蓝牙开发

首先，要操作蓝牙，先要在AndroidManifest.xml里加入权限

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

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

然后，看下api，Android所有关于蓝牙开发的类都在android.bluetooth包下，只有8个类

**1.BluetoothAdapter**

顾名思义，蓝牙适配器，直到我们建立bluetoothSocket连接之前，都要不断操作它BluetoothAdapter里的方法很多，常用的有以下几个：

cancelDiscovery() 根据字面意思，是取消发现，也就是说当我们正在搜索设备的时候调用这个方法将不再继续搜索

      disable()关闭蓝牙

      enable()打开蓝牙，这个方法打开蓝牙不会弹出提示，更多的时候我们需要问下用户是否打开，一下这两行代码同样是打开蓝牙，不过会提示用户**：**

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

startActivityForResult(enabler,reCode);//同startActivity(enabler);

      getAddress()获取本地蓝牙地址

      getDefaultAdapter()获取默认BluetoothAdapter，实际上，也只有这一种方法获取BluetoothAdapter

      getName()获取本地蓝牙名称

      getRemoteDevice(String address)根据蓝牙地址获取远程蓝牙设备

      getState()获取本地蓝牙适配器当前状态（感觉可能调试的时候更需要）

      isDiscovering()判断当前是否正在查找设备，是返回true

      isEnabled()判断蓝牙是否打开，已打开返回true，否则，返回false

     listenUsingRfcommWithServiceRecord(String name,UUID uuid)根据名称，UUID创建并返回BluetoothServerSocket，这是创建BluetoothSocket服务器端的第一步

      startDiscovery()开始搜索，这是搜索的第一步

**2.BluetoothDevice**

看名字就知道，这个类描述了一个蓝牙设备

      createRfcommSocketToServiceRecord(UUIDuuid)根据UUID创建并返回一个BluetoothSocket

getState() 蓝牙状态这里要说一下，只有在 BluetoothAdapter.STATE\_ON 状态下才可以监听，具体可以看andrid api;

这个方法也是我们获取BluetoothDevice的目的——创建BluetoothSocket  
这个类其他的方法，如getAddress(),getName(),同BluetoothAdapter

**3.BluetoothServerSocket**

如果去除了Bluetooth相信大家一定再熟悉不过了，既然是Socket，方法就应该都差不多，这个类一种只有三个方法两个重载的accept(),accept(inttimeout)两者的区别在于后面的方法指定了过时时间，需要注意的是，执行这两个方法的时候，直到接收到了客户端的请求（或是过期之后），都会阻塞线程，应该放在新线程里运行！

还有一点需要注意的是，这两个方法都返回一个BluetoothSocket，最后的连接也是服务器端与客户端的两个BluetoothSocket的连接

      close()这个就不用说了吧，翻译一下——关闭！

**4.BluetoothSocket**

跟BluetoothServerSocket相对，是客户端一共5个方法，不出意外，都会用到

      close(),关闭

      connect()连接

      getInptuStream()获取输入流

      getOutputStream()获取输出流

      getRemoteDevice()获取远程设备，这里指的是获取bluetoothSocket指定连接的那个远程蓝牙设备

1、获取本地蓝牙适配器

      BluetoothAdapter  
mAdapter= BluetoothAdapter.getDefaultAdapter();

 2、打开蓝牙

      if(!mAdapter.isEnabled()){

//弹出对话框提示用户是后打开

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

startActivityForResult(enabler, REQUEST\_ENABLE);

      //不做提示，强行打开

      // mAdapter.enable();

}

3、搜索设备  
   1)刚才说过了mAdapter.startDiscovery()

是第一步,可以你会发现没有返回的蓝牙设备，怎么知道查找到了呢？向下看，不要急

2)定义BroadcastReceiver,关于BroadcastReceiver不多讲了，不是今天的讨论内容，代码如下

**[java]** [view plain](http://blog.csdn.net/q610098308/article/details/45248423) [copy](http://blog.csdn.net/q610098308/article/details/45248423)

[print?](http://blog.csdn.net/q610098308/article/details/45248423)

1. BroadcastReceiver mReceiver = **new** BroadcastReceiver() {
2. **public** **void** onReceive(Context context, Intent intent) {
3. String action = intent.getAction();
4. //找到设备
5. **if** (BluetoothDevice.ACTION\_FOUND.equals(action)) {
6. BluetoothDevice device = intent
7. .getParcelableExtra(BluetoothDevice.EXTRA\_DEVICE);
9. **if** (device.getBondState() != BluetoothDevice.BOND\_BONDED) {
11. Log.v(TAG, "find device:" + device.getName()
12. + device.getAddress());
13. }
14. }
15. //搜索完成
16. **else** **if** (BluetoothAdapter.ACTION\_DISCOVERY\_FINISHED
17. .equals(action)) {
18. setTitle("搜索完成");
19. **if** (mNewDevicesAdapter.getCount() == 0) {
20. Log.v(TAG, "find over");
21. }
22. }
23. }
24. };

  这样，没当查找到新设备或是搜索完成，相应的操作都在上段代码的两个if里执行了，不过前提是你要先注册

BroadcastReceiver，具体代码如下

**[java]** [view plain](http://blog.csdn.net/q610098308/article/details/45248423) [copy](http://blog.csdn.net/q610098308/article/details/45248423)

[print?](http://blog.csdn.net/q610098308/article/details/45248423)

1. IntentFilter filter = **new** IntentFilter(BluetoothDevice.ACTION\_FOUND);
2. registerReceiver(mReceiver, filter);
3. filter = **new** IntentFilter(BluetoothAdapter.ACTION\_DISCOVERY\_FINISHED);
4. registerReceiver(mReceiver, filter)<span style="font-family:SimSun;">;</span>

（这段代码，一般写在onCreate（）里..）  4、建立连接，首先Android sdk（2.0以上版本）支持的蓝牙连接是通过BluetoothSocket建立连接（说的不对请高人指正），服务器端（BluetoothServerSocket）和客户端（BluetoothSocket）需指定同样的UUID，才能建立连接，因为建立连接的方法会阻塞线程，所以服务器端和客户端都应启动新线程连接

1）服务器端：

//UUID格式一般是"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"可到

        //http://www.uuidgenerator.com 申请

BluetoothServerSocket serverSocket = mAdapter. listenUsingRfcommWithServiceRecord(serverSocketName,UUID);  
serverSocket.accept();

2)客户端：  
//还记得我们刚才在BroadcastReceiver获取了BLuetoothDevice么？  
BluetoothSocket clienSocket=dcvice. createRfcommSocketToServiceRecord(UUID);  
clienSocket.connect();

5、数据传递，通过以上操作，就已经建立的BluetoothSocket连接了，数据传递无非是通过流的形式  
1）获取流  
inputStream = socket.getInputStream();  
outputStream = socket.getOutputStream();2）写出、读入  
这是基础的东西，在这就不多赘述了  
终于写完了，这是我这两天的学习经验，希望对有蓝牙需求的朋友有所

补充一下，使设备能够被搜索

Intent enabler = new Intent(BluetoothAdapter.ACTION\_REQUEST\_DISCOVERABLE);

startActivityForResult(enabler,REQUEST\_DISCOVERABLE);

在做android蓝牙串口连接的时候一般会使用

  BluetoothSocket tmp = null;  
            // Get a BluetoothSocket for a connection with the  
            // given BluetoothDevice  
            try {  
              tmp = device.createRfcommSocketToServiceRecord(MY\_UUID);  
           } catch (IOException e) {  
                Log.e(TAG, "create() failed", e);  
          }

然后是tmp赋给BluetoothSocket,接着调用connect方法进行蓝牙设备的连接。

可是 BluetoothSocket 的connect方法本身就会报很多异常错误。

以下根据对蓝牙开发的一点研究可通过以下方法解决：

方法1.先进行蓝牙自动配对，配对成功，通过UUID获得BluetoothSocket,然后执行connect()方法。

方法2.通过UUID获得BluetoothSocket,然后先根据mDevice.getBondState()进行判断是否需要配对，最后执行connnect()方法。

[制代码](javascript:void(0);)

