置为向 client 通知 characteristic value，且需要成功接收 indication 的应答的情况。

#### 1. 结果未挂起的 Characteristic Value Indication

回调函数 `notify_ind_result_cb()` 的返回结果不为 `APP_RESULT_PENDING`。该流程中各层之间的交互如图 3-27 所示。

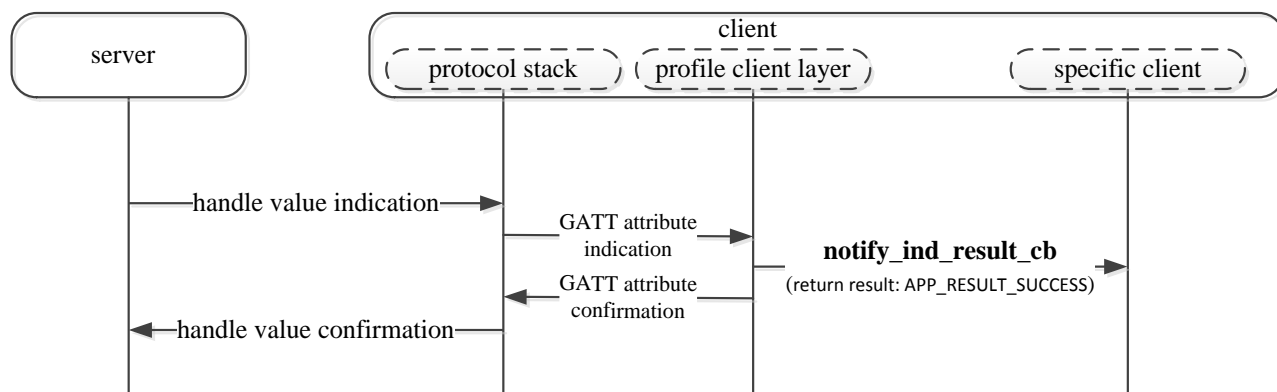


图 3-27 结果未挂起的 Characteristic Value Indication 流程

#### 2. 结果挂起的 Characteristic Value Indication

回调函数 `notify_ind_result_cb()` 的返回结果为 `APP_RESULT_PENDING`。APP 需要调用

client\_attr\_ind\_confirm()以发送 confirmation。该流程中各层之间的交互如图 3-28 所示。

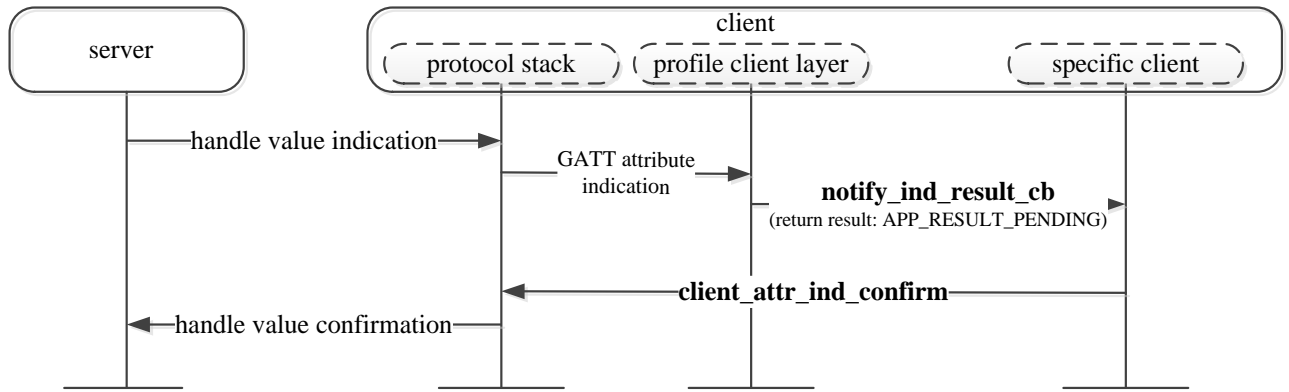


图 3-28 结果挂起的 Characteristic Value Indication 流程

profile client layer 未存储 service handle 信息，因此 profile client layer 无法确定发送该 indication 的 specific client。profile client layer 将调用所有注册的 specific clients 回调函数，因此 specific client 需要检查是否需要处理该 indication。

