**Android GATT 相关API整理**

# A．扫描(扫描蓝牙设备)

# BluetoothManager

位置：android.bluetooth.BluetoothManager

说明：BLE操作的入口,管理蓝牙设备。可在此类上获取BluetoothAdapter的实例。

主要方法：

## 1.1 getAdapter()

说明：获取当前设备的默认蓝牙适配器。

## 1.2 openGattServer(Context context, BluetoothGattServerCallback callback)

说明：利用该方法可以返回一个GATT Server实例, 可以利用返回的BluetoothGattServer实例进行GATT服务端操作。

PS：在Android 4.3 版本，手机仅可以作为中心设备来使用，Android 5.0之后手机才可以作为外围设备使用，即此时手机可以作为BLE设备为中心设备提供数据。

# BluetoothAdapter

位置：android.bluetooth.BluetoothAdapter

说明：指代本地设备上的蓝牙适配器。该类提供了打开&关闭蓝牙，获取本地蓝牙模块的Mac地址，开始和终止扫描外围BLE设备的方法(Android 4.3)。

主要方法：

## 2.1 startDiscovery()

说明：调用该方法去发现蓝牙设备。调用此方法的前提是必须打开蓝牙适配器，即(BluetoothAdapter.enable()方法必须返回true), 此方法是一个异步方法，调用此方法后系统会发送(BluetoothAdapter.ACTIOIN\_DISCOVERY\_STARTED)广播，如果发现了一个设备,系统会发送(BluetoothDevice.ACTION\_FOUND)广播，可在此广播的意图上获取搜索到的蓝牙设备，如果扫描完毕，系统会发送(BluetoothAdapter.ACTION\_FINISHED)广播。可通过调用cancelDiscovery()方法来结束扫描。

PS: Android 6.0 + 或更高版本需要提供定位权限，如：android.permission.ACCESS\_COARSE\_LOCATION。

## 2.2 cancelDiscovery()

说明: 结束扫描。

## 2.3 enable()

说明：调用该方法打开本地蓝牙适配器。直观表现是系统会弹窗询问用户是否允许打开当前蓝牙适配器。

## 2.4 disable()

说明：调用该方法关闭本地蓝牙适配器。

## 2.5 getDefaultAdapter ()

说明：调用此方法获取本地蓝牙适配器。若硬件平台不支持蓝牙会返回null,。

PS：此方法Android 2.0 就已提供，可藉由此方法扫描经典蓝牙设备，若要扫描BLE设备，推荐使用下列方式(Android 4.3提供)：

BluetoothManager manager = (BluetoothManager) getSystemService(Context.BLUETOOTH\_SERVICE);

BluetoothAdapter adapter1 = manager.getAdapter();

不过实际测试两种方式都可以获取到Adapter。

## 2.6 getBondedDevices ()

说明：获取本地蓝牙适配器已经配对的蓝牙设备。

## 2.7 getState ()

说明：获取本地蓝牙适配器当前的状态，状态可能是下列结果之一：**STATE\_OF**F, **STATE\_TURNING\_ON**, **STATE\_ON**, **STATE\_TURNING\_OFF**。

## 2.8 startLeScan(BluetoothAdapter.LeScanCallback callback)

说明：开始扫描BLE设备。通过onLeScan(BluetoothDevice, int, byte[]) 函数回调扫描的结果。

PS：需要 android.permission.BLUETOOTH\_ADMIN 权限。

PS：Android 5.0（API21）之上推荐使用BluetoothAdapter# getBluetoothLeScanner# startScan() 方法来扫描BLE设备。

## 2.9 startLeScan(UUID[] serviceUuids, BluetoothAdapter.LeScanCallback callback)

说明：功能同上。区别在于只有当外围设备的广播包含参数所给定的服务时，该外围设备才会被返回。

PS：需要 android.permission.BLUETOOTH\_ADMIN 权限。

## 2.10 public void stopLeScan (BluetoothAdapter.LeScanCallback callback)

说明：停止扫描BLE设备。

PS：需要 android.permission.BLUETOOTH\_ADMIN 权限。

## 2.11 getRemoteDevice(String address)

说明：根据指定的Mac地址返回一个蓝牙设备对象。只要传入的MAC地址合法就能返回一个蓝牙设备对象，无论本地蓝牙适配器是否曾发现过该设备。

## 2.12 getBluetoothLeScanner()

说明：用于获取一个扫描BLE设备的帮助类。它会返回一个BluetoothLeScanner类型, 该类型提供了

startScan(ScanCallback callback)、

startScan(List<ScanFilter> filters, ScanSettings settings, ScanCallback callback)

stopScan(ScanCallback callback)

三个方法来扫描和获取BLE设备。

## 2.13 setName(String name)

说明：可通过此方法设置蓝牙设备的名称。该名称最多可包含248个字节(UTF-8编码)

## 2.14 getBluetoothLeAdvertiser()

说明：获取BLE广播的操作对象。通过该对象可进行接下来的开启广播和结束广播操作。

PS：这个对象在以下两种情况下会返回为NULL，1) 手机蓝牙模块不支持BLE广播；2）蓝牙未开启。所以在调用该方法前，一定要判断蓝牙是否已开启，然后再判断获取到的BluetoothLeAdvertiser是否为NULL，然后再进行接下来的操作。

## 2.15 isMultipleAdvertisementSupported

说明：判断芯片是否支持多播，若支持则返回true。

# BluetoothLeScanner

位置：android.bluetooth.le.BluetoothLeScanner

说明：这个类提供了一些扫描BLE设备的方法。

主要方法：

## 3.1 startScan(ScanCallback callback)

说明：开始扫描BLE设备。

PS：BluetoothLeScanner类是Android 5.0（API21）提供的最新的用于BLE扫描的帮助类，在Android 5.0 及之后的版本，建议使用该API进行BLE设备扫描。

## 3.2 startScan(List<ScanFilter> filters, ScanSettings settings, ScanCallback callback)

说明：开始扫描BLE设备。此方法同样可用于BLE设备扫描。与前一个方法的区别在于：该方法提供了一个fiters参数，用于设置扫描的过滤条件，通过ScanSettings.Builer类可进一步指定的需要过滤的设备名(DeviceName), Mac地址、ServiceUUID等信息。

