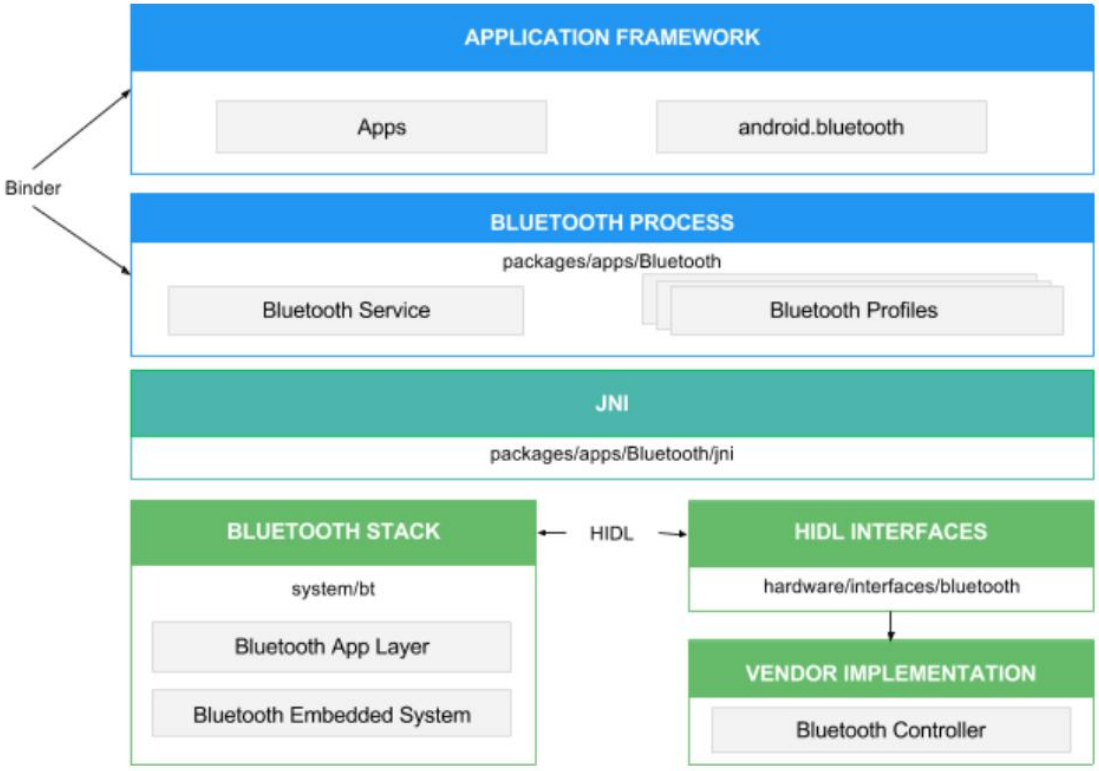
# 蓝牙框架原理以及通讯数据交互流程

## 一、背景

众所周知，蓝牙是 Android 系统里面一个非常重要的模块，也是我们日常生活应用的非常多的一个功能，所以，蓝牙模块应用的稳定性直接影响到一个系统的满意度的高度。然而由于蓝牙模块的开发涉及面也是比较广泛，问题也是比较多的一个模块。为了能快速认知此蓝牙模块并快速进入到蓝牙模块的开发，本文就针对个人对蓝牙模块认识，做一个整理。第一部分对蓝牙框架的介绍，第二部分是蓝牙通讯数据交互流程举例。

## 二、Android 蓝牙框架

蓝牙应用通过 Binder 与蓝牙进程进行通信。蓝牙进程使用 Java 原生接口（JNI）与蓝牙堆栈通信，并向开发者提供对各种蓝牙配置文件的访问权限。下图显示了蓝牙堆栈的常规结构：




### 1、应用框架

处于应用框架级别的是应用代码，它使用 android.bluetooth API 与蓝牙硬件进行交互。此代码在内部通过 Binder IPC 机制调用蓝牙进程。结合我们现在目前的项目，这里面包含了两部分，其中一部分如我们上层应用的 YX_Bluetooth，一部分为我们的 Frameworks 中间件，YX_Bluetooth 通过 android.bluetooth API 调用到 Frameworks 中间件（如下图 1、