示例代码如下所示：

```

static T_APP_RESULT simp_ble_client_notif_ind_result_cb(uint8_t conn_id, bool notify,
    uint16_t handle, uint16_t value_size, uint8_t *p_value)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    T_SIMP_CLIENT_CB_DATA cb_data;
    uint16_t *hdl_cache;
    hdl_cache = simp_table[conn_id].hdl_cache;
    cb_data.cb_type = SIMP_CLIENT_CB_TYPE_NOTIF_IND_RESULT;
    if (handle == hdl_cache[HDL_SIMBLE_V3_NOTIFY])
    {
        cb_data.cb_content.notif_ind_data.type = SIMP_V3_NOTIFY;
        cb_data.cb_content.notif_ind_data.data.value_size = value_size;
        cb_data.cb_content.notif_ind_data.data.p_value = p_value;
    }
    else if (handle == hdl_cache[HDL_SIMBLE_V4_INDICATE])
    {
        cb_data.cb_content.notif_ind_data.type = SIMP_V4_INDICATE;
        cb_data.cb_content.notif_ind_data.data.value_size = value_size;
        cb_data.cb_content.notif_ind_data.data.p_value = p_value;
    }
    else
    {
        return app_result;
    }
}
    
```

```

/* Inform application the notif/ind result. */
if (simp_client_cb)
{
    app_result = (*simp_client_cb)(simp_client, conn_id, &cb_data);
}
return app_result;
}

```

### 3.2.3.8 Sequential Protocol

#### 3.2.3.8.1 Request-response protocol

许多 ATT PDUs 是 sequential request-response protocol。一旦 client 向 server 发送 request, 在收到该 server 发送的 response 之前, client 不能发送 request。Server 发送的 indication 同样是 sequential request-response protocol。

以下流程均是 sequential request-response protocol:

- Discovery 流程
- Read Characteristic Value By Handle 流程
- Read Characteristic Value By UUID 流程
- Write Characteristic Value 流程
- Write Long Characteristic Values 流程

在当前流程完成之前, APP 不能启动其它流程。否则, 其它流程会启动失败。

当建立 connection 成功后时, 蓝牙协议层可能会发送 exchange MTU request。GAP 层将发送 GAP\_MSG\_LE\_CONN\_MTU\_INFO 消息以通知 APP, exchange MTU 流程已完成。在收到 GAP\_MSG\_LE\_CONN\_MTU\_INFO 消息后, APP 可以启动以上流程。

```

void app_handle_conn_mtu_info_evt(uint8_t conn_id, uint16_t mtu_size)
{
    APP_PRINT_INFO2("app_handle_conn_mtu_info_evt: conn_id %d, mtu_size %d", conn_id, mtu_size);
    app_discov_services(conn_id, true);
}

```

#### 3.2.3.8.2 Commands

在 ATT 中, 不要求 response 的 command 没有流控机制。

- Write Without Response
- Signed Write Without Response

由于资源有限, 蓝牙协议层对 commands 采用流控机制。

在 GAP 层中维护 credits 数目实现对 Write Command 和 Signed Write Command 的流控, 在收到蓝牙协议层的响应之前, 允许 APP 发送 credits 笔 command。蓝牙协议层能够缓存 credits 笔 command 的数据。

- 当 profile client layer 向协议层发送 command 时, credits 数目减 1

- 当 command 发送给 server 时，协议层会发送响应给 profile client layer，credits 数目加 1
- credits 数目大于 0 时，才可以发送 command

回调函数 `write_result_cb()` 可以通知当前的 credits 数目。APP 也可以将参数类型设为 `GAP_PARAM_LE_REMAIN_CREDITS`，调用 `le_get_gap_param()` 函数获取 credits 数目。

```
void test(void)
{
    uint8_t wds_credits;
    le_get_gap_param(GAP_PARAM_LE_REMAIN_CREDITS, &wds_credits);
}
```

## 4 BLE 示例工程

使用 LE physical transport 的设备可以定义为四种 GAP 角色。

1. Broadcaster
  - 1) 发送 advertisement
  - 2) 不能建立 connection
2. Observer
  - 1) 对 advertisement 进行 scan
  - 2) 不能发起 connection
3. Peripheral
  - 1) 发送 advertisement
  - 2) 作为 slave 角色建立一条 LE 链路
  - 3) 示例 APP: *BLE Peripheral Application*
4. Central
  - 1) 对 advertisement 进行 scan
  - 2) 作为 master 角色发起 connection