## 3.3 stopScan(ScanCallback callback)

说明：停止扫描BLE设备。

# BluetoothDevice

位置：android.bluetooth.BluetoothDevice

说明：蓝牙设备，代表一个具体的蓝牙外设。

主要方法：

## 4.1 connectGatt(Context context, boolean autoConnect, BluetoothGattCallback callback)

说明：连接到此设备托管的GATT服务端。当前设备作为GATT Client端，该方法提供了一个BluetoothGattCallback抽象类用于接收连接的状态和结果。该抽象类提供了很多回调方法，如：获取Service的发现状态、Characteristic的读取和写入结果、Descriptor的读取和写入结果，当前连接状态的改变(是否成功连接到远端的GATT Server)

## 4.2 connectGatt(Context context, boolean autoConnect, BluetoothGattCallback callback, int transport)

说明：连接到此设备托管的GATT服务端。相较于上一个方法，该方法多了一个参数 int transport, 用于指定GATT与远端双模设备连接时的首选传输类型。共提供了以下三种可选类型：

TRANSPORT\_AUTO：没有优先选择。

TRANSPORT\_BREDR：首选BR/EDR传输。

TRANSPORT\_LE：首选LE传输。

## 4.3 createBond()

说明：与远端设备开始配对。

PS：这是一个异步操作，也就是说调用此方法后系统会立即返回结果。可以通过在广播接收者中注册ACTION\_BOND\_STATE\_CHANGED 这个Action来监听配对过程是否已完成。

## 4.4 getAddress()

说明：获取该蓝牙设备的Mac地址。

## 4.4 getName()

说明：获取该蓝牙设备的名称。

PS：设备名可能为空。

## 4.5 getType()

说明：获取该蓝牙设备的类型。共有4种可能的结果：

DEVICE\_TYPE\_CLASSIC：经典蓝牙设备。

DEVICE\_TYPE\_LE：BLE设备。

DEVICE\_TYPE\_DUAL: 多模设备(BR/EDR/LE)。

DEVICE\_TYPE\_UNKNOWN: 未知设备。

## 4.6 getUuids()

说明：获取该设备所支持的所有UUID。

# BluetoothAdapter.LeScanCallback

位置：android.bluetooth.BluetoothAdapter

说明：BluetoothAdapter的一个内部接口，在调用BluetoothAdapter# startLeScan() 方法时传入。

主要方法：

## 5.1 onLeScan(BluetoothDevice device, int rssi, byte[] scanRecord)

说明：如果在扫描过程中发现了一个BLE设备，系统将会自动调用该方法。该方法的参数包含了远端BLE设备的一些相关信息(设备名、设备地址、信号强度、支持的Server等等)。

# ScanCallback

位置：android.bluetooth.le.ScanCallback

说明：一个抽象类，在调用 BluetoothLeScanner # startScan（）方法时传入。

主要方法：

## 6.1 onScanResult(int callbackType, ScanResult result)

说明：回调方法，当一个外围设备的广播被发现时系统自动回调该方法。

## 6.2 onBatchScanResults(List<ScanResult> results)

说明：批量返回扫描结果。

PS：一般蓝牙设备对象都是通过onScanResult(int,ScanResult)返回，而不会在onBatchScanResults(List)方法中返回，除非手机支持批量扫描模式并且开启了批量扫描模式。批处理的开启请查看ScanSettings。

## 6.3 onScanFailed(int errorCode)

说明：扫描失败，并返回错误码。

# 扫描示例Code

## 声明蓝牙权限和定位权限

<!--蓝牙权限-->

<uses-permission android:name="android.permission.BLUETOOTH"/>

<uses-permission android:name="android.permission.BLUETOOTH\_ADMIN"/>

<!-- LE Beacons位置相关权限-->

<uses-permission android:name="android.permission.ACCESS\_FINE\_LOCATION" />

<uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION" />

<!--蓝牙模块 设置为true表示只有支持蓝牙的手机才能安装-->

<uses-feature

android:name="android.hardware.bluetooth\_le"

android:required="true" />

## 打开蓝牙

//初始化ble设配器

BluetoothManager manager = (BluetoothManager) getSystemService(Context.BLUETOOTH\_SERVICE);

BluetoothAdapter mBluetoothAdapter = manager.getAdapter();

//判断蓝牙是否开启，如果关闭则请求打开蓝牙

if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {

//方式一：请求打开蓝牙

Intent intent = new Intent(BluetoothAdapter.ACTION\_REQUEST\_ENABLE);

startActivityForResult(intent, 1);

//方式二：半静默打开蓝牙

//低版本android会静默打开蓝牙，高版本android会请求打开蓝牙

//mBluetoothAdapter.enable();

}

PS：  mBluetoothAdapter.isEnabled()判断当前蓝牙是否打开，如果蓝牙处于打开状态返回true。同时可以在activity层通过广播监听蓝牙的关闭与开启，进行自己的逻辑处理：

new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

//获取蓝牙广播 本地蓝牙适配器的状态改变时触发

String action = intent.getAction();

if (action.equals(BluetoothAdapter.ACTION\_STATE\_CHANGED)) {

//获取蓝牙广播中的蓝牙新状态

int blueNewState = intent.getIntExtra(BluetoothAdapter.EXTRA\_STATE, 0);

//获取蓝牙广播中的蓝牙旧状态

int blueOldState = intent.getIntExtra(BluetoothAdapter.EXTRA\_STATE, 0);

switch (blueNewState) {

//正在打开蓝牙

case BluetoothAdapter.STATE\_TURNING\_ON:

break;

//蓝牙已打开

case BluetoothAdapter.STATE\_ON:

break;

//正在关闭蓝牙

case BluetoothAdapter.STATE\_TURNING\_OFF:

break;

//蓝牙已关闭

case BluetoothAdapter.STATE\_OFF:

break;

}

}

}

};

