杰理 ac69 系列 OTA 库开发说明

声明

- 本项目所参考、使用技术必须全部来源于公知技术信息，或自主创新设计。
- 本项目不得使用任何未经授权的第三方知识产权的技术信息。
- 如个人使用未经授权的第三方知识产权的技术信息，造成的经济损失和法律后果由个人承担。

目 录

版本历史

版本	日期	修改者	修改记录
1.6.0	2022/02/16	钟卓成	1、修改文档结构 2、增加调试说明和开源资源 3、补充使用示例说明 4、增加使用流程图
1.5.0	2021/01/08	钟卓成	1、增加蓝牙事件回调说明 2、增加升级操作使用说明
1.4.0	2020/11/05	钟卓成	1、增加错误码说明 2、修改部分错误描述
1.3.0	2020/09/29	钟卓成	1、重构 OTA 库结构 2、修改回连机制 3、更新接口
1.0.3	2020/04/30	钟卓成	1、增加 TWS 设备 OTA 升级的错误码 2、接口更新
1.0.1	2019/09/09	钟卓成	1、增加特殊字段的说明； 比如，onBtDeviceConnection 的 status 和 onMtuChanged 的 status 的区别； 2、IUpgradeCallback 增加回连回调和进度类型; 3、兼容 ufw 的新的升级流程。
1.0.0	2019/03/26	钟卓成	1、初始版本，接口说明，开发说明

第 1 章 概述

本文档是为了方便后续项目维护和管理、记录开发内容而创建。
本文档描述的 OTA 库是基于杰理的 RCSP 协议而实现。同时适应于单双备份的 OTA 流程。

第 2 章 接口说明

主要讲述比较重要的接口，只是作为参考。

2.1 重要接口 IBluetoothManager

IBluetoothManager 是用户必须实现的接口类，需要通过该接口类获取用户 SDK 的蓝牙控制接口，蓝牙传输数据等重要内容。

方法名	参数	返回	描述	作用方式
onBtDeviceConnection	device：蓝牙设备对象 status：连接状态，注意：需要转换为 StateCode 的连接状态	void	传递蓝牙连接状态	用户调用，实现状态传递
onReceiveDeviceData	device：蓝牙设备对象 data： 原始数据	void	传递接收到的蓝牙数据	用户调用，实现数据传递
onMtuChanged	gatt：BluetoothGatt 控制对象 mtu：协商的 BLE 的 mtu status：状态	void	传递协商的 BLE 的 MTU	用户调用，实现数据传递，可以不实现
onError	error：错误事件	void	传递错误事件	用户调用，实现数据传递，可以不实现
getConnectedDevice	void	BluetoothDevice - 已连接的蓝牙设备对象，没有设备连接时，返回 null	获取已连接的蓝牙设备对象	需要用户实现
getConnectedBluetoothGatt	void	BluetoothGatt - 已连接的 BLE 的	获取已连接的 BLE 的 GATT 控制对象	需要用户实现，Spp 方式不实现

		GATT 控制对象，没有设备连接时，返回 null		
connectBluetoothDevice	device - 蓝牙设备对象	void	连接蓝牙设备	需要用户实现
disconnectBluetoothDevice	device - 蓝牙设备对象	void	断开蓝牙设备的连接	需要用户实现
sendDataToDevice	device - 蓝牙设备对象 data: 数据包	boolean - 操作结果	向蓝牙设备发送数据	需要用户实现
queryMandatoryUpdate	IActionCallback<TargetInfoResponse> - 结果回调	void	查询设备升级状态	直接调用

颜色说明

1. 红色表示状态或数据传递的接口
2. 蓝色表示用户实现接口
3. 黑色表示用户可以直接调用的 API

注意事项

1. connectBluetoothDevice 必须保证单独连接，不会影响到其他连接
    1. BLE 方式，只连接 BLE，不连接经典蓝牙
    2. SPP 方式，只连接 SPP，不连接 A2DP、HFP 等
    3. 传递的连接状态需要转换为 StateCode 的连接状态
2. sendDataToDevice 需要保证数据完整的发送
    1. BLE 方式，需要根据 MTU 分包和队列式发数。API 会通知发送的数据，不考虑 MTU。
    2. SPP 方式，无需特殊处理。
3. BLE 方式实现 sendDataToDevice 接口需要注意以下几点
    1. 需要根据 BLE 的 MTU 值，进行 MTU 分包
    2. 不能并发式发数，需要队列式发数
    3. 因为 OTA 功能需要发送大量数据，需要队列式发数，保证数据完整的正确的发送。