### 4.1 BLE Peripheral Application

#### 4.1.1 简介

本节内容是 BLE peripheral application 的概述。BLE peripheral 工程实现简单的 BLE peripheral 设备，可以作为开发基于 peripheral 角色的 APP 的框架。

##### 1. Peripheral 角色的特征：

- 1) 发送 advertising 数据包
- 2) 作为 slave 角色建立 LE 链路

##### 2. 可配特征：

- 1) 支持的 GATT services

GAP Service 和 GATT Service (mandatory)、Battery Service、Simple BLE Service

#### 4.1.2 工程概述

本节内容介绍工程的路径和结构，相关文件路径如下所示：

- 工程源代码路径为 component\common\bluetooth\realtek\sdk\example\ble\_peripheral

### 4.1.3 源代码概述

以下章节介绍该 APP 的重要组成部分。

#### 4.1.3.1 初始化

当上电或芯片重启后，ble\_app\_main ()函数会被调用，并执行以下初始化函数：

```
int ble_app_main(void)
{
    osif_signal_init();
    trace_init();
    bte_init();
    board_init();
    le_gap_init(APP_MAX_LINKS);
    app_le_gap_init();
    app_le_profile_init();
    pwr_mgr_init();
    task_init();
    return 0;
}
```

GAP 和 GATT Profiles 初始化流程如下所示：

1. le\_gap\_init() - 初始化 GAP 并设置 link 数目
2. app\_le\_gap\_init() - GAP 参数的初始化，用户可以通过修改以下参数自定义 application
  - 1) [Device Name 和 Device Appearance 的配置](#)
  - 2) [Advertising 参数的配置](#)
  - 3) [Bond Manager 参数的配置](#)
  - 4) [其它参数的配置](#)
3. app\_le\_profile\_init() - 初始化基于 GATT 的 Profile

更多关于 GAP 的初始化和启动流程的信息参见 [GAP 的初始化和启动流程](#)。

#### 4.1.3.2 GAP 消息处理

一旦收到 GAP message， app\_handle\_gap\_msg()将会被调用。更多关于 GAP message 的信息参见 [BLE GAP 消息](#)。

当收到 GAP\_INIT\_STATE\_STACK\_READY 消息时，peripheral APP 将调用 le\_adv\_start()启动 advertising。若此时 BLE peripheral application 在 evolution board 上运行，设备将是可连接的。对端设备可以对 peripheral 设备进行 scan，并创建 connection。

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
```

```

if (gap_dev_state.gap_init_state != new_state.gap_init_state)
{
    if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
    {
        APP_PRINT_INFO0("GAP stack ready");
        /*stack ready*/
        le_adv_start();
    }
}
.....
}

```

当 peripheral APP 收到 GAP\_CONN\_STATE\_DISCONNECTED 消息时，APP 将调用 le\_adv\_start() 启动 advertising。在连接断开之后，peripheral APP 将恢复为可连接状态。

```

void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    switch (new_state)
    {
        case GAP_CONN_STATE_DISCONNECTED:
        {
            if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
                && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
            {
                APP_PRINT_ERROR1("app_handle_conn_state_evt: connection lost cause 0x%x", disc_cause);
            }
            le_adv_start();
        }
        break;
        .....
    }
}

```

### 4.1.3.3 GAP 回调函数处理

app\_gap\_callback() 用于处理 GAP 回调函数消息，更多关于 GAP 回调函数的信息参见 [BLE GAP 回调函数](#)。

### 4.1.3.4 Profile 消息回调函数

当 APP 使用 xxx\_add\_service 注册 specific service 时，APP 需要注册回调函数以处理 specific service 的消息。APP 需要调用 server\_register\_app\_cb 注册回调函数以处理 profile server layer 的信息。

APP 可以针对不同的 services 注册不同的回调函数，也可以注册通用回调函数以处理 specific services 和 profile server layer 的消息。

app\_profile\_callback()是通用回调函数，根据 service id 区分不同的 services。

