|  |
| --- |
| **蓝牙开发文档** |

|  |  |  |
| --- | --- | --- |
| Date  日期 | Version  版本 | Comments  备注 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Function**  **职位** | **Name**  **姓名** | **Date**  **日期** | **Signature**  **签名** |
| **Written by**  **拟定** |  | 吴东海 | 2016-4-19 | 吴东海 |
| **Verified by**  审核 |  |  |  |  |
| **Verified by**  审核 |  |  |  |  |

所谓蓝牙（Bluetooth）技术，实际上是一种短距离无线通信技术，利用“蓝牙”技术，能够有效地简化掌上电脑、笔记本电脑和移动电话手机等移动通信终端设备之间的通信，也能够成功地简化以上这些设备与Internet之间的通信，从而使这些现代通信设备与因特网之间的数据传输变得更加迅速高效，为无线通信拓宽道路。说得通俗一点，就是蓝牙技术使得现代一些轻易携带的移动通信设备和电脑设备，不必借助电缆就能联网，并且能够实现无线上因特网，其实际应用范围还可以拓展到各种家电产品、消费电子产品和汽车等信息家电，组成一个巨大的无线通信网络。

关于蓝牙的名字的由来有个有趣的故事，“蓝牙”（Bluetooth）原是一位在10世纪统一丹麦的国王，他将当时的瑞典、芬兰与丹麦统一起来，因为他爱吃蓝莓，所以他的牙齿是蓝色的。用他的名字来命名这种新的技术标准，含有蓝牙将取消杂乱无章的众多线路，用无线连接来取代有线连接。

蓝牙目前暂时共有很多个版本： V1.0/1.1/1.2/2.0/2.1/3.0/4.0/4.2，我司使用的MTK平台，蓝牙的最高版本是4.0。通常蓝牙3.0以及以前的版本称为传统蓝牙，蓝牙4.0就是以前的传统蓝牙加上低功耗蓝牙（BLE）。顾名思义，低功耗蓝牙就是功耗很低，有的BLE设备甚至可以使用纽扣电池，但是低功耗蓝牙功能比较有限，我们熟悉的应用，比如收发文件、使用蓝牙打电话听歌，使用的还是传统蓝牙技术。

支持BTE技术的设备分为单模和双模，单模就是只支持BLE，不支持传统蓝牙技术，比如现在流行的手表、手环；双模就是既支持传统蓝牙又支持BLE，我们开发的手机属于双模设备。

整个蓝牙协议体系结构可分为底层硬件模块、中间协议层和高端应用层。我们维护的蓝牙代码属于应用层，本文的内容也是针对应用层的。



在MTK平台上，android L及以前平台，蓝牙模块使用的都是MTK自己开发的成果，但是在android M平台，由于BRODCOM（博通）公司公布了更多的蓝牙源码，所以，MTK的协议栈也采用了博通源码。

一般来说，本地蓝牙设备要与别的远端蓝牙设备进行操作，要经历搜索—〉配对—〉连接这几个步骤。

**几个常用蓝牙术语：**

**搜索：**一个蓝牙通过搜索操作去查询四周的蓝牙设备，并把搜索结果反馈用户，以便用户选择需要连接的设备来进行配对。

**配对：**在两个蓝牙设备间建立一种信任关系的过程。此过程中将交换链路密钥（在请求建立连接之前或在连接阶段），一般说来，一个蓝牙设备要是准备对另一个蓝牙设备进行操作，需要两个蓝牙设备完成配对操作。

**已配对设备：**已与其交换了链路密钥的蓝牙设备（在请求建立连接之前或在连接阶段中）。

**蓝牙地址：**用于识别每个蓝牙设备的 48 位地址。这在技术规格中通常被称为BD\_ADDR。如：00：11：22：33：44：55。蓝牙地址就像人的身份证号码一样，用于唯一标识一个蓝牙设备，世界上的正规蓝牙设备都有一个全球唯一的地址，这个地址由蓝牙联盟提供。

**蓝牙名字：**虽然标识一个蓝牙设备的是蓝牙地址，但是蓝牙地址是一串数字，较难记忆，所以有了蓝牙名字，它是为了便于用户操作蓝牙而产生的，它并不能标识一个蓝牙设备，只是一个最大长度248个字节、采用UTF8编码的字符串。

**配置文件（PROFILE）：**蓝牙有很多应用，PROFILE就是具体的一个蓝牙应用。我们的手机支持的PROFILE分为两种：android默认支持的PROFILE和MTK自己实现的PROFILE。由于我司的产品只支持android默认的PROFILE，

所以本文只对android默认支持的PROFILE做以简介。

**高级音频分发配置文件 (A2DP) ：**A2DP 配置文件描述了立体声质量音频如何从媒体源流化传送至汇点。配置文件定义了音频源和汇点两个角色。典型的使用如“随身听”类的媒体播放器。音频源可以是音乐播放器，音频汇点则是无线耳机。A2DP 定义了可在 ACL 信道上实现单声道或立体声高质量音频内容分发的协议和程序。A2DP分为SOURCE与SINK两个角色，当我们用手机听音乐时，手机是SOURCE，耳机是SINK。

**音频/视频远程控制配置文件 (AVRCP) ：**AVRCP 设计用于提供控制 TV、Hi-fi 设备等的标准接口。此配置文件用于许可单个远程控制设备（或其它设备）控制所有用户可以接入的 A/V 设备。AVRCP 定义了如何控制流媒体的特征。包括暂停、停止、启动重放、音量控制及其它类型的远程控制操作。AVRCP分为CONTROL和TARGET两个角色，当我们用蓝牙耳机遥控手机端的音乐播放时，手机是TARGET，耳机是CONTROL。

**对象推送配置文件（OPP）：**OPP描述了如何在蓝牙设备之间发送和接收文件，分为PULL和PUSH两种操作，蓝牙SPEC只要求实现PUSH操作，PULL可以不实现。根据PUSH操作区分，OPP分为OPS和OPC两种角色，当蓝牙设备间在收发文件时，发送方是OPC，接收方是OPS。

**免提配置文件 (HFP) ：**HFP 描述了网关设备如何用于供免提设备拨打和接听呼叫。典型配置如汽车使用手机作为网关设备。在车内，立体声系统用于电话音频，而车内安装的麦克风则用于通话时发送输出音频。HFP 还可用于个人计算机在家中或办公环境中作为手机扬声器的情况。 HFP分为AG和HF两个角色，当用蓝牙耳机听电话时，手机是AG，耳机是HF。

**耳机配置文件 (HSP)** ：HSP 描述了蓝牙耳机如何与计算机或其它蓝牙设备（如手机）通信。连接和配置好后，耳机可以作为远程设备的音频输入和输出接口。 HFP分为AG和HS，当用蓝牙耳机听电话时，手机是AG，耳机是HS。

**人机接口设备配置文件 (HID) ：**HID 配置文件定义了蓝牙HID（如键盘、指向设备、游戏设备及远程监视设备）使用的协议、程序及功能。通过HID，可以让蓝牙键盘等设备控制手机。

**电话本访问配置文**件**（PBAP）：**PBAP定义了访问手机电话本和通话记录的协议。PBAP分为SERVER和CLIENT两种角色，SERVER端提供电话本和通话记录的数据，CLIENT端是请求数据一端。我们的手机是SERVER端。

**个人局域网配置文件 (PAN) ：**PAN 描述了两个或更多个蓝牙设备如何构成一个即时网络，以及如何使用同一机制通过网络接入点接入远程网络。配置文件角色包括网络接入点、组即时网络及个人局域网用户。PAN的一个常用应用是如果一个手机通过数据连接或者WIFI连上了互联网，另外一个手机可以通过与这个联网手机建立PAN连接来上网。

**信息访问配置文件（MAP）：**MAP定义了访问手机短信息的协议。MAP也分为SERVER和CLIENT两种角色，SERVER端提供短信息的数据，CLIENT端是请求数据一端。我们的手机是SERVER端。

**通用属性配置文件（GATT）：**GATT是BLE上的应用，它定义了BLE设备之间数据传输的协议。现在流行的蓝牙手表、蓝牙手环就是通过GATT来与手机进行信息交互。

**一、蓝牙代码的位置及相关说明**

在6580上，蓝牙相关代码存在于很多地方，但是，绝大部分代码位于以下三个位置：

**1.packages/apps/Settings/src/com/android/settings/Bluetooth**

这个文件夹下的代码是google原生的代码，主要负责setting下蓝牙的基本功能及界面

**2.packages/apps/Bluetooth**

这个文件夹下的代码是google原生的代码，主要负责蓝牙的一些profile及jni

**3.vendor/mediatek/proprietary/packages/apps/Bluetooth**

这个文件夹下的代码是MTK写的，主要负责一些google没有涉及的profile，在我们的项目中，并没有使用这些 MTK提供的profile。

4.**system\bt\**

博通蓝牙源码

**二、蓝牙PORFILE的配置说明**

蓝牙的上层应用有很多，每个应用都被称之为PROFILE。

在MTK6580 平台可以对蓝牙的profile进行配置，也就是说可以根据项目 的实际需要来配置自己的项目支持哪些profile。

Profile的配置文件有两个，一个文件是google的默认android工程带有的；一个文件是MTK自己添加的。

Google规定的prifile配置文件路径如下：

packages/apps/Bluetooth/res/values/config.xml

内容如下：

|  |  |  |
| --- | --- | --- |
|  |  | <resources>           <bool name="profile\_supported\_a2dp">true</bool>           <bool name="profile\_supported\_a2dp\_sink">false</bool>           <bool name="profile\_supported\_hdp">true</bool>           <bool name="profile\_supported\_hs\_hfp">true</bool>           <bool name="profile\_supported\_hfpclient">false</bool>           <bool name="profile\_supported\_hid">true</bool>           <bool name="profile\_supported\_opp">true</bool>           <bool name="profile\_supported\_pan">true</bool>           <bool name="profile\_supported\_pbap">true</bool>           <bool name="profile\_supported\_gatt">true</bool>           <bool name="pbap\_include\_photos\_in\_vcard">false</bool>           <bool name="pbap\_use\_profile\_for\_owner\_vcard">true</bool>           <bool name="profile\_supported\_map">true</bool>           <bool name="profile\_supported\_avrcp\_controller">false</bool>  </resources> |

 MTK自己添加的profile的配置文件的路径如下：

vendor/mediatek/proprietary/external/bluetooth/bt\_cust/btconfig.xml

内容如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 |  | <resources>         <profile>             <name>profile\_supported\_simap</name>             <value>false</value>         </profile>         <profile>             <name>profile\_supported\_prxm</name>             <value>false</value>         </profile>         <profile>             <name>profile\_supported\_prxr</name>             <value>false</value>         </profile>         <profile>             <name>profile\_supported\_ftp</name>             <value>false</value>         </profile>         <profile>             <name>profile\_supported\_bpp</name>             <value>false</value>         </profile>         <profile>             <name>profile\_supported\_bip</name>             <value>false</value>         </profile>         <profile>             <name>profile\_supported\_dun</name>             <value>false</value>         </profile> </resources> |

在这个配置文件中：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 |  | <profile>        <name>profile\_supported\_dun</name>         <value>false</value>  </profile> |

