FBeacon iOS SDK 接入文档

适用SDK版本	编写日期	编写人员
2.0	2024年4月11日	陈昌华
2.1	2024年11月21日	陈昌华
2.1.2.4	2025年12月17日	陈昌华

1 第一大类FBBluetoothBrowser

涉及到FBBluetoothBrowser、FBFilter类的理解。

1.1 初始化FBBluetoothBrowser对象

```
/// 初始化
/// - Parameter type: 设备数组'peripheralItems'中只放置一种类型的设备
- (instancetype)initWithType:(const FBBluetoothBrowserType)type;
```

说明1：关于type参数的选择

```
typedef enum __FBBluetoothBrowserType : char {
    FBBluetoothBrowserTypeBeacon, // 标准Beacon广播（iBeacon、UID、URL、TLM、ATLBeacon）
    FBBluetoothBrowserTypeSensor, // 自定义广播（e.g 传感器数据）
    FBBluetoothBrowserTypeSetting // Beacon模块
} FBBluetoothBrowserType;
```

【1】如果选择FBBluetoothBrowserTypeBeacon，那么只会将广播了标准Beacon广播（iBeacon、UID、URL、TLM、ATLBeacon）的设备加入到外设数组（peripheralItems），另外将iBeacon广播也作为FBPeripheralItem实例对象加入到外设数组（peripheralItems）。

【2】如果选择FBBluetoothBrowserTypeSensor，那么只会将广播了自定义广播（e.g 传感器数据）的设备加入到外设数组（peripheralItems）。

【3】如果选择FBBluetoothBrowserTypeSetting，那么只会将Beacon设备加入到外设数组（peripheralItems）。

1.2 配置FBBluetoothBrowser对象的delegate

```
/// 委托对象
@property (nonatomic, weak, nullable) id<FBBluetoothBrowserDelegate>
delegate;
```

说明1:关于FBBluetoothBrowserDelegate的理解

```
@protocol FBBluetoothBrowserDelegate <NSObject>

@required
/// 已经完成新外设的添加（外设数组'peripheralItems'中已经添加此外设）
- (void)bluetoothBrowser:(FBBluetoothBrowser *)bluetoothBrowser
didAddPeripheralItem:(FBPeripheralItem *)peripheralItem;
/// 外设数组'peripheralItems'已更新
- (void)bluetoothBrowserDidUpdatePeripheralItems:(FBBluetoothBrowser
*)bluetoothBrowser;

@optional
/// 中心蓝牙状态发生改变
- (void)bluetoothBrowserDidChangeState:(FBBluetoothBrowser
*)bluetoothBrowser;
/// 是否添加新外设'peripheralItem'
- (BOOL)bluetoothBrowser:(FBBluetoothBrowser *)bluetoothBrowser
shouldAddPeripheralItem:(FBPeripheralItem *)peripheralItem;
/// 外设数组'peripheralItems'中的某个外设'peripheralItem'的本地名字发生改变
- (void)bluetoothBrowser:(FBBluetoothBrowser *)bluetoothBrowser
didUpdatePeripheralItemName:(FBPeripheralItem *)peripheralItem;
/// 外设数组'peripheralItems'中的某个外设'peripheralItem'广播包里的名字发生改变
- (void)bluetoothBrowser:(FBBluetoothBrowser *)bluetoothBrowser
didUpdatePeripheralItemAdvertisingName:(FBPeripheralItem *)peripheralItem;
/// 外设数组'peripheralItems'中的某个外设'peripheralItem'信号强度发生改变
- (void)bluetoothBrowser:(FBBluetoothBrowser *)bluetoothBrowser
didUpdatePeripheralItemRSSI:(FBPeripheralItem *)peripheralItem;
@end
```

1.3 FBBluetoothBrowser对象的扫描配置

初始化FBBluetoothBrowser对象之后，就可以拿到对象下面的如下属性：

```objc
/// 外设数组'peripheralItems'中只放置一种类型的设备
@property (nonatomic, readonly) FBBluetoothBrowserType type;
/// 当前的过滤器
@property (nonatomic, readonly, strong, nonnull) FBFilter *filter;
/// 中心蓝牙的状态
@property (nonatomic, readonly, assign) CBManagerState state;
/// 外设数组（调用startScanning、startScanningWithServices:方法后，此属性值会清空）
@property (nonatomic, readonly, strong, nonnull) NSArray<FBPeripheralItem *> *peripheralItems;
/// 已建立蓝牙连接的外设（调用startScanning、startScanningWithServices:方法后，此属性值会清空）
@property (nonatomic, readonly, strong, nonnull) NSArray<FBPeripheralItem *> *connectedPeripheralItems;
/// 委托对象
@property (nonatomic, weak, nullable) id<FBBluetoothBrowserDelegate> delegate;
```

