Faytech BluetoothKit---Android Bluetooth Framework

This library allows for easy access to Bluetooth device scan and connection, support customizing scan policy

and solved many android bluetooth inherent compatibility

Requirements

- minSdkVersion should be not less than 18

Usage

1. In Android Studio's build.gradle, add a line in dependencies:

```
implementation 'com.inuker.bluetooth:library:1.4.0'
```

If it is Eclipse, you can import bluetoothkit.jar and add the following to AndroidManifest.xml:

- Permission in AndroidManifest.xml

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>

<uses-feature
    android:name="android.hardware.bluetooth_le"
    android:required="true" />
```

Usage

1、If you are building with Gradle, simply add the following line to the `dependencies` section of your `build.gradle` file:

```
implementation 'com.inuker.bluetooth:library:1.1.4'
```

2、Create a BluetoothClient as below:

```
BluetoothClient mClient = new BluetoothClient(context);
```

Scan Device

This library support both Bluetooth LE device scan and Classic device scan, you could customize the scan policy as below:

```
SearchRequest request = new SearchRequest.Builder()
```

```
        .searchBluetoothLeDevice(3000, 3)   // scan Bluetooth LE device for
3000ms, 3 times
        .searchBluetoothClassicDevice(5000) // then scan Bluetooth Classic device
for 5000ms, 1 time
        .searchBluetoothLeDevice(2000)      // at last scan Bluetooth LE device
for 2000ms
        .build();

mClient.search(request, new SearchResponse() {
    @Override
    public void onSearchStarted() {

    }

    @Override
    public void onDeviceFounded(SearchResult device) {

    }

    @Override
    public void onSearchStopped() {

    }

    @Override
    public void onSearchCanceled() {

    }
});
```

You could stop the whole scan by just one line:

```
mClient.stopSearch();
```

Bluetooth LE Connection

• Connect

BleGattProfile contains all service and characteristic uuid.

Bluetooth switch

Turn Bluetooth on and off:

```
mClient.openBluetooth();
mClient.closeBluetooth();
```

Judge whether Bluetooth is on:

```
mClient.isBluetoothOpened();
```

It takes a while to turn Bluetooth on or off. You can register a callback to monitor the status. If the callback parameter is true, it means Bluetooth is on,
and false means Bluetooth is off

```
mClient.registerBluetoothStateListener(mBluetoothStateListener);

private final BluetoothStateListener mBluetoothStateListener = new
BluetoothStateListener() {
@Override
public void onBluetoothStateChanged(boolean openOrClosed) {

}

};

mClient.unregisterBluetoothStateListener(mBluetoothStateListener);
```

Device pairing

Monitor device pairing status changes

```
private final BluetoothBondListener mBluetoothBondListener = new
BluetoothBondListener() {
@Override
public void onBondStateChanged(String mac, int bondState) {
// bondState = Constants.BOND_NONE, BOND_BONDING, BOND_BONDED
}
};

mClient.registerBluetoothBondListener(mBluetoothBondListener);
mClient.unregisterBluetoothBondListener(mBluetoothBondListener);
```

Beacon analysis

Customized data of the device can be carried in the broadcast for device identification, data broadcasting, event notification, etc.,
so that the mobile phone can obtain the data pushed by the device without connecting to the device.

The scanned beacon data is byte[]. In the scanRecord of SearchResult, the Beacon object is generated as follows:

```
Beacon beacon = new Beacon(device.scanRecord);
```

The Beacon data structure is as follows:

```
public class Beacon {

public byte[] mBytes;

public List<BeaconItem> mItems;
}
```

BeaconItem is distinguished by type,

```
public class BeaconItem {
/**
* Length declared in broadcast
*/
public int len;

/**
* Type declared in broadcast
*/
public int type;

/**
* Data part in broadcast
*/
public byte[] bytes;
}
```

Then, according to the custom protocol, parse the bytes in the corresponding BeaconItem.
First, create a BeaconParser, pass in the corresponding BeaconItem,
and then read data continuously according to the protocol.
If a field in the protocol occupies 1 byte, call readByte.
If it occupies two bytes, call readShort. If you want to get a bit of a byte, call getBit.
Note that the parser
Every time the data is read, the pointer will move backward accordingly. You can call setPosition to set the current pointer position.

```
BeaconItem beaconItem; // Set to the corresponding item in beacon
BeaconParser beaconParser = new BeaconParser(beaconItem);
int firstByte = beaconParser.readByte(); // Read the first byte
int secondByte = beaconParser.readByte(); // Read the second byte
int productId = beaconParser.readShort(); // Read the third and fourth bytes
boolean bit1 = beaconParser.getBit(firstByte, 0); // Get the first bit of the
first byte
boolean bit2 = beaconParser.getBit(firstByte, 1); // Get the second bit of the
first byte
beaconParser.setPosition(0); // Set the reading starting point to the first byte
```

BLE device communication

● Connection

The connection process includes normal connection (connectGatt) and service discovery (discoverServices). When a callback is received here, it means that the service discovery is completed. The callback parameter BleGattProfile includes the uuids of all services and characteristics. The returned code indicates the operation status, including success, failure or timeout, etc. All constants are in the Constants class.