表示把该项目设置为不支持DUN profile。

如果把该项变为：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 |  | <profile>       <name>profile\_supported\_dun</name>       <value>true</value>  </profile> |

表示把该项目设置为支持DUN profile。

还有一个蓝牙性能配置文件比较重要：vendor/mediatek/ proprietary/external/ bluetooth/bt\_cust/bt\_cust\_table.h，这个配置文件配置了很多蓝牙的性能，这个文件是MTK公开以方便用户定制的配置文件，由于这个文件比较长，就不呈现了，在实际开发中，曾经为了满足海外一个运营商的需要使用过该文件，改动的相关项如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 |  | {    .name = "InquiryTimeout",    .value = (void \*)0x30, } |

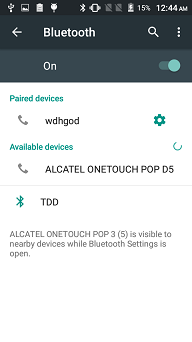
这一项，定义了蓝牙的搜索设备的时间，默认这个时间是0x30\*1.28s = 61.44s,这几年，公司的各个产品的蓝牙搜索设备时间都是61.44s，但是海外有个新的运营商客户觉得这个时间比较长，该运营商要求蓝牙搜索设备的时间为20s左右。根据该运营商的需求，我们设置了一个定制值来控制这个值，在该运营商的产品上：value = (void \*)0x0E,这样蓝牙搜索设备的时间就是0x0E\*1.28 = 14\*1.28 = 17.92s，再算上蓝牙搜索设备时的其它时间开销，总的蓝牙搜索设备时间共计20s左右，满足了该运营商的需求。

**三、蓝牙权限**  
 在开发安卓应用时，如果要使用蓝牙功能，需要申明蓝牙权限，具体如下：

|  |  |  |
| --- | --- | --- |
| 1 2 |  | <uses-permission android:name="android.permission.BLUETOOTH" /> <uses-permission android:name="android.permission.BLUETOOTH\_ADMIN" /> |

**四、重要的蓝牙界面实现介绍及相关蓝牙规则**

**1．Setting下的蓝牙主界面**

****

这个界面是蓝牙模块中使用最频繁、功能最多的界面，这个界面包含的功能主要有：

1.打开和关闭蓝牙

2.进入蓝牙的主菜单

3.发起对已搜索的设备的配对请求

4.进入已配对设备的设置

这个界面由文件BluetoothSettings.java中的BluetoothSettings类实现，该类是个Fragment,属于设置模块的Subsetting的一部分,Subsetting是Activity。该类的定义为：

public final class BluetoothSettings extends DeviceListPreferenceFragment implements Indexable

DeviceListPreferenceFragment的定义为：

public abstract class DeviceListPreferenceFragment extends

RestrictedSettingsFragment implements BluetoothCallback

BluetoothCallback是一个接口的代码如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 |  | interface BluetoothCallback {     // 蓝牙的状态变化时会回调     void onBluetoothStateChanged(int bluetoothState);     //蓝牙的可见性变化时会回调     void onScanningStateChanged(boolean started);     //当有新设备加入时回调     void onDeviceAdded(CachedBluetoothDevice cachedDevice);     //当设备消失时回调     void onDeviceDeleted(CachedBluetoothDevice cachedDevice);     //当设备的绑定发生变化时回调     void onDeviceBondStateChanged(CachedBluetoothDevice            cachedDevice, int bondState); } |

这个接口在蓝牙模块中被广泛应用，BluetoothSettings也间接继承了该接口。

该界面主要由两部分，一个是负责蓝牙打开与关闭的Switch bar，一个是设备列表。设备列表分为两个部分：Paired devices（已配对的设备）和Available devices（已搜索到的设备）.

Switch bar由以下代码加载：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 |  | public void onActivityCreated(Bundle savedInstanceState) {         ......         mBluetoothEnabler = new BluetoothEnabler(activity,            mSwitchBar);         mBluetoothEnabler.setupSwitchBar(); } |

设备列表由以下代码加载：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 |  | private void updateContent(int bluetoothState) {   ......   // Paired devices category   if (mPairedDevicesCategory == null) {   mPairedDevicesCategory = new PreferenceCategory(getActivity());   } else    {      mPairedDevicesCategory.removeAll();   }   addDeviceCategory(mPairedDevicesCategory,      R.string.bluetooth\_preference\_paired\_devices,      BluetoothDeviceFilter.BONDED\_DEVICE\_FILTER, true);   int numberOfPairedDevices = mPairedDevicesCategory.getPreferenceCount();    if (isUiRestricted() || numberOfPairedDevices <= 0) {      preferenceScreen.removePreference(mPairedDevicesCategory);   }    // Available devices category   if (mAvailableDevicesCategory == null) {   mAvailableDevicesCategory = new BluetoothProgressCategory(getActivity());   mAvailableDevicesCategory.setSelectable(false);   } else {   mAvailableDevicesCategory.removeAll();   }   addDeviceCategory(mAvailableDevicesCategory,      R.string.bluetooth\_preference\_found\_devices,      BluetoothDeviceFilter.UNBONDED\_DEVICE\_FILTER, mInitialScanStarted);   int numberOfAvailableDevices = mAvailableDevicesCategory.      getPreferenceCount();    ...... } |

在Android L平台，配对设备列表最多有40个设备，搜索到设备列表最多25个设备，这个在native层不开源代码中设定

可以看到每个设备前有个小图标，上图上有两种小图标，一个是电话状，一个是蓝牙标志，那么手机系统是如何给判别加载什么图标呢？

判断代码在BluetoothDevicePreference.java中：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 5 |  | private int getBtClassDrawable() {         BluetoothClass btClass = mCachedDevice.getBtClass();         if (btClass != null) {             switch (btClass.getMajorDeviceClass()) {                 case BluetoothClass.Device.Major.COMPUTER:                     //该设备是电脑，返回电脑图标                     return R.drawable.ic\_bt\_laptop;                  case BluetoothClass.Device.Major.PHONE:                     //该设备是手机，返回手机图标                     return R.drawable.ic\_bt\_cellphone;                  case BluetoothClass.Device.Major.PERIPHERAL:                     //该设备是外围设备，返回外设图标                     return HidProfile.getHidClassDrawable(btClass);                  case BluetoothClass.Device.Major.IMAGING:                     //该设备是图片设备，返回图片设备图标                     return R.drawable.ic\_bt\_imaging;                  default:                     // unrecognized device class; continue                     Log.d(TAG, "unrecognized device class " + btClass);             }         } else {             Log.w(TAG, "mBtClass is null");         }          List<LocalBluetoothProfile> profiles = mCachedDevice.getProfiles();         for (LocalBluetoothProfile profile : profiles) {             int resId = profile.getDrawableResource(btClass);             if (resId != 0) {                 return resId;             }         }         if (btClass != null) {             if (btClass.doesClassMatch(BluetoothClass.PROFILE\_A2DP)) {                 //该设备是立体声耳机，返回立体声耳机图标                 return R.drawable.ic\_bt\_headphones\_a2dp;              }             if (btClass.doesClassMatch(BluetoothClass.PROFILE\_HEADSET)) {                 //该设备是免提设备，返回免提设备图标                 return R.drawable.ic\_bt\_headset\_hfp;             }         }         return R.drawable.ic\_settings\_bluetooth2;  } |

要理解上述代码，先介绍蓝牙设备class的相关知识。

每个蓝牙设备都有一个class，这个class是一个四字节的字符串，它代表这个蓝牙设备的通用属性和具备的功能。通过分析一个蓝牙设备的class，我们可以知道这个蓝牙设备是一个手机还是一个计算机还是一个耳机等等，还可以知道

这个蓝牙设备具备的功能，这些功能比如：音频功能，电话功能等等。

每个class由0个或者几个services class和一个设备class构成。一个设备class

可以进一步拆分为major device class和minor device class

在文件BluetoothClass.java中，对service和major devices class做了描述，至于minor device class

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 |  | public static final class Service {         private static final int BITMASK                 = 0xFFE000;          public static final int LIMITED\_DISCOVERABILITY = 0x002000;         public static final int POSITIONING             = 0x010000;         //网络服务         public static final int NETWORKING              = 0x020000;         public static final int RENDER                  = 0x040000;         //照相服务         public static final int CAPTURE                 = 0x080000;         //文件传输服务         public static final int OBJECT\_TRANSFER         = 0x100000;         //音频服务         public static final int AUDIO                   = 0x200000;         //电话服务         public static final int TELEPHONY               = 0x400000;         //信息服务         public static final int INFORMATION             = 0x800000;     } |

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 |  | public static class Major {             private static final int BITMASK           = 0x1F00;              public static final int MISC              = 0x0000;             //计算机             public static final int COMPUTER          = 0x0100;             //电话             public static final int PHONE             = 0x0200;             //网络设备             public static final int NETWORKING        = 0x0300;             //音频视频设备             public static final int AUDIO\_VIDEO       = 0x0400;             //外围设备             public static final int PERIPHERAL        = 0x0500;             //图像设备             public static final int IMAGING           = 0x0600;             //可穿戴设备             public static final int WEARABLE          = 0x0700;             //玩具             public static final int TOY               = 0x0800;             //健康设备             public static final int HEALTH            = 0x0900;             public static final int UNCATEGORIZED     = 0x1F00; } |

在Android平台，设备的可见性分SCAN\_MODE\_CONNECTABLE\_DISCOVERABLE

（既可连接又可被发现）和SCAN\_MODE\_CONNECTABLE（只能被连接，不可见），

根据Android L的设计，只有当主界面位于最前面并可以获得焦点时，手机才是 可见的，当主界面退出时，手机会被设置为不可见。

控制设置可见性的变量是mInitiateDiscoverable。

当主界面不处于最前端或者退出时，如下代码将手机设置为不可见：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 |  | public void onPause() {    ......    // Make the device only visible to connected devices.    mLocalAdapter.setScanMode(BluetoothAdapter.SCAN\_MODE\_CONNECTABLE);    ...... } |

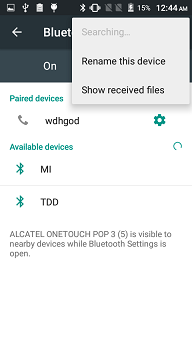
当主界面处于最前端并获得焦点时，如下代码将手机设置为可见：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 |  | private void updateContent(int bluetoothState) {    ......    if (mInitiateDiscoverable) {      // Make the device visible to other devices.      mLocalAdapter.setScanMode(BluetoothAdapter.         SCAN\_MODE\_CONNECTABLE\_DISCOVERABLE);      mInitiateDiscoverable = false;   }   ......  } |

当用户点击设备列表中的设备时，会有如下响应，在文件BluetoothDevicePreference.java中，代码如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 |  | void onClicked() {   int bondState = mCachedDevice.getBondState();   //如果该设备与本机连接，则询问是否断开连接   if (mCachedDevice.isConnected()) {      askDisconnect();   }    //如果该设备没有与本机连接，但已配对，则发起对该设备的连接   else if (bondState == BluetoothDevice.BOND\_BONDED) {     Log.d(TAG, mCachedDevice.getName() + " connect");             mCachedDevice.connect(true);   }   //如果该设备属于搜索到的设备，则发起对该设备的配对请求   else if (bondState == BluetoothDevice.BOND\_NONE) {     pair();   } } |

