# 1、蓝牙应用的打开流程：

系统在启动的时候会添加蓝牙服务，在BluetoothManagerService的构造函数里，调用loadStoredNameAndAddress()，在这个BluetoothHandler里，绑定了 Ibluetooth.class.getName，通过aidl启动了远程 AdapterService服务, AdapterService.java里的静态块实现蓝牙的启动/Settings/src/com/android/settings/

Settings.java为设置界面，更新设置里面的目录，判断系统是否有蓝牙服务，在这个类获取蓝牙的设置界面/Settings/src/com/android/settings/BluetoothSettings.java

if (SystemProperties.get("ro.kernel.qemu").equals("1")) {

Slog.i(TAG, "No Bluetooh Service (emulator)");

} else if (factoryTest == SystemServer.FACTORY\_TEST\_LOW\_LEVEL) {

Slog.i(TAG, "No Bluetooth Service (factory test)");

} else {

Slog.i(TAG, "Bluetooth Manager Service");

bluetooth = new BluetoothManagerService(context);

ServiceManager.addService(BluetoothAdapter.BLUETOOTH\_MANAGER\_SERVICE, bluetooth);

}

/frameworks/base/services/java/com/android/server/BluetoothManagerService.java，这部分代码，生成libandroid\_runtime.so完成功能，中转BluetoothAdapter和Bluetooth.apk，所有来自其他应用的请求，都通过IBluetooth接口，转发到Bluetooth.apk

启动方式：

Intent i = new Intent(IBluetooth.class.getName());

if (!mContext.bindService(i, mConnection,Context.BIND\_AUTO\_CREATE,  
 UserHandle.USER\_CURRENT)) {

这里实际返回android.bluetooth.IBluetooth，这个action正好是AdapterService要处理的action。

BluetoothManagerService保持一个BluetoothServiceConnection的回调，当AdapterService启动时，就可以拿到IBluetooth接口了。

packages/apps/Settings/src/com/android/settings/bluetooth/BluetoothSettings.java是进入蓝牙的界面，点击”关闭“启动蓝牙服务

public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {

// Show toast message if Bluetooth is not allowed in airplane mode

if (isChecked &&

!WirelessSettings.isRadioAllowed(mContext, Settings.Global.RADIO\_BLUETOOTH)) {

Toast.makeText(mContext, R.string.wifi\_in\_airplane\_mode, Toast.LENGTH\_SHORT).show();

// Reset switch to off

buttonView.setChecked(false);

}

if (mLocalAdapter != null) {

mLocalAdapter.setBluetoothEnabled(isChecked);

}

mSwitch.setEnabled(false);

}

/frameworks/base/core/java/android/bluetooth/BluetoothAdapter.java

public boolean enable() {

if (isEnabled() == true){

if (DBG) Log.d(TAG, "enable(): BT is already enabled..!");

return true;

}

try {

return mManagerService.enable(); //mManagerService其实是bluetoothAdapter的一个代理

} catch (RemoteException e) {Log.e(TAG, "", e);}

return false;

}

/frameworks/base/services/java/com/android/server/BluetoothManagerService.java

enable-->sendEnableMsg(){

mHandler.sendMessage(mHandler.obtainMessage(MESSAGE\_ENABLE,quietMode?1:0,0)))

}

case MESSAGE\_ENABLE:

mEnable = true;

handleEnable(msg.arg1 == 1);

break;

private void handleEnable(boolean quietMode) {

mQuietEnable = quietMode;

synchronized(mConnection) {

if ((mBluetooth == null) && (!mBinding)) {

//Start bind timeout and bind

Message timeoutMsg=mHandler.obtainMessage(MESSAGE\_TIMEOUT\_BIND);

mHandler.sendMessageDelayed(timeoutMsg,TIMEOUT\_BIND\_MS);

mConnection.setGetNameAddressOnly(false);

Intent i = new Intent(IBluetooth.class.getName());

if (!mContext.bindService(i, mConnection,Context.BIND\_AUTO\_CREATE,

UserHandle.USER\_CURRENT)) {

mHandler.removeMessages(MESSAGE\_TIMEOUT\_BIND);

Log.e(TAG, "Fail to bind to: " + IBluetooth.class.getName());

} else {

mBinding = true;

}

} else if (mBluetooth != null) {

if (mConnection.isGetNameAddressOnly()) {

mConnection.setGetNameAddressOnly(false);

try {

mBluetooth.registerCallback(mBluetoothCallback);

} catch (RemoteException re) {

Log.e(TAG, "Unable to register BluetoothCallback",re);

}

//Inform BluetoothAdapter instances that service is up

sendBluetoothServiceUpCallback();

}

//Enable bluetooth

try {

if (!mQuietEnable) {

if(!mBluetooth.enable()) {

Log.e(TAG,"IBluetooth.enable() returned false");

}

}

else {

if(!mBluetooth.enableNoAutoConnect()) {

Log.e(TAG,"IBluetooth.enableNoAutoConnect() returned false");

}

}

} catch (RemoteException e) {

Log.e(TAG,"Unable to call enable()",e);

}

}

}

}

mBluetooth在注册回调函数之后调用sendBluetoothServiceUpCallback方法进行广播，通知BluetoothAdapter，AdapterService已经up，因此调用的是BluetoothAdapter的onBluetoothServiceUp方法

private void sendBluetoothServiceUpCallback() {

int n = mCallbacks.beginBroadcast();

if (!mConnection.isGetNameAddressOnly()) {

for (int i=0; i <n;i++) {

try { mCallbacks.getBroadcastItem(i).onBluetoothServiceUp(mBluetooth); } }

mCallbacks.finishBroadcast();

}

}

/frameworks/base/core/java/android/bluetooth/BluetoothAdapter.java用于对BluetoothMangerService中回调函数的一个代理

final private IBluetoothManagerCallback mManagerCallback =

new IBluetoothManagerCallback.Stub() {

public void onBluetoothServiceUp(IBluetooth bluetoothService) {

synchronized (mManagerCallback) {

mService = bluetoothService;

for (IBluetoothManagerCallback cb : mProxyServiceStateCallbacks ){

try {

if (cb != null) {

cb.onBluetoothServiceUp(bluetoothService);

} 。。

}

}

}

下一步调用了AdapterService中的enable,这里静态的调用ClassInitNative函数，然后调用initNative的作用主要是启动相应sock、协议栈初始化、启动btif\_task,监听处理蓝牙接口相关的状态消息。，里面有三个enable函数

**public** **synchronized** **boolean** enable(**boolean** quietMode) {

enforceCallingOrSelfPermission(*BLUETOOTH\_ADMIN\_PERM*,

"Need BLUETOOTH ADMIN permission");

**if** (*DBG*)debugLog("Enable called with quiet mode status = " + mQuietmode);

mQuietmode = quietMode;

Message m =

mAdapterStateMachine.obtainMessage(AdapterState.*USER\_TURN\_ON*);

mAdapterStateMachine.sendMessage(m);

**return** **true**;

}

AdapterService里面的enable方法进入AdapterState状态机中，OffState为初始状态，在这里面启动所有profiles服务，并转入PendingCommandState状态