说明1：type、delegate在初始化和配置代理时赋值进去的。

说明2：peripheralItems、connectedPeripheralItems目前肯定是空数组，因为到目前为止没有进行扫描操作和连接操作。

说明3：计划执行扫描操作前，需要根据state属性值确定是否可以进行扫描，枚举值如下：

```objc
typedef NS_ENUM(NSInteger, CBManagerState) {
    CBManagerStateUnknown = 0,
    CBManagerStateResetting,
    CBManagerStateUnsupported,
    CBManagerStateUnauthorized,
    CBManagerStatePoweredOff,
    CBManagerStatePoweredOn,
} NS_ENUM_AVAILABLE(10_13, 10_0);
```

建议为CBManagerStatePoweredOn才执行扫描操作，可根据FBBluetoothBrowser对象的delegate方法及时跟踪state属性值的变化：

```objc
/// 中心蓝牙状态发生改变
- (void)bluetoothBrowserDidChangeState:(FBBluetoothBrowser *)bluetoothBrowser;
```

说明4：关于filter属性的类结构定义如下

```objc
@interface FBFilter : NSObject

/// 是否开启「名称过滤」
@property (nonatomic, assign) BOOL filterByNameEnabled;
/// 筛选的「名称」
@property (nonatomic, strong) NSString *filterName;
/// 筛选的「RSSI最弱值」
@property (nonatomic, assign) CGFloat minimumRSSI;

/// 保存当前各属性值到缓存中，初始化FBBluetoothBrowser对象时默认使用上一次保存的FBFilter各属性值
- (void)saveData;

@end
```

可以拿到filter对象后，配置过滤条件后再执行扫描操作，否则使用上一次保存的FBFilter各属性值。

1.4 开始扫描外设广播

```objc
/// 开始扫描外设
- (BOOL)startScanning;
```

或者

```objc
/// 开始对广播包中包含特定UUID的外设进行扫描
- (BOOL)startScanningWithServices:(nullable NSArray<NSString *> *)serviceUUIDs;
```

说明1：根据初始化配置的FBBluetoothBrowserType枚举值，每当发现新外设时，会询问代理是否添加到外设数组peripheralItems。

```objc
/// 是否添加新外设'peripheralItem'
- (BOOL)bluetoothBrowser:(FBBluetoothBrowser *)bluetoothBrowser shouldAddPeripheralItem:(FBPeripheralItem *)peripheralItem;
```

说明2：完成新外设的添加后，会告知代理

```objc
/// 已经完成新外设的添加（外设数组'peripheralItems'中已经添加此外设）
- (void)bluetoothBrowser:(FBBluetoothBrowser *)bluetoothBrowser didAddPeripheralItem:(FBPeripheralItem *)peripheralItem;
```

1.5 外设数组中外设信息的更新

蓝牙模块会不断的发出广播，包括标准Beacon广播数据的更新、自定义广播比如温湿度数据的更新、RSSI值更新等，都会通过代理方法反馈到delegate对象，App层面可以根据此类回调刷新UI信息：

```
/// 外设数组'peripheralItems'中的某个外设'peripheralItem'的本地名字发生改变
- (void)bluetoothBrowser:(FBBluetoothBrowser *)bluetoothBrowser didUpdatePeripheralItemName:(FBPeripheralItem *)peripheralItem;
/// 外设数组'peripheralItems'中的某个外设'peripheralItem'广播包里的名字发生改变
- (void)bluetoothBrowser:(FBBluetoothBrowser *)bluetoothBrowser didUpdatePeripheralItemAdvertisingName:(FBPeripheralItem *)peripheralItem;
/// 外设数组'peripheralItems'中的某个外设'peripheralItem'信号强度发生改变
- (void)bluetoothBrowser:(FBBluetoothBrowser *)bluetoothBrowser didUpdatePeripheralItemRSSI:(FBPeripheralItem *)peripheralItem;
```

说明1：虽然标准Beacon广播数据的更新、自定义广播比如温湿度数据的更新没有专门的回调方法告知代理，但是可以在外设数组'peripheralItems'中的某个外设'peripheralItem'信号强度发生改变时，使用FBPeripheralItem中的属性获取到最新的其他最新广播数据进行UI刷新。