## 开始扫描(Android 4.3)

//开始扫描

mBluetoothAdapter.startLeScan(mLeScanCallback);

//停止扫描

mBluetoothAdapter.stopLeScan(mLeScanCallback);

## 开始扫描(Android 5.0+)

//获取 5.0 的扫描类实例

mBLEScanner = mBluetoothAdapter.getBluetoothLeScanner();

//开始扫描

//可设置过滤条件，在第一个参数传入，但一般不设置过滤。

mBLEScanner.startScan(null,mScanSettings,mScanCallback);

//停止扫描

mBLEScanner.stopScan(mScanCallback);

## 扫描回调(Android 4.3)

//5.0以下

mLeScanCallback = new BluetoothAdapter.LeScanCallback() {

@Override

public void onLeScan(BluetoothDevice device, int rssi, byte[] scanRecord) {

//对扫描到的设备进行操作。如：获取设备信息。

}

};

## 扫描回调(Android 5.0)

mScanCallback = new ScanCallback() {

//当一个蓝牙ble广播被发现时回调

@Override

public void onScanResult(int callbackType, ScanResult result) {

super.onScanResult(callbackType, result);

//扫描类型有开始扫描时传入的ScanSettings相关

//对扫描到的设备进行操作。如：获取设备信息。

}

//批量返回扫描结果

//@param results 以前扫描到的扫描结果列表。

@Override

public void onBatchScanResults(List<ScanResult> results) {

super.onBatchScanResults(results);

}

//当扫描不能开启时回调

@Override

public void onScanFailed(int errorCode) {

super.onScanFailed(errorCode);

//扫描太频繁会返回ScanCallback.SCAN\_FAILED\_APPLICATION\_REGISTRATION\_FAILED，表示app无法注册，无法开始扫描。

}

};

# B．当手机作为中心设备时(建立GATT连接、发现服务、读写特征值)

# BluetoothGatt

位置：android.bluetooth.BluetoothGatt

说明：通用属性协议，代表BLE通讯的基本规则和操作。

主要方法：

## 7.1 connect()

说明：重新连接到远端设备。连接断开后，使用此方法重新连接到远端设备。如果设备不在范围内，则一旦设备回到范围内，就会重新触发连接。

## 7.2 disconnect()

说明：断开一个已经建立的连接，或者取消一个当前正在进行的连接尝试。

## 7.3 discoverServices()

说明：发现服务。包括远端设备提供的Service、characteristics、descriptors。

PS：这是一个异步方法，服务发现完毕后，将会触发 BluetoothGattCallback # onServicesDiscovered() 方法，如果发现成功，接下来就可以调用BluetoothGatt# getDevice()方法来检索远端设备所支持的服务。

## 7.4 getDevice()

说明：返回此GATT客户端定位到的远端蓝牙设备。

## 7.4 getServices()

说明：返回由远端设备提供的GATT服务的列表。

PS： 直接调用此方法是无意义的，必须首先调用 discoverServices() 方法等待远程设备扫描服务完毕。

## 7.5 getServices(UUID uuid)

说明：根据指定的UUID返回一个BluetoothGattService对象。前提是指定的UUID远端设备是支持的。

## 7.6 readCharacteristic(BluetoothGattCharacteristic characteristic)

说明：从关联的设备读取请求的Characteristic。这同样也是一个异步方法，如果读取完毕，读取的结果将会包含在BluetoothGattCallback # onCharacteristicRead(BluetoothGatt, BluetoothGattCharacteristic, int) 这个回调方法中。

## 7.7 readDescriptor(BluetoothGattDescriptor descriptor)

说明：从关联的设备读取给定的descriptor的值。一旦读取操作完成，将会触发BluetoothGattCallback # onDescriptorRead(BluetoothGatt, BluetoothGattDescriptor, int) 这个回调方法。

## 7.8 readRemoteRssi()

说明：读取已连接远端设备的信号强度。它将会触发BluetoothGattCallback # onReadRemoteRssi(BluetoothGatt, int, int) 这个回调方法。

## 7.8 requestMtu()

说明：在执行写请求操作(这个写的操作不需要响应)时，发送的数据将被截断成MTU size的大小，这个函数可用于请求更大的MTU大小，以便一次能够发送更多数据。如果此操作成功完成，它将会触发 BluetoothGattCallback # onMtuChanged（）方法。

## 7.9 setCharacteristicNotification (BluetoothGattCharacteristic characteristic, boolean enable)

说明：启用或禁用给定Characteristic的通知。一旦为Characteristic 启用通知后，如果远端设备指定的Characteristic 已更改，它将会触发BluetoothGattCallback # onCharacteristicChanged(BluetoothGatt, BluetoothGattCharacteristic) 回调方法。

## 7.10 boolean writeCharacteristic (BluetoothGattCharacteristic characteristic)

说明：给关联的远程设备写入指定的Characteristic 和值。一旦写入操作成功完成，BluetoothGattCallback # onCharacteristicWrite(BluetoothGatt, BluetoothGattCharacteristic, int) 方法将会被回调。