**case** *USER\_TURN\_ON*:

**if** (*DBG*) Log.*d*(*TAG*,"CURRENT\_STATE=OFF, MESSAGE = USER\_TURN\_ON");

notifyAdapterStateChange(BluetoothAdapter.*STATE\_TURNING\_ON*);

mPendingCommandState.setTurningOn(**true**);

transitionTo(mPendingCommandState);

sendMessageDelayed(*START\_TIMEOUT*, *START\_TIMEOUT\_DELAY*);

mAdapterService.processStart(); //启动系统支持的所有profiles的services

**break**;

发出updateAdapterState广播，这个由AdapterService来接收并更新服务的状态

**private** **void** notifyAdapterStateChange(**int** newState) {

**int** oldState = mAdapterProperties.getState();

mAdapterProperties.setState(newState);

infoLog("Bluetooth adapter state changed: " + oldState + "-> " + newState);

mAdapterService.updateAdapterState(oldState, newState);

}

启动所有服务

**void** processStart() {

**if** (*DBG*) debugLog("processStart()");

Class[] supportedProfileServices = Config.*getSupportedProfiles*();

//Initialize data objects

**for** (**int** i=0; i < supportedProfileServices.length;i++) {

mProfileServicesState.put(supportedProfileServices[i].getName(),BluetoothAdapter.*STATE\_OFF*);

}

mRemoteDevices = **new** RemoteDevices(**this**);

mAdapterProperties.init(mRemoteDevices);

**if** (*DBG*) {debugLog("processStart(): Make Bond State Machine");}

mBondStateMachine = BondStateMachine.*make*(**this**, mAdapterProperties, mRemoteDevices);

mJniCallbacks.init(mBondStateMachine,mRemoteDevices);

//**初始化远程设备和绑定状态机之间的关系**

*setAdapterService*(**this**);

//Start profile services

**if** (!mProfilesStarted && supportedProfileServices.length >0) {

//Startup all profile services

setProfileServiceState(supportedProfileServices,BluetoothAdapter.*STATE\_ON*);

}**else** {

**if** (*DBG*) {debugLog("processStart(): Profile Services alreay started");}

mAdapterStateMachine.sendMessage(mAdapterStateMachine.obtainMessage(AdapterState.*STARTED*));

}

}

Starting service com.android.bluetooth.a2dp.A2dpService

Starting service com.android.bluetooth.hid.HidService

Starting service com.android.bluetooth.hdp.HealthService

Starting service com.android.bluetooth.pan.PanService

Starting service com.android.bluetooth.gatt.GattService

回到AdapterProperties中回调adapterPropertyChangeCallback方法，BondStateMachine状态机进入StableState，AdapterState状态机进入PendingCommandState状态

**void** adapterPropertyChangedCallback(**int**[] types, **byte**[][] values) {

Intent intent;

**int** type;

**byte**[] val;

**for** (**int** i = 0; i < types.length; i++) {

val = values[i];

type = types[i];

infoLog("adapterPropertyChangedCallback with type:" + type + " len:" + val.length);

**synchronized** (mObject) {

**switch** (type) {

**case** AbstractionLayer.*BT\_PROPERTY\_BDNAME*:

mName = **new** String(val);

intent = **new** Intent(BluetoothAdapter.*ACTION\_LOCAL\_NAME\_CHANGED*);

intent.putExtra(BluetoothAdapter.*EXTRA\_LOCAL\_NAME*, mName);

intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT);

mService.sendBroadcastAsUser(intent, UserHandle.ALL,

mService.*BLUETOOTH\_PERM*);

debugLog("Name is: " + mName);

**break**;

**case** AbstractionLayer.*BT\_PROPERTY\_BDADDR*:

mAddress = val;

debugLog("Address is:" + Utils.*getAddressStringFromByte*(mAddress));

**break**;

**case** AbstractionLayer.*BT\_PROPERTY\_CLASS\_OF\_DEVICE*:

mBluetoothClass = Utils.*byteArrayToInt*(val, 0);

debugLog("BT Class:" + mBluetoothClass);

**break**;

**case** AbstractionLayer.*BT\_PROPERTY\_ADAPTER\_SCAN\_MODE*:

**int** mode = Utils.*byteArrayToInt*(val, 0);

mScanMode = mService.*convertScanModeFromHal*(mode);

intent = **new** Intent(BluetoothAdapter.*ACTION\_SCAN\_MODE\_CHANGED*);

intent.putExtra(BluetoothAdapter.*EXTRA\_SCAN\_MODE*, mScanMode);

intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT);

mService.sendBroadcast(intent, mService.*BLUETOOTH\_PERM*);

debugLog("Scan Mode:" + mScanMode);

**if** (mBluetoothDisabling) {

mBluetoothDisabling=**false**;

mService.startBluetoothDisable();

}

**break**;

**case** AbstractionLayer.*BT\_PROPERTY\_UUIDS*:

mUuids = Utils.*byteArrayToUuid*(val);

**break**;

**case** AbstractionLayer.*BT\_PROPERTY\_ADAPTER\_BONDED\_DEVICES*:

**int** number = val.length/*BD\_ADDR\_LEN*;

**byte**[] addrByte = **new** **byte**[*BD\_ADDR\_LEN*];

**for** (**int** j = 0; j < number; j++) {

System.*arraycopy*(val, j \* *BD\_ADDR\_LEN*, addrByte, 0, *BD\_ADDR\_LEN*);

onBondStateChanged(mAdapter.getRemoteDevice(

Utils.*getAddressStringFromByte*(addrByte)),

BluetoothDevice.*BOND\_BONDED*);

}

**break**;

**case** AbstractionLayer.*BT\_PROPERTY\_ADAPTER\_DISCOVERABLE\_TIMEOUT*:

mDiscoverableTimeout = Utils.*byteArrayToInt*(val, 0);

debugLog("Discoverable Timeout:" + mDiscoverableTimeout);

**break**;

**default**:

errorLog("Property change not handled in Java land:" + type);

}

}

}

}

**public** **void** onProfileServiceStateChanged(String serviceName, **int** state) {

Message m = mHandler.obtainMessage(*MESSAGE\_PROFILE\_SERVICE\_STATE\_CHANGED*);

m.obj=serviceName;

m.arg1 = state;

mHandler.sendMessage(m);

}

**private** **void** processProfileServiceStateChanged(String serviceName, **int** state) {

**boolean** doUpdate=**false**;

**boolean** isTurningOn;

**boolean** isTurningOff;

**synchronized** (mProfileServicesState) {

Integer prevState = mProfileServicesState.get(serviceName);

**if** (prevState != **null** && prevState != state) {

mProfileServicesState.put(serviceName,state);

doUpdate=**true**;

}

}

。。。

**synchronized** (mAdapterStateMachine) {

isTurningOff = mAdapterStateMachine.isTurningOff();

isTurningOn = mAdapterStateMachine.isTurningOn();

}

**if** (isTurningOff) {

**synchronized** (mProfileServicesState) {

Iterator<Map.Entry<String,Integer>> i = mProfileServicesState.entrySet().iterator();

**while** (i.hasNext()) {

Map.Entry<String,Integer> entry = i.next();

**if** (BluetoothAdapter.*STATE\_OFF* != entry.getValue()) {

Log.*d*(*TAG*, "Profile still running: " + entry.getKey());

**return**;

}

}

}

mProfilesStarted=**false**;

mAdapterStateMachine.sendMessage(mAdapterStateMachine.obtainMessage(AdapterState.*STOPPED*));