图 2），而 Frameworks 中间件又通过 Binder IPC 进程间通讯调用到系统应用 Bluetooth process（如下图 3）：

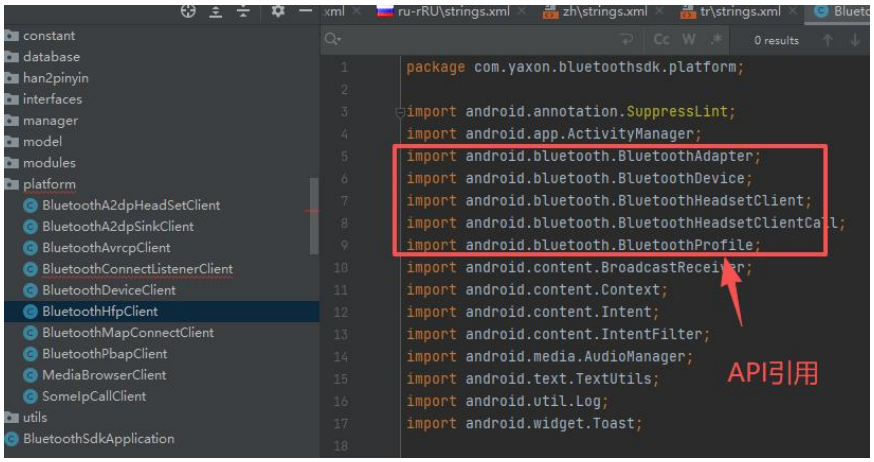



图 1.




图 2.

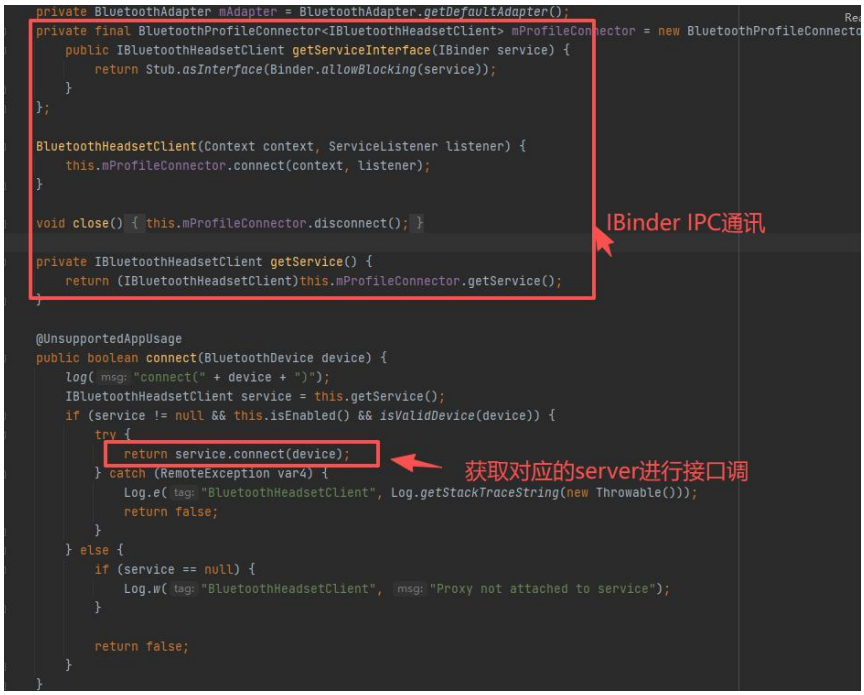


图 3.

## 2、系统蓝牙应用

系统蓝牙应用，位于系统源码 packages/apps/Bluetooth/路径下，被打包为 Android 应用，并在 Android 框架层实现蓝牙配置文件。此应用起到一个承上启下的作用，承上，实现相关服务，并提供 IBinder 相关接口供 Frameworks 调用（如下图 4），下着通过 JNI 调用蓝牙协议栈（如下图 5、图 6）。




图 4.