1.6 外设数组'peripheralItems'按RSSI从强到弱排序

默认情况下外设数组'peripheralItems'中的对象，是按照被扫描到的顺序添加，在实际应用中经常需要将外设数组'peripheralItems'按RSSI从强到弱排序，使用如下方法：

```
/// 将外设数组'peripheralItems'按RSSI从强到弱排序
- (void)sortByRSSI;
```

说明1：SDK完成数据重排之后会通过代理方式告知delegate，App层可在此时机刷新UI：

```
/// 外设数组'peripheralItems'整体发生改变（比如调用sortByRSSI、startScanning、startScanningWithServices:方法）
- (void)bluetoothBrowserDidUpdatePeripheralItems:(FBBluetoothBrowser *)bluetoothBrowser;
```

1.7 停止扫描外设

```
/// 停止扫描外设
- (void)stopScanning;
```

2 第二大类FBPeripheralItem

涉及到FBPeripheralItem、FBBeacon类的理解。

2.1 设备信息结构

通过第一大类的操作步骤，已经可以拿到设备FBPeripheralItem对象，关于该对象中包含的信息结构如下：

```
/// 搜索到的蓝牙外设对象
@interface FBPeripheralItem : NSObject

```

```objc
#pragma mark - 设备基本信息
/// 是否可建立蓝牙连接
@property (nonatomic, assign, readonly, getter = isConnectable) BOOL connectable;
/// 本地名字
@property (nonatomic, strong, readonly, nullable) NSString *name;
/// 唯一标识
@property (nonatomic, strong, readonly, nullable) NSString *UUID;
/// 信号强度
@property (nonatomic, assign, readonly) int RSSI;
/// 广播包里面的名字
@property (nonatomic, strong, readonly, nullable) NSString *advertisingName;
/// 广播包里面的服务UUID
@property (nonatomic, strong, readonly, nullable) NSArray *serviceUUIDs;
/// 最新广播的时间戳
@property (nonatomic, assign, readonly) NSTimeInterval advertisementTimestamp;
/// 广播间隔时间数组
@property (nonatomic, strong, readonly, nullable) NSMutableArray *advertisementSpaceTimestampArray;
/// 广播频率（可根据广播间隔时间数组‘advertisementSpaceTimestampArray’按照App层算法另行估算）
@property (nonatomic, assign, readonly) NSTimeInterval advertisementRate;
/// 用于显示的名字，优先取advertisingName，若advertisingName长度为0则取name
@property (nonatomic, copy, readonly) NSString *displayedName;

#pragma mark - Beacon设备信息
/// 设备是否为Beacon设备
@property (nonatomic, assign, readonly, getter = isBeaconMoudle) BOOL beaconMoudle;
/// 设备模块类型编号
@property (nonatomic, assign, readonly) int modelIndex;
/// 设备MAC地址
@property (nonatomic, strong, readonly, nullable) NSString *macAddress;
/// 取MAC地址后6位
@property (nonatomic, strong, readonly, nullable) NSString *nameSuffix;
/// 设备固件版本号
@property (nonatomic, strong, readonly, nullable) NSString *firmwareVersion;
/// 设备电量
@property (nonatomic, assign, readonly) int quantityOfElectricity;
/// 设备是否包含PIN码（可连接但需要密码）
@property (nonatomic, assign, readonly) BOOL hasPINCode;
/// 设备是否包含NFC
@property (nonatomic, assign, readonly) BOOL hasNFC;
/// 设备是否包含LongRange125kbps广播
@property (nonatomic, assign, readonly) BOOL hasLongRange;
/// 设备是否包含LED
@property (nonatomic, assign, readonly) BOOL hasLED;
/// 设备是否包含蜂鸣器
@property (nonatomic, assign, readonly) BOOL hasBuzzer;
/// 设备是否包含Gsensor
@property (nonatomic, assign, readonly) BOOL hasGsensor;
/// 设备是否包含按键
@property (nonatomic, assign, readonly) BOOL hasKey;

#pragma mark - 标准Beacon广播
/// 标准Beacon广播（iBeacon、UID、URL、TLM、ATLBeacon）数组
```

```objc
@property (nonatomic, strong, readonly, nullable) NSMutableArray<FBBeacon *> *beacons;
/// 最新的标准Beacon广播（iBeacon、UID、URL、TLM、ATLBeacon）
@property (nonatomic, strong, readonly, nullable) FBBeacon *lastestBeacon;

#pragma mark - 自定义广播（e.g 传感器信息）
/// 包含温湿度广播
@property (nonatomic, assign, readonly, getter = isSensor) BOOL sensor;
/// 温度
@property (nonatomic, strong, readonly) NSString *temperature;
/// 湿度
@property (nonatomic, strong, readonly) NSString *humidity;

#pragma mark - 通讯模块
/// 是否打开，即是否可读写
@property (nonatomic, assign, readonly, getter = isOpened) BOOL opened;
/// 写数据时每包数据的间隔，单位为秒，支持的最小值为0.01
@property (nonatomic, assign) NSTimeInterval writeInterval;
/// 指定发送数据的特征UUID
@property (nonatomic, strong) NSString *writeCharacteristicUUID;
/// 发送数据带不带响应
@property (nonatomic, assign) BOOL writeWithoutResponse;

@end
```