## 7.11 writeDescriptor(BluetoothGattDescriptor descriptor)

说明：将给定Descriptor 的值写入到关联的远程设备。

# BluetoothGattCallback

位置：android.bluetooth.BluetoothGattCallback

说明：客户端GATT通信的回调类。用于回调各种状态和结果。

主要方法：

## 8.1 onConnectionStateChange(BluetoothGatt gatt, int status, int newState)

说明：连接状态更新回调。可在此方法中判断与远端设备的连接是否建立、断开，还可以拿到执行建立连接操作时返回的错误码。

## 8.2 onServicesDiscovered(BluetoothGatt gatt, int status)

说明：服务发现回调。

## 8.3 onCharacteristicWrite(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic, int status)

说明：特征写入回调。当写入操作完毕后会回调此方法。

## 8.4 onCharacteristicChanged(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic)

说明：远端设备特征值改变回调。

## 8.5 onDescriptorWrite(BluetoothGatt gatt, BluetoothGattDescriptor descriptor, int status)

说明：Descriptor 写入完毕回调。

# BluetoothGattService

位置：android.bluetooth.BluetoothGattService

说明：代表一个Gatt 服务，由0个或多个Characteristic 特征值组成。

主要方法：

## 9.1 addCharacteristic(BluetoothGattCharacteristic characteristic)

说明：添加一个特征值到这个服务。

## 9.2 addService (BluetoothGattService service)

说明：添加一个服务到这个服务。

## 9.3 getCharacteristics()

说明：返回当前服务中所包含的特征列表。

## 9.4 getCharacteristic (UUID uuid)

说明：根据给定的UUID返回该服务所包含的特征列表中的指定的特征。

## 9.5 getIncludedServices()

说明：返回该服务中内置的GATT服务列表。

## 9.6 getInstanceId()

说明：返回此服务的实例ID。如果远程设备提供具有相同UUID的多种服务(例如：针对不同电池的多种电池服务)，则该实例ID可用于区分这些服务。

## 9.7 getType()

说明：返回此服务的类型。可能的值有两种：SERVICE\_TYPE\_PRIMARY (主要服务)、SERVICE\_TYPE\_SECONDARY(次要服务，被主要服务包含)。

## 9.8 getUuid()

说明：返回此服务的UUID。

# BluetoothGattCharacteristic

位置：android.bluetooth.BluetoothGattCharacteristic

说明：特征。里面包含了一组或多组数据，是GATT通信中的最小数据单元。

主要方法：

## 10.1 addDescriptor(BluetoothGattDescriptor descriptor)

说明：添加一个 Descriptor 到这个特征。

## 10.2 getDescriptors()

说明：返回当前特征中的descriptors 列表。

## 10.3 getDescriptor(UUID uuid)

说明：根据指定的UUID返回当前特征中指定的Descriptor。

## 10.4 getFloatValue(int formatType, int offset)

说明：返回此特征中指定位置指定格式的存储值。FormatType代表以何种形式取值。比如：将formatType设置为FORMAT\_UINT16y就说明在偏移量offset处取2个字节的值来生成返回值。 第二个参数代表这个数值中当前特征中的偏移量 (偏移多少能够找到这个值)。

## 10.4 getIntValue(int formatType, int offset)

说明：见上个方法。

## 10.5 getPermissions()

说明：返回此特征的权限。可选值有：

[PERMISSION\_READ](file:///D:\android-sdk\docs\reference\android\bluetooth\BluetoothGattCharacteristic.html#PERMISSION_READ)

[PERMISSION\_READ\_ENCRYPTED](file:///D:\android-sdk\docs\reference\android\bluetooth\BluetoothGattCharacteristic.html#PERMISSION_READ_ENCRYPTED)

[PERMISSION\_READ\_ENCRYPTED\_MITM](file:///D:\android-sdk\docs\reference\android\bluetooth\BluetoothGattCharacteristic.html#PERMISSION_READ_ENCRYPTED_MITM)

[PERMISSION\_WRITE](file:///D:\android-sdk\docs\reference\android\bluetooth\BluetoothGattCharacteristic.html#PERMISSION_WRITE)

….

etc

## 10.6 getProperties()

说明：返回此特征的属性。该属性包含指示该特性的特性标志的位掩码。

## 10.7 getService()

说明：返回此特征所属的服务。

## 10.8 getStringValue(int offset)

说明：见getFloatValue。

## 10.9 getUuid()

说明：返回当前特征的UUID。

## 10.10 getValue()

说明：获取该特征的存储值。当调用BluetoothGatt # readCharacteristic()方法，且收到了BluetoothGattCallback中的更新通知，该值将会更新。

## 10.11 getWriteType()

说明：获取当前特征值的定入类型。可选的值有 WRITE\_TYPE\_DEFAULT、WRITE\_TYPE\_NO\_RESPONSE、WRITE\_TYPE\_SIGNED。

## 10.12 setValue(String value)

说明：设置该特征的存储值。

## 10.13 setValue(int value, int formatType, int offset)

说明：设置该特征的存储值。

Value : 新值。

FormatType: 该值的类型。

Offst: 从哪个位置开始替换。

## 10.14 setValue(byte[] value)

说明：更新此特征的本地存储值。

PS：如果要将该值发送的远程设备，需要调用BluetoothGatt的 writeCharacteristic (BluetoothGattCharacteristic characteristic) 方法。

## 10.15 setWriteType(int writeType)

说明：设置该特征的写入类型。

# BluetoothGattDescriptor

位置：android.bluetooth.BluetoothGattDescriptor

说明：特征描述符。对特征的额外描述，包括但不限于特征的单位、属性等。

## 11.1 getCharacteristic()

说明：返回此描述符所属的特征。

## 11.2 getPermissions()

说明：返回此描述符的权限。

## 11.3 getUuid()

说明：返回此描述符的UUID。

## 11.4 getValue()

说明：返回此描述符存储的值。当调用BlueGatt # readDescriptor（）方法且在BluetoothGattCallback中得到通知之后，该值将会得到更新。

## 11.5 setValue()

说明：更新该描述符的本地存储值。

# GATT客户端通信示例Code

## 根据Mac地址获取到BluetoothDevice用于连接

BluetoothManager bluetoothmanager = (BluetoothManager)context.getSystemService(Context.BLUETOOTH\_SERVICE);

mBluetoothAdapter = bluetoothmanager.getAdapter();

//获取蓝牙设备对象进行连接

mBluetoothDevice = mBluetoothAdapter.getRemoteDevice(macAddressStr)

## 蓝牙GATT回调

//定义子线程handle，用于在BluetoothGattCallback中回调方法中的操作抛到该线程工作。

private Handler mHandler;

//定义handler工作的子线程

private HandlerThread mHandlerThread;

初始化handler

mHandlerThread = new HandlerThread("daqi");

mHandlerThread.start();

//将handler绑定到子线程中

mHandler = new Handler(mHandlerThread.getLooper());

//定义蓝牙Gatt回调类

public class daqiBluetoothGattCallback extends BluetoothGattCallback{

//连接状态回调

@Override

public void onConnectionStateChange(BluetoothGatt gatt, int status, int newState) {

super.onConnectionStateChange(gatt, status, newState);