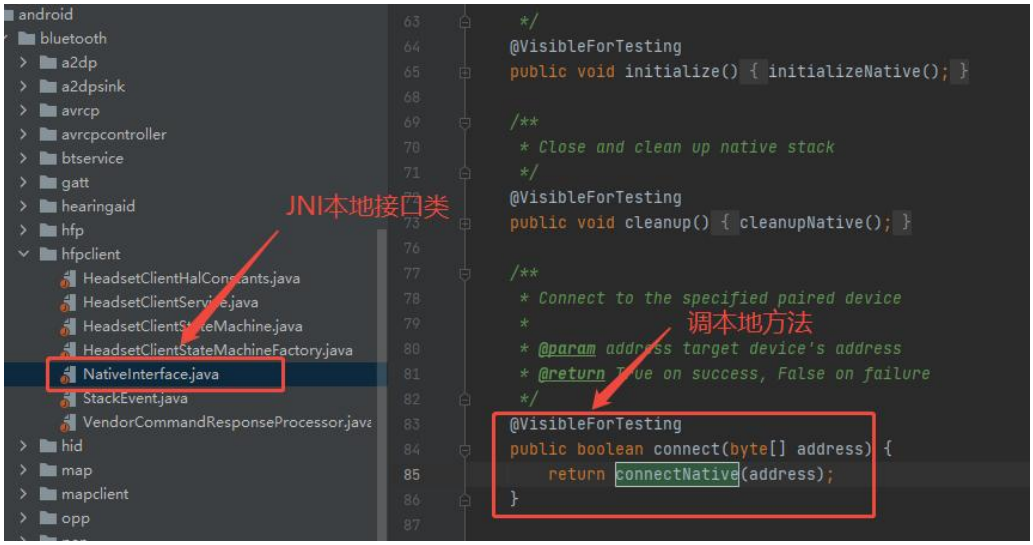


图 5.




图 6.

### 3、蓝牙协议栈 JNI 接口

与蓝牙协议栈相关联的 JNI 接口代码位于 packages/apps/Bluetooth/jni 中（如下图 7）。当程序启动时，通过 JAVA 初始化加载 JNI 程序，JNI 在初始化过程中通过注册 Java 类与 Java 程序进行方法绑定，如下图 8、图 9，当发生特定蓝牙操作时（例如发现设备时），JNI 代码会调用蓝牙协议栈。