说明1：信息结构主要分为四部分，信息结构中数据是否有效由初始化FBBluetoothBrowser对象时传入的FBBluetoothBrowserType枚举值决定，相应关系如下：

```
FBBluetoothBrowserTypeBeacon：
#pragma mark - 设备基本信息 && #pragma mark - Beacon设备信息 && #pragma mark - 标准Beacon广播

FBBluetoothBrowserTypeSensor：
#pragma mark - 设备基本信息 && #pragma mark - Beacon设备信息 && #pragma mark - 自定义广播（e.g 传感器信息）

FBBluetoothBrowserTypeSetting：
#pragma mark - 设备基本信息 && #pragma mark - Beacon设备信息

```

2.2 标准Beacon广播

目前支持iBeacon、UID、URL、TLM、ATLBeacon广播数据，由FBBeacon类封装，数据结构如下：

```objc
typedef enum __FBBeaconType : char {
    FBBeaconTypeUnknown,
    FBBeaconTypeIBeacon,
    FBBeaconTypeURL,
    FBBeaconTypeUID,
    FBBeaconTypeTLM,
    FBBeaconTypeAltBeacon
} FBBeaconType;

typedef enum __FBProximityType: NSUInteger {
```

```
    FBProximityTypeUnknown,
    FBProximityTypeImmediate,
    FBProximityTypeNear,
    FBProximityTypeFar
} FBProximityType;

/// 标准Beacon广播（iBeacon、UID、URL、TLM、ATLBeacon）
@interface FBBeacon : NSObject

/// 类型（iBeacon、URL、UID、TLM、AltBeacon）
@property (nonatomic, assign, readonly) FBBeaconType type;
/// 两个 Beacon 的相等判断
- (BOOL)isEqual:(FBBeacon *)otherBeacon;

#pragma mark - iBeacon

/// iBeacon
@property (nonatomic, strong, readonly, nullable) NSString *proximityUUID;
/// iBeacon
@property (nonatomic, assign, readonly) int major;
/// iBeacon
@property (nonatomic, assign, readonly) int minor;
/// iBeacon
@property (nonatomic, assign, readonly) FBProximityType proximityType;
/// iBeacon
@property (nonatomic, assign, readonly) double accuracy;

#pragma mark - UID<0x00>

/// UID<0x00>
@property (nonatomic, assign, readonly) NSInteger calibratedTxPowerAt0m_uid;
/// UID<0x00>
@property (nonatomic, strong, readonly, nullable) NSString *namespaceString;
/// UID<0x00>
@property (nonatomic, strong, readonly, nullable) NSString *instanceString;
/// UID<0x00>
@property (nonatomic, strong, readonly, nullable) NSString *reservedString;

#pragma mark - URL<0x10>

/// URL<0x10>
@property (nonatomic, assign, readonly) NSInteger calibratedTxPowerAt0m_url;
/// URL<0x10>
@property (nonatomic, strong, readonly, nullable) NSString *URLString;

#pragma mark - TLM<0x20>

/// TLM<0x20>
@property (nonatomic, assign, readonly) NSInteger tlmVersion;
/// TLM<0x20>
@property (nonatomic, assign, readonly) NSInteger batteryVoltage;
/// TLM<0x20>
@property (nonatomic, assign, readonly) NSInteger beaconTemperature;
/// TLM<0x20>
@property (nonatomic, assign, readonly) NSInteger advertisingPDUCount;
/// TLM<0x20>
@property (nonatomic, assign, readonly) NSInteger timeSincePowerOnOrReboot;