// status 用于返回操作是否成功,会返回异常码。

// newState 返回连接状态，如BluetoothProfile#STATE\_DISCONNECTED、BluetoothProfile#STATE\_CONNECTED

//操作成功的情况下

if (status == BluetoothGatt.GATT\_SUCCESS){

//判断是否连接码

if (newState == BluetoothProfile.STATE\_CONNECTED) {

}else if(newState == BluetoothProfile.STATE\_DISCONNECTED){

//判断是否断开连接码

}

}else{

//异常码

}

}

//服务发现回调

@Override

public void onServicesDiscovered(BluetoothGatt gatt, int status) {

super.onServicesDiscovered(gatt, status);

}

//特征写入回调

@Override

public void onCharacteristicWrite(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic, int status) {

super.onCharacteristicWrite(gatt, characteristic, status);

}

//外设特征值改变回调

@Override

public void onCharacteristicChanged(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic) {

super.onCharacteristicChanged(gatt, characteristic);

}

//描述写入回调

@Override

public void onDescriptorWrite(BluetoothGatt gatt, BluetoothGattDescriptor descriptor, int status) {

super.onDescriptorWrite(gatt, descriptor, status);

}

}

## 连接设备

 调用BluetoothDevice#connectGatt()进行BLE连接，第二个参数默认选择false,不自动连接。并定义BluetoothGatt变量，存储BluetoothDevice#connectGatt()返回的对象

//定义Gatt实现类  
private BluetoothGatt mBluetoothGatt;   
//创建Gatt回调  
private BluetoothGattCallback mGattCallback = new daqiBluetoothGattCallback();  
//连接设备  
if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.M) {  
 mBluetoothGatt = mBluetoothDevice.connectGatt(mContext,  
 false, mGattCallback, BluetoothDevice.TRANSPORT\_LE);  
} else {  
 mBluetoothGatt = mBluetoothDevice.connectGatt(mContext, false, mGattCallback);  
}

## 连接异常处理

蓝牙连接时，不一定百分百连接成功。连接出错时，会返回异常码进行错误描述。对于大多数异常码，可以通过重连来达到连接成功的目的。

错误代码:

 133 ：连接超时或未找到设备。

8 ： 设备超出范围。

22 ：表示本地设备终止了连接。

//定义重连次数

private int reConnectionNum = 0;

//最多重连次数

private int maxConnectionNum = 3;

public class daqiBluetoothGattCallback extends BluetoothGattCallback{

//连接状态回调

@Override

public void onConnectionStateChange(BluetoothGatt gatt, int status, int newState) {

super.onConnectionStateChange(gatt, status, newState);

// status 用于返回操作是否成功,会返回异常码。

//操作成功的情况下

if (status == BluetoothGatt.GATT\_SUCCESS){

}else{

//重连次数不大于最大重连次数

if(reConnectionNum < maxConnectionNum){

//重连次数自增

reConnectionNum++

//连接设备

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.M) {

mBluetoothGatt = mBluetoothDevice.connectGatt(mContext,

false, mGattCallback, BluetoothDevice.TRANSPORT\_LE);

} else {

mBluetoothGatt = mBluetoothDevice.connectGatt(mContext, false, mGattCallback);

}

}else{

//断开连接，返回连接失败回调

}

}

}

//其他回调方法

｝

## 发现服务

连接成功后，将会触发BluetoothGattCallback#onConnectionStateChange()方法，可在此方法中进行发现服务的操作。

public class MyBluetoothGattCallback extends BluetoothGattCallback{

//连接状态回调

@Override

public void onConnectionStateChange(BluetoothGatt gatt, int status, int newState) {

super.onConnectionStateChange(gatt, status, newState);

// status 用于返回操作是否成功,会返回异常码。

//操作成功的情况下

if (status == BluetoothGatt.GATT\_SUCCESS){

//判断是否连接码

if (newState == BluetoothProfile.STATE\_CONNECTED) {

//可延迟发现服务，也可不延迟

mHandler.post(() ->

//发现服务

mBluetoothGatt.discoverServices();

);

} else if(newState == BluetoothProfile.STATE\_DISCONNECTED){

//判断是否断开连接码

}

}

}

//其他回调方法

｝

当发现服务成功后，会触发BluetoothGattCallback#onServicesDiscovered()回调：

//定义需要进行通信的ServiceUUID

private UUID mServiceUUID = UUID.fromString("0000xxxx-0000-1000-8000-00805f9b34fb");

//定义蓝牙Gatt回调类

public class MyBluetoothGattCallback extends BluetoothGattCallback{

//服务发现回调

@Override

public void onServicesDiscovered(BluetoothGatt gatt, int status) {

super.onServicesDiscovered(gatt, status);

if (status == BluetoothGatt.GATT\_SUCCESS) {

mHandler.post(() ->

//获取指定uuid的service

BluetoothGattService gattService = mBluetoothGatt.getService(mServiceUUID);

//获取到特定的服务不为空

If (gattService != null){

} else {

//获取特定服务失败

}

);

}

}

｝

发现服务时，会存在发现不了特定服务的情况。或者说，整个BluetoothGatt对象中的服务列表为空。BluetoothGatt类中存在一个隐藏的方法refresh（），用于刷新Gatt的服务列表。当发现不了服务时，可以通过反射去调用该方法。

## 修改特征值并监听外部特征值改变

读写特征值

//定义需要进行通信的ServiceUUID

private UUID mServiceUUID = UUID.fromString("0000xxxx-0000-1000-8000-00805f9b34fb");

//定义需要进行通信的CharacteristicUUID

private UUID mCharacteristicUUID = UUID.fromString("0000yyyy-0000-1000-8000-00805f9b34fb");

//定义蓝牙Gatt回调类

public class daqiBluetoothGattCallback extends BluetoothGattCallback{

//服务发现回调

@Override

public void onServicesDiscovered(BluetoothGatt gatt, int status) {

super.onServicesDiscovered(gatt, status);

if (status == BluetoothGatt.GATT\_SUCCESS) {

mHandler.post(() ->

//获取指定uuid的service

BluetoothGattService gattService = mBluetoothGatt.getService(mServiceUUID);