/\*\* \* <p> \* 蓝牙连接线程 \* </p> \* \* @author lsw \* \*/ private class ConnectThread extends Thread { String macAddress = ""; public ConnectThread(String mac) { macAddress = mac; } public void run() { connecting = true; connected = false; if(mBluetoothAdapter == null){ mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter(); } mBluetoothDevice = mBluetoothAdapter.getRemoteDevice(macAddress); mBluetoothAdapter.cancelDiscovery(); //initSocket(); try { socket = mBluetoothDevice.createRfcommSocketToServiceRecord(uuid); } catch (IOException e) { // TODO Auto-generated catch block //e.printStackTrace(); Log.e(TAG, "Socket", e); } //adapter.cancelDiscovery(); while (!connected && connetTime <= 10) { connectDevice(); } // 重置ConnectThread //synchronized (BluetoothService.this) { //ConnectThread = null; //} } public void cancel() { try { socket.close(); socket = null; } catch (Exception e) { e.printStackTrace(); } finally { connecting = false; } } }

[制代码](javascript:void(0);)

　　接下来是调用的连接设备方法connectDevice():

[制代码](javascript:void(0);)

protected void connectDevice() { try { // 连接建立之前的先配对 if (mBluetoothDevice.getBondState() == BluetoothDevice.BOND\_NONE) { Method creMethod = BluetoothDevice.class .getMethod("createBond"); Log.e("TAG", "开始配对"); creMethod.invoke(mBluetoothDevice); } else { } } catch (Exception e) { // TODO: handle exception //DisplayMessage("无法配对！"); e.printStackTrace(); } mBluetoothAdapter.cancelDiscovery(); try { socket.connect(); //DisplayMessage("连接成功!"); //connetTime++; connected = true; } catch (IOException e) { // TODO: handle exception //DisplayMessage("连接失败！"); connetTime++; connected = false; try { socket.close(); socket = null; } catch (IOException e2) { // TODO: handle exception Log.e(TAG, "Cannot close connection when connection failed"); } } finally { connecting = false; } }

[制代码](javascript:void(0);)

方法3.利用反射通过端口获得BluetoothSocket,然后执行connect()方法。

[制代码](javascript:void(0);)

/\*\* \* <p> \* 蓝牙连接线程 \* </p> \* \* @author lsw \* \*/ private class ConnectThread extends Thread { String macAddress = ""; public ConnectThread(String mac) { macAddress = mac; } public void run() { connecting = true; connected = false; if(mBluetoothAdapter == null){ mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter(); } mBluetoothDevice = mBluetoothAdapter.getRemoteDevice(macAddress); mBluetoothAdapter.cancelDiscovery(); initSocket(); //adapter.cancelDiscovery(); while (!connected && connetTime <= 10) { try { socket.connect(); connected = true; } catch (IOException e1) { connetTime++; connected = false; // 关闭 socket try { socket.close(); socket = null; } catch (IOException e2) { //TODO: handle exception Log.e(TAG, "Socket", e2); } } finally { connecting = false; } //connectDevice(); } // 重置ConnectThread //synchronized (BluetoothService.this) { //ConnectThread = null; //} } public void cancel() { try { socket.close(); socket = null; } catch (Exception e) { e.printStackTrace(); } finally { connecting = false; } } }

[制代码](javascript:void(0);)

　　接下来是初始化并得到BluetoothSocket的方法

[制代码](javascript:void(0);)

/\*\* \* 取得BluetoothSocket \*/ private void initSocket() { BluetoothSocket temp = null; try { Method m = mBluetoothDevice.getClass().getMethod( "createRfcommSocket", new Class[] { int.class }); temp = (BluetoothSocket) m.invoke(mBluetoothDevice, 1);//这里端口为1 } catch (SecurityException e) { e.printStackTrace(); } catch (NoSuchMethodException e) { e.printStackTrace(); } catch (IllegalArgumentException e) { e.printStackTrace(); } catch (IllegalAccessException e) { e.printStackTrace(); } catch (InvocationTargetException e) { e.printStackTrace(); } socket = temp; }

[制代码](javascript:void(0);)

要点：1.蓝牙配对和连接是两回事，不可混为一谈。

　　　2.蓝牙串口连接可通过端口 （1-30）和UUID两种方法进行操作。

　　　3.通过UUID进行蓝牙连接最好先进行配对操作。

1. 蓝牙连接，不需要配对

由于被曾经使用蓝牙的思路所误导，一直以为使用蓝牙是必须一个配对的过程的。实际上不是这样。搜索到设备以后，直接去connect设备而不去配对，目前在我这里是没问题的，搜索到设备以后，可以直接用一下代码进行连接：

final String SPP\_UUID = "00001101-0000-1000-8000-00805F9B34FB"; UUID uuid = UUID.fromString(SPP\_UUID); BluetoothSocket **socket**; **socket** = device.createInsecureRfcommSocketToServiceRecord(uuid); adapter.cancelDiscovery(); **socket**.**connect**(); 这里的UUID是比较好用的一个，设备都可以识别。

1. startDiscovey有可能启动失败

一般程序中会有两步：开启蓝牙、开始寻找设备。之前我写的代码是用户按下按钮就直接顺序执行这两步，导致的结果就是经常性搜索失败。仔细看了一下API，发现adapter.startDiscovery()函数是有一个boolean返回值的，也就是说如果启动失败就返回false。这就解释了为什么会启动失败了：顺序执行了开启蓝牙-寻找设备的步骤，但是由于蓝牙还没有完全打开，就开始寻找设备，导致寻找失败。于是最后我把代码改成了这样，问题解决：

adapter = BluetoothAdapter.getDefaultAdapter(); **if** (adapter == **null**) { *// 设备不支持蓝牙*  } *// 打开蓝牙*  **if** (!adapter.isEnabled()) { adapter.enable(); adapter.cancelDiscovery(); } *// 寻找蓝牙设备，android会将查找到的设备以广播形式发出去*  **while** (!adapter.startDiscovery()){ Log.e("BlueTooth", "尝试失败"); **try** { Thread.sleep(100); } **catch** (InterruptedException e) { e.printStackTrace(); } }

1. 接收数据转换

使用socket.getInputStream接收到的数据是字节流，这样的数据是没法分析的。又由于一般来说厂家给的协议都是类似于"FA 22 89 D0"这样的十六进制数据，所以很多情况需要一个byte转十六进制String的函数：

**public** **static** String **bytesToHex**(**byte**[] bytes) { **char**[] hexChars = **new** **char**[bytes.length \* 2]; **for** ( **int** j = 0; j < bytes.length; j++ ) { **int** v = bytes[j] & 0xFF; hexChars[j \* 2] = hexArray[v >>> 4]; hexChars[j \* 2 + 1] = hexArray[v & 0x0F]; } **return** **new** String(hexChars);}

**1. 使用蓝牙的响应权限**

复制代码代码如下:

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

**2. 配置本机蓝牙模块**

在这里首先要了解对蓝牙操作一个核心类BluetoothAdapter

复制代码代码如下:

BluetoothAdapter adapter = BluetoothAdapter.getDefaultAdapter();  
//直接打开系统的蓝牙设置面板  
Intent intent = new Intent(BluetoothAdapter.ACTION\_REQUEST\_ENABLE);  
startActivityForResult(intent, 0x1);  
//直接打开蓝牙  
adapter.enable();  
//关闭蓝牙  
adapter.disable();  
//打开本机的蓝牙发现功能（默认打开120秒，可以将时间最多延长至300秒）  
Intent discoveryIntent = new Intent(BluetoothAdapter.ACTION\_REQUEST\_DISCOVERABLE);  
discoverableIntent.putExtra(BluetoothAdapter.EXTRA\_DISCOVERABLE\_DURATION, 300);//设置持续时间（最多300秒）

**3.搜索蓝牙设备**

使用BluetoothAdapter的startDiscovery()方法来搜索蓝牙设备

startDiscovery()方法是一个异步方法，调用后会立即返回。该方法会进行对其他蓝牙设备的搜索，该过程会持续12秒。该方法调用后，搜索过程实际上是在一个System Service中进行的，所以可以调用cancelDiscovery()方法来停止搜索（该方法可以在未执行discovery请求时调用）。