重点：参考 3.1 实现相关接口

2.2 OTA 接口 IUpgradeManager

主要是实现 OTA 的流程功能的，由 OTA 库实现，用户调用对应接口即可实现 OTA 升级。

方法名	参数	描述
configure	BluetoothOTAConfigure - OTA 参数	配置 OTA 的实现参数
startOTA	IUpgradeCallback - OTA 的状态回调	开始 OTA 流程
cancelOTA	void	取消 OTA 流程
release	void	释放资源

注意事项

1. cancelOTA：如果是单备份方案，API 无效；如果是双备份方案，API 才有效

重点 ：BluetoothOTAConfigure

BluetoothOTAConfigure 必须在 OTA 前配置。

属性名	类型	描述	参考值
priority	int	OTA 的通讯方式	BluetoothOTAConfigure#PREFER_BLE BluetoothOTAConfigure#PREFER_SPP
isUseReconnect	boolean	是否使用自定义回连方式	默认是不使用
isUseAuthDevice	boolean	是否启用设备认证	默认是开启
firmwareFilePath	String	固件升级文件存放路径	默认为空，升级前需要设置
firmwareFileData	byte[]	固件升级文件数据	默认为空，升级前需要设置， 与 firmwareFilePath 一样，两者选其一即可
mtu	int	调节后的 BLE 的 MTU 值	默认为 20，范围：20-509
isNeedChangeMtu	boolean	是否需要调节 MTU	默认不调节 MTU

2.3 OTA 流程回调 IUpgradeCallback

IUpgradeCallback 是 OTA 流程的状态回调，用于更新 UI。

```
public interface IUpgradeCallback {
    /**
     * OTA 开始
     */
    void onStartOTA();

    /**
     * 需要回连的回调
     * <p>注意: 1.仅连接通讯通道（BLE or SPP）
     * 2.用于单备份 OTA</p>
     *
     * @param addr 回连设备的 MAC 地址
     */
    void onNeedReconnect(String addr);
    /**
     * 进度回调
     *
     * @param type 类型
     *                 <p>0 -- 下载 boot
     *                 or 1 -- 固件升级</p>
     * @param progress 进度
     */
    void onProgress(int type, float progress);
    /**
     * OTA 结束
     */
    void onStopOTA();
    /**
     * OTA 取消
     */
    void onCancelOTA();
    /**
     * OTA 失败
     *
     * @param error 错误信息
     */
    void onError(BaseError error);
}
```

2.4 蓝牙事件回调 IBluetoothCallback

```
public interface IBluetoothCallback {
    /**
     * 蓝牙适配器开关回调
     *
     * @param bEnabled 开关
     * @param bHasBle  是否支持 BLE 功能
     */
    void onAdapterStatus(boolean bEnabled, boolean bHasBle);

    /**
     * 搜索蓝牙设备的状态回调
     *
     * @param bBle 是否搜索 BLE
     * @param bStart 搜索状态
     */
    void onDiscoveryStatus(boolean bBle, boolean bStart);

    /**
     * 发现蓝牙设备的回调
     *
     * @param device 蓝牙设备对象
     */
    @Deprecated
    void onDiscovery(BluetoothDevice device);

    /**
     * 发现蓝牙设备的回调
     * @param device 蓝牙设备对象
     * @param bleScanMessage BLE 扫描数据
     */
    void onDiscovery(BluetoothDevice device, BleScanMessage bleScanMessage);

    /**
     * 蓝牙设备配对状态回调
     *
     * @param device 蓝牙设备对象
     * @param status 配对状态
     */
    void onBondStatus(BluetoothDevice device, int status);

    /**
```

```
     * BLE 数据缓冲区送数据设置回调
     *
     * @param device: BLE 设备
     * @param block 缓冲区大小
     * @param status: 0 - 成功, 其他失败
     */
    void onBleDataBlockChanged(BluetoothDevice device, int block, int status);

    /**
     * 蓝牙设备连接回调
     *
     * @param device 蓝牙设备对象
     * @param status 连接状态
     */
    void onBtDeviceConnection(BluetoothDevice device, int status);

    /**
     * 连接回调
     *
     * @param device 蓝牙设备对象
     * @param status 连接状态
     */
    void onConnection(BluetoothDevice device, int status);

    /**
     * 从蓝牙设备接收到的数据
     *
     * @param device 蓝牙设备对象
     * @param data   原始数据
     */
    void onReceiveData(BluetoothDevice device, byte[] data);

    /**
     * 经典蓝牙的A2DP 服务连接状态
     *
     * @param device 经典蓝牙设备对象
     * @param status 连接状态
     */
    void onA2dpStatus(BluetoothDevice device, int status);

    /**
     * 经典蓝牙的HFP 服务连接状态
     *
     * @param device 经典蓝牙设备对象
```

```
     * @param status 连接状态
     */
    void onHfpStatus(BluetoothDevice device, int status);

    /**
     * 错误事件回调
     *
     * @param error 错误事件
     */
    void onError(BaseError error);  //错误事件回调
}
```