//获取到特定的服务不为空

if(gattService != null){

//获取指定uuid的Characteristic

BluetoothGattCharacteristic gattCharacteristic = gattService.getCharacteristic(mCharacteristicUUID);

//获取特定特征成功

if（gattCharacteristic != null）{

//写入你需要传递给外设的特征值（即传递给外设的信息）

gattCharacteristic.setValue(bytes);

//通过GATt实体类将，特征值写入到外设中。

mBluetoothGatt.writeCharacteristic(gattCharacteristic);

//如果只是需要读取外设的特征值：

//通过Gatt对象读取特定特征（Characteristic）的特征值

String readValue = mBluetoothGatt.readCharacteristic(gattCharacteristic);

}

}else{

//获取特定服务失败

}

);

}

}

｝

当成功读取特征值时，会触发BluetoothGattCallback#onCharacteristicRead()回调。

@Override

public void onCharacteristicRead(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic, int status) {

super.onCharacteristicRead(gatt, characteristic, status);

if (status == BluetoothGatt.GATT\_SUCCESS) {

//获取读取到的特征值

characteristic.getValue()

｝

}

当成功写入特征值到外设时，会触发BluetoothGattCallback#onCharacteristicWrite()回调。

@Override

public void onCharacteristicWrite(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic, int status) {

super.onCharacteristicWrite(gatt, characteristic, status);

if (status == BluetoothGatt.GATT\_SUCCESS) {

//获取写入到外设的特征值

characteristic.getValue()

｝

}

当写入完特征值后，外设会修改自己的特征值，手机会触发BluetoothGattCallback#onCharacteristicChanged()方法，进行双向通信。

@Override

public void onCharacteristicChanged(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic) {

if (status == BluetoothGatt.GATT\_SUCCESS) {

//获取外设修改的特征值

String value = characteristic.getValue()

//对特征值进行解析

｝

｝

## 断开连接

断开连接的操作分为两步：

1. mBluetoothGatt.disconnect();
2. mBluetoothGatt.close();

调用disconnect()后，会触发手机会触发BluetoothGattCallback#onConnectionStateChange()的回调，回调断开连接信息，newState = BluetoothProfile.STATE\_DISCONNECTED。但接着马上调用close()，会终止BluetoothGattCallback#onConnectionStateChange()的回调，可以看情况将两个进行拆分调用，但一般都是成对出现。 例如： 需要在外设修改特征值触发BluetoothGattCallback#onCharacteristicChanged()时，断开连接。可以先在BluetoothGattCallback#onCharacteristicChanged()中调用disconnect(),并等调用BluetoothGattCallback#onConnectionStateChange()回调，返回断开连接信息后，再调用close()对Gatt资源进行关闭。

当和外设进行BLE通信时，如出现任何意外情况，马上调用断开连接操作

# C．当手机作为外围设备时(启动GATT Service、发送广播)

Note: 自Android 5.0 +后，手机可以作为外设广播数据。

# AdvertiseSettings(AdvertiseSettings.Builder)

位置: android.bluetooth.le.AdvertiseSettings

说明：广播设置。该类使用建造者模式构建，提供了一系列配置广播的方法。

主要方法：

## 12.1 setAdvertiseMode(int advertiseMode)

说明：设置广播模式，以控制广播的功率和延迟。可选值有：

[ADVERTISE\_MODE\_LOW\_POWER](file:///D:\android-sdk\docs\reference\android\bluetooth\le\AdvertiseSettings.html#ADVERTISE_MODE_LOW_POWER) ： 在低功耗模式下执行蓝牙LE广播。

[ADVERTISE\_MODE\_LOW\_LATENCY](file:///D:\android-sdk\docs\reference\android\bluetooth\le\AdvertiseSettings.html#ADVERTISE_MODE_LOW_LATENCY)：在低延迟、高功率模式下执行蓝牙LE广播。

[ADVERTISE\_MODE\_BALANCED](file:///D:\android-sdk\docs\reference\android\bluetooth\le\AdvertiseSettings.html#ADVERTISE_MODE_BALANCED)： 在均衡电源模式下执行蓝牙LE广播。

## 12.2 setTxPowerLevel(int txPowerLevel)

说明：设置发射功率级别。可选值有：

ADVERTISE\_TX\_POWER\_ULTRA\_LOW：使用最低传输功率级别进行广播

ADVERTISE\_TX\_POWER\_LOW： 使用低功率级别进行广播

ADVERTISE\_TX\_POWER\_MEDIUM： 使用中等功率级别进行广播

ADVERTISE\_TX\_POWER\_HIGH： 使用高功率级别进行广播

## 12.3 setTimeout(int timeoutMillis)

说明：设置广播的持续时间，单位为毫秒。最多为180000毫秒，当设置为0时则无时间限制，持续广播，除非调用BluetoothLeAdvertiser # stopAdvertising()。

PS：不得超过180000ms, 值为0将禁用时间限制。

## 12.4 setConnectable(boolean connectable)

说明：设置该广播是否可以被连接。

# AdvertiseData(AdvertiseData.Builder)

位置：android.bluetooth.le.AdvertiseData

说明：该类封装了广播的具体内容。它也使用建造者模式，则AdvertiseData.Builder类构建。

主要方法：

## 13.1 setIncludeDeviceName(boolean includeDeviceName)

说明：设置广播包中是否包含蓝牙的名称 。

## 13.2 setIncludeTxPowerLevel(boolean includeTxPowerLevel)

说明：设置广播包中是否包含蓝牙的发射功率。

## 13.3 addServiceUuid(ParcelUuid serviceUuid)

说明：添加特定的UUID在该广播包中。

## 13.4 addServiceData(ParcelUuid serviceDataUuid, byte[] serviceData)

说明：添加特定的UUID和数据在广播包中。

## 13.5 addManufacturerData(int manufacturerId, byte[] manufacturerSpecificData)

说明：添加特定的厂商ID和其数据在广播包中。

# BluetoothLeAdvertiser

位置：android.bluetooth.le.BluetoothLeAdvertiser

说明：通过这个类可以执行BLE广播的操作。比如开始广播和停止广播。一个Advertiser最多可以广播31个字节的用AdvertiseData表示的广播数据。

主要方法：

## 14.1 startAdvertising(AdvertiseSettings settings, AdvertiseData advertiseData, AdvertiseCallback callback)

说明：开始BLE广播。如果操作成功，那么构造的advertiseData 将会被立即广播出去，并且操作的状态码将会通过 AdvertiseCallback 回调类的回调方法返回。