请求Discovery后，系统开始搜索蓝牙设备，在这个过程中，系统会发送以下三个广播：

ACTION\_DISCOVERY\_START：开始搜索

ACTION\_DISCOVERY\_FINISHED：搜索结束

ACTION\_FOUND：找到设备，这个Intent中包含两个extra fields：EXTRA\_DEVICE和EXTRA\_CLASS，分别包含BluetooDevice和BluetoothClass。

我们可以自己注册相应的BroadcastReceiver来接收响应的广播，以便实现某些功能

复制代码代码如下:

// 创建一个接收ACTION\_FOUND广播的BroadcastReceiver  
private final BroadcastReceiver mReceiver = new BroadcastReceiver() {  
    public void onReceive(Context context, Intent intent) {  
        String action = intent.getAction();  
        // 发现设备  
        if (BluetoothDevice.ACTION\_FOUND.equals(action)) {  
            // 从Intent中获取设备对象  
            BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA\_DEVICE);  
            // 将设备名称和地址放入array adapter，以便在ListView中显示  
            mArrayAdapter.add(device.getName() + "\n" + device.getAddress());  
        }  
    }  
};  
// 注册BroadcastReceiver  
IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION\_FOUND);  
registerReceiver(mReceiver, filter); // 不要忘了之后解除绑定

**4. 蓝牙Socket通信**

如果打算建议两个蓝牙设备之间的连接，则必须实现服务器端与客户端的机制。当两个设备在同一个RFCOMM channel下分别拥有一个连接的BluetoothSocket，这两个设备才可以说是建立了连接。

服务器设备与客户端设备获取BluetoothSocket的途径是不同的。服务器设备是通过accepted一个incoming connection来获取的，而客户端设备则是通过打开一个到服务器的RFCOMM channel来获取的。

**服务器端的实现**

通过调用BluetoothAdapter的listenUsingRfcommWithServiceRecord(String, UUID)方法来获取BluetoothServerSocket（UUID用于客户端与服务器端之间的配对）

调用BluetoothServerSocket的accept()方法监听连接请求，如果收到请求，则返回一个BluetoothSocket实例（此方法为block方法，应置于新线程中）

如果不想在accept其他的连接，则调用BluetoothServerSocket的close()方法释放资源（调用该方法后，之前获得的BluetoothSocket实例并没有close。但由于RFCOMM一个时刻只允许在一条channel中有一个连接，则一般在accept一个连接后，便close掉BluetoothServerSocket）

复制代码代码如下:

private class AcceptThread extends Thread {  
    private final BluetoothServerSocket mmServerSocket;

    public AcceptThread() {  
        // Use a temporary object that is later assigned to mmServerSocket,  
        // because mmServerSocket is final  
        BluetoothServerSocket tmp = null;  
        try {  
            // MY\_UUID is the app's UUID string, also used by the client code  
            tmp = mBluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME, MY\_UUID);  
        } catch (IOException e) { }  
        mmServerSocket = tmp;  
    }

    public void run() {  
        BluetoothSocket socket = null;  
        // Keep listening until exception occurs or a socket is returned  
        while (true) {  
            try {  
                socket = mmServerSocket.accept();  
            } catch (IOException e) {  
                break;  
            }  
            // If a connection was accepted  
            if (socket != null) {  
                // Do work to manage the connection (in a separate thread)  
                manageConnectedSocket(socket);  
                mmServerSocket.close();  
                break;  
            }  
        }  
    }

    /\*\* Will cancel the listening socket, and cause the thread to finish \*/  
    public void cancel() {  
        try {  
            mmServerSocket.close();  
        } catch (IOException e) { }  
    }  
}

**客户端的实现**通过搜索得到服务器端的BluetoothService

调用BluetoothService的listenUsingRfcommWithServiceRecord(String, UUID)方法获取BluetoothSocket（该UUID应该同于服务器端的UUID）

调用BluetoothSocket的connect()方法（该方法为block方法），如果UUID同服务器端的UUID匹配，并且连接被服务器端accept，则connect()方法返回

注意：在调用connect()方法之前，应当确定当前没有搜索设备，否则连接会变得非常慢并且容易失败

复制代码代码如下:

private class ConnectThread extends Thread {  
    private final BluetoothSocket mmSocket;  
    private final BluetoothDevice mmDevice;

    public ConnectThread(BluetoothDevice device) {  
        // Use a temporary object that is later assigned to mmSocket,  
        // because mmSocket is final  
        BluetoothSocket tmp = null;  
        mmDevice = device;

        // Get a BluetoothSocket to connect with the given BluetoothDevice  
        try {  
            // MY\_UUID is the app's UUID string, also used by the server code  
            tmp = device.createRfcommSocketToServiceRecord(MY\_UUID);  
        } catch (IOException e) { }  
        mmSocket = tmp;  
    }

    public void run() {  
        // Cancel discovery because it will slow down the connection  
        mBluetoothAdapter.cancelDiscovery();

        try {  
            // Connect the device through the socket. This will block  
            // until it succeeds or throws an exception  
            mmSocket.connect();  
        } catch (IOException connectException) {  
            // Unable to connect; close the socket and get out  
            try {  
                mmSocket.close();  
            } catch (IOException closeException) { }  
            return;  
        }

        // Do work to manage the connection (in a separate thread)  
        manageConnectedSocket(mmSocket);  
    }

    /\*\* Will cancel an in-progress connection, and close the socket \*/  
    public void cancel() {  
        try {  
            mmSocket.close();  
        } catch (IOException e) { }  
    }  
}

连接管理（数据通信）   
分别通过BluetoothSocket的getInputStream()和getOutputStream()方法获取InputStream和OutputStream

使用read(bytes[])和write(bytes[])方法分别进行读写操作

注意：read(bytes[])方法会一直block，知道从流中读取到信息，而write(bytes[])方法并不是经常的block（比如在另一设备没有及时read或者中间缓冲区已满的情况下，write方法会block）

复制代码代码如下:

private class ConnectedThread extends Thread {  
    private final BluetoothSocket mmSocket;  
    private final InputStream mmInStream;  
    private final OutputStream mmOutStream;

    public ConnectedThread(BluetoothSocket socket) {  
        mmSocket = socket;  
        InputStream tmpIn = null;  
        OutputStream tmpOut = null;

        // Get the input and output streams, using temp objects because  
        // member streams are final  
        try {  
            tmpIn = socket.getInputStream();  
            tmpOut = socket.getOutputStream();  
        } catch (IOException e) { }

        mmInStream = tmpIn;  
        mmOutStream = tmpOut;  
    }

    public void run() {  
        byte[] buffer = new byte[1024];  // buffer store for the stream  
        int bytes; // bytes returned from read()

        // Keep listening to the InputStream until an exception occurs  
        while (true) {  
            try {  
                // Read from the InputStream  
                bytes = mmInStream.read(buffer);  
                // Send the obtained bytes to the UI Activity  
                mHandler.obtainMessage(MESSAGE\_READ, bytes, -1, buffer)  
                        .sendToTarget();  
            } catch (IOException e) {  
                break;  
            }  
        }  
    }

    /\* Call this from the main Activity to send data to the remote device \*/  
    public void write(byte[] bytes) {  
        try {  
            mmOutStream.write(bytes);  
        } catch (IOException e) { }  
    }

    /\* Call this from the main Activity to shutdown the connection \*/  
    public void cancel() {  
        try {  
            mmSocket.close();  
        } catch (IOException e) { }  
    }  
}