2.5 常量定义

2.5.1. 错误码 ErrorCode

错误码参考 **ErrorCode**

主错误码

错误码	名称	描述
-1	ErrorCode#ERR_UNKNOWN	未知错误
0	ErrorCode#ERR_NONE	没有错误
1	ErrorCode#ERR_COMMON	通用错误
2	ErrorCode#ERR_STATUS	状态错误
3	ErrorCode#ERR_COMMUNICATION	通讯错误
4	ErrorCode#ERR_OTA	OTA 错误
5	ErrorCode#ERR_OTHER	其他错误

详细错误码：(格式：0x[主码值][序号])

错误码	名称	描述
0x3002	ErrorCode#SUB_ERR_SEND_FAILED	发送数据失败
0x3003	ErrorCode#SUB_ERR_PAIR_TIMEOUT	配对超时
0x3004	ErrorCode#SUB_ERR_DATA_FORMAT	数据格式异常
0x3005	ErrorCode#SUB_ERR_PARSE_DATA	解包异常
0x3007	ErrorCode#SUB_ERR_SEND_TIMEOUT	发送数据超时
0x3008	ErrorCode#SUB_ERR_RESPONSE_BAD_STATUS	回复失败状态
0x4001	ErrorCode#SUB_ERR_OTA_FAILED	OTA 升级失败
0x4002	ErrorCode#SUB_ERR_DEVICE_LOW_VOLTAGE	设备低电压
0x4003	ErrorCode#SUB_ERR_CHECK_UPGRADE_FILE	升级文件错误
0x4004	ErrorCode#SUB_ERR_OFFSET_OVER	读取偏移量失败
0x4005	ErrorCode#SUB_ERR_CHECK_RECEIVED_DATA_FAILED	数据校验失败
0x4006	ErrorCode#SUB_ERR_UPGRADE_KEY_NOT_MATCH	加密 key 不匹配
0x4007	ErrorCode#SUB_ERR_UPGRADE_TYPE_NOT_MATCH	升级类型出错
0x4008	ErrorCode#SUB_ERR_OTA_IN_HANDLE	升级程序正在进行
0x4009	ErrorCode#SUB_ERR_UPGRADE_DATA_LEN	升级过程中出现长度错误
0x400A	ErrorCode#SUB_ERR_UPGRADE_FLASH_READ	flash 读写错误

0x400B	ErrorCode#SUB_ERR_UPGRADE_CMD_TIMEOUT	命令超时
0x400C	ErrorCode#SUB_ERR_UPGRADE_FILE_VERSION_SAME	升级文件的固件版本一致
0x400D	ErrorCode#SUB_ERR_TWS_NOT_CONNECT	TWS 未连接
0x400E	ErrorCode#SUB_ERR_HEADSET_NOT_IN_CHARGING_BIN	耳机未在充电仓
0x5001	ErrorCode#SUB_ERR_AUTH_DEVICE	认证设备失败
0x5005	ErrorCode#SUB_ERR_FILE_NOT_FOUND	未找到升级数据
0x5006	ErrorCode#SUB_ERR_IO_EXCEPTION	IO 异常

2.5.2. 状态码 StateCode

1.连接状态

状态码	名称	描述
0	StateCode#CONNECTION_DISCONNECT	断开连接
1	StateCode#CONNECTION_OK	连接成功
2	StateCode#CONNECTION_FAILED	连接失败
3	StateCode#CONNECTION_CONNECTING	连接中
4	StateCode#CONNECTION_CONNECTED	已连接