```

```
#pragma mark - AltBeacon<0xBEAC>

/// AltBeacon<0xBEAC>
@property (nonatomic, strong, readonly, nullable) NSString *manufacturerID;
/// AltBeacon<0xBEAC>
@property (nonatomic, strong, readonly, nullable) NSString *IDString;
/// AltBeacon<0xBEAC>
@property (nonatomic, assign, readonly) NSInteger calibratedTxPowerAt1m_alt;
/// AltBeacon<0xBEAC>
@property (nonatomic, strong, readonly, nullable) NSString
*manufacturerReservedString;

@end
```

说明1：需要判断FBBeaconType类型，使用对应的有效数据。

3 第三大类FBPeripheralManager

涉及到FBPeripheralManager、FBConfiguration、FBMutableBeacon、FBSession类的理解。

3.1 逻辑概要

如果需要对FBPeripheralItem对象进行配置，就需要第三大类的使用。一般操作逻辑如下：

（1）可配置的参数列表

（2）获取参数的当前配置值

（3）获取参数值的配置范围

（4）设置并保存新的配置值

3.2 初始化FBPeripheralManager对象

```
/// 初始化
- (instancetype)initWithPeripheralItem:(FBPeripheralItem *)peripheralItem
PINCode:(nullable NSString *)PINCode;
```

说明1：初始化完成后，可以通过FBPeripheralManager对象拿到如下对象：

```
/// 外设
@property (nonatomic, strong, readonly) FBPeripheralItem *peripheralItem;
/// 通信会话层
@property (nonatomic, strong, readonly) FBSession *session;
```

同时，可以做好「蓝牙断开回调」属性的配置：

```
/// 蓝牙断开的回调
@property (nonatomic, copy) void(^closeHandler)(NSError * _Nullable);
```

3.3 可配置的参数列表与参数值的配置范围

通过FBPeripheralManager的如下方法和属性可拿到当前FBPeripheralManager对象的可配置参数的列表和参数值的配置范围：

```
/// （外路操作）加载型号、参数取值范围等
- (void)loadConfigurationWithCompletionHandler:(void (^)(FBConfiguration *
_Nullable))completionHandler;
/// 配置参数取值范围
@property (nonatomic, strong, readonly) FBConfiguration *configuration;
/// 可配置参数的名称
@property (nonatomic, strong, readonly) NSArray *paramKeys;
```

注意：在FBPeripheralManager接口文件中会发现有4个操作属于「外路操作」，在进行此类操作方法的调用前，需要根据以下方法判断「操作是否占线」，只有当「handleBusy」属性值为NO时，才可执行。

```
/// 外路操作是否占线
@property (nonatomic, assign, readonly, getter=isHandleBusy) BOOL handleBusy;
```

3.4 获取参数的当前配置值

```
/// （外路操作）从设备读取所有参数的当前配置值
- (void)loadParametersWithCompletionHandler:(void (^)(NSError *
_Nullable))completionHandler;
/// 配置参数的值
@property (nonatomic, strong, readonly) NSArray *paramValues;
/// 标准Beacon广播（iBeacon、UID、URL、TLM、ATLBeacon）数组
@property (nonatomic, strong, readonly) NSMutableArray<FBMutableBeacon *>
*beacons;
/// （闭路操作）查询参数
- (id)valueForName:(NSString *)name;
```

3.5 设置并保存新的配置值

```
/// （闭路操作）更新参数的配置值
- (void)setValue:(id)value forName:(NSString *)name;
/// （外路操作）向设备写入所有参数（包括标准Beacon广播的配置）
- (void)saveParametersWithCompletionHandler:(void (^)(NSError *
_Nullable))completionHandler;
```

说明1：需要注意的是修改Beacon广播内容，需要对「beacons」数组属性中的对象进行修改。

3.6 重置/恢复默认出厂设置

```
/// （外路操作）重置设备
- (void)restoreWithCompletionHandler:(void (^)(NSError *
_Nullable))completionHandler;
```

4 第四大类FBUpgradeManager

涉及到FBUpgradeManager类的理解。

4.1 初始化

```
/// 初始化
- (instancetype)initWithPeripheralItem:(FBPeripheralItem *)peripheralItem
PINCode:(nullable NSString *)PINCode;
```

说明1：初始化完成后可以拿到通讯会话层对象：

```
/// 通讯会话层
@property (nonatomic, strong, readonly) FBSession *session;
```