} **else** **if** (isTurningOn) {

**synchronized** (mProfileServicesState) {

Iterator<Map.Entry<String,Integer>> i = mProfileServicesState.entrySet().iterator();

**while** (i.hasNext()) {

Map.Entry<String,Integer> entry = i.next();

**if** (BluetoothAdapter.*STATE\_ON* != entry.getValue()) {

Log.*d*(*TAG*, "Profile still not running:" + entry.getKey());

**return**;

}

}

}

**if** (*DBG*) Log.*d*(*TAG*, "All profile services started.");

mProfilesStarted=**true**;

mAdapterStateMachine.sendMessage(mAdapterStateMachine.obtainMessage(AdapterState.*STARTED*));

}

}

状态机继续调用STARTED分支，里面的enableNative()函数开始进入JNI调用

case STARTED: {

if (DBG) Log.d(TAG,"CURRENT\_STATE=PENDING, MESSAGE = STARTED, isTurningOn=" + isTurningOn + ", isTurningOff=" + isTurningOff);

//Remove start timeout

removeMessages(START\_TIMEOUT);

//Enable

boolean ret = mAdapterService.enableNative();

if (!ret) {

Log.e(TAG, "Error while turning Bluetooth On");

notifyAdapterStateChange(BluetoothAdapter.STATE\_OFF);

transitionTo(mOffState);

} else {

sendMessageDelayed(ENABLE\_TIMEOUT, ENABLE\_TIMEOUT\_DELAY);

}

}

break;

enableNative函数实现的作用是初始化HCI、串口相关，启动HCI工作主线程：bt\_hc\_callback，芯片上电、RF参数初始化

函数的定义在/jni/com\_android\_bluetooth\_btservice\_AdapterService.cpp

**static** jboolean **enableNative**(JNIEnv\* env, jobject obj) {

ALOGV("%s:",\_\_FUNCTION\_\_);

jboolean result = JNI\_FALSE;

**if** (!sBluetoothInterface) **return** result;

**int** ret = sBluetoothInterface->enable();

result = (ret == BT\_STATUS\_SUCCESS) ? JNI\_TRUE : JNI\_FALSE;

**return** result;

}

/external/bluetooth/bluedroid/btif/src/bluetooth.c中对bt\_interface\_t的enable的定义，函数继续调用btif\_enable\_bluetooth()，这个函数在btif\_core.c中定义

static int enable( void )

{

ALOGI("enable");

/\* sanity check \*/

if (interface\_ready() == FALSE)

return BT\_STATUS\_NOT\_READY;

return btif\_enable\_bluetooth();

}

bt\_status\_t btif\_enable\_bluetooth(void)

{

BTIF\_TRACE\_DEBUG0("BTIF ENABLE BLUETOOTH");

if (btif\_core\_state != BTIF\_CORE\_STATE\_DISABLED)

{

ALOGD("not disabled\n");

return BT\_STATUS\_DONE;

}

btif\_core\_state = BTIF\_CORE\_STATE\_ENABLING;

/\* Create the GKI tasks and run them \*/

bte\_main\_enable(btif\_local\_bd\_addr.address);

return BT\_STATUS\_SUCCESS;

}

/external/bluetooth/bluedroid/main/bte\_main.c

void bte\_main\_enable(uint8\_t \*local\_addr)

{

APPL\_TRACE\_DEBUG1("%s", \_\_FUNCTION\_\_);

/\* Initialize BTE control block \*/

BTE\_Init();

lpm\_enabled = FALSE;

if (bt\_hc\_if)

{

int result = bt\_hc\_if->init(&hc\_callbacks, local\_addr);

APPL\_TRACE\_EVENT1("libbt-hci init returns %d", result);

assert(result == BT\_HC\_STATUS\_SUCCESS);

if (hci\_logging\_enabled == TRUE)

bt\_hc\_if->logging(BT\_HC\_LOGGING\_ON, hci\_logfile);

#if (defined (BT\_CLEAN\_TURN\_ON\_DISABLED) && BT\_CLEAN\_TURN\_ON\_DISABLED == TRUE)

APPL\_TRACE\_DEBUG1("%s Not Turninig Off the BT before Turninig ON", \_\_FUNCTION\_\_);

bt\_hc\_if->set\_power(BT\_HC\_CHIP\_PWR\_OFF);

#endif

bt\_hc\_if->set\_power(BT\_HC\_CHIP\_PWR\_ON);

bt\_hc\_if->preload(NULL);

}

GKI\_create\_task((TASKPTR)btu\_task, BTU\_TASK, BTE\_BTU\_TASK\_STR,

(UINT16 \*) ((UINT8 \*)bte\_btu\_stack + BTE\_BTU\_STACK\_SIZE),

sizeof(bte\_btu\_stack));

GKI\_run(0);

}

/external/bluetooth/bluedroid/hci/src/bt\_hci\_bdroid.c

static void set\_power(bt\_hc\_chip\_power\_state\_t state)

{

int pwr\_state;

BTHCDBG("set\_power %d", state);

/\* Calling vendor-specific part \*/

pwr\_state = (state == BT\_HC\_CHIP\_PWR\_ON) ? BT\_VND\_PWR\_ON : BT\_VND\_PWR\_OFF;

if (bt\_vnd\_if)

bt\_vnd\_if->op(BT\_VND\_OP\_POWER\_CTRL, &pwr\_state);

else

ALOGE("vendor lib is missing!");

}

android/device/softwinner/common/hardware/realtak/bluetooth/libbt-vendor/rtl8723as/bt\_vendor\_rtk.c

static int op(bt\_vendor\_opcode\_t opcode, void \*param)

{

int retval = 0;

BTVNDDBG("op for %d", opcode);

switch(opcode)

{

case BT\_VND\_OP\_POWER\_CTRL:

{

BTVNDDBG("op for BT\_VND\_OP\_POWER\_CTRL");

int \*state = (int \*) param;

if (\*state == BT\_VND\_PWR\_OFF){

upio\_set\_bluetooth\_power(UPIO\_BT\_POWER\_OFF);

usleep(200000);

}

else if (\*state == BT\_VND\_PWR\_ON){

upio\_set\_bluetooth\_power(UPIO\_BT\_POWER\_ON);

usleep(500000);

}

}

break;

case BT\_VND\_OP\_FW\_CFG:

{

BTVNDDBG("op for BT\_VND\_OP\_FW\_CFG");

hw\_config\_start();

}

break;

case BT\_VND\_OP\_SCO\_CFG:

{

BTVNDDBG("op for BT\_VND\_OP\_SCO\_CFG");

retval = -1;

}

break;

case BT\_VND\_OP\_USERIAL\_OPEN:

{

BTVNDDBG("op for BT\_VND\_OP\_USERIAL\_OPEN");

int (\*fd\_array)[] = (int (\*)[]) param;

int fd, idx;

fd = userial\_vendor\_open((tUSERIAL\_CFG \*) &userial\_init\_cfg);

if (fd != -1)

{

for (idx=0; idx < CH\_MAX; idx++)

(\*fd\_array)[idx] = fd;

retval = 1;

}

/\* retval contains numbers of open fd of HCI channels \*/

}

break;

case BT\_VND\_OP\_USERIAL\_CLOSE:

{

BTVNDDBG("op for BT\_VND\_OP\_USERIAL\_CLOSE");

userial\_vendor\_close();

}

break;

case BT\_VND\_OP\_GET\_LPM\_IDLE\_TIMEOUT:

{

BTVNDDBG("op for BT\_VND\_OP\_GET\_LPM\_IDLE\_TIMEOUT");

}

break;

case BT\_VND\_OP\_LPM\_SET\_MODE:

{

BTVNDDBG("op for BT\_VND\_OP\_LPM\_SET\_MODE");

}

break;

case BT\_VND\_OP\_LPM\_WAKE\_SET\_STATE:

{

BTVNDDBG("op for BT\_VND\_OP\_LPM\_WAKE\_SET\_STATE");

}

break;

}

return retval;

}

# 2、获取已经配对的蓝牙设备

**public** **boolean** processMessage(Message msg) {

**switch**(msg.what) {

**case** *USER\_TURN\_ON*:

**if** (*DBG*) Log.*d*(*TAG*,"CURRENT\_STATE=OFF, MESSAGE = USER\_TURN\_ON");

notifyAdapterStateChange(BluetoothAdapter.*STATE\_TURNING\_ON*);

mPendingCommandState.setTurningOn(**true**);

transitionTo(mPendingCommandState);

sendMessageDelayed(*START\_TIMEOUT*, *START\_TIMEOUT\_DELAY*);

mAdapterService.processStart(); //启动系统支持的所有profiles的services

**break**;

**case** *USER\_TURN\_OFF*:

**if** (*DBG*) Log.*d*(*TAG*,"CURRENT\_STATE=OFF, MESSAGE = USER\_TURN\_OFF");

//**TODO**: Handle case of service started and stopped without enable

**break**;

**default**:

**if** (*DBG*) Log.*d*(*TAG*,"ERROR: UNEXPECTED MESSAGE: CURRENT\_STATE=OFF, MESSAGE = " + msg.what );

**return** **false**;

}

**return** **true**;

}

}

**private** **void** notifyAdapterStateChange(**int** newState) {

**int** oldState = mAdapterProperties.getState();

mAdapterProperties.setState(newState);

infoLog("Bluetooth adapter state changed: " + oldState + "-> " + newState);

mAdapterService.updateAdapterState(oldState, newState);

}

**void** updateAdapterState(**int** prevState, **int** newState){

**if** (mCallbacks !=**null**) {

**int** n=mCallbacks.beginBroadcast();

Log.*d*(*TAG*,"Broadcasting updateAdapterState() to " + n + " receivers.");

**for** (**int** i=0; i <n;i++) {

**try** {

mCallbacks.getBroadcastItem(i).onBluetoothStateChange(prevState,newState);

} **catch** (RemoteException e) {

Log.*e*(*TAG*, "Unable to call onBluetoothStateChange() on callback #" + i, e);

}

}

mCallbacks.finishBroadcast();

}

}

public void onBluetoothStateChanged(int bluetoothState) {

super.onBluetoothStateChanged(bluetoothState);

updateContent(bluetoothState, true);

}

private void updateContent(int bluetoothState, boolean scanState) {

。。。

case BluetoothAdapter.STATE\_ON:

preferenceScreen.removeAll();

preferenceScreen.setOrderingAsAdded(true);

mDevicePreferenceMap.clear();

// This device

if (mMyDevicePreference == null) {

mMyDevicePreference = new Preference(getActivity());

}

mMyDevicePreference.setTitle(mLocalAdapter.getName());

if (getResources().getBoolean(com.android.internal.R.bool.config\_voice\_capable)) {

mMyDevicePreference.setIcon(R.drawable.ic\_bt\_cellphone); // for phones

} else {

mMyDevicePreference.setIcon(R.drawable.ic\_bt\_laptop); // for tablets, etc.

}

mMyDevicePreference.setPersistent(false);

mMyDevicePreference.setEnabled(true);

preferenceScreen.addPreference(mMyDevicePreference);

。。。

// 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);

int numberOfPairedDevices = mPairedDevicesCategory.getPreferenceCount();

mDiscoverableEnabler.setNumberOfPairedDevices(numberOfPairedDevices);

。。。

｝

在这里显示已经配对过的蓝牙设备

# 3、搜索设备

http://blog.csdn.net/baimy1985/article/details/8996653

/packages/apps/Settings/src/com/android/settings/bluetooth/BluetoothSettings.java

初始化，IntentFilter类表示Intent过滤器, 大部分情况下, 每一个component都会定义一个或多个IntentFilter, 用于表明其可处理的Intent.

public BluetoothSettings() {

mIntentFilter = new IntentFilter(BluetoothAdapter.ACTION\_LOCAL\_NAME\_CHANGED);

}

private void startScanning() {

if (!mAvailableDevicesCategoryIsPresent) {

getPreferenceScreen().addPreference(mAvailableDevicesCategory);

}

mLocalAdapter.startScanning(true);

}

BluetoothLocalAdapter.startScanning()

---->BluetoothAdapter.startDiscovery()

---->AdapterService.startDiscovety()

--->com\_android\_bluetooth\_btservice\_AdapterService.cpp(**startDiscoveryNative**)

--->bluetooth.c(start\_discovery)

---->btif\_dm.c(btif\_dm\_start\_discovery) /bluedroid/btif/src/btif\_dm.c

bte\_search\_devices\_evt(tBTA\_DM\_SEARCH\_EVT event, tBTA\_DM\_SEARCH \*p\_data)

btif\_dm\_search\_devices\_evt (UINT16 event, char \*p\_param) 扫描到蓝牙设备的mac地址和设备名

---->bta\_dm\_api.c(BTA\_DmSearch)在这里获取远程的设备 /bluedroid/bta/dm/ bta\_dm\_api.c

/\* find nearby devices \*/

BTA\_DmSearch(&inq\_params, services, bte\_search\_devices\_evt);

void BTA\_DmSearch(tBTA\_DM\_INQ \*p\_dm\_inq, tBTA\_SERVICE\_MASK services, tBTA\_DM\_SEARCH\_CBACK \*p\_cback)

--->method\_deviceFoundCallback=env->GetMethodID(jniCallbackClass,"deviceFoundCallback", "([B)V");

---->com.android.bluetooth.btservice.JniCallbacks.java(deviceFoundCallback)

---->com.android.bluetooth.btservice.RemoteDevices.java(deviceFoundCallback)

Action\_Found

----->/bluetooth/BluetoothEventManager.java(DeviceFoundHandler())

private void updateContent(int bluetoothState, boolean scanState)用于更新“已配对”和“可用设备”

设备的存储：

/packages/apps/Settings/src/com/android/settings/bluetooth/BluetoothDevicePreference.java显示在蓝牙设置列表里面的已配对和可用蓝牙设备

/frameworks/base/core/java/android/bluetooth/BluetoothDevice.java代表远端连接的设备

mAddress用来保存蓝牙设备的物理地址

public void writeToParcel(Parcel out, int flags) {

out.writeString(mAddress);

}

/packages/apps/Bluetooth/src/com/android/bluetooth/btservice/RemoteDevice.java添加远程设备，BluetoothEventManager.java 处理RemoteDevice.java 里面的intent事件