2.回复状态

状态码	名称	描述
0	StateCode#STATUS_SUCCESS	成功状态
1	StateCode#STATUS_FAIL	失败状态
2	StateCode#STATUS_UNKOWN_CMD	未知命令
3	StateCode#STATUS_BUSY	繁忙状态
4	StateCode#STATUS_NO_RESOURCE	没有资源
5	StateCode#STATUS_CRC_ERROR	CRC 校验错误
6	StateCode#STATUS_ALL_DATA_CRC_ERROR	全部数据 CRC 错误
7	StateCode#STATUS_PARAMETER_ERROR	参数错误
8	StateCode#STATUS_RESPONSE_DATA_OVER_LIMIT	回复数据超出限制

第 3 章 开发说明

用户在开发时只需要实现部分接口和通过部分接口传输蓝牙数据即可，具体的流程由 OTA 库实现。详细需要实现的接口请参考 IBluetoothManager

开发步骤：

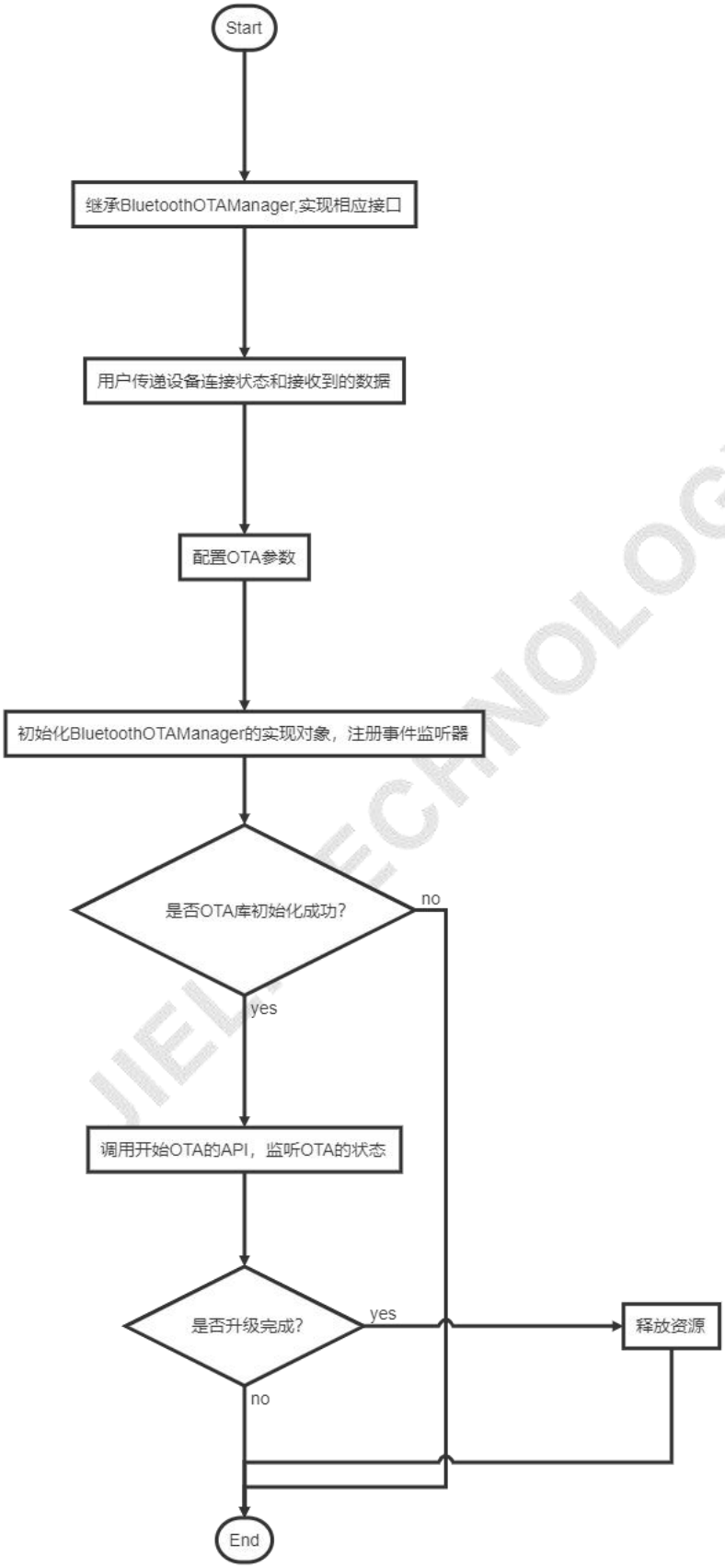
Start
继承BluetoothOTAManager,实现相应接口
用户传递设备连接状态和接收到的数据
配置OTA参数
初始化BluetoothOTAManager的实现对象，注册事件监听器
是否OTA库初始化成功?
no
yes
调用开始OTA的API，监听OTA的状态
是否升级完成?
yes
释放资源
no
End