**2．主界面中的菜单界面**

****

当用户点击主界面右上角的option小图标时，会出现如上图的菜单列表，

文件BluetoothSettings.java中的如下代码加载了菜单列表：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 |  | public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {   if (mLocalAdapter == null) return;   //If the user is not allowed to configure bluetooth,do not show the menu.   if (isUiRestricted()) return;    boolean bluetoothIsEnabled = mLocalAdapter.getBluetoothState() ==          BluetoothAdapter.STATE\_ON;   boolean isDiscovering = mLocalAdapter.isDiscovering();   Log.d(TAG, "onCreateOptionsMenu, isDiscovering " + isDiscovering);   int textId = isDiscovering ? R.string.bluetooth\_searching\_for\_devices :             R.string.bluetooth\_search\_for\_devices;   menu.add(Menu.NONE, MENU\_ID\_SCAN, 0, textId)                 .setEnabled(bluetoothIsEnabled && !isDiscovering)                 .setShowAsAction(MenuItem.SHOW\_AS\_ACTION\_NEVER);   menu.add(Menu.NONE, MENU\_ID\_RENAME\_DEVICE, 0,                  R.string.bluetooth\_rename\_device)                 .setEnabled(bluetoothIsEnabled)                 .setShowAsAction(MenuItem.SHOW\_AS\_ACTION\_NEVER);   menu.add(Menu.NONE, MENU\_ID\_SHOW\_RECEIVED, 0,                  R.string.bluetooth\_show\_received\_files)                 .setShowAsAction(MenuItem.SHOW\_AS\_ACTION\_NEVER);   /// M: Add for Advance Settings   if (ConfigHelper.isAdvanceSettingEnabled()) {   menu.add(Menu.NONE, MENU\_ID\_ADVANCED\_SETTING, 0,                 R.string.bluetooth\_advanced\_settings)                 .setEnabled(bluetoothIsEnabled)                 .setShowAsAction(MenuItem.SHOW\_AS\_ACTION\_NEVER);   }         super.onCreateOptionsMenu(menu, inflater); } |

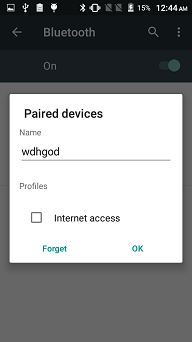
文件BluetoothSettings.java中的如下代码实现了菜单列表的响应：

 NormalText Code

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 |  | public boolean onOptionsItemSelected(MenuItem item) {   switch (item.getItemId()) {   case MENU\_ID\_SCAN:     if(mLocalAdapter.getBluetoothState() == BluetoothAdapter.STATE\_ON)     {       //开始搜索蓝牙设备       startScanning();     }     return true;    case MENU\_ID\_RENAME\_DEVICE:     //重命名本机蓝牙的名字     new BluetoothNameDialogFragment().show(                         getFragmentManager(), "rename device");     return true;    case MENU\_ID\_SHOW\_RECEIVED:     //显示已接收文件的列表     Intent intent = new Intent(BTOPP\_ACTION\_OPEN\_RECEIVED\_FILES);     getActivity().sendBroadcast(intent);     return true;    //M:when click menu "Advanced settings",start corresponding activity   case MENU\_ID\_ADVANCED\_SETTING:     //进入蓝牙高级设置     Intent i = new Intent(Intent.ACTION\_MAIN);     i.setAction(LAUNCH\_BT\_ADVANCED\_SETTINGS\_ACTION);     try {           startActivity(i);         } catch (ActivityNotFoundException e) {         Log.e(TAG, "Unable to start activity " + i.toString());         return false;         }     return true;             /// @}     }   return super.onOptionsItemSelected(item); } |

要注意，从代码看，菜单列表是四项，包括高级设置这一项，但是上图列表中并没有这一项，是因为受到函数ConfigHelper.isAdvanceSettingEnabled()的限制，如果系统里打开mtk自己实现的PROFILE，那么高级设置这一菜单项就出现了。

**3．已配对设备的设置界面**

****

当用户点击已配对设备的右边小图标时，pair.bmp，就会进入如上界面。在该界面，用户点击Forget，来解除与该设备的配对关系。在该界面，用户也可以修改该设备的名字，当用户修改了该设备的蓝牙名字，并点击OK，就可以完成对该设备的蓝牙名字的修改。

在该界面，还列出了该设备与本机相关的profile，在该界面的Internet

Access表明，该设备与本机可以建立PAN连接。

绘出这个小图标的代码如下：

在文件BluetoothDevicePreference.java中，

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 |  | protected void onBindView(View view) {         // Disable this view if the bluetooth enable/disable preference         // view is off         if (null != findPreferenceInHierarchy("bt\_checkbox")) {             setDependency("bt\_checkbox");         }          if (mCachedDevice.getBondState() == BluetoothDevice.BOND\_BONDED) {             ImageView deviceDetails = (ImageView) view.findViewById(R.id.                                                            deviceDetails);              if (deviceDetails != null) {                 deviceDetails.setOnClickListener(this);                 deviceDetails.setTag(mCachedDevice);             }         }          super.onBindView(view);  } |

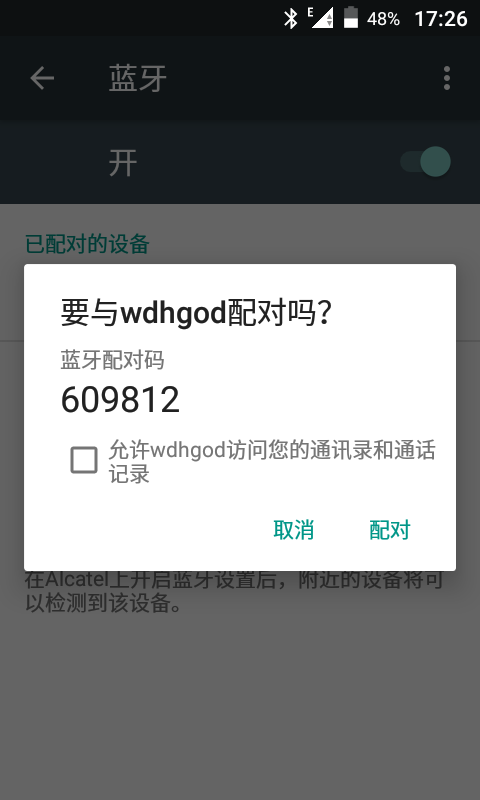
  设置点击该小图标的响应的代码如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 |  | 在文件BluetoothDevicePreference.java中：  public void setOnSettingsClickListener(OnClickListener listener) {     mOnSettingsClickListener = listener;  }  public void onClick(View v) {     // Should never be null by construction     if (mOnSettingsClickListener != null) {     mOnSettingsClickListener.onClick(v);    }  }  在文件BluetoothSettings.java中调用方法setOnSettingsClickListener  来设置点击该小图标的相应函数：  void initDevicePreference(BluetoothDevicePreference preference) {     CachedBluetoothDevice cachedDevice = preference.getCachedDevice();     if (cachedDevice.getBondState() == BluetoothDevice.BOND\_BONDED) {     // Only paired device have an associated advanced settings screen     preference.setOnSettingsClickListener(mDeviceProfilesListener);   }  } |

该界面由BluetoothSettings.java中的如下代码绘出：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 |  | private final View.OnClickListener mDeviceProfilesListener = new View. OnClickListener() {  .......      if (profileFragment == null || !profileFragment.isInLayout()) {      LayoutInflater inflater = getActivity().getLayoutInflater();      Log.d(TAG, "LayoutInflater inflate the dialog fragment");      mSettingsDialogView = inflater.inflate(R.layout.        bluetooth\_device\_settings,null);      profileFragment = (DeviceProfilesSettings)activity.getFragmentManager()             .findFragmentById(R.id.bluetooth\_fragment\_settings);       //To enable scrolling we store the name field in a seperate header and      // add to      // the ListView of the profileFragment.      View header =inflater.inflate(R.layout.bluetooth\_device\_settings\_header,          null);      /// M: ALPS01849600 {@       if (mSettingsDialogView == null || header == null) {           Log.d(TAG, "Layout inflate fail, just exit");           return ;        }        /// @}        profileFragment.getListView().addHeaderView(header);    }     final View dialogLayout = mSettingsDialogView;    AlertDialog.Builder settingsDialog = new AlertDialog.Builder(activity);    profileFragment.setDevice(device);    final EditText deviceName = (EditText)dialogLayout.findViewById(R.id.name);    deviceName.setText(device.getName(), TextView.BufferType.EDITABLE);     final DeviceProfilesSettings dpsFragment = profileFragment;    final Context context = v.getContext();    settingsDialog.setView(dialogLayout);    settingsDialog.setTitle(R.string.bluetooth\_preference\_paired\_devices);    settingsDialog.setPositiveButton(R.string.okay,       new DialogInterface.OnClickListener() {       @Override         public void onClick(DialogInterface dialog, int which) {         EditText deviceName = (EditText)dialogLayout.findViewById(R.id.name);         device.setName(deviceName.getText().toString());         }    });     settingsDialog.setNegativeButton(R.string.forget,       new DialogInterface.OnClickListener() {       @Override       public void onClick(DialogInterface dialog, int which) {         device.unpair();         com.android.settings.bluetooth.Utils.updateSearchIndex(         activity,BluetoothSettings.class.getName(), device.getName(),         context.getResources().getString(R.string.         bluetooth\_settings),         R.drawable.ic\_settings\_bluetooth2, false);      }    });     // We must ensure that the fragment gets destroyed     //to avoid duplicate fragments.    settingsDialog.setOnDismissListener(new DialogInterface.OnDismissListener()     {      public void onDismiss(final DialogInterface dialog) {        if (!activity.isDestroyed()) {          activity.getFragmentManager().beginTransaction().remove(dpsFragment)          .commitAllowingStateLoss();          Log.d(TAG, "Remove the fragment view");          }        }   });    final AlertDialog dialog = settingsDialog.create();   mProfileDialog = dialog;   ///M: Avoid unEffective name   deviceName.addTextChangedListener(new TextWatcher() {   public void afterTextChanged(Editable s) {     boolean isEffective = s.toString().trim().length() > 0;     dialog.getButton(AlertDialog.BUTTON\_POSITIVE).setEnabled(isEffective);   } ......  } |

**4.配对确认界面**

****

在使用蓝牙设备时，为了保证信息安全，产生了蓝牙配对这个概念，配对就是在双方蓝牙设备之间建立一种信任机制，在较早的蓝牙设备，用户要手动输入蓝牙配对码，在蓝牙2.1及其之后，引入了蓝牙安全简单配对，这样，用户就不必手动输入配对码，蓝牙设备会自动生成配对码，以此让用户确认，当然蓝牙设备双方的蓝牙配对码是一样的，用户通过点击屏幕，来确认就好。效果界面如上图。

MMI部分在文件BluetoothPairingRequest.java中接收广播：BluetoothDevice.ACTION\_PAIRING\_REQUEST开启BluetoothPairingDialog这个