DeviceProperties addDeviceProperties(**byte**[] address) {

**synchronized** (mDevices) {

DeviceProperties prop = **new** DeviceProperties();

BluetoothDevice device =

*mAdapter*.getRemoteDevice(Utils.*getAddressStringFromByte*(address));

prop.mAddress = address;

mDevices.put(device, prop);

**return** prop;

}

}

连接的远端设备的属性

**class** DeviceProperties {

**private** String mName;

**private** **byte**[] mAddress;

**private** **int** mBluetoothClass;

**private** **short** mRssi;

**private** ParcelUuid[] mUuids;

**private** **int** mDeviceType;

**private** String mAlias;

**private** **int** mBondState;

}

# 4、文件传递

http://blog.csdn.net/baimy1985/article/details/9124657

如果想要更好理解蓝牙OPP文件传输，需要了解OBEX基础协议

蓝牙在发送文件时发送端先来到packages/apps/bluetooth/com.android.bluetooth.opp.BluetoothOppLauncherActivity.java判断ACTION\_SEND和ACTION\_SEND\_MULTIPLE，根据发送文件的个数调用相应的分支，在每个分支最后都会创建一个线程，调用BLuetoothOppManager的saveSendingFileInfo（）函数，根据文件的个数调用不同的函数。

Thread t = **new** Thread(**new** Runnable() {

**public** **void** run() {

BluetoothOppManager.*getInstance*(BluetoothOppLauncherActivity.**this**)

.saveSendingFileInfo(mimeType,uris, **false**); //单个文件为例

//Done getting file info..Launch device picker

//and finish this activity

launchDevicePicker(); //

finish();

}

});

//保存要发送文件的信息

**public** **void** saveSendingFileInfo(String mimeType, String uriString, **boolean** isHandover) {

**synchronized** (BluetoothOppManager.**this**) {

mMultipleFlag = **false**;

mMimeTypeOfSendingFile = mimeType;

mUriOfSendingFile = uriString;

mIsHandoverInitiated = isHandover;

Uri uri = Uri.*parse*(uriString);

BluetoothOppUtility.*putSendFileInfo*(uri,

BluetoothOppSendFileInfo.*generateFileInfo*(mContext, uri, mimeType));

storeApplicationData();

}

}

发送端首先判断蓝牙设备是否打开，如果已经打开则直接进入已配对蓝牙设备界面，然后选择需要发送文件的设备，否则打开蓝牙设备进入BluetoothOppBtEnableActivity

**private** **final** **void** launchDevicePicker() {

**if** (!BluetoothOppManager.*getInstance*(**this**).isEnabled()) {

**if** (*V*) Log.*v*(*TAG*, "Prepare Enable BT!! ");

Intent in = **new** Intent(**this**, BluetoothOppBtEnableActivity.**class**);

in.setFlags(Intent.*FLAG\_ACTIVITY\_NEW\_TASK*);

startActivity(in);

} **else** {

**if** (*V*) Log.*v*(*TAG*, "BT already enabled!! ");

Intent in1 = **new** Intent(BluetoothDevicePicker.*ACTION\_LAUNCH*);

in1.setFlags(Intent.*FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS*);

in1.putExtra(BluetoothDevicePicker.*EXTRA\_NEED\_AUTH*, **false**);

in1.putExtra(BluetoothDevicePicker.*EXTRA\_FILTER\_TYPE*,

BluetoothDevicePicker.*FILTER\_TYPE\_TRANSFER*);

in1.putExtra(BluetoothDevicePicker.*EXTRA\_LAUNCH\_PACKAGE*,

Constants.*THIS\_PACKAGE\_NAME*);

in1.putExtra(BluetoothDevicePicker.*EXTRA\_LAUNCH\_CLASS*,

BluetoothOppReceiver.**class**.getName());

**if** (*V*) {Log.*d*(*TAG*,"Launching " +BluetoothDevicePicker.*ACTION\_LAUNCH* );}

startActivity(in1);

}

}

假设已经打开蓝牙设备，根据BluetoothDevicePicker.*ACTION\_LAUNCH定以跳转到bluetooth.DevicePickerActivity,*这里面加载了DevicePickerFragment类，选择一个设备调用

---->void onDevicePreferenceClick(BluetoothDevicePreference btPreference)

/packages/apps/Settings/src/com/android/settings/bluetooth/DevicePickerFragment.java

---->sendDevicePickedIntent(BluetoothDevice device)

发送BluetoothDevicePicker.ACTION\_DEVICE\_SELECTED广播

---->/packages/apps/bluetooth/com/android/bluetooth/opp/BluetoothOppReceiver.java接收广播

**if** (action.equals(BluetoothDevicePicker.*ACTION\_DEVICE\_SELECTED*)) {

BluetoothOppManager mOppManager = BluetoothOppManager.*getInstance*(context);

BluetoothDevice remoteDevice = intent.getParcelableExtra(BluetoothDevice.*EXTRA\_DEVICE*); //远程设备

。。

mOppManager.startTransfer(remoteDevice); //开始向远程设备传送文件

String deviceName = mOppManager.getDeviceName(remoteDevice);

String toastMsg;

**int** batchSize = mOppManager.getBatchSize();

**if** (mOppManager.mMultipleFlag) {

toastMsg = context.getString(R.string.*bt\_toast\_5*, Integer.*toString*(batchSize),

deviceName);

} **else** {

toastMsg = context.getString(R.string.*bt\_toast\_4*, deviceName);

}

Toast.*makeText*(context, toastMsg, Toast.*LENGTH\_SHORT*).show();

}

--->InsertShareInfoThread insertThread.start() 根据前面传送文件的个数选择发送函数

**public** **void** run() {

Process.*setThreadPriority*(Process.*THREAD\_PRIORITY\_BACKGROUND*);

**if** (mRemoteDevice == **null**) { Log.*e*(*TAG*, "Target bt device is null!"); **return**; }

**if** (mIsMultiple) { insertMultipleShare(); }

**else** { insertSingleShare(); }

**synchronized** (BluetoothOppManager.**this**) {

mInsertShareThreadNum--;

}

}

---->将共享文件的信息保存到数据库*opp.BluetoothOppProvider*中

**private** **void** insertSingleShare() {

ContentValues values = **new** ContentValues();

values.put(BluetoothShare.*URI*, mUri);

values.put(BluetoothShare.*MIMETYPE*, mTypeOfSingleFile);

values.put(BluetoothShare.*DESTINATION*, mRemoteDevice.getAddress());

**if** (mIsHandoverInitiated) {

values.put(BluetoothShare.*USER\_CONFIRMATION*,

BluetoothShare.*USER\_CONFIRMATION\_HANDOVER\_CONFIRMED*);

}

**final** Uri contentUri = mContext.getContentResolver().insert(BluetoothShare.*CONTENT\_URI*,

values);

**if** (*V*) Log.*v*(*TAG*, "Insert contentUri: " + contentUri + " to device: "

+ getDeviceName(mRemoteDevice));

}

---->BluetoothOppProvider.java中调用insert函数，将这条记录插入btopp表中，然后启动BluetoothOppService服务，调用onStartCommand函数，

**public** **int** onStartCommand(Intent intent, **int** flags, **int** startId) {

**if** (*V*) Log.*v*(*TAG*, "onStartCommand");

**if** (mAdapter == **null**) {

Log.*w*(*TAG*, "Local BT device is not enabled");