图 3-1

3.1 继承 BluetoothOTAManager,实现相应接口

```
/**
 * OTA 管理器实现
 */
public class OTAManager extends BluetoothOTAManager {
    private final BleManager bleManager = BleManager.getInstance();//BLE 连接的实现

    public OTAManager(Context context) {
        super(context);
        //TODO:用户通过自行实现的连接库对象完成传递设备连接状态和接收到的数据
         bleManager.registerBleEventCallback(new BleEventCallback() {
            @Override
            public void onBleConnection(BluetoothDevice device, int status) {
                super.onBleConnection(device, status);
                int connectStatus = changeConnectStatus(status); //注意：转变成 OTA 库
的连接状态
                //传递设备的连接状态
                onBtDeviceConnection(device, connectStatus);
            }

            @Override
            public void onBleDataNotification(BluetoothDevice device, UUID serviceUuid,
UUID characteristicsUuid, byte[] data) {
                super.onBleDataNotification(device, serviceUuid, characteristicsUuid, data);
                //传递设备的接收数据
                onReceiveDeviceData(device, data);
            }

            @Override
            public void onBleDataBlockChanged(BluetoothDevice device, int block, int sta
tus) {
                super.onBleDataBlockChanged(device, block, status);
                //传递 BLE 的 MTU 改变
                onMtuChanged(getConnectedBluetoothGatt(), block, status);
            }
        });
    }

    /**
     * 获取已连接的蓝牙设备
     * <p>
     *  注意：是通讯方式对应的蓝牙设备对象
```

```
     * </p>
     */
    @Override
public BluetoothDevice getConnectedDevice() {
        //TODO:用户自行实现
        return bleManager.getConnectedBtDevice();
}

/**
 * 获取已连接的 BluetoothGatt 对象
 * <p>
 * 若选择 BLE 方式 OTA，需要实现此方法。反之，SPP 方式不需要实现
 * </p>
 */
@Override
public BluetoothGatt getConnectedBluetoothGatt() {
        //TODO:用户自行实现
        return bleManager.getConnectedBtGatt();
}

 /**
     * 连接蓝牙设备
     * <p>
     *  注意:这里必须是单纯连接蓝牙设备的通讯方式。
     *  例如，BLE 方式，只连接 BLE，不应联动连接经典蓝牙。
     *  SPP 方式，只连接 SPP，不能连接 A2DP,HFP 和 BLE。
     * </p>
     * @param device 通讯方式的蓝牙设备
     */
@Override
public void connectBluetoothDevice(BluetoothDevice device) {
    //TODO:用户自行实现连接设备
    bleManager.connectBleDevice(device);
}

 /**
     * 断开蓝牙设备的连接
     *
     * @param device 通讯方式的蓝牙设备
     */
@Override
public void disconnectBluetoothDevice(BluetoothDevice device) {
    //TODO:用户自行实现断开设备
    bleManager.disconnectBleDevice(device);
```

```
    }

     /**
      *发送数据到蓝牙设备
      *<p>
      *  注意: 如果是BLE 发送数据，应该注意MTU 限制。BLE 方式会主动把MTU 重新设置为OTA 配置参数设置的值。
      *</p>
      * @param device 已连接的蓝牙设备
      * @param data 数据包
      * @return 操作结果
      */
    @Override
    public boolean sendDataToDevice(BluetoothDevice device, byte[] data) {
        //TODO:用户自行实现发送数据，BLE 方式，需要注意MTU 分包和队列式发数
        bleManager.writeDataByBleAsync(device, BleManager.BLE_UUID_SERVICE, BleManager.BLE_UUID_WRITE, data, new OnWriteDataCallback() {
            @Override
            public void onBleResult(BluetoothDevice device, UUID serviceUUID, UUID characteristicUUID, boolean result, byte[] data) {
                //返回结果
            }
        });
        //也可以阻塞等待结果
        return true;
    }

    /**
     * 用户通知OTA 库的错误事件
     */
    @Override
    public void errorEventCallback(BaseError error) {

    }

    /**
     * 用于释放资源
     */
    @Override
    public void release() {

    }

    //连接状态转换
```

```
    private int changeConnectStatus(int status) {
        int changeStatus = StateCode.CONNECTION_DISCONNECT;
        switch (status) {
            case BluetoothProfile.STATE_DISCONNECTED:
            case BluetoothProfile.STATE_DISCONNECTING: {
                changeStatus = StateCode.CONNECTION_DISCONNECT;
                break;
            }
            case BluetoothProfile.STATE_CONNECTED:
                changeStatus = StateCode.CONNECTION_OK;
                break;
            case BluetoothProfile.STATE_CONNECTING:
                changeStatus = StateCode.CONNECTION_CONNECTING;
                break;
        }
        return changeStatus;
    }
}
```