Activity。相关代码如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 |  | public final class BluetoothPairingRequest extends BroadcastReceiver {   private static final int NOTIFICATION\_ID = android.R.drawable.        stat\_sys\_data\_bluetooth;   @Override  public void onReceive(Context context, Intent intent) {    String action = intent.getAction();    if (action.equals(BluetoothDevice.ACTION\_PAIRING\_REQUEST)) {    // convert broadcast intent into activity intent (same action string)    BluetoothDevice device =      intent.getParcelableExtra(BluetoothDevice.EXTRA\_DEVICE);    int type = intent.getIntExtra(BluetoothDevice.EXTRA\_PAIRING\_VARIANT,      BluetoothDevice.ERROR);    Intent pairingIntent = new Intent();    pairingIntent.setClass(context, BluetoothPairingDialog.class);    pairingIntent.putExtra(BluetoothDevice.EXTRA\_DEVICE, device);    pairingIntent.putExtra(BluetoothDevice.EXTRA\_PAIRING\_VARIANT, type);    if (type == BluetoothDevice.PAIRING\_VARIANT\_PASSKEY\_CONFIRMATION ||        type == BluetoothDevice.PAIRING\_VARIANT\_DISPLAY\_PASSKEY ||        type == BluetoothDevice.PAIRING\_VARIANT\_DISPLAY\_PIN) {        int pairingKey = intent.getIntExtra(BluetoothDevice.EXTRA\_PAIRING\_KEY,        BluetoothDevice.ERROR);        pairingIntent.putExtra(BluetoothDevice.EXTRA\_PAIRING\_KEY, pairingKey);     }     pairingIntent.setAction(BluetoothDevice.ACTION\_PAIRING\_REQUEST);     pairingIntent.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);      PowerManager powerManager =        (PowerManager)context.getSystemService(Context.POWER\_SERVICE);     String deviceAddress = device != null ? device.getAddress() : null;     if (powerManager.isScreenOn() &&       LocalBluetoothPreferences.shouldShowDialogInForeground(context,      deviceAddress)) {     // Since the screen is on and the BT-related activity is in the     foreground,     // just open the dialog     context.startActivity(pairingIntent);  } |

在文件BluetoothPairingDialog.java中，处理了配对中的相关界面，包括需要输入配对码的和不须输入配对码的，由于蓝牙设备发展的很快，本文档只介绍如上图的界面。

  在蓝牙的配对模块中，把配对的各种方式都包括了，并以常量类型区分各种配对方式，如上图的确认配对码的配对方式的类型是BluetoothDevice.PAIRING\_VARIANT\_PASSKEY\_CONFIRMATION，在方法onCreate中进行区分：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 |  | protected void onCreate(Bundle savedInstanceState) {  ......    case BluetoothDevice.PAIRING\_VARIANT\_PASSKEY\_CONFIRMATION:    int passkey =      intent.getIntExtra(BluetoothDevice.EXTRA\_PAIRING\_KEY, BluetoothDevice.ERROR);      if (passkey == BluetoothDevice.ERROR) {      Log.e(TAG, "Invalid Confirmation Passkey received, not showing any dialog");      return;    }    mPairingKey = String.format(Locale.US, "%06d", passkey);    createConfirmationDialog();    break;  ...... } |

如下代码，实现了配对确认界面：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 |  | private void createConfirmationDialog() {   final AlertController.AlertParams p = mAlertParams;   p.mTitle = getString(R.string.bluetooth\_pairing\_request,     mCachedDeviceManager.getName(mDevice));   p.mView = createView();   p.mPositiveButtonText = getString(R.string.bluetooth\_pairing\_accept);   p.mPositiveButtonListener = this;   p.mNegativeButtonText = getString(R.string.bluetooth\_pairing\_decline);   p.mNegativeButtonListener = this;   setupAlert(); } |

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 3 |  | private View createView() {   View view = getLayoutInflater().inflate(R.layout.bluetooth\_pin\_confirm, null);   TextView pairingViewCaption = (TextView) view.findViewById(R.id.pairing\_caption);   TextView pairingViewContent = (TextView) view.findViewById(R.id.pairing\_subhead);   TextView messagePairing = (TextView) view.findViewById(R.id.pairing\_code\_message);   CheckBox contactSharing = (CheckBox) view.findViewById(      R.id.phonebook\_sharing\_message\_confirm\_pin);   contactSharing.setText(getString(R.string.bluetooth\_pairing\_shares\_phonebook,                 mCachedDeviceManager.getName(mDevice)));   if (mDevice.getPhonebookAccessPermission() == BluetoothDevice.ACCESS\_ALLOWED) {             contactSharing.setChecked(true);   } else if (mDevice.getPhonebookAccessPermission() == BluetoothDevice.ACCESS\_REJECTED){             contactSharing.setChecked(false);   } else {   if (mDevice.getBluetoothClass().getDeviceClass()          == BluetoothClass.Device.AUDIO\_VIDEO\_HANDSFREE) {     contactSharing.setChecked(true);     mDevice.setPhonebookAccessPermission(BluetoothDevice.ACCESS\_ALLOWED);   } else {     contactSharing.setChecked(false);     mDevice.setPhonebookAccessPermission(BluetoothDevice.ACCESS\_REJECTED);    }  } //注册pbap相关的点击监听  contactSharing.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {    @Override    public void onCheckedChanged(CompoundButton compoundButton, boolean isChecked) {    if (isChecked) {      mDevice.setPhonebookAccessPermission(BluetoothDevice.ACCESS\_ALLOWED);    } else {    mDevice.setPhonebookAccessPermission(BluetoothDevice.ACCESS\_REJECTED);    }  } });  String messageCaption = null; String pairingContent = null; switch (mType) {   ......   case BluetoothDevice.PAIRING\_VARIANT\_PASSKEY\_CONFIRMATION:     pairingContent = mPairingKey;     break;    ......                  }  if (pairingContent != null) {   pairingViewCaption.setVisibility(View.VISIBLE);   pairingViewContent.setVisibility(View.VISIBLE);   pairingViewContent.setText(pairingContent); }    return view; } |

该界面的布局文件是Bluetooth\_pin\_confirm.xml，布局内容如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 3 |  | <ScrollView     xmlns:android="http://schemas.android.com/apk/res/android"     android:layout\_height="match\_parent"     android:layout\_width="match\_parent">      <LinearLayout         xmlns:android="http://schemas.android.com/apk/res/android"         android:paddingStart="16dip"         android:layout\_height="match\_parent"         android:layout\_width="match\_parent"         android:orientation="vertical">          <TextView             android:id="@+id/pairing\_caption"             android:layout\_width="wrap\_content"             android:layout\_height="wrap\_content"             android:layout\_marginTop="@dimen/bluetooth\_dialog\_padding"             android:layout\_marginStart="@dimen/bluetooth\_dialog\_padding"             android:layout\_marginEnd="@dimen/bluetooth\_dialog\_padding"             android:gravity="center\_vertical"             android:text="@string/bluetooth\_pairing\_key\_msg"             android:visibility="gone"             android:textAppearance="@android:style/TextAppearance.Material.Body1"             android:textColor="@\*android:color/secondary\_text\_material\_light"  />          <TextView             android:id="@+id/pairing\_subhead"             android:layout\_width="wrap\_content"             android:layout\_height="wrap\_content"             android:layout\_marginStart="@dimen/bluetooth\_dialog\_padding"             android:layout\_marginEnd="@dimen/bluetooth\_dialog\_padding"             android:layout\_marginBottom="@dimen/bluetooth\_dialog\_padding"             android:gravity="center\_vertical"             android:visibility="gone"             android:textAppearance="@android:style/TextAppearance.Material.Headline" />          <TextView             android:id="@+id/pairing\_code\_message"             android:layout\_width="wrap\_content"             android:layout\_height="wrap\_content"             android:layout\_marginStart="@dimen/bluetooth\_dialog\_padding"             android:layout\_marginEnd="@dimen/bluetooth\_dialog\_padding"             android:layout\_marginBottom="@dimen/bluetooth\_dialog\_padding"             android:gravity="center\_vertical"             android:text="@string/bluetooth\_enter\_passkey\_msg"             android:textAppearance="@android:style/TextAppearance.Material.Subhead"             android:textColor="@\*android:color/secondary\_text\_material\_light"             android:visibility="gone" />          <CheckBox             android:id="@+id/phonebook\_sharing\_message\_confirm\_pin"             android:layout\_width="wrap\_content"             android:layout\_height="wrap\_content"             android:layout\_marginStart="@dimen/bluetooth\_dialog\_padding"             android:layout\_marginEnd="@dimen/bluetooth\_dialog\_padding"             android:textAppearance="@android:style/TextAppearance.Material.Body1"             android:textColor="@\*android:color/secondary\_text\_material\_light"  />     </LinearLayout> </ScrollView> |

可以看到很多控件的属性都是gone，这就使程序变得灵活，在需要该控件可见时，可以通过API：setVisibility(View.VISIBLE)进行设置。

在配对界面，对softkey的点击处理如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 |  | public void onClick(DialogInterface dialog, int which) {   switch (which) {     case BUTTON\_POSITIVE:       if (mPairingView != null) {         onPair(mPairingView.getText().toString());         } else {           onPair(null);         }         break;          case BUTTON\_NEGATIVE:          default:           onCancel();         break;      } } |

**5．发送文件界面**

****

一般来说，用户想要通过蓝牙传输文件时，要通过share来完成，在share中选择蓝牙，就会跳到如下页面：

****

在该界面，会显示已经配对的设备，并且进行对新设备进行搜索，该界面的显示代码如下，在文件：packages\apps\settings\src\com\android\settings\

Bluetooth\ DevicePickerFragment.java：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 |  | public void onCreate(Bundle savedInstanceState) {    super.onCreate(savedInstanceState);    getActivity().setTitle(getString(R.string.device\_picker));    UserManager um = (UserManager) getSystemService(Context.USER\_SERVICE);    mStartScanOnResume = !um.hasUserRestriction(DISALLOW\_CONFIG\_BLUETOOTH)                 && (savedInstanceState == null);  // don't start scan after rotation    setHasOptionsMenu(true);     mProgressCategory = (ProgressCategory) findPreference(KEY\_BT\_DEVICE\_LIST); } |

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 |  | public void onResume() {   super.onResume();   ///M: before add device pref, firstly clear the screen   mProgressCategory.setNoDeviceFoundAdded(false);   removeAllDevices();   //显示已配对设备   addCachedDevices();   if (mStartScanOnResume) {   //展开蓝牙设备的搜索   mLocalAdapter.startScanning(true);   mStartScanOnResume = false;  } } |

在这个界面，当用户点击设备会对选中的设备进行蓝牙文件传输，方法是sendDevicePickedIntent，

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 |  | void onDevicePreferenceClick(BluetoothDevicePreference btPreference) {   mLocalAdapter.stopScanning();   LocalBluetoothPreferences.persistSelectedDeviceInPicker(   getActivity(), mSelectedDevice.getAddress());   if ((btPreference.getCachedDevice().getBondState() ==     BluetoothDevice.BOND\_BONDED) || !mNeedAuth) {   sendDevicePickedIntent(mSelectedDevice);   finish();  } else {          super.onDevicePreferenceClick(btPreference);  } } |