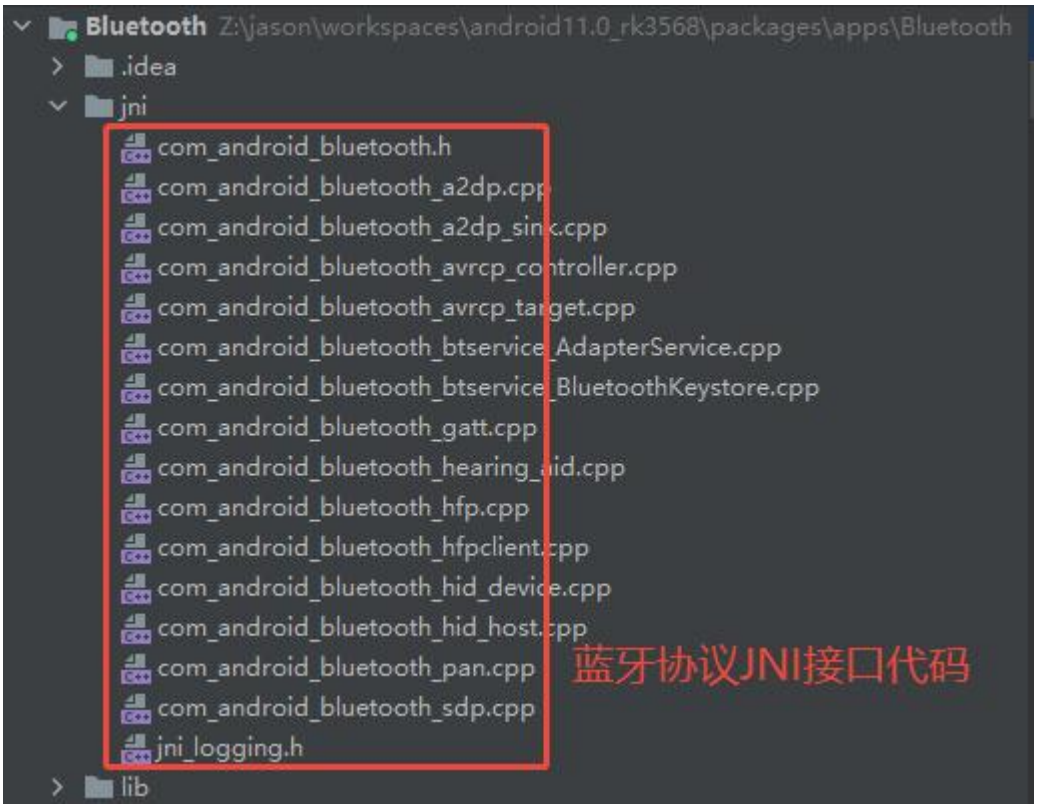



图 7.


/*
* JNI Initialization
*/
jint JNI_OnLoad(JavaVM* jvm, void* reserved) {
JNIEnv* e;
int status;
ALOGV("Bluetooth Adapter Service : loading JNI\n");
// Check JNI version
if (jvm->GetEnv((void**)&e, JNI_VERSION_1_6)) {
ALOGE("JNI version mismatch error");
return JNI_ERR;
}
status = android::register_com_android_bluetooth_btservice_AdapterService(e);
if (status < 0) {
ALOGE("jni adapter service registration failure, status: %d", status);
return JNI_ERR;
}
status =
android::register_com_android_bluetooth_btservice_BluetoothKeystore(e);
if (status < 0) {
ALOGE("jni BluetoothKeystore registration failure: %d", status);
return JNI_ERR;
}
status = android::register_com_android_bluetooth_hfp(e);
if (status < 0) {
ALOGE("jni hfp registration failure, status: %d", status);
return JNI_ERR;
}
status = android::register_com_android_bluetooth_hfpclient(e);
JNI初始化加载入口
注册JAVA类绑定

图 8


static JNINativeMethod sMethods[] = {
{"classInitNative", "()V", (void*)classInitNative},
{"initializeNative", "()Z", (void*)initializeNative},
{"cleanupNative", "()V", (void*)cleanupNative},
{"connectNative", "([B)Z", (void*)connectNative},
{"disconnectNative", "([B)Z", (void*)disconnectNative},
{"connectAudioNative", "([B)Z", (void*)connectAudioNative},
{"disconnectAudioNative", "([B)Z", (void*)disconnectAudioNative},
{"startVoiceRecognitionNative", "([B)Z",
(void*)startVoiceRecognitionNative},
{"stopVoiceRecognitionNative", "([B)Z", (void*)stopVoiceRecognitionNative},
{"setVolumeNative", "([BII)Z", (void*)setVolumeNative},
{"dialNative", "([BLjava/lang/String;)Z", (void*)dialNative},
{"dialMemoryNative", "([BI)Z", (void*)dialMemoryNative},
{"handleCallActionNative", "([BII)Z", (void*)handleCallActionNative},
{"queryCurrentCallsNative", "([B)Z", (void*)queryCurrentCallsNative},
{"queryCurrentOperatorNameNative", "([B)Z",
(void*)queryCurrentOperatorNameNative},
{"retrieveSubscriberInfoNative", "([B)Z",
(void*)retrieveSubscriberInfoNative},
{"sendDtmfNative", "([BB)Z", (void*)sendDtmfNative},
{"requestLastVoiceTagNumberNative", "([B)Z",
(void*)requestLastVoiceTagNumberNative},
{"sendATCmdNative", "([BIILjava/lang/String;)Z", (void*)sendATCmdNative},
};
int register_com_android_bluetooth_hfpclient(JNIEnv* env) {
return jniRegisterNativeMethods(
env, "com/android/bluetooth/hfpclient/NativeInterface",
sMethods, NELEM(sMethods));
}
与Java Native方法一一对应
对应Java类路径

图 9.