3.2 用户传递设备连接状态和接收到的数据

3.2.1. 传递连接状态

```
final OTAManager manager = new OTAManager();

//TODO:用户通过自行实现的连接库对象完成传递设备连接状态
BleManager.getInstance().registerBleEventCallback(new BleEventCallback() {

    @Override
    public void onBleConnection(BluetoothDevice device, int status) {
        super.onBleConnection(device, status);
        int connectStatus = changeConnectStatus(status); //注意：转变成 OTA 库的连接状
态
        //传递设备的连接状态
        manager.onBtDeviceConnection(device, connectStatus);
    }
});

//连接状态转换
private int changeConnectStatus(int status) {
    int changeStatus = StateCode.CONNECTION_DISCONNECT;
    switch (status) {
        case BluetoothProfile.STATE_DISCONNECTED:
        case BluetoothProfile.STATE_DISCONNECTING: {
            changeStatus = StateCode.CONNECTION_DISCONNECT;
            break;
        }
        case BluetoothProfile.STATE_CONNECTED:
            changeStatus = StateCode.CONNECTION_OK;
            break;
        case BluetoothProfile.STATE_CONNECTING:
            changeStatus = StateCode.CONNECTION_CONNECTING;
            break;
    }
    return changeStatus;
}
```

3.2.2. 传递接收到的蓝牙数据

```
final OTAManager manager = new OTAManager();

//TODO:用户通过自行实现的连接库对象完成传递接收到的数据
BleManager.getInstance().registerBleEventCallback(new BleEventCallback() {

    @Override
    public void onBleDataNotification(BluetoothDevice device, UUID serviceUuid, UUID cha
racteristicsUuid, byte[] data) {
        super.onBleDataNotification(device, serviceUuid, characteristicsUuid, data);
        //传递设备的接收数据
        manager.onReceiveDeviceData(device, data);
    }
});
```

3.2.3. 传递 BLE 的 MTU 改变

```
final OTAManager manager = new OTAManager();

//TODO:用户通过自行实现的连接库对象完成传递 BLE 的 MTU 改变
BleManager.getInstance().registerBleEventCallback(new BleEventCallback() {

    @Override
    public void onBleDataBlockChanged(BluetoothDevice device, int block, int status) {
        super.onBleDataBlockChanged(device, block, status);
        //传递 BLE 的 MTU 改变
        manager.onMtuChanged(getConnectedBluetoothGatt(), block, status);
    }
});
```

3.3 配置 OTA 参数

```
OTAManager otaManager = new OTAManager();
BluetoothOTAConfigure bluetoothOption = BluetoothOTAConfigure.createDefault();
bluetoothOption.setPriority(BluetoothOTAConfigure.PREFER_BLE) //请按照项目需要选择
                .setUseAuthDevice(true) //具体根据固件的配置选择
                .setBleIntervalMs(500) //默认是 500 毫秒
                .setTimeoutMs(3000) //超时时间
                .setMtu(514) //BLE 底层通讯 MTU 值，会影响 BLE 传输数据的速率。建议用 512。该 MTU 值会使 OTA 库在 BLE 连接时改变 MTU，所以用户 SDK 需要对此处理。
                .setNeedChangeMtu(false); //不需要调整 MTU，建议客户连接时调整好 BLE 的 MTU
bluetoothOption.setFirmwareFilePath(firmwarePath); //设置本地存储 OTA 文件的路径
//          bluetoothOption.setFirmwareFileData(firmwareData);//设置本地存储 OTA 文件的数据，与 setFirmwareFilePath，二者选其一
otaManager.configure(bluetoothOption); //设置 OTA 参数
```