**6．接收文件确认界面**



该界面的代码在packages\apps\bluetooth\src\com\android\Bluetooth\opp

\BluetoothOppIncomingFileConfirmActivity.java中。

当用户发送和接收文件时，接收方会弹出如上界面，在这个界面，用户可以选择是否接收发来的文件。如果在该界面，用户不做选择，那么在两分钟后，该界面会消失，从而拒绝了接收文件。

   如下代码是用户点击了拒绝和接受的处理方法。

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 1 |  | public void onClick(DialogInterface dialog, int which) {   switch (which) {     case DialogInterface.BUTTON\_POSITIVE:       if (!mTimeout) {       // Update database       mUpdateValues = new ContentValues();       mUpdateValues.put(BluetoothShare.USER\_CONFIRMATION,         BluetoothShare.USER\_CONFIRMATION\_CONFIRMED);       this.getContentResolver().update(mUri, mUpdateValues, null, null);        Toast.makeText(this, getString(R.string.bt\_toast\_1), Toast.LENGTH\_SHORT).show();     }     break;      case DialogInterface.BUTTON\_NEGATIVE:     // Update database     mUpdateValues = new ContentValues();     mUpdateValues.put(BluetoothShare.USER\_CONFIRMATION,       BluetoothShare.USER\_CONFIRMATION\_DENIED);     this.getContentResolver().update(mUri, mUpdateValues, null, null);     break;   } } |

在其他蓝牙设备给本机发送文件时，探出了如上界面，如果用户在2分钟内没有处理，那么本机会收到超时广播：BluetoothShare.USER\_CONFIRMATION\_

TIMEOUT\_ACTION，本机在收到超时广播后，进行超时处理，相应的方法是onTimeout。

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 |  | private BroadcastReceiver mReceiver = new BroadcastReceiver() {    @Override    public void onReceive(Context context, Intent intent) {    if (!BluetoothShare.USER\_CONFIRMATION\_TIMEOUT\_ACTION.      equals(intent.getAction())) {      return;    }    onTimeout();   }  }; |

  在接收到超时广播后，本机通过ui显示提醒用户超时，并发送消息DISMISS\_TIMEOUT\_DIALOG。

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 1 |  | private void onTimeout() {   if (V)Log.v(TAG, "onTimeout ++");   mTimeout = true;    mAlert.setTitle(getString(R.string.incoming\_file\_confirm\_timeout\_content,     mTransInfo.mDeviceName));   mAlert.getButton(DialogInterface.BUTTON\_NEGATIVE).setVisibility(View.GONE);   mAlert.getButton(DialogInterface.BUTTON\_POSITIVE).setText(   getString(R.string.incoming\_file\_confirm\_timeout\_ok));    mTimeoutHandler.sendMessageDelayed(mTimeoutHandler.obtainMessage(DISMISS\_TIMEOUT\_DIALOG),                 DISMISS\_TIMEOUT\_DIALOG\_VALUE); } |

在handle中，处理消息DISMISS\_TIMEOUT\_DIALOG，使用finish()关闭当前界面。

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 |  | private final Handler mTimeoutHandler = new Handler() {   @Override   public void handleMessage(Message msg) {     switch (msg.what) {     case DISMISS\_TIMEOUT\_DIALOG:     if (V) Log.v(TAG, "Received DISMISS\_TIMEOUT\_DIALOG msg.");     finish();     break;     default:       break;    }  } }; |

**四、蓝牙的JNI调用说明**

JNI：在蓝牙模块中，上层代码是java，通过jni，来实现对底层功能代码的调用：

Java->c++->c

在android L平台部分代码，在文件com\_android\_bluetooth\_btservice\_

AdapterService.cpp中如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 |  | static JNINativeMethod sMethods[] = {     /\* name, signature, funcPtr \*/     {"classInitNative", "()V", (void \*) classInitNative},     {"initNative", "()Z", (void \*) initNative},     {"cleanupNative", "()V", (void\*) cleanupNative},     {"enableNative", "()Z",  (void\*) enableNative},     {"disableNative", "()Z",  (void\*) disableNative},     {"setAdapterPropertyNative", "(I[B)Z", (void\*) setAdapterPropertyNative},     {"getAdapterPropertiesNative", "()Z", (void\*) getAdapterPropertiesNative},     {"getAdapterPropertyNative", "(I)Z", (void\*) getAdapterPropertyNative},     {"getDevicePropertyNative", "([BI)Z", (void\*) getDevicePropertyNative},     {"setDevicePropertyNative", "([BI[B)Z", (void\*) setDevicePropertyNative},     {"startDiscoveryNative", "()Z", (void\*) startDiscoveryNative},     {"cancelDiscoveryNative", "()Z", (void\*) cancelDiscoveryNative},     {"createBondNative", "([BI)Z", (void\*) createBondNative},     {"removeBondNative", "([B)Z", (void\*) removeBondNative},     {"cancelBondNative", "([B)Z", (void\*) cancelBondNative},     {"getConnectionStateNative", "([B)I", (void\*) getConnectionStateNative},     {"pinReplyNative", "([BZI[B)Z", (void\*) pinReplyNative},     {"sspReplyNative", "([BIZI)Z", (void\*) sspReplyNative},     {"getRemoteServicesNative", "([B)Z", (void\*) getRemoteServicesNative},     {"getRemoteMasInstancesNative", "([B)Z", (void\*) getRemoteMasInstancesNative},     {"connectSocketNative", "([BI[BII)I", (void\*) connectSocketNative},     {"createSocketChannelNative", "(ILjava/lang/String;[BII)I",      (void\*) createSocketChannelNative},     {"configHciSnoopLogNative", "(Z)Z", (void\*) configHciSnoopLogNative},     {"alarmFiredNative", "()V", (void \*) alarmFiredNative},     {"readEnergyInfo", "()I", (void\*) readEnergyInfo},     /// M: ALPS01293574: Dump FTrace     {"dumpFTraceNative", "()V", (void \*) dumpFTraceNative} }; |

而底层代码通过jni中的callback来实现对上层java代码的调用：

c->c++->java

在Android L平台部分代码，在文件com\_android\_bluetooth\_btservice\_

AdapterService.cpp中如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 |  | static void classInitNative(JNIEnv\* env, jclass clazz) {     int err;     hw\_module\_t\* module;      jclass jniCallbackClass =         env->FindClass("com/android/bluetooth/btservice/JniCallbacks");     sJniCallbacksField = env->GetFieldID(clazz, "mJniCallbacks",         "Lcom/android/bluetooth/btservice/JniCallbacks;");      method\_stateChangeCallback = env->GetMethodID(jniCallbackClass,                "stateChangeCallback", "(I)V");      method\_adapterPropertyChangedCallback = env->GetMethodID(jniCallbackClass,                "adapterPropertyChangedCallback",                "([I[[B)V");     method\_discoveryStateChangeCallback = env->GetMethodID(jniCallbackClass,                "discoveryStateChangeCallback", "(I)V");      method\_devicePropertyChangedCallback = env->GetMethodID(jniCallbackClass,                "devicePropertyChangedCallback",                ([B[I[[B)V");     method\_deviceFoundCallback = env->GetMethodID(jniCallbackClass,                "deviceFoundCallback", "([B)V");     method\_pinRequestCallback = env->GetMethodID(jniCallbackClass,                 "pinRequestCallback",                "([B[BI)V");     method\_sspRequestCallback = env->GetMethodID(jniCallbackClass,               "sspRequestCallback",               "([B[BIII)V");      method\_bondStateChangeCallback = env->GetMethodID(jniCallbackClass,               "bondStateChangeCallback", "(I[BI)V");      method\_aclStateChangeCallback = env->GetMethodID(jniCallbackClass,               "aclStateChangeCallback", "(I[BI)V"); |

事实上，在智能机上，蓝牙的真正的功能代码还是底层的c代码，当然这些代码都在库里，我们是看不到的，所以当蓝牙模块出现功能问题时，往往要merge MTK的patch。

**七、蓝牙定制值的实现**

在6580中蓝牙模块有三个定制值，一个是用于定制蓝牙的默认名字；一个是用于定制本地蓝牙适配器 的第一次开机的状态，也就是当用户第一次开机时，本地蓝牙是否被打开；一个是用于定制蓝牙搜索设备的时间。

1. **定制蓝牙名字**

**在android L使用Def\_framework\_bt\_name**

机制：在蓝牙被开启后，执行对蓝牙名字进行修改的代码，当然这段代码只会被执行一次。如下代码，是设置蓝牙默认名字的功能代码。

在文件isdm\_Jrdshared.plf中定义Def\_framework\_bt\_name

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 |  | <VAR> <SIMPLE\_VAR> <SDMID>Def\_framework\_bt\_name</SDMID> <C\_NAME>Def\_framework\_bt\_name</C\_NAME> <C\_TYPE>AsciiString</C\_TYPE> <ARRAY>NOT\_ARRAY</ARRAY> <METATYPE>AsciiString,60</METATYPE> <IS\_CUSTO>1</IS\_CUSTO> <FEATURE>SystemProperty</FEATURE> <DESC>If this item value is "" the bluetooth default name is  PRODUCT\_MODEL, otherwise the bluetooth name will use this value</DESC> <VALUE>"ALCATEL ONETOUCH POP 3 (5) Relook"</VALUE> </SIMPLE\_VAR> </VAR> |

在文件AdapterService.java中

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 |  | public void setBluetoothDefaultName() {    mPref=this.getSharedPreferences("bluetooth", Context.MODE\_PRIVATE);    //判断是否第一次执行这个操作    if (true == mPref.getBoolean("isFirstPowerOn", true))     {       Context = getApplicationContext();       //取出蓝牙的默认名字            bluetooth\_default\_name=mContext.getResources().getString          (com.jrdcom.internal.R.string.Def\_framework\_bt\_name);        if (bluetooth\_default\_name != null)             {                    //根据默认值设置蓝牙名字              setName(bluetooth\_default\_name);              SharedPreferences.Editor editor = mPref.edit();              editor.putBoolean("isFirstPowerOn", false);              editor.commit();             }       } } |

这个设置蓝牙默认名字的功能代码，很多项目上都是一样的，但是把这段代码放到什么位置是很重要的，最新的做法是把这段代码放到蓝牙被打开后的callback中，在蓝牙被打开后，系统通过JNI，把蓝牙已经打开的事件通过callback到上层应用，因为以上的代码只会执行一次，所以，在用户拿到手机，第一次打开蓝牙后，应用程序就会取出预先在perso中设置的蓝牙名字对蓝牙名字进行设置，从而实现蓝牙名字定制的功能。

在文件AdapterState.java中，相关代码如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 |  | void stateChangeCallback(int status) {   if (status == AbstractionLayer.BT\_STATE\_OFF) {    sendMessage(DISABLED);   }   //判断蓝牙已经被打开    else if (status == AbstractionLayer.BT\_STATE\_ON) {        sendMessage(ENABLED\_READY);        if(mAdapterService != null){        Log.d(TAG, "setBluetoothDefaultName mAdapterService != null");         //在蓝牙被激活后，设置蓝牙默认名字         mAdapterService.setBluetoothDefaultName();        }      } ...... } |