4.2 解析升级文件

```
/// 解析升级文件信息
/// @param data 文件数据
/// @param complete 完成回调
/// @param faile 失败回调
- (void)infoFromData:(NSData *)data complete:(void(^)(NSString *bootloader,
NSString *binCrc, NSInteger length, NSString *versionRange, NSString *
_Nullable modelType, NSInteger modelTypeNum, NSString *uploadModel, NSString
*crc))complete faile:(void (^)(void))faile;
```

说明1：此方法会校验升级文件的合法性，通过合法性后再由App应用层决定是否使用该文件进行固件升级。

4.3 空中升级

```
/// （外路操作）升级
/// @param path 固件路径
/// @param restore 是否恢复出厂设置
/// @param infoHandler 当前模块的信息（模块型号、固件版本、BT版本）
/// @param progressHandler 升级进度
/// @param completionHandler 升级操作结果
- (void)upgradeWithFilePath:(NSString *)path restore:(BOOL)restore
infoHandler:(void (^)(NSDictionary * _Nullable))infoHandler progressHandler:
(void (^)(CGFloat))progressHandler completionHandler:(void (^)(NSError *
_Nullable))completionHandler;
```

说明1：空中升级属于「外路操作」，但考虑到经过对FBUpgradeManager对象的初始化后，session对象不与FBPeripheralManager对象中的session对象共用，因此不用考虑操作是否占线。

5 第五大类FBSession

该类为App向设备读写数据的一个中间会话层，涉及到蓝牙连接和设备鉴权。在FBPeripheralManager和FBUpgradeManager类的初始化后，会在内部实现初始化一个对应的FBSession对象，并且FBSession对象的delegate为FBPeripheralManager和FBUpgradeManager对象持有，因此在对设备进行配置和空中升级过程，App应用层不需要对FBSession直接进行管理，并且在设备配置和空中升级完成之后，在SDK层会关闭这个中间会话，断开蓝牙连接。在App应用层也有需要主动关闭会话的需要，因此在对设备进行配置和空中升级过程，App层应在适当时候用下面方法关闭会话：

```
/// 关闭会话
- (void)closeSession;
```

当然，根据App应用层的具体使用场景，App层如果确实需要持有FBSession对象的delegate完成通讯操作。分两种情况：

1、如果依然要使用当前的FBSession对象完成对设备进行配置和空中升级过程，请在适当时候将delegate持有交还给FBPeripheralManager和FBUpgradeManager对象。

2、在非第一种情况的前提下，App层可以通过初始化FBSession对象完成自由的数据通讯。

5.1 初始化

```
/// 初始化
- (instancetype)initWithPeripheralItem:(FBPeripheralItem *)peripheralItem
pinCode:(NSString *)pinCode;
/// 代理
@property (nonatomic, weak) id<FBSessionDelegate> delegate;
```

5.2 打开会话

```
/// 打开会话
- (void)openSession;
```

说明1：根据代理回调确定会话层Session的状态变化：

```objc
@protocol FBSessionDelegate <NSObject>
@optional
/// 通话已打开
- (void)sessionDidOpen:(FBSession *)session;
/// 通话已结束
- (void)sessionDidClose:(FBSession *)session error:(NSError *)error;
/// 通话数据写入完成
- (void)sessionDidWrite:(FBSession *)session;
/// 通话接收数据
- (void)session:(FBSession *)session didReceiveData:(NSData *)data;
@end
```

5.3 往设备写入数据/接收数据

写入

```objc
/// 通过会话写数据
- (BOOL)writeDataInSession:(NSData *)data;
```

接收

```objc
/// 通话接收数据
- (void)session:(FBSession *)session didReceiveData:(NSData *)data;
```

5.4 关闭会话

```objc
/// 关闭会话
- (void)closeSession;
```

第六大类 Suota升级

6.1 待升级模块完成Parameters对象初始化

将待升级模块信息完成Parameters单例类部分属性赋值，其中Paramters类数据结构如下：

```objc
@interface Parameters : NSObject

+ (Parameters*) instance;

@property SuotaManager* suotaManager;
@property CBPeripheral* peripheral;
@property NSString *pinCode;
@property NSString *macAddress;
@property CBCentralManager *centralManager;

@end
```

示例：

```
Parameters.instance.peripheral = _peripheralItem.peripheral;
Parameters.instance.pinCode = _pinCode;
Parameters.instance.macAddress = _peripheralItem.macAddress;
```