3.4 初始化 BluetoothOTAManager 的实现对象，注册事件监听器

```
//1.构建 OTAManager 对象
OTAManager otaManager = new OTAManager();
//2.注册事件监听器
otaManager.registerBluetoothCallback(new BtEventCallback() {
    @Override
    public void onConnection(BluetoothDevice device, int status) {
        //必须等待库回调连接成功才可以开始 OTA 库操作
        if (status == StateCode.CONNECTION_OK) {
            //...
        }
    }
});
```

3.5 升级操作使用说明

用户开始升级之前必须先等待 OTA 库准备流程跑完，否则容易出现 OTA 异常或者 OTA 开始不了的情况。

```
//OTAManager 继承 BluetoothOTAManager 并实现对应方法
public class OTAManager extends BluetoothOTAManager{
    ...
}

//OTA 库的使用
//1.构建 OTAManager 对象
OTAManager otaManager = new OTAManager();
//2.注册事件监听器
otaManager.registerBluetoothCallback(new BtEventCallback() {
    @Override
    public void onConnection(BluetoothDevice device, int status) {
        //必须等待库回调连接成功才可以开始 OTA 操作
        if (status == StateCode.CONNECTION_OK) {
            //1.可以查询是否需要强制升级
            otaManager.queryMandatoryUpdate(new IActionCallback<TargetInfoResponse>()
 {
                @Override
                public void onSuccess(TargetInfoResponse targetInfoResponse) {
                    //TODO:说明设备需要强制升级，请跳转到 OTA 界面，引导用户升级
固件
                    targetInfoResponse.getVersionCode(); //设备版本号
                    targetInfoResponse.getVersionName();  //设备版本名
                    targetInfoResponse.getProjectCode(); //设备产品 ID(默认是 0，如果设
备支持会改变)
                }

                @Override
                public void onError(BaseError baseError) {
                    //可以不用处理，也可以获取设备信息
                    //没有错误，可以获取设备信息
                    if (baseError.getCode() == ErrorCode.ERR_NONE && baseError.getS
ubCode() == ErrorCode.ERR_NONE){
                        TargetInfoResponse deviceInfo = otaManager.getDeviceInfo();
                        deviceInfo.getVersionCode(); //设备版本号
                        deviceInfo.getVersionName();  //设备版本名
                        deviceInfo.getProjectCode(); //设备产品 ID(默认是 0，如果设备
支持会改变)
                    }
```

```
                }
            });
            //2.也可以直接进行 OTA 升级
            //* 需要先设置升级文件路径 - filePath
            otaManager.getBluetoothOption().setFirmwareFilePath(filePath);
            //* 进行 OTA 升级，然后根据回调进行 UI 更新
            otaManager.startOTA(new IUpgradeCallback() {
                @Override
                public void onStartOTA() {
                    //回调开始 OTA
                }

                @Override
                public void onNeedReconnect(String addr) {
                    //回调需要回连的设备地址
                }

                @Override
                public void onProgress(int type, float progress) {
                    //回调 OTA 进度
                    //type : 0 --- 下载 uboot  1 --- 升级固件
                }

                @Override
                public void onStopOTA() {
                    //回调 OTA 升级完成
                }

                @Override
                public void onCancelOTA() {
                    //回调 OTA 升级被取消
                }

                @Override
                public void onError(BaseError error) {
                    //回调 OTA 升级发生的错误事件
                }
            });
        }
    }
});
```