**4、蓝牙协议栈**

AOSP 中提供了默认蓝牙堆栈（位于 system/bt 中）。该堆栈会实现常规蓝牙 HAL，并通过扩展和更改配置对其进行自定义。

**5、供应商实现**

供应商设备使用 HAL 接口定义语言 (HIDL) 与蓝牙堆栈交互。

**6、HIDL**

HIDL 定义了蓝牙堆栈和供应商实现之间的接口。如需生成蓝牙 HIDL 文件，请将蓝牙接口文件传入 HIDL 生成工具中。接口文件位于 hardware/interfaces/bluetooth 中。

**7、蓝牙堆栈开发**

Android 蓝牙堆栈是一个完全限定的蓝牙堆栈。核心蓝牙堆栈位于 system/bt 中。

## 三、通讯数据交互流程

通过以上蓝牙框架层级接受，对系统蓝牙框架有了一定的了解，也明白了各层级之间是如何建立连接的。 下面再详细说明我们是如何接受底层数据，即接受协议栈数据的。上面举例了如果调用蓝牙连接，我们再以如何接受连接状态变化举例：

1、当调用蓝牙连接之后，底层会将连接数据通过 JNI 向上反馈，如下图 10

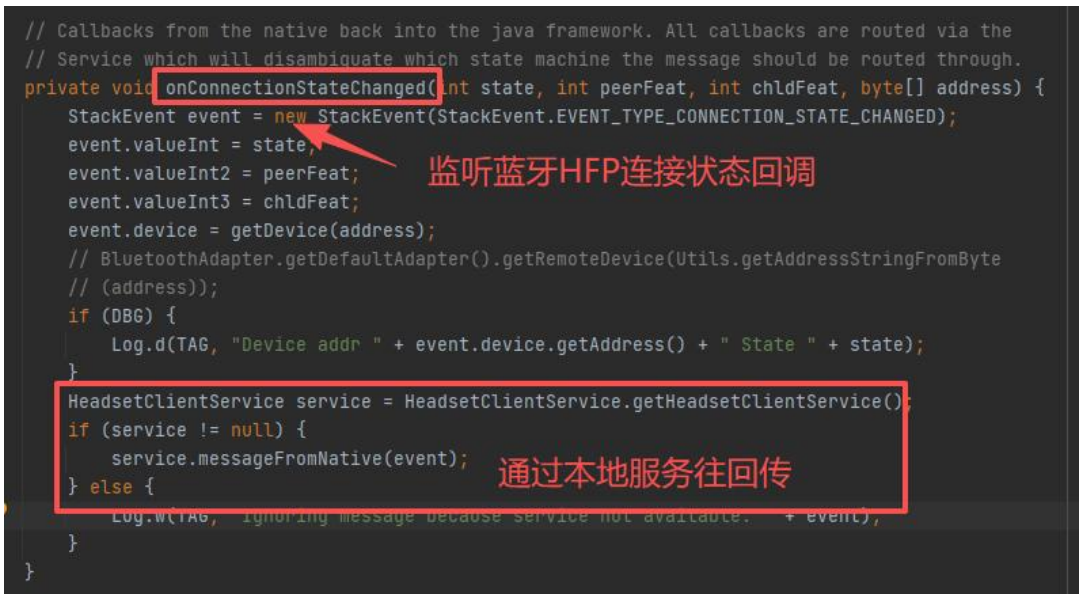



图 10.

2、从上图可以知，数据已经传回到了本地服务，而 HeadsetClientService 此服务是类似一个中转服务器，他是一个长驻 Server，他接受到数据后，会把数据直接转发到对应的业务类去，如下图 11.

```
// Handle messages from native (JNI) to java
public void messageFromNative(StackEvent stackEvent) {
    HeadsetClientStateMachine sm = getStateMachine(stackEvent.device);
    if (sm == null) {
        Log.w(TAG, "No SM found for event " + stackEvent);
    }

    sm.sendMessage(StackEvent.STACK_EVENT, stackEvent);
}
```

图 11.

3、HeadsetClientStateMachine 属于业务类，蓝牙电话基本上所有的需要的调整都会在此类中修改，HeadsetClientService 会获取到当前的连接状态对象，根据 Event.type 类型进行分发，如下图 12、图 13、图 14