**在android M 使用ro.bluetooth\_default\_name**

在android M，由于mtk使用了博通的部分代码，所以，定制代码是在c代码中实现的，定制值是一个属性。

在文件Isdm\_sys\_properties.plf中定义ro.bluetooth\_default\_name

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 |  | <VAR> <SIMPLE\_VAR> <SDMID>ro.bluetooth\_default\_name</SDMID> <C\_TYPE></C\_TYPE> <C\_NAME>ro.bluetooth\_default\_name</C\_NAME> <ARRAY>NOT\_ARRAY</ARRAY> <METATYPE>AsciiString,60</METATYPE> <IS\_CUSTO>1</IS\_CUSTO> <VALUE>"Alcatel Pixi 4 (4)"</VALUE> <FEATURE>SystemProperty</FEATURE> <DESC> [FR/PR]:730544 [Function]:the bluetooth name will use this value [Value]:defa bluetooth name will use this value [Special C]:single sim and dual sim are the same </DESC> </SIMPLE\_VAR> </VAR> |

在文件Btif\_dm.c中，去掉函数btif\_get\_default\_local\_name中的代码，再加上如下代码：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 2 |  | static char\* btif\_get\_default\_local\_name() {  if (btif\_default\_local\_name[0] == '\0')  {    char prop\_name[PROPERTY\_VALUE\_MAX] = {'\0'};;    if ((property\_get("ro.bluetooth\_default\_name", prop\_name, NULL)))    {      strncpy(btif\_default\_local\_name, prop\_name, DEFAULT\_LOCAL\_NAME\_MAX);    }    else    {      char prop\_model[PROPERTY\_VALUE\_MAX] = {'\0'};      property\_get(PROPERTY\_PRODUCT\_MODEL, prop\_model, "");      strncpy(btif\_default\_local\_name, prop\_model, DEFAULT\_LOCAL\_NAME\_MAX);    }    btif\_default\_local\_name[DEFAULT\_LOCAL\_NAME\_MAX] = '\0';  }  LOG\_DEBUG("BTA\_DM\_ENABLE\_EVT  PROPERTY\_VALUE\_MAX : %d ", PROPERTY\_VALUE\_MAX);  LOG\_DEBUG("BTA\_DM\_ENABLE\_EVT  btif\_default\_local\_name : %s ",   btif\_default\_local\_name);  return btif\_default\_local\_name; } |

**2．定制第一次开机时蓝牙的初始状态**

这个定制是实现当用户第一次开机时，蓝牙是打开还是关闭。

使用Def\_framework\_bt\_activation

定制代码在文件BluetoothManagerService.java中，定制代码如下：

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 |  | BluetoothManagerService(Context context) {    ......   int bootCount=1;   //取出定制值   Boolean isAcitivateByDefault= mContext.getResources().getBoolean     (com.android.internal.R.bool.Def\_framework\_bt\_activation);   String preferenceName="Bluetooth\_customization";   try{     bootCount=Settings.Secure.getInt(mContentResolver,     preferenceName);    }catch(android.provider.Settings.SettingNotFoundException e){    bootCount=0;    Settings.Secure.putInt(mContentResolver,preferenceName,1);   }   //如果第一次开机而且定制蓝牙初始状态为打开，则设置蓝牙状态   //为打开   if(bootCount==0 && isAcitivateByDefault){   Settings.Global.putInt(mContentResolver,Settings.Global.        BLUETOOTH\_ON, 1);   }    ...... } |

手机系统开机会执行BluetoothManagerService，上述代码也是确保功 能代码Settings.Global.putInt(mContentResolver,Settings.Global.  
BLUETOOTH\_ON, 1)只执行一次，当进行这样设置后，后面的程序会对Settings.Global.BLUETOOTH\_ON进行判断，当发现Settings.Global.

BLUETOOTH\_ON是1时，就会打开蓝牙，当发现Settings.Global.BLUETOOTH

\_ON是0时，就不打开蓝牙，从而实现定制用户第一次开机是否打开蓝牙的功能。  
**3. 定制蓝牙搜索设备的时间**

在android M平台上：

  在文件Isdm\_sys\_properties.plf中定义ro.bluetooth\_shortinquire

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 |  | <VAR> <SIMPLE\_VAR> <SDMID>ro.bluetooth\_shortinquire</SDMID> <C\_TYPE>AsciiString</C\_TYPE> <C\_NAME>ro.bluetooth\_shortinquire</C\_NAME> <ARRAY>NOT\_ARRAY</ARRAY> <METATYPE>AsciiString,104</METATYPE> <IS\_CUSTO>1</IS\_CUSTO> <VALUE>"false"</VALUE> <FEATURE>SystemProperty</FEATURE> <DESC> [FR/PR]:1711758 [Function]:"false": the time of bt require is  normal,the time is 60s ; "true": the time of bt require is short,the time is 15s.the isdm  is "false" by default [Value]:"false": the time of bt require is  normal,the time is 60s;  "true": the time of bt require is short,the time is 15s.the isdm is  "false" by default [Special C]:single sim and dual sim are the same </DESC> </SIMPLE\_VAR> </VAR> |

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 |  | #define BTIF\_DM\_DEFAULT\_INQ\_MAX\_DURATION    10  #define BTIF\_DM\_DEFAULT\_INQ\_MAX\_DURATION\_LONG 44  bt\_status\_t btif\_dm\_start\_discovery(void) {   ......    property\_get("ro.bluetooth\_shortinquire", property\_inquire\_time,"false");    if(0 == strcmp(property\_inquire\_time, "true"))   {     inq\_params.duration = BTIF\_DM\_DEFAULT\_INQ\_MAX\_DURATION;   }   else     {     inq\_params.duration = BTIF\_DM\_DEFAULT\_INQ\_MAX\_DURATION\_LONG;   }    ...... } |

上述代码可以实现对蓝牙搜索设备时间的定制，在这里，时间单位是1.28秒。如果设定ro.bluetooth\_shortinquire为true，则蓝牙搜索设备的时间就是：10\*1.28=12.8秒，如果设定ro.bluetooth\_shortinquire为false，则蓝牙搜索设备的时间就是：44\*12.8= 563.2秒。

**五、分析蓝牙trace**

很多蓝牙问题都需要分析trace，一般来说，分析蓝牙问题，要用到两个trace：

bt\_log和main\_log。

在main\_log中，包括了关于蓝牙java层的trace以及一些稍微层的c与c++层的trace。

在bt\_log中，包括了蓝牙的信令交互，要分析bt\_log,需要数据库，在pixi3\_4中可以在如下路径找到这个数据库BTCatacherDB：

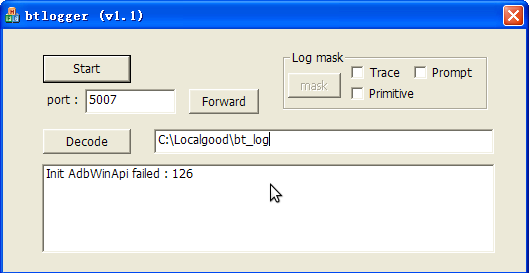
vendor/mediatek/pixi3\_4/artifacts/out/target/product/pixi3\_4/obj/CODEGEN/btcodegen/database/ BTCatacherDB

在给mtk提交E\_sevice时，mtk通常需要根据bt\_log来分析蓝牙的信令交互。在我们平时的开发中，也需要分析蓝牙的信令的交互。有些时候，即使我们没有底层代码，但是我们可以通过分析trace，来加速问题的解决。

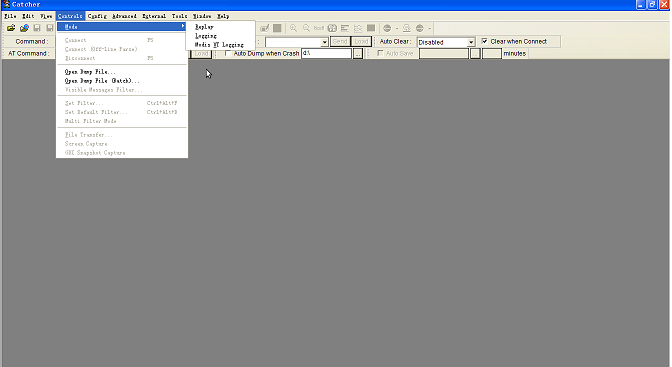
分析bt\_log，除了要catcher外，还需要一个小工具btlogger，这个小工具是MTK发布的。现在介绍一下分析bt\_log的方法。

**分析bt\_log:**

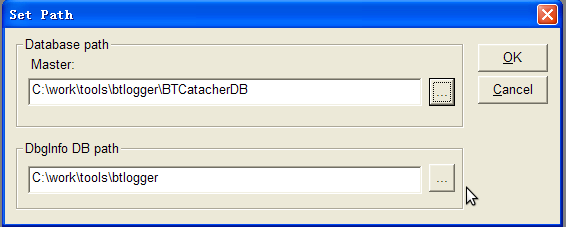
**1.打开工具bt\_logger,在输入框里填入bt\_log的绝对路径**



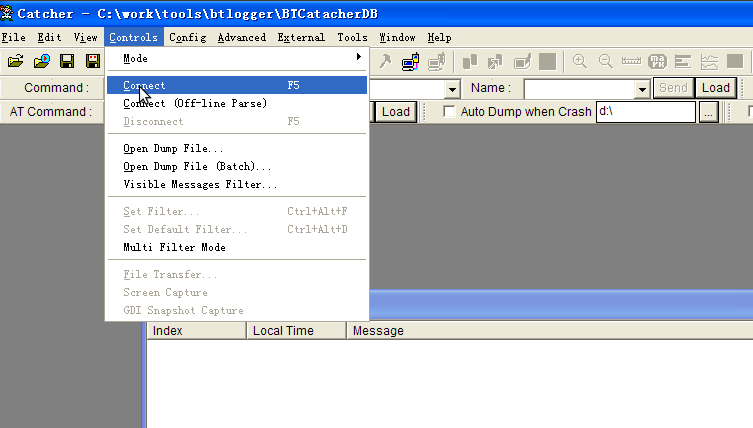
**2．打开catcher，选择：Controls-〉Mode-〉Modis UT Logging**