} **else** {

startListener();

}

updateFromProvider();

**return** *START\_NOT\_STICKY*;

}

---->UpdateThread.start()在这里面调用insertShare()函数，updateThread就是cursor和mShares的同步。而这其中显然是以cursor为主的，可以认为mShare用于更新cursor的内容。他们同步的关键就是固定的按照一定顺序排列的id，他们一起从0开始增加，若是mshares中的id小于cursor中的id，则意味这mShares中由cursor中没有的内容，那就使用deleteShare来删除对应的内容，若是两者id相等，则使用updateShare来更新cursor中内容，若是mShares的id大于cursor中的id，则意味着cursor中有内容美柚增加到mShares中，则需要使用insertShare来添加。

--->insertShare()函数就是把cursor加入到对应mShares中，然后启动相应发送和接收文件的transfer函数

**if** (mBatchs.size() == 0) {

//第一次则会新建oppBatch

BluetoothOppBatch newBatch = **new** BluetoothOppBatch(**this**, info);

newBatch.mId = mBatchId;

mBatchId++;

mBatchs.add(newBatch); //加入到mBatchs中

**if** (info.mDirection == BluetoothShare.*DIRECTION\_OUTBOUND*) {

**if** (*V*) Log.*v*(*TAG*, "Service create new Batch " + newBatch.mId

+ " for OUTBOUND info " + info.mId);

//若是发送，新建发送的transfer

mTransfer = **new** BluetoothOppTransfer(**this**, mPowerManager, newBatch);

} **else** **if** (info.mDirection == BluetoothShare.*DIRECTION\_INBOUND*) {

**if** (*V*) Log.*v*(*TAG*, "Service create new Batch " + newBatch.mId

+ " for INBOUND info " + info.mId);

//若是接收，新建接收的transfer，差别就在于接受没有Batch参数

mServerTransfer = **new** BluetoothOppTransfer(**this**, mPowerManager, newBatch,

mServerSession);

}

**if** (info.mDirection == BluetoothShare.*DIRECTION\_OUTBOUND* && mTransfer != **null**) {

**if** (*V*) Log.*v*(*TAG*, "Service start transfer new Batch " + newBatch.mId

+ " for info " + info.mId);

mTransfer.start(); //开始传输

} **else** **if** (info.mDirection == BluetoothShare.*DIRECTION\_INBOUND*

&& mServerTransfer != **null**) {

**if** (*V*) Log.*v*(*TAG*, "Service start server transfer new Batch " + newBatch.mId

+ " for info " + info.mId);

mServerTransfer.start();

}

/com/android/bluetooth/opp/BluetoothOppTransfer.java

首先检查蓝牙是否已经打开，然后根据接收文件调用startConnectSession()，发送文件调用startObexSession()。

**public** **void** start() {

**if** (!mAdapter.isEnabled()) {

Log.*e*(*TAG*, "Can't start transfer when Bluetooth is disabled for " + mBatch.mId);

markBatchFailed();

mBatch.mStatus = Constants.*BATCH\_STATUS\_FAILED*;

**return**;

}

**if** (mHandlerThread == **null**) {

**if** (*V*) Log.*v*(*TAG*, "Create handler thread for batch " + mBatch.mId);

mHandlerThread = **new** HandlerThread("BtOpp Transfer Handler",

Process.*THREAD\_PRIORITY\_BACKGROUND*);

mHandlerThread.start();

mSessionHandler = **new** EventHandler(mHandlerThread.getLooper());

**if** (mBatch.mDirection == BluetoothShare.*DIRECTION\_OUTBOUND*) {

/\* for outbound transfer, we do connect first \*/

startConnectSession();

} **else** **if** (mBatch.mDirection == BluetoothShare.*DIRECTION\_INBOUND*) {

/\*

\* for inbound transfer, it's already connected, so we start

\* OBEX session directly

\*/

startObexSession();

}

}

}

下面方法的作用是启动一个新的线程来连接远端设备，代码没有贴出来。如果创建RFCOMM的SocketConnectThread，则根据连接类型调用createInsecureRfcommSocket方法，创建BluetoothSocket对象之后调用socket.connect()方法连接，BluetoothOppRfcommTransport

**private** **void** startConnectSession() {

**if** (Constants.*USE\_TCP\_DEBUG*) {

mConnectThread = **new** SocketConnectThread("localhost", Constants.*TCP\_DEBUG\_PORT*, 0);

mConnectThread.start();

} **else** {

mConnectThread = **new** SocketConnectThread(mBatch.mDestination,**false**);

mConnectThread.start();

}

}

取得与远程设备的连接之后，会发送RF\_CONNECTED消息给自身，然后继续调用startObexSession()

**private** **void** startObexSession() {

mBatch.mStatus = Constants.*BATCH\_STATUS\_RUNNING*;

mCurrentShare = mBatch.getPendingShare();

…..

**if** (mBatch.mDirection == BluetoothShare.*DIRECTION\_OUTBOUND*) {

**if** (*V*) Log.*v*(*TAG*, "Create Client session with transport " + mTransport.toString());

mSession = **new** BluetoothOppObexClientSession(mContext, mTransport);

} **else** **if** (mBatch.mDirection == BluetoothShare.*DIRECTION\_INBOUND*) {

/\*

\* For inbounds transfer, a server session should already exists

\* before BluetoothOppTransfer is initialized. We should pass in a

\* mSession instance.

\*/

**if** (mSession == **null**) {

/\*\* set current share as error \*/

Log.*e*(*TAG*, "Unexpected error happened !");

markBatchFailed();

mBatch.mStatus = Constants.*BATCH\_STATUS\_FAILED*;

**return**;

}

**if** (*V*) Log.*v*(*TAG*, "Transfer has Server session" + mSession.toString());

}

mSession.start(mSessionHandler);

processCurrentShare();

}

跟踪发送文件的代码则需要创建一个客户端

/src/com/android/bluetooth/opp/BluetoothOppObexClientSession.java

**public** **void** start(Handler handler) {

**if** (*D*) Log.*d*(*TAG*, "Start!");

mCallback = handler;

mThread = **new** ClientThread(mContext, mTransport);

mThread.start();

}

创建完ClientThread线程之后继续执行run，发送文件。整个过程一共有以下几个关键步骤：1）wakelock锁的抓住和释放；2）obex层的连接和断开；3）执行sendFile函数

**public** **void** run() {

Process.*setThreadPriority*(Process.*THREAD\_PRIORITY\_BACKGROUND*);

**if** (*V*) Log.*v*(*TAG*, "acquire partial WakeLock");

wakeLock.acquire();

…....