引用资料：Android官方SDK、《Android/OPhone完全开发讲义》

android4.0蓝牙使能的详细解析 本文详细分析了android4.0 中蓝牙使能的过程，相比较android2.3，4.0中的蓝牙最大的差别在于UI上on/off的伪开关。在android4.0中加入了 adapter的状态机。所谓的状态机就类似于状态转换图，在一个状态收到某个特定的命令会变成另外一个状态，不同的命令可以跳转到不同的状态（当然也有 可能到同一状态）。adapter的初始状态为poweroff，在android系统启动的时候会进入warmup状态，同时会进行UUID的add， 该操作会引起propertychanged的UUID signal，该signal会使得状态从warmup变换到hotoff状态。因此在UI端off时其实adapter已经处于hotoff状态而不是 poweroff状态。这一点是很关键的。在正文中，我会从假如我不知道这些开始来描绘整个使能的过程。  
正文：  
毫无疑问，bluetooth的打开是在Settings中进行的操作。因此，冤有头，债有主，我们来到了Settings.java中，果然发现了相关的代码如下：  
mBluetoothEnabler =new BluetoothEnabler(context, new Switch(context));  
于是，我们得以进入真正的蓝牙操作的殿堂，好好进去看看吧。  
1、BluetoothEnabler的构造函数  
    public BluetoothEnabler(Context context,Switch switch\_) {  
        mContext = context;  
        mSwitch = switch\_;  
//很简单了，去调用一个LocalBluetoothManager类的getInstance，其实会构造该类的  
        LocalBluetoothManager manager =LocalBluetoothManager.getInstance(context);  
        if (manager == null) {  
            // Bluetooth is not supported  
            mLocalAdapter = null;  
            mSwitch.setEnabled(false);  
        } else {  
//构造成功后，通过manager得到bluetooth的adapter  
            mLocalAdapter =manager.getBluetoothAdapter();  
        }  
//同时新建一个intent，用于接收ACTION\_STATE\_CHANGED  
        mIntentFilter = newIntentFilter(BluetoothAdapter.ACTION\_STATE\_CHANGED);  
    }  
2、LocalBluetoothManager类的getInstance  
     public static synchronizedLocalBluetoothManager getInstance(Context context) {  
        if (sInstance == null) {  
//2.1同样的，这个会去调用LocalBluetoothAdapter的getInstance，也会构造该类  
            LocalBluetoothAdapter adapter =LocalBluetoothAdapter.getInstance();  
            if (adapter == null) {  
                return null;  
            }  
            // This will be around as long asthis process is  
            Context appContext =context.getApplicationContext();  
//2.2构造LocalBluetoothManager类  
            sInstance = newLocalBluetoothManager(adapter, appContext);  
        }  
        return sInstance;  
    }  
2.1LocalBluetoothAdapter的getInstance  
    static synchronized LocalBluetoothAdaptergetInstance() {  
        if (sInstance == null) {  
//2.1.1通过BluetoothAdapter得到DefaultAdapter  
            BluetoothAdapter adapter =BluetoothAdapter.getDefaultAdapter();  
            if (adapter != null) {  
//2.1.2若有该DefaultAdapter，则构造LocalBluetoothAdapter  
                sInstance = newLocalBluetoothAdapter(adapter);  
            }  
        }  
        return sInstance;  
    }  
2.1.1BluetoothAdapter得到DefaultAdapter  
    public static synchronized BluetoothAdaptergetDefaultAdapter() {  
        if (sAdapter == null) {  
            IBinder b =ServiceManager.getService(BluetoothAdapter.BLUETOOTH\_SERVICE);  
            if (b != null) {  
                IBluetooth service =IBluetooth.Stub.asInterface(b);  
                sAdapter = newBluetoothAdapter(service);  
            }  
        }  
        return sAdapter;  
    }  
2.1.2构造LocalBluetoothAdapter  
//其实就是 mAdapter的初始化而已  
    privateLocalBluetoothAdapter(BluetoothAdapter adapter) {  
        mAdapter = adapter;  
    }  
2.2构造LocalBluetoothManager类  
//管理本地蓝牙类，用来在蓝牙API子类上面再封装一个接口  
    privateLocalBluetoothManager(LocalBluetoothAdapter adapter, Context context) {  
        mContext = context;  
//mLocalAdapter初始化为DefaultAdapter中得到的值  
mLocalAdapter= adapter;  
//构造CachedBluetoothDeviceManager，用来管理远程蓝牙设备  
        mCachedDeviceManager = newCachedBluetoothDeviceManager(context);  
//2.2.1构建BluetoothEventManager，该类是用来管理广播消息和回调函数的，即分发不同的消息去对UI进行处理  
        mEventManager = newBluetoothEventManager(mLocalAdapter,  
                mCachedDeviceManager, context);  
//2.2.2该类提供对不同LocalBluetoothProfile object的访问  
        mProfileManager = newLocalBluetoothProfileManager(context,  
                mLocalAdapter,mCachedDeviceManager, mEventManager);  
    }  
2.2.1构建BluetoothEventManager  
    BluetoothEventManager(LocalBluetoothAdapteradapter,  
            CachedBluetoothDeviceManagerdeviceManager, Context context) {  
        mLocalAdapter = adapter;  
        mDeviceManager = deviceManager;  
//创建两个IntentFilter  
        mAdapterIntentFilter = newIntentFilter();  
//这里没有对mProfileIntentFilter进行初始化，这个在LocalBluetoothProfileManager的addProfile中实现  
        mProfileIntentFilter = newIntentFilter();  
//创建一个Handler的Hash表  
        mHandlerMap = new HashMap<String,Handler>();  
        mContext = context;  
//注册对adapter和Device的几个广播消息的处理回调函数  
//add action到mAdapterIntentFilter  
        // Bluetooth on/off broadcasts  
       addHandler(BluetoothAdapter.ACTION\_STATE\_CHANGED, newAdapterStateChangedHandler());  
        // Discovery broadcasts  
       addHandler(BluetoothAdapter.ACTION\_DISCOVERY\_STARTED, newScanningStateChangedHandler(true));  
       addHandler(BluetoothAdapter.ACTION\_DISCOVERY\_FINISHED, newScanningStateChangedHandler(false));  
       addHandler(BluetoothDevice.ACTION\_FOUND, new DeviceFoundHandler());  
       addHandler(BluetoothDevice.ACTION\_DISAPPEARED, newDeviceDisappearedHandler());  
       addHandler(BluetoothDevice.ACTION\_NAME\_CHANGED, newNameChangedHandler());  
        // Pairing broadcasts  
       addHandler(BluetoothDevice.ACTION\_BOND\_STATE\_CHANGED, newBondStateChangedHandler());  
       addHandler(BluetoothDevice.ACTION\_PAIRING\_CANCEL, newPairingCancelHandler());  
        // Fine-grained state broadcasts  
       addHandler(BluetoothDevice.ACTION\_CLASS\_CHANGED, newClassChangedHandler());  
        addHandler(BluetoothDevice.ACTION\_UUID,new UuidChangedHandler());  
        // Dock event broadcasts  
        addHandler(Intent.ACTION\_DOCK\_EVENT,new DockEventHandler());  
//mAdapterIntentFilter的接收处理函数  
       mContext.registerReceiver(mBroadcastReceiver, mAdapterIntentFilter);  
    }  
2.2.2构造LocalBluetoothProfileManager类  
    LocalBluetoothProfileManager(Contextcontext,  
            LocalBluetoothAdapter adapter,  
            CachedBluetoothDeviceManagerdeviceManager,  
            BluetoothEventManager eventManager){  
        mContext = context;  
//各个类之间进行关联  
        mLocalAdapter = adapter;  
        mDeviceManager = deviceManager;  
        mEventManager = eventManager;  
        // pass this reference to adapter andevent manager (circular dependency)  
        mLocalAdapter.setProfileManager(this);  
        mEventManager.setProfileManager(this);  
        ParcelUuid[] uuids =adapter.getUuids();  
        // uuids may be null if Bluetooth isturned off  
        if (uuids != null) {  
//假如已经有了uuid，根据uuid来add并new对应的profile，只针对A2DP,HFP,HSP,OPP四个profile，HID和PAN在下面，每次都add  
            updateLocalProfiles(uuids);  
        }  
        // Always add HID and PAN profiles  
//加入HID和PAN两个profile  
        mHidProfile = new HidProfile(context,mLocalAdapter);  
        addProfile(mHidProfile,HidProfile.NAME,  
               BluetoothInputDevice.ACTION\_CONNECTION\_STATE\_CHANGED);  
        mPanProfile = new PanProfile(context);  
        addPanProfile(mPanProfile,PanProfile.NAME,  