## 14.2 startAdvertising(AdvertiseSettings settings, AdvertiseData advertiseData, AdvertiseData scanResponse, AdvertiseCallback callback)

说明：开始BLE广播。

## 14.3 stopAdvertising(AdvertiseCallback callback)

说明：结束BLE广播。这里的callback必须和开始广播的回调是同一个。

# AdvertiseCallback

位置：android.bluetooth.le.AdvertiseCallback

说明：蓝牙BLE广播的回调类，可用于传递广播的操作状态。比如开启广播和结束广播的操作码。

主要方法：

## 15.1 onStartFailure(int errorCode)

说明：在广播无法启动的时候会回调此方法。

## 15.2 onStartSuccess (AdvertiseSettings settingsInEffect)

说明：在广播启动成功的时候会回调此方法。其参数代表该广播的实际配置，可能跟请求的不同。

# BluetoothGattServer

位置：android.bluetooth.BluetoothGattServer

说明：代表蓝牙GATT的Server端。可在该Server端添加相关的GATT Service 和 GATT Characteristic，让其它中心设备读取。

主要方法：

## 16.1 addService(BluetoothGattService service)

说明：添加一条Service到该服务端的Service 列表。如果在调用该方法时本地设备已经公开了服务，则会向所有客户端发送服务更新通知。

## 16.2 cancelConnection(BluetoothDevice device)

说明：断开一个已经建立的连接，或者取消当前正在进行的连接尝试。

## 16.3 clearServices()

说明：从提供的服务列表中删除所有服务。

## 16.4 close()

说明：关闭当前GATT 服务端实例。

## 16.5 connect(BluetoothDevice device, boolean autoConnect)

说明：启动一个支持蓝牙GATT设备的连接。连接可能无法立即建立，但是当远程设备可用时将完成连接。当通过此方法导致连接状态更改时，将会调用

BluetoothGattServerCallback # void onConnectionStateChange (BluetoothDevice device, int status, int newState) 方法。

## 16.6 getConnectedDevices()

说明：已废弃，获取连接的设备。使用BluetoothManager # getConnectedDevices (int profile) 替代，参数传GATT。

## 16.7 getService(UUID uuid)

说明：根据指定的UUID从该服务列表中返回相应的Service。

## 16.8 getServices()

说明：返回此设备提供的GATT Servcie列表。

## 16.9 removeService(BluetoothGattService service)

说明：从提供的GATT Service列表中移除一个Service。

## 16.10 notifyCharacteristicChanged(BluetoothDevice device, BluetoothGattCharacteristic characteristic, boolean confirm)

说明：发送一条GATT Characteristic 已更新的通知。

## 16.11 sendResponse(BluetoothDevice device, int requestId, int status, int offset, byte[] value)

说明：将对读取或写入请求的响应发送到远程设备。当下列这些回调方法接收到读/写请求时，必须调用此函数：

BluetoothGattServerCallback # onCharacteristicReadRequest(BluetoothDevice, int, int, BluetoothGattCharacteristic)

BluetoothGattServerCallback #onCharacteristicWriteRequest(BluetoothDevice, int, BluetoothGattCharacteristic, boolean, boolean, int, byte[])

BluetoothGattServerCallback #onDescriptorReadRequest(BluetoothDevice, int, int, BluetoothGattDescriptor)

BluetoothGattServerCallback #onDescriptorWriteRequest(BluetoothDevice, int, BluetoothGattDescriptor, boolean, boolean, int, byte[])

# BluetoothGattServerCallback

位置：android.bluetooth.BluetoothGattServerCallback

说明：服务端GATT通信的Callback。

主要方法:

## 17.1 onCharacteristicReadRequest(BluetoothDevice device, int requestId, int offset, BluetoothGattCharacteristic characteristic)

说明：一个远程的客户端请求读取本地特征。

## 17.2 onCharacteristicWriteRequest(BluetoothDevice device, int requestId, BluetoothGattCharacteristic characteristic, boolean preparedWrite, boolean responseNeeded, int offset, byte[] value)

说明：一个远程的客户端请求写入本地特征。

## 17.3 onConnectionStateChange(BluetoothDevice device, int status, int newState)

说明：当连接或断开远程设备时会回调此方法。

## 17.4 onDescriptorReadRequest(BluetoothDevice device, int requestId, int offset, BluetoothGattDescriptor descriptor)

说明：一个远程的客户端请求读取本地的描述符。

## 17.5 onDescriptorWriteRequest(BluetoothDevice device, int requestId, BluetoothGattDescriptor descriptor, boolean preparedWrite, boolean responseNeeded, int offset, byte[] value)

说明：一个远程的客户端请求写入本地的描述符。

## 17.6 onExecuteWrite(BluetoothDevice device, int requestId, boolean execute)

说明：标识写入的状态。

## 17.7 onMtuChanged(BluetoothDevice device, int mtu)

说明：若远程请求更改MTU的size,且该MTU的size已更改完毕时调用此方法。第二个参数代表已更新的新的MTU的size。

## 17.8 onNotificationSent(BluetoothDevice device, int status)

说明：当一个通知或指示已经发送给远程设备将会回调此方法。注意，如果要发送多个通知给远程设备，应该在该回调中收到status为GATT\_SUCCESS时再发送其它通知。

## 17.9 onServiceAdded(int status, BluetoothGattService service)

说明：当一个本地服务已经被成功添加时回调此方法。

# GATT服务端通信示例Code

## 设置BLE广播

//自定义UUID

public static UUID UUID\_SERVICE = UUID.fromString("0000fff7-0000-1000-8000-00805f9b34fb");