．

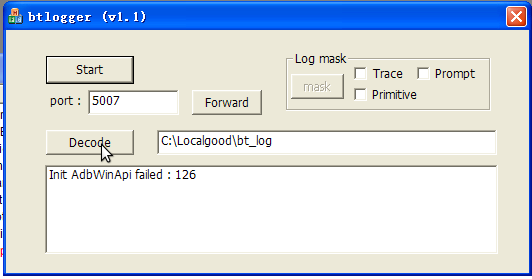
**3．设置数据库，该数据库在工程里有**



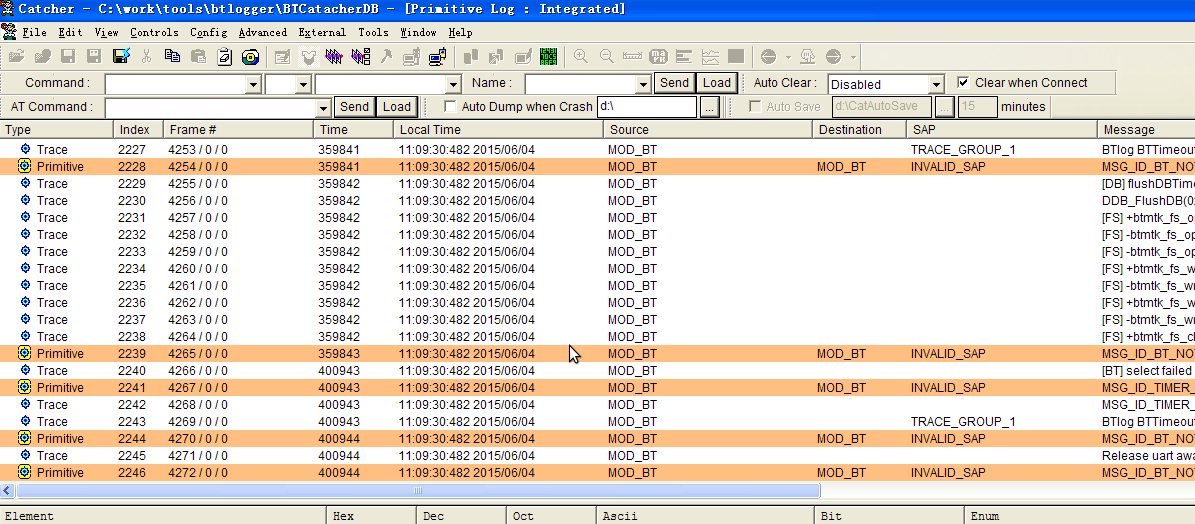
**4．选择Controls->Connect,从而将catcher与bt\_logger相连**



**5．在bt\_logger里点击按钮：Decode，从而执行解析trace的操作**

．

**6．解码后的trace，这些蓝牙trace与功能机上的trace有很多都是一样的。**



**六、蓝牙类库介绍**

本文只对传统蓝牙的类做以介绍，并不涉及BLE（低功耗蓝牙，在蓝牙4.0上引入）。在蓝牙模块里，package是android.bluetooth.里面有如下重要的类：

|  |  |
| --- | --- |
| [BluetoothA2dp](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothA2dp.html) | This class provides the public APIs to control the Bluetooth A2DP profile. |
| [BluetoothAdapter](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothAdapter.html) | Represents the local device Bluetooth adapter. |
| [BluetoothAssignedNumbers](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothAssignedNumbers.html) | Bluetooth Assigned Numbers. |
| [BluetoothClass](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothClass.html) | Represents a Bluetooth class, which describes general characteristics and capabilities of a device. |
| [BluetoothClass.Device](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothClass.Device.html) | Defines all device class constants. |
| [BluetoothClass.Device.Major](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothClass.Device.Major.html) | Defines all major device class constants. |
| [BluetoothClass.Service](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothClass.Service.html) | Defines all service class constants. |
| [BluetoothDevice](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothDevice.html) | Represents a remote Bluetooth device. |
| [BluetoothHeadset](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothHeadset.html) | Public API for controlling the Bluetooth Headset Service. |
| [BluetoothHealth](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothHealth.html) | Public API for Bluetooth Health Profile. |
| [BluetoothHealthAppConfiguration](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothHealthAppConfiguration.html) | The Bluetooth Health Application Configuration that is used in conjunction with the [BluetoothHealth](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothHealth.html) class. |
| [BluetoothHealthCallback](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothHealthCallback.html) | This abstract class is used to implement [BluetoothHealth](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothHealth.html) callbacks. |
| [BluetoothServerSocket](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothServerSocket.html) | A listening Bluetooth socket. |
| [BluetoothSocket](file:///C:\work\doc\docs_low\reference\android\bluetooth\BluetoothSocket.html) | A connected or connecting Bluetooth socket. |

限于篇幅，本文只介绍BluetoothAdapter类，至于其他的类，有兴趣的人可参考andoroid的开发文档。

1. **BluetoothAdapter类基础**

类BluetoothAdapter的格式如下：

Public final class BluetoothAdapter extends Object

BluetoothAdapter代表本地的蓝牙适配器设备，通过这个类可以让用户能执行基本的蓝牙任务。例如初始化设备的搜索、查询可配对的设备、使用一个已知的MAC地址来初始化一个BluetoothDevice类、创建一个BluetoothServerSocket类以监听其他设备对本机连接请求等等。

为了得到这个代表本地蓝牙适配器的BluetoothAdapter类，需要调用静态方法getDefaultAdapter(),这是所有蓝牙动作使用的第一步。当拥有本地适配器以后，用户可以获得一系列的BluetoothDevice对象，这些对象代表所有拥有getBondedDevice()方法的已经匹配的设备；用startDiscovery()方法来开始设备的搜索；或者创建一个BluetoothServerSocket类，通过listenUseingRfcommWithServiceRecord(String ,UUID)方法来监听新来的连接请求。

在这个类里，大部分方法需要BLUETOOTH权限，一些方法同时需要BLUETOOTH\_ADMIN权限。

1. **BluetoothAdapter类的常量**
2. **String ACTION\_DISCOVERY\_FINISHED**

广播事件：本地蓝牙适配器已经完成设备的搜寻过程，需要BLUETOOTH权限接收。

常量值："android.bluetooth.adapter.action.DISCOVERY\_FINISHED"

1. **String ACTION\_DISCOVERY\_STARTED**

广播事件：本地蓝牙适配器已经开始对远程设备的搜寻过程。它通常涉及一个大概12秒的查询扫描过程，紧跟着是一个对每个获取到自身蓝牙名称的新设备的页面扫描。ACTION\_FOUND常量被注册用来通知有新的被搜索到的远程蓝牙设备。、

设备搜索是一个重量级的过程，蓝牙负担很重，当搜索在进行时，用户不能尝试对新的远程蓝牙设备进行连接，同时存在的连接只能获得有限制的带宽，还会带来高等待时间。

在必要的时候，用户可以用cancelDiscovery()方法来取消正在进行的查找进程。

这个ACTION需要BLUETOOTH权限接收。

常量值："android.bluetooth.adapter.action.DISCOVERY\_STARTED"

**3）String ACTION\_ LOCAL\_NAME\_CHANGED**

广播事件：本地蓝牙适配器已经改变了它自己的蓝牙名字，该名字对于

远程蓝牙设备是可见的，它总是包含了一个带有名称的EXTRA\_Local\_NA

ME的附加域。

这个ACTION需要BULETOOTH权限接收

常量值："android.bluetooth.adapter.action.LOCAL\_NAME\_CHANGED"

* 1. **String ACTION\_REQUEST\_DISCORERABLE**

ACTIVITY 事件：展示一个要求能被发现的系统活动。如果蓝牙模块当前未打开，该Action也将请求用户打开蓝牙模块。能被发现模式和SCAN\_MODE\_CONNECTABLE\_DISCOVERABLE等价。

当远程设备执行查找设备进程操作时，这个ACTION允许远程设备发现该蓝牙适配器。从隐私安全考虑，Android系统没有将能被搜寻模式设为默认状态。

该ACTION的发送者可以选择性地运用EXTRA\_DISCOVERY\_DURATION这个附加域去请求能被发现的持续时间。

Android运用onActivityResult（int，int，Intent）回收方法来传递该活动结果的通知。被搜寻的时间（以秒为单位）将通过resultCode值来显示。如果用户拒绝被搜询，或者设备产生了错误，则通过RESULT\_CANCELED值来显示。

每当设备的扫描模式变化的时候，应用程序可以通过ACTION\_SCAN\_MODE\_CHANGED值来监听全局的消息通知。比如，当设备停止被搜寻以后，该变化可以被系统通知给应用程序。

这个ACTION需要BULETOOTH权限

常值值："android.bluetooth.adapter.action.REQUEST\_DISCOVERABLE

"

* 1. **String ACTION\_REQUEST\_ENABLE**

ACTIVITY 事件：展示一个要求用户打开蓝牙的系统activity。当蓝牙模块完成打开工作，或者用户决定不打开蓝牙模块时，这个系统activity将返回。Activity运用onActivityResult（int，int，Intent）来传递该系统activity的结果。如果蓝牙模块被打开，返回的resultCode将是RESULT\_OK;如果用户拒绝打开蓝牙或者产生了错误。返回的resultCode将是RESULT\_CANCELED.

每当蓝牙模块被打开或者关闭，应用程序可以通过ACTION\_STATE\_CHANGED来监听全局的消息通知。

这个需要BULETOOTH权限。

常量值："android.bluetooth.adapter.action.REQUEST\_ENABLE"

* 1. **String ACTION\_SCAN\_MODE\_CHANGED**

广播活动：指名本地蓝牙适配器的蓝牙扫描模式发生了变化。它总是包含EXTRA\_ SCAN\_MODE和EXTRA\_PREVIOUS\_SCAN\_MODE两个附加域，这两个附加域分别包含了过去的和现在的扫描模式。

这个需要BLUETOOTH权限。

常量值： "android.bluetooth.adapter.action.SCAN\_MODE\_CHANGED"

* 1. **String ACTION\_STATE\_CHANGED**

广播活动：指明蓝牙适配器的状态已经改变。例如，蓝牙模块被打开或者被关闭。它总是包含EXTRA\_STATE和EXTRA\_PREVIOUS\_STATE两个附加域。这两个附加域分别包含了本地蓝牙的当前的和过去的状态。

这个需要BLUETOOTH权限。

常量值："android.bluetooth.adapter.action.STATE\_CHANGED"

* 1. **public static final String EXTRA\_CONNECTION\_STATE**

是ACTION\_CONNECTION\_STATE\_CHANGED 的一个附加域，表示了当前蓝牙适配器的连接状态。

* 1. **String EXTRA\_LOCAL\_NAME**

在ACTION\_LOCAL\_NAME\_CHANGED中作为一个附加域，表示了本地蓝牙名字。

**10）String EXTRA\_SCAN\_MODE**

在ACTION\_SCAN\_MODE\_CHANGED中作为一个整形附加域，来表征当前

的扫描模式，可以取的值如下：

SCAN\_MODE\_NONE

SCAN\_MODE\_CONNECTABLE

SCAN\_MODE\_CONNECTABLE\_DISCOVERABLE

**11) int SCAN\_MODE\_CONNECTABLE**

指明本地蓝牙适配器的查询功能失效，但是page扫描功能有效。所以，

本地蓝牙不能被别的远程蓝牙设备发现。不过，对于以前曾经发现过本

远端设备可以连接本地蓝牙。

常量值： 21 (0x00000015)

**12）int SCAN\_MODE\_CONNECTABLE\_DISCOVERABLE**

指明在本地蓝牙适配器中，查询扫描功能和page扫描功能都有效。因此，

本地蓝牙适配器既可以被远程蓝牙设备发现，也可以被其连接。

常量值：23 (0x00000017)