               BluetoothPan.ACTION\_CONNECTION\_STATE\_CHANGED);  
        Log.d(TAG,"LocalBluetoothProfileManager construction complete");  
    }  
好吧，其实我们被骗了，刚刚只是一个路引，不是真正的操作，真正的操作向来都是从你滑动界面那个on/off键开始的，因此我们决定把这个键的处理给揪出来。在Settings界面上一共就只有两个on/off键，一个是wifi，另一个就是蓝牙了，我们从这个代码入手：  
                case HEADER\_TYPE\_SWITCH:  
//其实写这个代码的人也比较心虚，假如switch多一点，下面就要重写了  
                    // Would need a differenttreatment if the main menu had more switches  
                    if (header.id ==R.id.wifi\_settings) {  
                       mWifiEnabler.setSwitch(holder.switch\_);  
                    } else {  
//这个就是处理了，上面的路引没有白做啊  
                       mBluetoothEnabler.setSwitch(holder.switch\_);  
                    }  
3、mBluetoothEnabler.setSwitch分析  
    public void setSwitch(Switch switch\_) {  
//若是和上次相同，则不做任何事情，可以理解，代码也懒嘛  
        if (mSwitch == switch\_) return;  
//把上次的switch的changelistener清空  
       mSwitch.setOnCheckedChangeListener(null);  
        mSwitch = switch\_;  
//重设这次的switch的changelistener  
       mSwitch.setOnCheckedChangeListener(this);  
        int bluetoothState =BluetoothAdapter.STATE\_OFF;  
//获取getBluetoothState，这个过程也会同步一下state，防止改变  
        if (mLocalAdapter != null)bluetoothState = mLocalAdapter.getBluetoothState();  
//根据状态设置一下两个标志位  
        boolean isOn = bluetoothState ==BluetoothAdapter.STATE\_ON;  
        boolean isOff = bluetoothState ==BluetoothAdapter.STATE\_OFF;  
//设置checked的状态位。注意，假如这里状态发生了改变，则会调用this.onCheckedChanged来进行处理  
        mSwitch.setChecked(isOn);  
        if(WirelessSettings.isRadioAllowed(mContext, Settings.System.RADIO\_BLUETOOTH)) {  
//有bluetooth或者不是airplane，则该switch不变灰，否则，灰的。  
            mSwitch.setEnabled(isOn || isOff);  
        } else {  
            mSwitch.setEnabled(false);  
        }  
    }  
4、onCheckedChanged  
在switch状态发生改变后，会调用这个地方的回调函数进行处理。  
    public void onCheckedChanged(CompoundButtonbuttonView, boolean isChecked) {  
        // Show toast message if Bluetooth isnot allowed in airplane mode  
//若是打开的话，就需要检查一下是否allow Bluetooth（radio，airplane的check）  
        if (isChecked &&  
               !WirelessSettings.isRadioAllowed(mContext,Settings.System.RADIO\_BLUETOOTH)) {  
            Toast.makeText(mContext,R.string.wifi\_in\_airplane\_mode, Toast.LENGTH\_SHORT).show();  
            // Reset switch to off  
//若是不对的话，reset为off  
            buttonView.setChecked(false);  
        }  
        if (mLocalAdapter != null) {  
//4.1设置scanmode，放心，它会判断state的，不是STATE\_ON，会直接返回false的  
           mLocalAdapter.setScanMode(BluetoothAdapter.SCAN\_MODE\_CONNECTABLE);  
//4.2使能或不使能Bluetooth了  
           mLocalAdapter.setBluetoothEnabled(isChecked);  
        }  
//过程中还是会反灰，直到setBluetoothEnabled的结果返回会改变switch的状态  
        mSwitch.setEnabled(false);  
    }  
4.1设置scanmod  
会调用adapter中的setScanMode，直接去看就可以了，事实上就是设置了两个property标志，没什么  
    public boolean setScanMode(int mode) {  
//这里把这个代码写出来就是证明一下，STATE\_ON才会真正做下去，否则免谈  
        if (getState() != STATE\_ON) returnfalse;  
//这里会调用对应server中的setScanMode  
        return setScanMode(mode, 120);  
    }  
    public synchronized boolean setScanMode(intmode, int duration) {  
//这里有个permission，好像和2.3中不一样，注意一下     mContext.enforceCallingOrSelfPermission(android.Manifest.permission.WRITE\_SECURE\_SETTINGS,  
"NeedWRITE\_SECURE\_SETTINGS permission");  
        boolean pairable;  
        boolean discoverable;  
        switch (mode) {  
        case BluetoothAdapter.SCAN\_MODE\_NONE:  
            pairable = false;  
            discoverable = false;  
            break;  
        caseBluetoothAdapter.SCAN\_MODE\_CONNECTABLE:  
//开始就是这里了，可pairable，但是不可discoverable  
            pairable = true;  
            discoverable = false;  
            break;  
        caseBluetoothAdapter.SCAN\_MODE\_CONNECTABLE\_DISCOVERABLE:  
            pairable = true;  
            discoverable = true;  
            if (DBG) Log.d(TAG, "BTDiscoverable for " + duration + " seconds");  
            break;  
        default:  
            Log.w(TAG, "Requested invalidscan mode " + mode);  
            return false;  
        }  
//设置这两个property标志  
       setPropertyBoolean("Discoverable", discoverable);  
       setPropertyBoolean("Pairable", pairable);  
        return true;  
    }  
4.2setBluetoothEnabled分析  
    public void setBluetoothEnabled(booleanenabled) {  
//根据enabled的标志设置是enable还是disable，在2.3中，这个地方就是bt\_enable哦，这里还不知道，我们在第5步进行详细的分析  
        boolean success = enabled  
                ? mAdapter.enable()  
                : mAdapter.disable();  
//成功了，设置对应的状态位  
        if (success) {  
            setBluetoothStateInt(enabled  
                ?BluetoothAdapter.STATE\_TURNING\_ON  
                :BluetoothAdapter.STATE\_TURNING\_OFF);  
        } else {  
            if (Utils.V) {  
                Log.v(TAG,"setBluetoothEnabled call, manager didn't return " +  
                        "success forenabled: " + enabled);  
            }  
//同步一下设置的状态  
            syncBluetoothState();  
        }  
    }  
}  
5、mAdapter.enable或者mAdapter.disable  
就先分析enable吧，它会调用对应server端的enable(ture),我们来看看源码  
    public synchronized boolean enable(booleansaveSetting) {  
       mContext.enforceCallingOrSelfPermission(BLUETOOTH\_ADMIN\_PERM,  
                                               "Need BLUETOOTH\_ADMIN permission");  
        // Airplane mode can prevent Bluetoothradio from being turned on.  
//检查是否是飞行模式  
        if (mIsAirplaneSensitive &&isAirplaneModeOn() && !mIsAirplaneToggleable) {  
            return false;  
        }  
//5.1注意与2.3的不同，在2.3中，这里会调用enablethread去调用native的bt\_enable，而4.0没有这么做。没事，我们来分析4.0怎么做的。  
       mBluetoothState.sendMessage(BluetoothAdapterStateMachine.USER\_TURN\_ON,saveSetting);  
        return true;  
    }  
5.1mBluetoothState.sendMessage  
简单理解一下，mBluetoothState是BluetoothAdapterStateMachine类。因此，在分析的之前，简单说一下，它其实就是类似一个状态转换图，根据你所处于的状态，然后再判断收到的操作，进行不同的处理。根据构造函数中的setInitialState(mPowerOff);可以知道初始状态是PowerOff。但是从它给出的状态机可以看出，在PowerOff的状态时，它是通过TURN\_HOT/TURN\_ON来改变到HotOff状态的，然后才会收到USER\_TURN\_ON，去该变到BluetootOn的状态。因此，可以肯定的是我们这里的USER\_TURN\_ON不是它收到的第一个message，因此我们去纠结一下它是从哪里开始改变PowerOff的状态：extra1，然后再来看这里的处理吧：5.2。  
extra1、mAdapter.enable之前的状态机转变  
众所周知，android在启动之后会启动一个serverThread的线程，通过这个线程会启动一系列的服务。我们的蓝牙服务也是在这里启动的，android4.0其实在这个地方对状态机进行了修改，我们来看一下源码：  
该代码位于framworks/base/services/java/com/android/server/systemserver.java  
BluetoothServicebluetooth = null;  
BluetoothA2dpServicebluetoothA2dp = null;  
//模拟器上是不支持Bluetooth的，工厂测试模式也没有Bluetooth（这个不了解）  
            // Skip Bluetooth if we have anemulator kernel  
            // TODO: Use a more reliable checkto see if this product should  
            // support Bluetooth - see bug988521  
            if(SystemProperties.get("ro.kernel.qemu").equals("1")) {  
                Slog.i(TAG, "No BluetoohService (emulator)");  
            } else if (factoryTest ==SystemServer.FACTORY\_TEST\_LOW\_LEVEL) {  
                Slog.i(TAG, "No BluetoothService (factory test)");  