6.2 OTA升级管理类SuotaManager

（1）使用Parameters单例类中部分信息初始化SuotaManager对象，示例如下：

```
strongSelf.suotaManager = [[SuotaManager alloc]
initWithPeripheral:strongSelf.peripheral suotaManagerDelegate:
(DeviceNavigationController*)strongSelf.parentViewController
pinCode:parameters.pinCode macAddress:parameters.macAddress];
```

（2）然后将SuotaManager对象保存至Parameters单例类中，方便全局管理，示例代码如下：

```
parameters.suotaManager = strongSelf.suotaManager;
```

（3）使用SuotaManager对象建立与待升级模块的蓝牙连接

```
if (strongSelf.suotaManager.state == DEVICE_DISCONNECTED) {
    [strongSelf.suotaManager connect];
}
```

（4）获取待升级模块的固件版本信息

信息通过回调方法进行返回，其中包括：

```
- (void) onCharacteristicRead:(CBUUID*)uuid value:(NSString*)value {
    if ([uuid isEqual:SuotaProfile.CHARACTERISTIC_MANUFACTURER_NAME_STRING])
{
        containerView.manufacturerNameTextLabel.text = [value
isEqualToString:@"Dialog Semi"] ? @"Dialog Semiconductor" : value;
        SuotaLog(TAG, @"Manufacturer: %@", value);
        return;
    }

    if ([uuid isEqual:SuotaProfile.CHARACTERISTIC_MODEL_NUMBER_STRING]) {
        containerView.modelNumberTextLabel.text = value;
        SuotaLog(TAG, @"Model Number: %@", value);
        return;
    }

    if ([uuid isEqual:SuotaProfile.CHARACTERISTIC_FIRMWARE_REVISION_STRING])
{
        containerView.firmwareRevisionTextLabel.text = value;
        SuotaLog(TAG, @"Firmware Revision: %@", value);
```

```
        return;
    }

    if ([uuid isEqual:SuotaProfile.CHARACTERISTIC_SOFTWARE_REVISION_STRING])
{
        containerView.softwareRevisionTextLabel.text = value;
        SuotaLog(TAG, @"Software Revision: %@", value);
        return;
    }

    SuotaLog(TAG, @"Unknown info: %@", value);
}
```

（5）选择OTA固件文件

在这个环节中有两个步骤需要执行，第一步是将OTA固件文件通过解析等方式转换为SuotaFile对象表示，默认将应用程序的 Documents 目录中的有效过滤后的OTA固件文件转为SuotaFile对象数组呈现，示例如下：

```
@property NSArray<SuotaFile*>* fileArray;
self.fileArray = [SuotaFile listFilesWithHeaderInfo];
```

第二步将选中的SuoFile对象赋值到SuotaManager对象的suotaFile属性中，相关代码示例如下：

```
self.suotaManager.suotaFile = [[SuotaFile alloc] initWithURL:url];
```

（6）OTA升级Gpio针脚配置

在Dialog OTA升级过程中的针脚等配置默认应如下：

```
Memory type: SPI
MISO GPIO: P0_3
MOSI GPIO: P0_0
CS GPIO: P0_1
SCK GPIO:   P0_4
Image bank: 2
Block size: 240
```

对应的相关代码如下：

```
[self.suotaManager initializeSuota:blockSize misoGpio:spiMISO
mosiGpio:spiMOSI csGpio:spiCS sckGpio:spiSCK imageBank:imageBank];
```