//初始化广播设置

mAdvertiseSettings = new AdvertiseSettings.Builder()

//设置广播模式，以控制广播的功率和延迟。

.setAdvertiseMode(AdvertiseSettings.ADVERTISE\_MODE\_LOW\_POWER)

//发射功率级别

.setTxPowerLevel(AdvertiseSettings.ADVERTISE\_TX\_POWER\_HIGH)

//不得超过180000毫秒。值为0将禁用时间限制。

.setTimeout(3000)

//设置是否可以连接

.setConnectable(false)

.build();

## 初始化广播包和扫描响应包

//初始化广播包

mAdvertiseData = new AdvertiseData.Builder()

//设置广播设备名称

.setIncludeDeviceName(true)

//设置发射功率级别

.setIncludeDeviceName(true)

.build();

//初始化扫描响应包

mScanResponseData = new AdvertiseData.Builder()

//隐藏广播设备名称

.setIncludeDeviceName(false)

//隐藏发射功率级别

.setIncludeDeviceName(false)

//设置广播的服务UUID

.addServiceUuid(new ParcelUuid(UUID\_SERVICE))

//设置厂商数据

.addManufacturerData(0x11,hexStrToByte(mData))

.build();

## 开始广播

//获取BLE广播的操作对象。

//如果蓝牙关闭或此设备不支持蓝牙LE广播，则返回null。

mBluetoothLeAdvertiser = mBluetoothAdapter.getBluetoothLeAdvertiser();

//mBluetoothLeAdvertiser不为空，且蓝牙已开启

if(mBluetoothAdapter.isEnabled()){

if (mBluetoothLeAdvertiser != null){

//开启广播

mBluetoothLeAdvertiser.startAdvertising(mAdvertiseSettings,

mAdvertiseData, mScanResponseData, mAdvertiseCallback);

}else {

Log.d("xp.chen","该手机不支持ble广播");

}

}else{

Log.d("xp.chen","手机蓝牙未开启");

}

private class daqiAdvertiseCallback extends AdvertiseCallback {

//开启广播成功回调

@Override

public void onStartSuccess(AdvertiseSettings settingsInEffect){

super.onStartSuccess(settingsInEffect);

Log.d("daqi","开启服务成功");

}

//无法启动广播回调。

@Override

public void onStartFailure(int errorCode) {

super.onStartFailure(errorCode);

Log.d("daqi","开启服务失败，失败码 = " + errorCode);

}

}

## 初始化GATT Service和Characteristic

// 初始化GATT Service

BluetoothGattService service = new BluetoothGattService(UUID\_SERVICE,

BluetoothGattService.SERVICE\_TYPE\_PRIMARY);

//初始化特征值

mGattCharacteristic = new BluetoothGattCharacteristic(UUID\_CHARACTERISTIC,

BluetoothGattCharacteristic.PROPERTY\_WRITE|

BluetoothGattCharacteristic.PROPERTY\_NOTIFY|

BluetoothGattCharacteristic.PROPERTY\_READ,

BluetoothGattCharacteristic.PERMISSION\_WRITE|

BluetoothGattCharacteristic.PERMISSION\_READ);

//初始化描述

mGattDescriptor = new BluetoothGattDescriptor(UUID\_DESCRIPTOR,BluetoothGattDescriptor.PERMISSION\_WRITE);

//Service添加特征值

mGattService.addCharacteristic(mGattCharacteristic);

mGattService.addCharacteristic(mGattReadCharacteristic);

//特征值添加描述

mGattCharacteristic.addDescriptor(mGattDescriptor);

## 启动GATT Service

通过蓝牙管理器mBluetoothManager获GATT Server，用来添加GATT Service。添加完Gatt Service后，外部中心设备连接手机时，将能获取到对应的GATT Service 和 获取对应的数据

//初始化GattServer回调

mBluetoothGattServerCallback = new daqiBluetoothGattServerCallback();

if (mBluetoothManager != null)

mBluetoothGattServer = mBluetoothManager.openGattServer(this, mBluetoothGattServerCallback);

boolean result = mBluetoothGattServer.addService(mGattService);

if (result){

Toast.makeText(daqiActivity.this,"添加服务成功",Toast.LENGTH\_SHORT).show();

}else {

Toast.makeText(daqiActivity.this,"添加服务失败",Toast.LENGTH\_SHORT).show();

}

private class daqiBluetoothGattServerCallback extends BluetoothGattServerCallback{

//设备连接/断开连接回调

@Override

public void onConnectionStateChange(BluetoothDevice device, int status, int newState) {

super.onConnectionStateChange(device, status, newState);

}

//添加本地服务回调

@Override

public void onServiceAdded(int status, BluetoothGattService service) {

super.onServiceAdded(status, service);

}

//特征值读取回调

@Override

public void onCharacteristicReadRequest(BluetoothDevice device, int requestId, int offset, BluetoothGattCharacteristic characteristic) {

super.onCharacteristicReadRequest(device, requestId, offset, characteristic);

}

//特征值写入回调

@Override

public void onCharacteristicWriteRequest(BluetoothDevice device, int requestId, BluetoothGattCharacteristic characteristic, boolean preparedWrite, boolean responseNeeded, int offset, byte[] value) {

super.onCharacteristicWriteRequest(device, requestId, characteristic, preparedWrite, responseNeeded, offset, value);

}

//描述读取回调

@Override

public void onDescriptorReadRequest(BluetoothDevice device, int requestId, int offset, BluetoothGattDescriptor descriptor) {

super.onDescriptorReadRequest(device, requestId, offset, descriptor);

}

//描述写入回调

@Override

public void onDescriptorWriteRequest(BluetoothDevice device, int requestId, BluetoothGattDescriptor descriptor, boolean preparedWrite, boolean responseNeeded, int offset, byte[] value) {

super.onDescriptorWriteRequest(device, requestId, descriptor, preparedWrite, responseNeeded, offset, value);

}

}

<END>