            } else {  
                Slog.i(TAG, "BluetoothService");  
//新建Bluetoothservice，并把他加入到ServiceManager中  
                bluetooth = newBluetoothService(context);  
               ServiceManager.addService(BluetoothAdapter.BLUETOOTH\_SERVICE,bluetooth);  
//extra1.1在启动Bluetooth服务后进行一些初始化，呵呵，这里就对状态机进行了改变  
               bluetooth.initAfterRegistration();  
//新建了BluetoothA2dpService，并把之加入到了ServiceManager中  
bluetoothA2dp= new BluetoothA2dpService(context, bluetooth);  
               ServiceManager.addService(BluetoothA2dpService.BLUETOOTH\_A2DP\_SERVICE,  
                                         bluetoothA2dp);  
//extra1.2同样的要在之后做些init的工作  
               bluetooth.initAfterA2dpRegistration();  
//得到是否飞行  
                int airplaneModeOn =Settings.System.getInt(mContentResolver,  
                       Settings.System.AIRPLANE\_MODE\_ON, 0);  
//看Bluetooth是否on，若是打开的状态（没有飞行），则这里会调用enable去打开  
                int bluetoothOn =Settings.Secure.getInt(mContentResolver,  
                   Settings.Secure.BLUETOOTH\_ON, 0);  
                if (airplaneModeOn == 0&& bluetoothOn != 0) {  
                    bluetooth.enable();  
                }  
            }  
extra1.1initAfterRegistration分析  
    public synchronized voidinitAfterRegistration() {  
//得到default的adapter  
        mAdapter =BluetoothAdapter.getDefaultAdapter();  
//创建BluetoothAdapterStateMachine，初始化几个状态，并设初始状态位POWEROFF,这里同时新建了一个EventLoop  
        mBluetoothState = newBluetoothAdapterStateMachine(mContext, this, mAdapter);  
        mBluetoothState.start();  
//根据这个xml的bool变量来决定是否先期TURN\_HOT，该变量位于frameworks/base/core/res/res/values/config.xml中，默认为true  
        if (mContext.getResources().getBoolean  
           (com.android.internal.R.bool.config\_bluetooth\_adapter\_quick\_switch)) {  
//extra1.2发送TURN\_HOT的状态变化message  
           mBluetoothState.sendMessage(BluetoothAdapterStateMachine.TURN\_HOT);  
        }  
//得到对应的EventLoop  
        mEventLoop =mBluetoothState.getBluetoothEventLoop();  
    }  
extra1.2  TURN\_HOT message的处理  
    /\*\*  
     \* Bluetooth module's power is off,firmware is not loaded.  
     \*/  
    private class PowerOff extends State {  
        @Override  
        public void enter() {  
            if (DBG) log("Enter PowerOff:" + getCurrentMessage().what);  
        }  
        @Override  
        public boolean processMessage(Messagemessage) {  
            log("PowerOff process message:" + message.what);  
            boolean retValue = HANDLED;  
            switch(message.what) {  
……  
               case TURN\_HOT:  
//extra1.3这里就是我们寻找了千年的bt\_enable所在的地方。我们去看看  
                    if (prepareBluetooth()) {  
//extra1.5转变状态到warmup，在prepareBluetooth真正完成后，这个状态还会发生改变  
                        transitionTo(mWarmUp);  
                    }  
                    break;  
……  
extra1.3prepareBluetooth分析  
看英文注释就知道了，不解释  
        /\*\*  
         \* Turn on Bluetooth Module, Loadfirmware, and do all the preparation  
         \* needed to get the Bluetooth Moduleready but keep it not discoverable  
         \* and not connectable.  
         \* The last step of this method sets upthe local service record DB.  
         \* There will be a event reporting thestatus of the SDP setup.  
         \*/  
        private boolean prepareBluetooth() {  
//extra1.4首先还是调用了enableNative的本地方法，到这里你会发现终于和2.3相似了（不过请注意调用的时机不同了，这个在初始化，而2.3在界面的on/off滑动的时候），它还是会调用bt\_enable，这个就会调用对应的set\_bluetooth\_power了  
            if(mBluetoothService.enableNative() != 0) {  
                return false;  
            }  
            // try to start event loop, give 2attempts  
//尝试两次去start event loop  
            int retryCount = 2;  
            boolean eventLoopStarted = false;  
            while ((retryCount-- > 0)&& !eventLoopStarted) {  
                mEventLoop.start();  
                // it may take a moment for theother thread to do its  
                // thing.  Check periodically for a while.  
                int pollCount = 5;  
                while ((pollCount-- > 0)&& !eventLoopStarted) {  
                    if(mEventLoop.isEventLoopRunning()) {  
                        eventLoopStarted =true;  
                        break;  
                    }  
                    try {  
                        Thread.sleep(100);  
                    } catch(InterruptedException e) {  
                       log("prepareBluetooth sleep interrupted: " + pollCount);  
                        break;  
                    }  
                }  
            }  
//出错处理  
            if (!eventLoopStarted) {  
               mBluetoothService.disableNative();  
                return false;  
            }  
            // get BluetoothService ready  
//建立native data以及SDP相关的一些操作，这里将会产生PropertyChanged的UUIDs的signal，对该信号的处理会对状态发生改变，详细分析见extra1.5  
            if(!mBluetoothService.prepareBluetooth()) {  
                mEventLoop.stop();  
               mBluetoothService.disableNative();  
                return false;  
            }  
//设置一个prepare的超时处理，在该时间内没有收到UUID changed的signal将会进行错误处理  
           sendMessageDelayed(PREPARE\_BLUETOOTH\_TIMEOUT,PREPARE\_BLUETOOTH\_TIMEOUT\_TIME);  
            return true;  
        }  
    }  
extra1.4bt\_enable分析  
intbt\_enable() {  
    LOGV(\_\_FUNCTION\_\_);  
    int ret = -1;  
    int hci\_sock = -1;  
    int attempt;  
//power的设置，on。不解释，可加入对应板子的gpio口的处理，默认就只用了rfkill的处理  
    if (set\_bluetooth\_power(1) < 0) gotoout;  
//开始hciattach服务，这个我们也做了修改，加入了rtk\_h5  
    LOGI("Starting hciattachdaemon");  
    if (property\_set("ctl.start","hciattach") < 0) {  
        LOGE("Failed to starthciattach");  
        set\_bluetooth\_power(0);  
        goto out;  
    }  
    // Try for 10 seconds, this can onlysucceed once hciattach has sent the  
    // firmware and then turned on hci devicevia HCIUARTSETPROTO ioctl  
    for (attempt = 1000; attempt > 0;  attempt--) {  
//创建hci\_sock  
        hci\_sock = create\_hci\_sock();  
        if (hci\_sock < 0) goto out;  
//调用ioctl的HCIDEVUP，来判断hciattach是否已经ok了。  
        ret = ioctl(hci\_sock, HCIDEVUP,HCI\_DEV\_ID);  
        LOGI("bt\_enable: ret: %d, errno:%d", ret, errno);  
        if (!ret) {  
            break;  
        } else if (errno == EALREADY) {  
            LOGW("Bluetoothd alreadystarted, unexpectedly!");  
            break;  
        }  
        close(hci\_sock);  
//等待10 ms后再试一次  
        usleep(100000);  // 100 ms retry delay  
    }  
//10s都没有搞定，需要做个失败的处理  
    if (attempt == 0) {  
        LOGE("%s: Timeout waiting for HCIdevice to come up, error- %d, ",  
            \_\_FUNCTION\_\_, ret);  
        if (property\_set("ctl.stop","hciattach") < 0) {  
            LOGE("Error stoppinghciattach");  
        }  
        set\_bluetooth\_power(0);  
        goto out;  
    }  
//启动bluetoothd服务  
    LOGI("Starting bluetoothddeamon");  
    if (property\_set("ctl.start","bluetoothd") < 0) {  
        LOGE("Failed to startbluetoothd");  