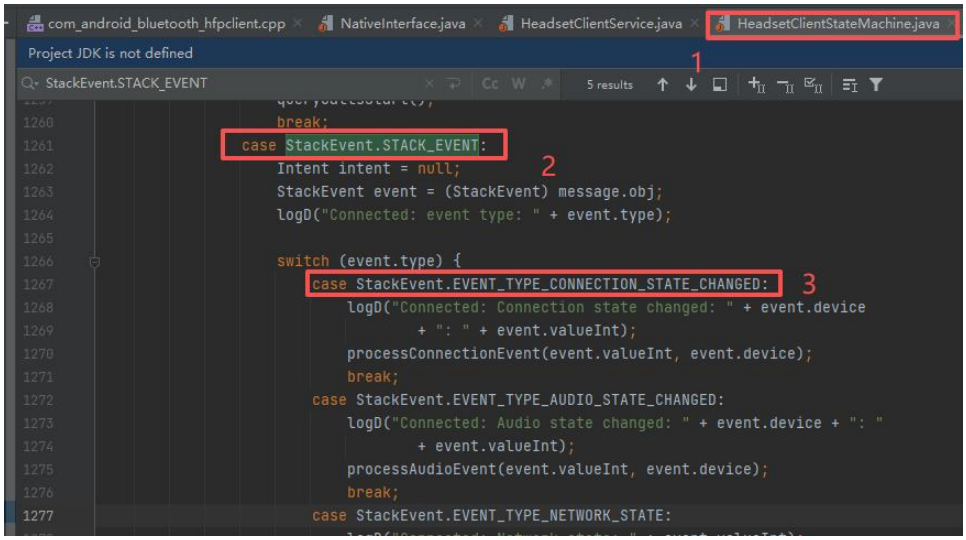



图 12.




图 13.

```
// This method does not check for error condition (newState == prevState)
private void broadcastConnectionState(BluetoothDevice device, int newState, int prevState) {
    logD("Connection state " + device + ": " + prevState + "->" + newState);
    /*
     * Notifying the connection state change of the profile before sending
     * the intent for connection state change, as it was causing a race
     * condition, with the UI not being updated with the correct connection
     * state.
     */
    Intent intent = new Intent(BluetoothHeadsetClient.ACTION_CONNECTION_STATE_CHANGED);
    intent.putExtra(BluetoothProfile.EXTRA_PREVIOUS_STATE, prevState);
    intent.putExtra(BluetoothProfile.EXTRA_STATE, newState);
    intent.putExtra(BluetoothDevice.EXTRA_DEVICE, device);

    // add feature extras when connected
    if (newState == BluetoothProfile.STATE_CONNECTED) {...}
    mService.sendBroadcast(intent, ProfileService.BLUETOOTH_PERM);
}
```

6

最终将连接状态以广播形式发送

图 14.

4、最后，上层应用通过注册广播接收器进行结束即可，如下图 15、图 16.

```
public void init(Context context) {
    YXLog.i(TAG, msg: "init");
    mContext = context;
    registerContentObserver();
    mStatusReceiverReceiver = new ConnectStatusReceiver();
    IntentFilter filter = new IntentFilter();
    filter.addAction(BluetoothDevice.ACTION_PAIRING_REQUEST);
    filter.addAction(BluetoothAdapter.ACTION_CONNECTION_STATE_CHANGED);
    filter.addAction(BluetoothHeadsetClient.ACTION_CONNECTION_STATE_CHANGED);
    filter.addAction(BluetoothAdapter.ACTION_STATE_CHANGED);
    filter.addAction(BluetoothDevice.ACTION_BOND_STATE_CHANGED);
    filter.addAction(BluetoothA2dp.ACTION_CONNECTION_STATE_CHANGED);
    filter.addAction(BluetoothA2dpSink.ACTION_CONNECTION_STATE_CHANGED);
    filter.addAction(BluetoothAdapter.ACTION_DISCOVERY_STARTED);
```

图 15

```
private class ConnectStatusReceiver extends BroadcastReceiver {
    @SuppressLint("MissingPermission")
    @Override
    public void onReceive(Context context, Intent intent) {
        try {
            String action = intent.getAction();
            Log.i(TAG, msg: "ConnectStatusReceiver action = " + action);
            LinkedInfo linkedInfo = null;
            BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            if (device != null) {...}

            switch (action) {
                case BluetoothHeadsetClient.ACTION_CONNECTION_STATE_CHANGED:
                    setHfpConnectState(linkedInfo, intent);
                    break;
                case BluetoothAdapter.ACTION_STATE_CHANGED:
                    //监听蓝牙开关
```

图 16.

完.