3.5.1. TargetInfoResponse

```
public class TargetInfoResponse {

    private String versionName;       //设备版本名称
    private int versionCode;          //设备版本信息
    private String protocolVersion; //协议版本

    //经典蓝牙相关信息
    private String edrAddr;     //经典蓝牙地址
    private int edrStatus = 0; //经典蓝牙的连接状态
    private int edrProfile = 0;//经典蓝牙支持的协议

    //BLE 相关信息
    private String bleAddr;    //BLE 地址
    private boolean bleOnly;  //是否仅仅连接 ble 设备

    //SysInfo 属性
    private int volume;        // 设备音量
    private int maxVol;        // 设备最大音量
    private int quantity;      // 设备电量
    private int functionMask;// 功能支持掩码
    private byte curFunction;// 当前模式
    private int sdkType;       // SDK 的标识

    private String name;                    // 名字
    private int pid;                        // 产品 ID
    private int vid;                        // 厂商 ID
    private int uid;                        // 客户 ID
    private int mandatoryUpgradeFlag;       // 强制升级
    private int requestOtaFlag;             // 请求升级标志
    private int ubootVersionCode;           // uboot 版本号
    private String ubootVersionName;        // uboot 版本名称
    private boolean isSupportDoubleBackup; // 是否支持双备份升级（单备份[false]，需要断开回连过程；双备份[true]，不需要断开回连过程）
    private boolean isNeedBootLoader;       // 是否需要下载 boot loader
    private int singleBackupOtaWay;         // 单备份OTA 连接方式(00 -- 不使能 01 -- BLE  02 -- SPP)
    // 拓展模式
    // 0：不启用
    // 1: 升级资源模式
    private int expandMode;
```

```
    private int allowConnectFlag; // 是否允许连接 0 -- 允许 1 -- 不允许

    //用于服务器校验产品信息
    private String authKey;       // 认证秘钥
    private String projectCode; // 项目标识码

    //用于自定义版本信息
    private String customVersionMsg;

    private boolean isSupportMD5;   // 是否支持 MD5 读取
    private boolean isGameMode;     // 是否游戏模式
    ...
}
```

3.5.2. 注意事项

1. 为了防止升级失败导致固件变"砖"，因此固件升级失败后会进入强制升级模式
2. 因为未连接设备时，不知道设备状态，所以库初始化成功后需要查询设备升级状态
3. 用户 SDK 或 APP 应该具备回连上一次连接的蓝牙设备的功能，用于强制升级时自动回连设备，再通过 OTA 升级更新固件
4. 单备份方案，进入 uboot 后，将不存在 A2DP 和 HFP。因此回连方式，不能回连经典蓝牙
5. 双备份方案，不需要回连过程，直接开始升级流程

3.6 释放资源

不再使用 OTA 功能，可以释放 OTAManager 的对象

```
//初始化 OTAManager
OTAManager otaManager = new OTAManager();
// ...
//OTA 操作完成后，需要注销事件监听器和释放资源
//          otaManager.unregisterBluetoothCallback(this);
otaManager.release();
```

第 4 章 调试说明

1. 开关 LOG: 可以使用 JL_Log.setIsLog(boolean bl)设置
2. 保存 LOG 到本地: 前提是 Log 已打开，并调用 JL_Log.setIsSaveLogFile(Context context, boolean isSaveFile)设置
    1. 若开启保存本地 log 文件，退出应用前记得关闭保存 Log 文件
    2. Log 文件保存位置：
        - 如果手机系统是 Android 10.0+，放到./Android/data/[包名]/files/[包名]/logcat/
        - 如果手机系统是 Android 10.0 以下，放到/[包名]/logcat/
    3. Log 文件每次开启保存文件都会创建，所以为了避免太多 Log 文件影响，注意定期清理
    4. Log 文件命名格式：ota_log_app_[时间戳].txt
3. 出现 OTA 问题，请提供以下资料
    1. 简单描述复现 OTA 问题的步骤
    2. 提供出现 OTA 问题的**时间戳最接近**的 Log 文件，可以多份，并说明 OTA 问题的**时间戳**
    3. 最好提供视频或者问题截图

第 5 章 开源资料

5.1 压缩包文件结构

```
apk -- 测试 APK
code -- 演示程序源码
doc -- 开发文档
libs -- 核心库
```

5.2 开源地址

1. 国外用户可以访问 GitHub
    1. 用户可以在 issues 提问，我司开发者会及时提供回复
    2. 用户也可以进入我司的开源群（钉钉：搜索“AC63/69 杰理开源社区”），找到对应负责人进行咨询
2. 国内用户可以访问 Gitee
    1. 用户可以先参考杰理 OTA 库答疑
    2. 用户可以在 issues 提问，我司开发者会及时提供回复
    3. 用户也可以进入我司的开源群（钉钉：搜索“AC63/69 杰理开源社区”），找到对应负责人进行咨询
3. 用户可以关注下我司的官方账号
    1. GitHub 账号：ZhuHai Jieli Technology Co.,Ltd
    2. Gitee 账号：珠海杰理科技

5.3 测试 Demo

应用名：杰理 OTA

包名：com.jieli.otasdk

注意：

1. 不得使用测试 demo 的包名上架任何平台
2. 不建议使用测试 Demo 作为商业用途