（7）开始进行OTA升级

```
[self.suotaManager startUpdate];
```

其中关于升级过程的各种回调参考SuotaManagerDelegate协议方法。

6.3 SuotaManagerDelegate协议方法介绍

```
@protocol SuotaManagerDelegate <NSObject>

@required

/*!
 * @method onConnectionStateChange:
 *
 * @param newStatus 新的连接状态。状态代码可以在SuotaManagerStatus中找到。
 *
 * @discussion 每次连接状态更改时触发此方法。
 */
- (void) onConnectionStateChange:(enum SuotaManagerStatus)newStatus;

@required

/*!
 * @method onServicesDiscovered
 *
 * @discussion 在服务发现时触发此方法。
 */
- (void) onServicesDiscovered;

@required

/*!
 * @method onCharacteristicRead:characteristic:
 *
 * @param characteristicGroup 当前的特征组。特征组可以在CharacteristicGroup中找到。
 * @param characteristic 读取到的特征。
 *
 * @discussion 在读取特征时触发此方法。
 */
- (void) onCharacteristicRead:(enum CharacteristicGroup)characteristicGroup characteristic:(CBCharacteristic*)characteristic;

@required

/*!
 * @method onDeviceInfoReadCompleted:
 *
 * @param status DeviceInfoReadStatus状态。指示是否成功读取到设备信息：SUCCESS表示成功，NO_DEVICE_INFO表示没有可读取的设备信息。
 *
 * @discussion 当所有设备信息读取完成时触发此方法。
 */
- (void) onDeviceInfoReadCompleted:(enum DeviceInfoReadStatus)status;

@required

/*!
 * @method onDeviceReady
 *
```

```
 * @discussion 当所有可用的SUOTA信息已读取完毕时触发此方法。如果AUTO_READ_DEVICE_INFO设置为<code>true</code>，则表示设备信息也已读取。
 */
- (void) onDeviceReady;

@required

/*!
 * @method onSuotaLog:type:log:
 *
 * @param state 当前的SuotaProtocolState。
 * @param type 日志类型。
 * @param log 关联的状态更新消息。
 *
 * @discussion 如果NOTIFY_SUOTA_STATUS为<code>true</code>，则会触发此方法。
 */
- (void) onSuotaLog:(enum SuotaProtocolState)state type:(enum SuotaLogType)type log:(NSString*)log;

@required

/*!
 * @method onChunkSend:totalChunks:chunk:block:blockChunks:totalBlocks:
 *
 * @param chunkCount 当前块的数据块数。
 * @param totalChunks 总的数据块数。
 * @param chunk 当前块的数据块数。
 * @param block 当前的数据块。
 * @param blockChunks 当前块中的数据块总数。
 * @param totalBlocks 总的数据块数。
 *
 * @discussion 如果NOTIFY_CHUNK_SEND为<code>true</code>，则在发送数据块时触发此方法。
 */
- (void) onChunkSend:(int)chunkCount totalChunks:(int)totalChunks chunk:(int)chunk block:(int)block blockChunks:(int)blockChunks totalBlocks:(int)totalBlocks;

@required

/*!
 * @method onBlockSent:
 *
 * @param block 当前的数据块数。
 * @param totalBlocks 总的数据块数。
 *
 * @discussion 在成功传输数据块后触发此方法。
 */
- (void) onBlockSent:(int)block totalBlocks:(int)totalBlocks;

@required

/*!
 * @method updateSpeedStatistics:max:min:avg:
 *
 * @param current 当前块的传输速度（Bps）。
 * @param max 最大传输速度（Bps）。
 * @param min 最小传输速度（Bps）。
```

```
 * @param avg 平均传输速度（Bps）。
 *
 * @discussion 如果CALCULATE_STATISTICS为<code>true</code>，则每500ms触发此方法。
 */
- (void) updateSpeedStatistics:(double)current max:(double)max min:(double)min avg:(double)avg;

@required

/*!
 * @method updateCurrentSpeed:
 *
 * @param currentSpeed 每秒发送的字节数。
 *
 * @discussion 如果CALCULATE_STATISTICS为<code>true</code>，则每1000ms触发此方法。这表示当前每秒发送的字节数，而不是平均值。
 */
- (void) updateCurrentSpeed:(double)currentSpeed;

@required

/*!
 * @method onUploadProgress:
 *
 * @param percent 已发送到设备的固件文件的百分比。
 *
 * @discussion 如果NOTIFY_UPLOAD_PROGRESS为<code>true</code>，则在上传进度更新时触发此方法。
 */
- (void) onUploadProgress:(float)percent;

@required

/*!
 * @method onSuccess:imageUploadElapsedSeconds:
 *
 * @param totalElapsedSeconds 执行SUOTA协议的总耗时。
 * @param imageUploadElapsedSeconds 图像上传期间的耗时。
 *
 * @discussion 在SUOTA过程成功完成后触发此方法。如果无法计算耗时，时间参数值为<code>-1</code>。
 */
- (void) onSuccess:(double)totalElapsedSeconds imageUploadElapsedSeconds:(double)imageUploadElapsedSeconds;

@required

/*!
 * @method onFailure:
 *
 * @param errorCode 导致失败的原因。错误代码可以在SuotaErrors和ApplicationErrors中找到。
 *
 * @discussion 在发生意外事件时触发此方法。
 */
- (void) onFailure:(int)errorCode;

```

```
@required

/*!
 * @method onRebootSent
 *
 * @discussion 当重启信号发送到设备时触发此方法。
 */
- (void) onRebootSent;

@end
```