**if** (!mInterrupted) {

connect();

}

**while** (!mInterrupted) {

**if** (!waitingForShare) {

doSend();

} **else** {

**try** {

**if** (*D*) Log.*d*(*TAG*, "Client thread waiting for next share, sleep for "

+ *sSleepTime*);

Thread.*sleep*(*sSleepTime*);

} **catch** (InterruptedException e) { }

}

}

disconnect();

**if** (wakeLock.isHeld()) {

**if** (*V*) Log.*v*(*TAG*, "release partial WakeLock");

wakeLock.release();

}

Message msg = Message.*obtain*(mCallback);

msg.what = BluetoothOppObexSession.*MSG\_SESSION\_COMPLETE*;

msg.obj = mInfo;

msg.sendToTarget();

}

**private** **void** doSend() {

**int** status = BluetoothShare.*STATUS\_SUCCESS*;

。。。。。

**if** (status == BluetoothShare.*STATUS\_SUCCESS*) {

/\* do real send \*/

**if** (mFileInfo.mFileName != **null**) { status = sendFile(mFileInfo); }

**else** { /\* this is invalid request \*/

status = mFileInfo.mStatus; }

waitingForShare = **true**;

} **else** { Constants.*updateShareStatus*(mContext1, mInfo.mId, status); }

**if** (status == BluetoothShare.*STATUS\_SUCCESS*) {

Message msg = Message.*obtain*(mCallback);

msg.what = BluetoothOppObexSession.*MSG\_SHARE\_COMPLETE*;

msg.obj = mInfo;

msg.sendToTarget();

} **else** {

Message msg = Message.*obtain*(mCallback);

msg.what = BluetoothOppObexSession.*MSG\_SESSION\_ERROR*;

mInfo.mStatus = status;

msg.obj = mInfo;

msg.sendToTarget();

}

}

sendFile是真正干活的，，执行完sendFile会把分享成功或失败的消息传回来，sendFile里会执行打包的过程，对于字段的含义要看Headset.java(frameworks/base/obex/javax/obex/HeaderSet.java)。

到这里发送文件的代码就完了，由于发送文件的时长是不确定的，所以在这个流程里面有很多线程代码。

发送完文件之后继续调用updateShareStatus()方法，其中调用sendIntentIfCompleted()方法去发送BluetoothShare.*TRANSFER\_COMPLETED\_ACTION*

**private** **int** sendFile(BluetoothOppSendFileInfo fileInfo) {

。。。。。。。

**while** (!mInterrupted && okToProceed && (position != fileInfo.mLength)) {

**if** (*V*) timestamp = System.*currentTimeMillis*();

readLength = a.read(buffer, 0, outputBufferSize);

outputStream.write(buffer, 0, readLength);

/\* check remote abort \*/

responseCode = putOperation.getResponseCode();

**if** (*V*) Log.*v*(*TAG*, "Response code is " + responseCode);

**if** (responseCode != ResponseCodes.OBEX\_HTTP\_CONTINUE

&& responseCode != ResponseCodes.OBEX\_HTTP\_OK) {

/\* abort happens \*/

okToProceed = **false**;

} **else** {

position += readLength;

**if** (*V*) {

Log.*v*(*TAG*, "Sending file position = " + position

+ " readLength " + readLength + " bytes took "

+ (System.*currentTimeMillis*() - timestamp) + " ms");

}

updateValues = **new** ContentValues();

updateValues.put(BluetoothShare.*CURRENT\_BYTES*, position);

mContext1.getContentResolver().update(contentUri, updateValues,

**null**, **null**);

}

}

}

。。。。。。

Constants.*updateShareStatus*(mContext1, mInfo.mId, status);

。。。。。。

}

## 4.2接收文件

/packages/apps/bluetooth/opp/BluetoothOppService.java

接收文件时由于在蓝牙打开，即蓝牙状态是BluetoothAdapter.STATE\_ON时会执行startSocketListener()，在这个函数调用了BluetoothOppRfcommListener.start()方法，该方法中新创建了线程监听客户端发送连接的请求，如果连接成功就会去accept该通道

**private** **void** startSocketListener() {

**if** (*V*) Log.*v*(*TAG*, "start RfcommListener");

mSocketListener.start(mHandler);

**if** (*V*) Log.*v*(*TAG*, "RfcommListener started");

}

**public** **synchronized** **boolean** start(Handler callback) {

**if** (mSocketAcceptThread == **null**) {

mCallback = callback;

mSocketAcceptThread = **new** Thread(*TAG*) {

**public** **void** run() {

。。。。

**for** (**int** i = 0; i < *CREATE\_RETRY\_TIME* && !mInterrupted; i++) {

**try** {

**if** (*V*) Log.*v*(*TAG*, "Starting RFCOMM listener....");

mBtServerSocket = mAdapter.listenUsingInsecureRfcommWithServiceRecord("OBEX Object Push", BluetoothUuid.*ObexObjectPush*.getUuid());

。。。。

BluetoothSocket clientSocket;

**while** (!mInterrupted) {

BluetoothServerSocket sSocket = mBtServerSocket;

**if** (sSocket ==**null**) {

mInterrupted = **true**;

} **else** {

clientSocket = sSocket.accept();

**if** (*V*) Log.*v*(*TAG*, "Accepted connection from "

+ clientSocket.getRemoteDevice());

BluetoothOppRfcommTransport transport = **new** BluetoothOppRfcommTransport(

clientSocket);

Message msg = Message.*obtain*();

msg.setTarget(mCallback);

msg.what = *MSG\_INCOMING\_BTOPP\_CONNECTION*;

msg.obj = transport;

msg.sendToTarget();

}

。。。。。

};

mInterrupted = **false**;

**if**(!Constants.*USE\_TCP\_SIMPLE\_SERVER*) {

mSocketAcceptThread.start();

}

}

**return** **true**;

}

监听完成之后回到BluetoothOppService收到MSG\_INCOMING\_BTOPP\_CONNECTION 消息，调用createServerSession()

**public** **void** handleMessage(Message msg) {

。。。。。

**case** BluetoothOppRfcommListener.*MSG\_INCOMING\_BTOPP\_CONNECTION*:

**if** (*D*) Log.*d*(*TAG*, "Get incoming connection");

ObexTransport transport = (ObexTransport)msg.obj;

/\*

\* Strategy for incoming connections:

\* 1. If there is no ongoing transfer, no on-hold connection, start it

\* 2. If there is ongoing transfer, hold it for 20 seconds(1 seconds \* 20 times)

\* 3. If there is on-hold connection, reject directly

\*/

**if** (mBatchs.size() == 0 && mPendingConnection == **null**) {

Log.*i*(*TAG*, "Start Obex Server");

createServerSession(transport);

} **else** {

**if** (mPendingConnection != **null**) {

Log.*w*(*TAG*, "OPP busy! Reject connection");

**try** {

transport.close();

} **catch** (IOException e) {

Log.*e*(*TAG*, "close tranport error");

}

} **else** **if** (Constants.*USE\_TCP\_DEBUG* && !Constants.*USE\_TCP\_SIMPLE\_SERVER*) {

Log.*i*(*TAG*, "Start Obex Server in TCP DEBUG mode");

createServerSession(transport);

} **else** {

Log.*i*(*TAG*, "OPP busy! Retry after 1 second");

mIncomingRetries = mIncomingRetries + 1;

mPendingConnection = transport;

Message msg1 = Message.*obtain*(mHandler);

msg1.what = *MSG\_INCOMING\_CONNECTION\_RETRY*;

mHandler.sendMessageDelayed(msg1, 1000);

}

}

**break**;

。。。。。。

}

**private** **void** createServerSession(ObexTransport transport) {

mServerSession = **new** BluetoothOppObexServerSession(**this**, transport);

mServerSession.preStart();