```
void app_le_profile_init(void)
{
    server_init(2);
    simp_srv_id = simp_ble_service_add_service(app_profile_callback);
    bas_srv_id = bas_add_service(app_profile_callback);
    server_register_app_cb(app_profile_callback);
}
```

#### 1. 通用 profile server 回调函数

SERVICE\_PROFILE\_GENERAL\_ID 是 profile server layer 使用的 service id。profile server layer 使用的消息包含以下两种消息类型：

- 1) PROFILE\_EVT\_SRV\_REG\_COMPLETE：在 GAP 启动流程中完成 service 注册流程。
- 2) PROFILE\_EVT\_SEND\_DATA\_COMPLETE：profile server layer 使用该消息向 APP 通知发送 notification/indication 的结果。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    if (service_id == SERVICE_PROFILE_GENERAL_ID)
    {
        T_SERVER_APP_CB_DATA *p_param = (T_SERVER_APP_CB_DATA *)p_data;
        switch (p_param->eventId)
        {
            case PROFILE_EVT_SRV_REG_COMPLETE:// srv register result event.
                APP_PRINT_INFO1("PROFILE_EVT_SRV_REG_COMPLETE: result %d",
                                p_param->event_data.service_reg_result);
                break;
            case PROFILE_EVT_SEND_DATA_COMPLETE:
                break;
        }
    }
}
```

#### 2. Battery Service

bas\_srv\_id 是 battery service 的 service id。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    .....
    else if (service_id == bas_srv_id)
    {
        T_BAS_CALLBACK_DATA *p_bas_cb_data = (T_BAS_CALLBACK_DATA *)p_data;
        switch (p_bas_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:
```



```

        .....
    }
}

```

### 3. Simple BLE Service

simp\_srv\_id 是 simple BLE service 的 service id。

```

T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    .....
    else if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:
                .....
        }
    }
}

```

## 4.1.4 测试步骤

首先，编译并将 BLE Peripheral application 下载到 evolution board。BLE Peripheral application 的基本功能如上所述，为实现其它复杂功能，用户可以参考 SDK 提供的使用手册和源代码进行开发。

当 BLE Peripheral application 在 evolution board 上运行时，设备将是可连接的。对端设备可以对 peripheral 设备进行 scan，并创建 connection。在连接断开之后，peripheral APP 将恢复为可连接状态。

### 4.1.4.1 与 iOS 设备测试

步骤简介：基于 iOS 的设备与 BLE 兼容，因此可以 discover 运行 BLE Peripheral Application 的设备。推荐从 App Store 下载 BLE 相关的 APP (例如 LightBlue) 以执行 scan 和 connection 测试。

测试步骤：在 iOS 设备上运行 LightBlue 进行 scan，与 BLE\_PERIPHERAL 设备创建 connection，如图 4-1 所示：

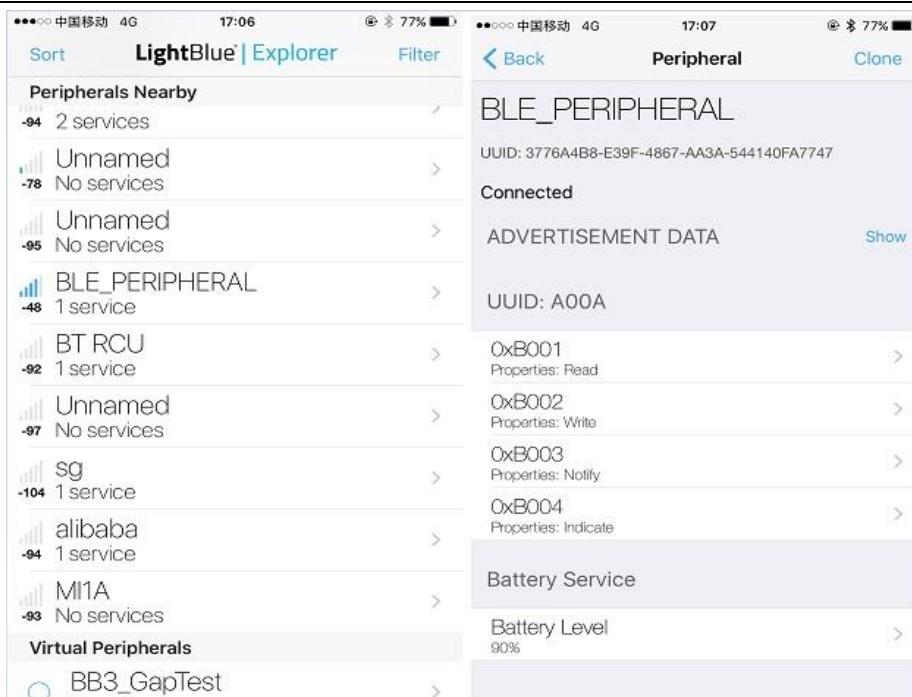


图 4-1 与 iOS 设备测试

## 参考文献

[1] Bluetooth SIG. Core\_v5.0 [M]. 2016, 169.