```
mClient.connect(MAC, new BleConnectResponse() {
    @Override
    public void onResponse(int code, BleGattProfile profile) {
        if (code == REQUEST_SUCCESS) {

        }
    }
});

You can configure the connection parameters as follows,
```

BleConnectOptions options = new BleConnectOptions.Builder()
.setConnectRetry(3)   // Retry 3 times if connection fails
.setConnectTimeout(30000)   // Connection timeout 30s
.setServiceDiscoverRetry(3) // Retry 3 times if service discovery fails
.setServiceDiscoverTimeout(20000) // Service discovery timeout 20s
.build();

mClient.connect(MAC, options, new BleConnectResponse() {
@Override
public void onResponse(int code, BleGattProfile data) {

}
});

```
### **● Connection status**

If you want to monitor the Bluetooth connection status, you can register a
callback. There are only two states: connected and disconnected.
```

mClient.registerConnectStatusListener(MAC, mBleConnectStatusListener);

private final BleConnectStatusListener mBleConnectStatusListener = new
BleConnectStatusListener() {

```
@Override
public void onConnectStatusChanged(int status) {
    if (status == STATUS_CONNECTED) {

    } else if (status == STATUS_DISCONNECTED) {

    }
}
```

};

mClient.unregisterConnectStatusListener(MAC, mBleConnectStatusListener);

```
You can also actively obtain the connection status:
```

```
int status = mClient.getConnectStatus(MAC);
// Constants.STATUS_UNKNOWN
// Constants.STATUS_DEVICE_CONNECTED
// Constants.STATUS_DEVICE_CONNECTING
// Constants.STATUS_DEVICE_DISCONNECTING
// Constants.STATUS_DEVICE_DISCONNECTED
```

```
### **• Disconnect**
```Java
mClient.disconnect(MAC);
```

• Read Characteristic

```
mClient.read(MAC, serviceUUID, characterUUID, new BleReadResponse() {
    @Override
    public void onResponse(int code, byte[] data) {
        if (code == REQUEST_SUCCESS) {

        }
    }
});
```

• Write Characteristic

The data to write should be no more than 20 bytes.

```
mClient.write(MAC, serviceUUID, characterUUID, bytes, new BleWriteResponse() {
    @Override
    public void onResponse(int code) {
        if (code == REQUEST_SUCCESS) {

        }
    }
});

// with WRITE_TYPE_NO_RESPONSE
mClient.writeNoRsp(MAC, serviceUUID, characterUUID, bytes, new BleWriteResponse()
{
    @Override
    public void onResponse(int code) {
        if (code == REQUEST_SUCCESS) {

        }
    }
});
```
```

• Read Descriptor

```
mClient.readDescriptor(MAC, serviceUUID, characterUUID, descriptorUUID, new
BleReadResponse() {
@Override
public void onResponse(int code, byte[] data) {

}
});
```

• Write Descriptor

```
mClient.writeDescriptor(MAC, serviceUUID, characterUUID, descriptorUUID, bytes,
new BleWriteResponse() {
@Override
public void onResponse(int code) {

}
});
```

• Open Notify

There are two callbacks here, onNotify is for receiving notifications.

```
mClient.notify(MAC, serviceUUID, characterUUID, new BleNotifyResponse() {
    @Override
    public void onNotify(UUID service, UUID character, byte[] value) {

    }

    @Override
    public void onResponse(int code) {
        if (code == REQUEST_SUCCESS) {

        }
    }
});
```

• Close Notify

```
mClient.unnotify(MAC, serviceUUID, characterUUID, new BleUnnotifyResponse() {
    @Override
    public void onResponse(int code) {
        if (code == REQUEST_SUCCESS) {

        }
    }
});
```

• open Indicate

Similar to Notify,

```
mClient.indicate(MAC, serviceUUID, characterUUID, new BleNotifyResponse() {
    @Override
    public void onNotify(UUID service, UUID character, byte[] value) {

    }

    @Override
    public void onResponse(int code) {
        if (code == REQUEST_SUCCESS) {

        }
    }
});
```

• Close Indicate

```
mClient.unindicate(MAC, serviceUUID, characterUUID, new BleUnnotifyResponse() {
    @Override
    public void onResponse(int code) {
        if (code == REQUEST_SUCCESS) {

        }
    }
});
```

• Read Rssi

```
mClient.readRssi(MAC, new BleReadRssiResponse() {
    @Override
    public void onResponse(int code, Integer rssi) {
        if (code == REQUEST_SUCCESS) {

        }
    }
});
```

```
### **• Clear the request queue**

If the device does not have time to process the requests sent to the device,
these requests will be saved in the queue. If you need to clear these requests in
some scenarios, you can call
```

mClient.clearRequest(MAC, clearType);

// Constants.REQUEST_READ, all read requests
// Constants.REQUEST_WRITE, all write requests
// Constants.REQUEST_NOTIFY, all notification-related requests
// Constants.REQUEST_RSSI, all read signal strength requests

```
clearType indicates the request type to be cleared. If you want to clear multiple
requests, you can OR multiple types. If you want to clear all requests, pass in
0.

### **• Refresh cache**

```Java
mClient.refreshCache(MAC);
```

Author

- Email: hogan.sun@faytech.com
```