**13）int SCAN\_MODE\_NONE**

指明在本地蓝牙适配器中，查询扫描功能和page扫描功能都失效。因

此，本地蓝牙适配器既不能被远程蓝牙设备发现，也不能被其连接。

**14）String EXTRA\_PREVIOUS\_STATE**

在ACTION\_STATE\_CHANGED中作为一个整形附加域，来表示本地蓝

牙适配器以前的激活状态。

可以取得值如下：

STATE\_OFF

STATE\_TURING\_ON

STATE\_ON

STATE\_TURNING\_OFF

**15) String EXTRA\_STATE**

在ACTION\_STATE\_CHANGED中作为一个整形附加域，来表示本地

蓝牙适配器当前的激活状态。

可以取得值如下：

STATE\_OFF

STATE\_TURNING\_ON

STATE\_ON

STATE\_TURNING\_OFF

**16）int STATE\_OFF**

指明本地蓝牙适配器已经关闭

常量值：10 (0x0000000a)

**17）int STATE\_ON**

指明本地蓝牙适配器已经打开

常量值：12 (0x0000000c)

**18）int STATE\_TURNING\_OFF**

指明本地蓝牙适配器正在被关闭

常量值：13 (0x0000000d)

**19）int STATE\_TURNING\_ON**

指明本地蓝牙适配器正在被打开

常量值：11 (0x0000000b)

1. **BluetoothAdapter类的公共方法**
2. **public Boolean cancelDiscovery（）**

**功能**：取消本地蓝牙设备查找其他设备的操作，需要BLUETOOTH\_ADMIN权限。

因为对于蓝牙适配器而言，查找设备是一个重量级的操作，因此这个方法必须在尝试连接到远程设备前进行调用。查找设备的操作不是由activity控制的，而是作为一个系统服务来运行。因此应用程序应该经常调用这个方法，来终止查找设备的操作，即使这个应用程序没有直接发起查找设备的操作，这样有助于应用程序的顺利执行。

如果本地蓝牙适配器的状态不是STATE\_ON,这个API将返回false。

**返回值：**

成功则返回true，有错误则返回false。

1. **public static Boolean checkBluetoothAddress(String address)**

**功能：**验证蓝牙地址是否有效，里面的字母必须为大写才有效

**参数：**address,字符串形式的蓝牙模块地址。

**返回值：**地址正确则返回true，否则返回false。

1. **public Boolean disable()**

**功能：**关闭本地蓝牙适配器，在用户没有明确的动作表示前，不要使用该方法。这个方法能够友好地停止所有的蓝牙连接、停止蓝牙系统服务、以及对底层的蓝牙硬件进行断电。没有用户的直接同意，蓝牙就不能被disable。

该方法是一个异步调用方法：该方法将马上获得返回值，用户需要通过监听ACTION\_STATE\_CHANGED来获取随后的蓝牙适配器的状态改变通知。如果方法返回true值，则该蓝牙适配器状态会立刻从STATE\_ON转向STATE\_TURNING\_OFF,稍后则会转向STATE\_OFF或者STATE\_ON.如果该方法调用返回false,那么说明系统已经有一个保护蓝牙适配器被关闭的问题存在，比如该蓝牙适配器已经被关闭了。

该方法需要BLUETOOTH\_ADMIN权限。

**返回值：**如果蓝牙适配器的关闭操作已经开启则返回true，如果产生错误则返回

False.

1. **public Boolean enable()**

**功能：**打开本地蓝牙适配器，在用户没有明确的动作表示前，不要使用该方法。该方法将为蓝牙硬件供电，并且启动所有的蓝牙系统服务。在没有用户的直接同意前，不要使用该方法。如果用户为了创建无线连接而需要打开蓝牙模块，则其需要使用ACTION\_REQUEST\_ENABLE，该ACTION会产生一个请求用户允许打开蓝牙的会话界面。

该方法是一个异步调用方法：该方法将马上获得返回值，用户要通过监听ACTION\_STATE\_CHANGED来获取随后的本地蓝牙适配器的状态改变的通知。如果该调用返回true，则该蓝牙适配器的状态会立刻从STATE\_OFF转向STATE\_TURNING\_ON,稍后则会转为STATE\_OFF或者STATE\_ON。如果该调用返回false，那么说明系统已经有一个保护蓝牙适配器被打开的问题存在，比如蓝牙适配器已经被打开。

该方法需要BLUETOOTH\_ADMIN权限。

**返回值：**如果蓝牙适配器的打开进程已经开启则返回true，如果产生错误则返回false。

1. **public String getAddress()**

**功能：**返回本地蓝牙适配器的硬件地址，例如：

00：11：22：AA：BB：CC

该方法需要BLUETOOTH权限。

**返回值**：字符串形式的蓝牙模块地址

1. **public Set<BluetoothDevice> getBondedDevices()**

功能：返回已经与本地蓝牙适配器配对的蓝牙设备的集合。如果蓝牙适配器的状态不是STATE\_ON,则这个API将返回一个空的集合。

该方法需要BLUETOOTH权限。

**返回值**：返回已经与本地蓝牙适配器配对的蓝牙设备集合，如果出错，返回null

1. **public static synchronized BluetoothAdapter getDefaultAdapter()**

**功能：**获取默认本地蓝牙设配器的handle。目前，Android只支持一个蓝牙适配器，但是API可以被扩展为支持多个适配器。该方法总是返回默认的蓝牙适配器。

**返回值**：返回默认的蓝牙适配器，如果蓝牙适配器在该硬件平台不被支持，则返回null。

1. **public String getName()**

**功能：**获取本地蓝牙适配器的蓝牙名称，该名称对于外界蓝牙设备是可见的。

该方法需要BLUETOOTH权限。

**返回值：**本地蓝牙适配器的名称，如果有错则返回null。

1. **public int getProfileConnectionState(int profile)**

**功能**：获取给定的profile的当前连接状态。这个方法用来检测对于一个给定的profile，本地蓝牙适配器与远端蓝牙设备之间的连接状态。这个profile可以是：

HEADTH、HEADSET、A2DP。

**返回值：**返回以下常量之一：

STATE\_DISCONNECTED

STATE\_CONNECTING

STATE\_CONNECTED

STATE\_DISCONNECTING

**10) public BluetoothDevice getRemoteDevice(String address)**

**功能：**根据给定的蓝牙地址获取一个BluetoothDevice对象。合法的

蓝牙硬件地址必须为大写，格式类似于：“00：11：22：33：AA：

BB”。checkBluetoothAddress (String)方法可以用来验证蓝牙地址的

正确性。

**参数：**

Address：合法的蓝牙MAC地址

**异常：**IIIegalArgumentException 如果蓝牙地址不合法。

**11) public int getScanMode()**

**功能：**获取本地蓝牙适配器的当前蓝牙扫描模式，蓝牙扫描模式决定

了本地蓝牙适配器可连接并且/或者可被远端蓝牙设备发现。可能的

返回值如下：

SCAN\_MODE\_NONE

SCAN\_MODE\_CONNECTABLE

SCAN\_MODE\_CONNECTABLE\_DISCOVERABLE

如果蓝牙状态不是STATE\_ON,则返回SCAN\_MODE\_NONE。

该方法需要BLUETOOTH权限。

**返回值：**扫描模式。

**12）public int getState()**

**功能：**获取本地蓝牙适配器的当前状态，可能的返回值如下：

STATE\_OFF

STATE\_TURNING\_ON

STATE\_ON

STATE\_TURNING\_OFF

该方法需要BLUETOOTH权限。

**返回值：**本地蓝牙适配器的当前状态。

**13）public boolean isDiscovering()**

**功能**：如果当前蓝牙适配器正处于搜索进程中，则返回true。设备

搜索是一个重量级过程，当查找正在进行时，用户不能对远程设备

建立新的连接，同时，存在的连接将只能获得有限制的带宽以及产

生高等待时间。用户可以使用cancelDiscovery()来取消正在进行的搜

索操作。

应用程序可以为ACTION\_DISCOVERY\_STARTEDACTION\_DISCOVERY\_

FINISHED进行注册，从而在查找开始或者完成的时候，可以得到

通知。

该方法需要BLUETOOTH权限。

如果本地蓝牙适配器的状态不是STATE\_ON,这个API将返回false。

**返回值：**如果正在搜索，则返回true。  
  **14）public boolean isEnable()**

**功能：**如果蓝牙正处于打开状态并且可用，则返回true。该方法与

getBluetoothState()==STATE\_ON等价。

该方法需要BLUETOOTH权限。

**返回值：**如果本地蓝牙适配器已经打开，则返回true。

**15) public BluetoothServerSocket listenUsingRfcommWithServiceRecord**

**(String name， UUID uuid)**

**功能：**创建一个正在监听的安全的带有服务记录的基于RFCOMM（串

口仿真） 的socket。一个对该socket进行连接的远端设备将被认

证，而对该socket的通信将被加密。使用accept()方法可以获取

从监听BluetoothServerSocket处新来的 连接。系统会分配一个未

被使用的RFCOMM通道来进行监听。

系统也将注册一个服务发现协议（SDP）记录，该记录带有一个包

含了特定的通用唯一识别码（UUID）、服务名称和自动分配通道的

本地SDP服务器。远端 蓝牙设备可以用相同的UUID来查询SDP

服务器，并搜寻要连接到哪个通道。如果该Socket已经关闭，或

者该应用程序已经异常退出，则SDP记录会被移除。

使用createRfcommSocketToServiceRecord(UUID)可以从另一个使用

相同UUID的设备来连接到这个socket。

该方法需要BLUETOOTH权限。

**参数：**

name：SDP记录的服务名称

uuid：SDP记录对应的UUID

**返回值：**一个正在监听的BluetoothServerSocket。

**异常**：IOException，表示产生了错误，比如蓝牙设备不可用或者没有

足够的许可或者通道被占用。

**16）public boolean setName(String name)**

**功能**：设置本地蓝牙适配器的昵称，这个昵称对于外界蓝牙设备而言

是可见的。

合法的蓝牙名称最多拥有248位UTF-8字符。

该方法需要BLUETOOTH\_ADMIN权限。

**参数：**

name:一个合法的蓝牙名称

**返回值：**如果该名称已经被成功设定，则返回true，否则返回false

**17）public boolean startDiscovery()**

**功能：**开始对远端设备进行搜索。这个操作通常涉及一个大概12s的

查询扫描过程，紧跟着的是一个page扫描以获得新设备的蓝牙名字。

应用程序可以注册 ACTION\_FOUND以获得远端蓝牙设备已经被搜

索到的通知。

这个搜索操作只能搜索那些可被发现的设备，所以如果想要搜索某个

蓝牙设备， 那就一定要把这个要被搜索的蓝牙设备设置为可被发现

的。

如果本地蓝牙的状态不是STATE\_ON,这个API将返回false。

**返回值：**成功返回true，错误返回false。

**八、结束语**

对于蓝牙模块来说，无论是功能机项目还是智能机项目，UI界面比起其他模块 ，并不复杂。但是蓝牙的SPEC（协议）的内容很多，而且蓝牙PROFILE越来越多，蓝牙工程师要持续学习，持续跟进。而且我们看到的代码大多是上层应用代码，至于底层的功能代码，在android L平台我们是看不到的。所以，一些功能性的bug只能通过系统供应商MTK提供patch来解决 。不过，即使对于蓝牙的上层应用来说，代码量也是比较大的。遇到bug需要结合trace认真跟踪，这样可以加快解决bug的速度。