        set\_bluetooth\_power(0);  
        goto out;  
    }  
    ret = 0;  
out:  
//关闭hci\_sock  
    if (hci\_sock >= 0) close(hci\_sock);  
    return ret;  
}  
extra 1.5 PropetyChanged的UUIDs的处理  
event\_filter是用来对bluez的dbus的signal进行监听的，有signal产生后，会在这里进行处理。因此，我们直接到这里看看该怎么处理。  
//Called by dbus during WaitForAndDispatchEventNative()  
staticDBusHandlerResult event\_filter(DBusConnection \*conn, DBusMessage \*msg,  
                                      void\*data) {  
    native\_data\_t \*nat;  
    JNIEnv \*env;  
    DBusError err;  
    DBusHandlerResult ret;  
//err的一个初始化  
    dbus\_error\_init(&err);  
//得到参数  
    nat = (native\_data\_t \*)data;  
    nat->vm->GetEnv((void\*\*)&env,nat->envVer);  
    if (dbus\_message\_get\_type(msg) !=DBUS\_MESSAGE\_TYPE\_SIGNAL) {  
        LOGV("%s: not interested (not asignal).", \_\_FUNCTION\_\_);  
        returnDBUS\_HANDLER\_RESULT\_NOT\_YET\_HANDLED;  
    }  
    LOGV("%s: Received signal %s:%s from%s", \_\_FUNCTION\_\_,  
        dbus\_message\_get\_interface(msg),dbus\_message\_get\_member(msg),  
        dbus\_message\_get\_path(msg));  
    env->PushLocalFrame(EVENT\_LOOP\_REFS);  
……  
//PropertyChanged这个signal的处理  
    } else if (dbus\_message\_is\_signal(msg,  
                                     "org.bluez.Adapter",  
                                     "PropertyChanged")) {  
//由msg解析参数  
        jobjectArray str\_array =parse\_adapter\_property\_change(env, msg);  
        if (str\_array != NULL) {  
            /\* Check if bluetoothd has(re)started, if so update the path. \*/  
            jstring property =(jstring)env->GetObjectArrayElement(str\_array, 0);  
            const char \*c\_property =env->GetStringUTFChars(property, NULL);  
//检查Property是否started  
            if (!strncmp(c\_property,"Powered", strlen("Powered"))) {  
//若是powered，则看value是否是true，是ture就得到对应的path  
                jstring value =  
                    (jstring)env->GetObjectArrayElement(str\_array, 1);  
                const char \*c\_value =env->GetStringUTFChars(value, NULL);  
                if (!strncmp(c\_value,"true", strlen("true")))  
                    nat->adapter =get\_adapter\_path(nat->conn);  
               env->ReleaseStringUTFChars(value, c\_value);  
            }  
           env->ReleaseStringUTFChars(property, c\_property);  
//extra1.6调用对应的method\_onPropertyChanged函数，该method对应的onPropertyChanged函数  
            env->CallVoidMethod(nat->me,  
                             method\_onPropertyChanged,  
                              str\_array);  
        } elseLOG\_AND\_FREE\_DBUS\_ERROR\_WITH\_MSG(&err, msg);  
        goto success;  
……  
extra1.6真正的处理函数onPropertyChanged分析  
 /\*\*  
     \* Called by native code on aPropertyChanged signal from  
     \* org.bluez.Adapter. This method is alsocalled from  
     \* {@link BluetoothAdapterStateMachine} toset the "Pairable"  
     \* property when Bluetooth is enabled.  
     \*  
     \* @param propValues a string arraycontaining the key and one or more  
     \* values.  
     \*/  
    /\*package\*/ void onPropertyChanged(String[]propValues) {  
        BluetoothAdapterPropertiesadapterProperties =  
               mBluetoothService.getAdapterProperties();  
//先fill up cache  
        if (adapterProperties.isEmpty()) {  
            // We have got a property changebefore  
            // we filled up our cache.  
           adapterProperties.getAllProperties();  
        }  
        log("Property Changed: " +propValues[0] + " : " + propValues[1]);  
        String name = propValues[0];  
……  
//对UUIDs的处理  
        } else if(name.equals("Devices") || name.equals("UUIDs")) {  
            String value = null;  
            int len =Integer.valueOf(propValues[1]);  
            if (len > 0) {  
                StringBuilder str = newStringBuilder();  
                for (int i = 2; i <propValues.length; i++) {  
                    str.append(propValues[i]);  
                    str.append(",");  
                }  
                value = str.toString();  
            }  
//把name和value值加入到property的map中  
            adapterProperties.setProperty(name,value);  
//extra1.7有UUIDs的change signal会刷新Bluetooth的State  
            if (name.equals("UUIDs")){  
               mBluetoothService.updateBluetoothState(value);  
            }  
//对Pairable和Discoverable的处理  
       } else if(name.equals("Pairable") || name.equals("Discoverable")) {  
            adapterProperties.setProperty(name,propValues[1]);  
            if(name.equals("Discoverable")) {  
   //5.6发送SCAN\_MODE\_CHANGED的msg，去改变状态机      mBluetoothState.sendMessage(BluetoothAdapterStateMachine.SCAN\_MODE\_CHANGED);  
            }  
//设置对应的property  
            String pairable =name.equals("Pairable") ? propValues[1] :  
               adapterProperties.getProperty("Pairable");  
            String discoverable =name.equals("Discoverable") ? propValues[1] :  
               adapterProperties.getProperty("Discoverable");  
            // This shouldn't happen, unlessAdapter Properties are null.  
            if (pairable == null ||discoverable == null)  
                return;  
            int mode =BluetoothService.bluezStringToScanMode(  
                   pairable.equals("true"),  
                   discoverable.equals("true"));  
            if (mode >= 0) {  
//当pairable和discoverable均为true的时候，会发送一个ACTION\_SCAN\_MODE\_CHANGED的广播消息  
                Intent intent = newIntent(BluetoothAdapter.ACTION\_SCAN\_MODE\_CHANGED);  
               intent.putExtra(BluetoothAdapter.EXTRA\_SCAN\_MODE, mode);  
               intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT);  
                mContext.sendBroadcast(intent,BLUETOOTH\_PERM);  
            }  
        }  
……  
extra1.7  UUIDs改变带来的State的刷新  
    /\*\*  
     \* This function is called from BluetoothEvent Loop when onPropertyChanged  
     \* for adapter comes in with UUID property.  
     \* @param uuidsThe uuids of adapter asreported by Bluez.  
     \*/  
    /\*package\*/ synchronized voidupdateBluetoothState(String uuids) {  
        ParcelUuid[] adapterUuids =convertStringToParcelUuid(uuids);  
//为什么必须包含所有已经有的uuid？？感觉有点反了，再看看  
        if (mAdapterUuids != null &&  
           BluetoothUuid.containsAllUuids(adapterUuids, mAdapterUuids)) {  
//放SERVICE\_RECORD\_LOADED的信息，此时，处于warm up状态，看extra1.8分析状态如何继续改变          mBluetoothState.sendMessage(BluetoothAdapterStateMachine.SERVICE\_RECORD\_LOADED);  
        }  
    }  
extra1.8 UUIDs对状态机改变  
   /\*\*  
     \* Turning on Bluetooth module's power,loading firmware, starting  
     \* event loop thread to listen on Bluetoothmodule event changes.  
     \*/  
    private class WarmUp extends State {  
        @Override  
        public void enter() {  
            if (DBG) log("Enter WarmUp:" + getCurrentMessage().what);  
        }  
        @Override  
        public boolean processMessage(Messagemessage) {  
            log("WarmUp process message:" + message.what);  
            boolean retValue = HANDLED;  
            switch(message.what) {  
                case SERVICE\_RECORD\_LOADED:  
//可以看到，首先会把当时从poweroff过来的一个超时message拿remove了。  
                   removeMessages(PREPARE\_BLUETOOTH\_TIMEOUT);  
//转到hotoff状态，在hotoff状态仍会接收到多个SERVICE\_RECORD\_LOADED的msg，但是那个状态下该msg将没有任何handled，因此会一直处于hotoff状态  
                    transitionTo(mHotOff);  
                    break;  
……  
5.2mAdapter.enable中mBluetoothState.sendMessage后的状态机处理  
由extra的分析可知，此时，Bluetooth的State已经处于HotOff状态了，所以，从这里开始处理State的变换。  
    /\*\*  
     \* Bluetooth Module has powered, firmwareloaded, event loop started,  
     \* SDP loaded, but the modules staysnon-discoverable and  
     \* non-connectable.  
     \*/  
    private class HotOff extends State {  
        @Override  
        public void enter() {  
            if (DBG) log("Enter HotOff:" + getCurrentMessage().what);  
        }  
        @Override  
        public boolean processMessage(Messagemessage) {  
            log("HotOff process message:" + message.what);  
            boolean retValue = HANDLED;  
            switch(message.what) {  
                case USER\_TURN\_ON:  
//发出BluetoothAdapter.STATE\_TURNING\_ON的广播消息  
                   broadcastState(BluetoothAdapter.STATE\_TURNING\_ON);  
                    if ((Boolean) message.obj){  
//就是把Settings.Secure.BLUETOOTH\_ON设为1。用于标志Bluetooth enable了  
                       persistSwitchSetting(true);  
                    }  
                    // let it fall toTURN\_ON\_CONTINUE:  
                    //$FALL-THROUGH$  
//注意上面没有break哦  
                case TURN\_ON\_CONTINUE:  
//这里就是把Bluetooth设为connectable就是Powered=1，这里就把prepareBluetooth中设置的不可连接重新设置回来了。这个重连会产生一些新的变化，它会发送WRITE\_SCAN\_ENABLE的cmd，因此在该cmd\_complete时会有一些新的处理：5.3，它会再次引起状态机的改变：5.6  
                   mBluetoothService.switchConnectable(true);  
//进入到Switching状态  
                    transitionTo(mSwitching);  
                    break;  
……  
5.3 WRITE\_SCAN\_ENABLE在cmd\_complete后的处理  
在bluez中是用cmd\_complete函数来监视发出cmd完成后的处理的。该函数具体如下：  
staticinline void cmd\_complete(int index, void \*ptr)  
{  
structdev\_info \*dev = &devs[index];  
evt\_cmd\_complete\*evt = ptr;  
uint16\_topcode = btohs(evt->opcode);  
uint8\_tstatus = \*((uint8\_t \*) ptr + EVT\_CMD\_COMPLETE\_SIZE);  
switch(opcode) {  
……  
//WRITE\_SCAN\_ENABLE命令完成的处理函数，会再发一个READ\_SCAN\_ENABLE的命令  
casecmd\_opcode\_pack(OGF\_HOST\_CTL, OCF\_WRITE\_SCAN\_ENABLE):  
hci\_send\_cmd(dev->sk,OGF\_HOST\_CTL, OCF\_READ\_SCAN\_ENABLE,  
0,NULL);  
break;  
//5.4紧接着就是对READ\_SCAN\_ENABLE命令完成的处理，它是通过read\_scan\_complete来实现的  
casecmd\_opcode\_pack(OGF\_HOST\_CTL, OCF\_READ\_SCAN\_ENABLE):  
ptr+= sizeof(evt\_cmd\_complete);  
read\_scan\_complete(index,status, ptr);  
break;  
……  
}  
5.4 read\_scan命令完成的处理  
staticvoid read\_scan\_complete(int index, uint8\_t status, void \*ptr)  
{  
structbtd\_adapter \*adapter;  
read\_scan\_enable\_rp\*rp = ptr;  
DBG("hci%dstatus %u", index, status);  
//由index得到对应的adapter  
adapter= manager\_find\_adapter\_by\_id(index);  
if(!adapter) {  
error("Unableto find matching adapter");  
return;  
}  
//5.5这里算是一个通知adapter，mode改变了。  
adapter\_mode\_changed(adapter,rp->enable);  
}  
5.5通知adapter，mode发生了改变  
voidadapter\_mode\_changed(struct btd\_adapter \*adapter, uint8\_t scan\_mode)  
{  
constgchar \*path = adapter\_get\_path(adapter);  
gbooleandiscoverable, pairable;  
DBG("old0x%02x new 0x%02x", adapter->scan\_mode, scan\_mode);  
//若相同，则nothing todo  
if(adapter->scan\_mode == scan\_mode){  
#ifdefBOARD\_HAVE\_BLUETOOTH\_BCM  
    /\*we may reset scan\_mode already inbtd\_adapter\_stop(), so comes to here\*/  
    set\_mode\_complete(adapter);  
#endif  
    return;  
}  
//把discoverable的timeout清空  
adapter\_remove\_discov\_timeout(adapter);  
//这里开始，是设为SCAN\_PAGE| SCAN\_INQUIRY  
switch(scan\_mode) {  
caseSCAN\_DISABLED:  
adapter->mode= MODE\_OFF;  
discoverable= FALSE;  
pairable= FALSE;  
break;  
caseSCAN\_PAGE:  
adapter->mode= MODE\_CONNECTABLE;  
discoverable= FALSE;  
pairable= adapter->pairable;  
break;  
case(SCAN\_PAGE | SCAN\_INQUIRY):  
//设一下模式，在有reply要求的情况下，该步骤还是很重要的  
adapter->mode= MODE\_DISCOVERABLE;  
discoverable= TRUE;  
pairable= adapter->pairable;  
//还要设一个discoverable的时间  
if(adapter->discov\_timeout != 0)  
adapter\_set\_discov\_timeout(adapter,  
adapter->discov\_timeout);  
break;  
caseSCAN\_INQUIRY:  
/\*Address the scenario where a low-level application like  
 \* hciconfig changed the scan mode \*/  
if(adapter->discov\_timeout != 0)  
adapter\_set\_discov\_timeout(adapter,  
adapter->discov\_timeout);  
/\*ignore, this event should not be sent \*/  
default:  
/\*ignore, reserved \*/  
return;  
}  
/\*If page scanning gets toggled emit the Pairable property \*/  
//这里会发一个property\_changed的pairable的signal  
if((adapter->scan\_mode & SCAN\_PAGE) != (scan\_mode & SCAN\_PAGE))  
emit\_property\_changed(connection,adapter->path,  
ADAPTER\_INTERFACE,"Pairable",  
DBUS\_TYPE\_BOOLEAN,&pairable);  
if(!discoverable)  
adapter\_set\_limited\_discoverable(adapter,FALSE);  
//这里会发一个property\_changed的discoverable的signal  
emit\_property\_changed(connection,path,  
ADAPTER\_INTERFACE,"Discoverable",  
DBUS\_TYPE\_BOOLEAN,&discoverable);  
adapter->scan\_mode= scan\_mode;  
set\_mode\_complete(adapter);  
}  
5.6 WRTIE\_SCAN\_ENABLE最终引起的状态机的变化  
在此之前，状态机处于switching的状态，收到了SCAN\_MODE\_CHANGED的msg。  
    private class Switching extends State {  
        @Override  
        public void enter() {  
            if (DBG) log("Enter Switching:" + getCurrentMessage().what);  
        }  
        @Override  
        public boolean processMessage(Messagemessage) {  
            log("Switching processmessage: " + message.what);  
            boolean retValue = HANDLED;  
            switch(message.what) {  
                case SCAN\_MODE\_CHANGED:  
                    // This event matchesmBluetoothService.switchConnectable action  
//mPublicState在hotoff到swtiching状态变化时已经被设为STATE\_TURNING\_ON了，所以这里if没有问题  
                    if (mPublicState ==BluetoothAdapter.STATE\_TURNING\_ON) {  
                        // set pairable if it'snot  
//设置为pairable假如还没有设置的话，这个会先在bluez中检查一下当前是否pairable，我们在前面已经设置好了，所以，这里只是一个检查而已，没有什么实际性的工作  
                       mBluetoothService.setPairable();  
//初始化bond state和profile state，这个会在adapter pairable之后，bluetooth turn on之前发生  
                       mBluetoothService.initBluetoothAfterTurningOn();  
//这边正式进入到bluetoothon的状态，终于进了这里，哎。。。  
                       transitionTo(mBluetoothOn);  
//发送STATE\_ON的broadcast  
                       broadcastState(BluetoothAdapter.STATE\_ON);  
                        // run bluetooth nowthat it's turned on  
                        // Note runBluetoothshould be called only in adapter STATE\_ON  
//连接那些可以自动连接的设备，通知battery，蓝牙打开了  
                       mBluetoothService.runBluetooth();  
                    }  
                    break;  
……  
至此，蓝牙的使能主要过程已经全部搞定。