**if** (*D*) Log.*d*(*TAG*, "Get ServerSession " + mServerSession.toString()

+ " for incoming connection" + transport.toString());

}

mSession在创建的时候根据传入的参数进行初始化，新建线程调用run方法接收数据

/frameworks/base/obex/javax/obex/ServerSession.java

**public** **void** preStart() {

**if** (*D*) Log.*d*(*TAG*, "acquire full WakeLock");

mWakeLock.acquire();

**try** {

**if** (*D*) Log.*d*(*TAG*, "Create ServerSession with transport " + mTransport.toString());

mSession = **new** ServerSession(mTransport, **this**, **null**);

} **catch** (IOException e) {

Log.*e*(*TAG*, "Create server session error" + e);

}

}

# ５、蓝牙耳机接听电话

http://blog.csdn.net/baimy1985/article/details/9275559

Bluetooth Handset StateMachine

\* (Disconnected)

\* | ^

\* CONNECT | | DISCONNECTED

\* V |

\* (Pending)

\* | ^

\* CONNECTED | | CONNECT

\* V |

\* (Connected)

\* | ^

\* CONNECT\_AUDIO | | DISCONNECT\_AUDIO

\* V |

\* (AudioOn)

蓝牙接打电话对应HFP(Free your hand)，在手机与蓝牙耳机已连接时会优先触发蓝牙接听电话的代码流程起步代码在phone\src\com\android\phone\nCallScreen.java的connectBluetoothAudio() /disconnectBluetoothAudio()

void connectBluetoothAudio() {

if (VDBG) log("connectBluetoothAudio()...");

if (mBluetoothHeadset != null) {

// TODO(BT) check return

mBluetoothHeadset.connectAudio();

}

mBluetoothConnectionPending = true;

mBluetoothConnectionRequestTime = SystemClock.elapsedRealtime();

}

/HeadSetStateMachine.java中首先进入PendingState，然后转移状态到Connected中处理CONNECT\_AUDIO消息

**case** *CONNECT\_AUDIO*:

connectAudioNative(getByteAddress(mCurrentDevice));

**break**;

在JNI层的com\_android\_bluetooth\_hfp.cpp中进行连接

**static** jboolean **connectAudioNative**(JNIEnv \*env, jobject object, jbyteArray address) {

jbyte \*addr;

bt\_status\_t status;

**if** (!sBluetoothHfpInterface) **return** JNI\_FALSE;

addr = env->GetByteArrayElements(address, NULL);

**if** (!addr) {

jniThrowIOException(env, EINVAL);

**return** JNI\_FALSE;

}

**if** ( (status = sBluetoothHfpInterface->connect\_audio((bt\_bdaddr\_t \*)addr)) !=

BT\_STATUS\_SUCCESS) {

ALOGE("Failed HF audio connection, status: %d", status);

}

env->ReleaseByteArrayElements(address, addr, 0);

**return** (status == BT\_STATUS\_SUCCESS) ? JNI\_TRUE : JNI\_FALSE;

}

/external/bluetooth/bluedroid/btif/src/btif\_hf.c

static bt\_status\_t connect\_audio( bt\_bdaddr\_t \*bd\_addr )

{

CHECK\_BTHF\_INIT();

if (is\_connected(bd\_addr))

{

BTA\_AgAudioOpen(btif\_hf\_cb.handle);

/\* Inform the application that the audio connection has been initiated successfully \*/

btif\_transfer\_context(btif\_in\_hf\_generic\_evt, BTIF\_HFP\_CB\_AUDIO\_CONNECTING,

(char \*)bd\_addr, sizeof(bt\_bdaddr\_t), NULL);

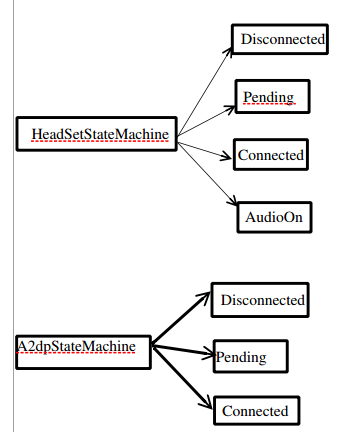
return BT\_STATUS\_SUCCESS;

}

return BT\_STATUS\_FAIL;

}

# 6、在蓝牙列表中连接蓝牙耳机



void onClicked() {

int bondState = mCachedDevice.getBondState();

if (mCachedDevice.isConnected()) {

askDisconnect();

} else if (bondState == BluetoothDevice.BOND\_BONDED) {

mCachedDevice.connect(true);

} else if (bondState == BluetoothDevice.BOND\_NONE) {

pair();

}

}

void connect(boolean connectAllProfiles) {

if (!ensurePaired()) {

return;

}

mConnectAttempted = SystemClock.elapsedRealtime();

connectWithoutResettingTimer(connectAllProfiles);

}

private void connectWithoutResettingTimer(boolean connectAllProfiles) {

。。

mIsConnectingErrorPossible = true;

int preferredProfiles = 0;

for (LocalBluetoothProfile profile : mProfiles) {

if (connectAllProfiles ? profile.isConnectable() : profile.isAutoConnectable()) {

if (profile.isPreferred(mDevice)) {

++preferredProfiles;

connectInt(profile);

}

}

}

if (DEBUG) Log.d(TAG, "Preferred profiles = " + preferredProfiles);

if (preferredProfiles == 0) {

connectAutoConnectableProfiles();

}

}

synchronized void connectInt(LocalBluetoothProfile profile) {

if (!ensurePaired()) {

return;

}

if (profile.connect(mDevice)) {

if (Utils.D) {

Log.d(TAG, "Command sent successfully:CONNECT " + describe(profile));

}

return;

}

Log.i(TAG, "Failed to connect " + profile.toString() + " to " + mName);

}

**public** **boolean** connect(BluetoothDevice device) {

A2dpService service = getService();

**if** (service == **null**) **return** **false**;

**return** service.connect(device);

}

**public** **boolean** connect(BluetoothDevice device) {

enforceCallingOrSelfPermission(*BLUETOOTH\_ADMIN\_PERM*,

"Need BLUETOOTH ADMIN permission");

**if** (getPriority(device) == BluetoothProfile.PRIORITY\_OFF) {

**return** **false**;

}

。。。

mStateMachine.sendMessage(A2dpStateMachine.*CONNECT*, device);

**return** **true**;

}

Disconnected状态

**case** *CONNECT*:

BluetoothDevice device = (BluetoothDevice) message.obj;

broadcastConnectionState(device, BluetoothProfile.*STATE\_CONNECTING*,

BluetoothProfile.*STATE\_DISCONNECTED*);

**if** (!connectA2dpNative(getByteAddress(device)) ) {

broadcastConnectionState(device, BluetoothProfile.*STATE\_DISCONNECTED*,

BluetoothProfile.*STATE\_CONNECTING*);

**break**;

}

**synchronized** (A2dpStateMachine.**this**) {

mTargetDevice = device;

transitionTo(mPending);

}

connectAdpNative()将继续进入JNI中com\_android\_bluetooth\_a2dp.cpp

**static** jboolean **connectA2dpNative**(JNIEnv \*env, jobject object, jbyteArray address) {

jbyte \*addr;

bt\_bdaddr\_t \* btAddr;

bt\_status\_t status;

ALOGI("%s: sBluetoothA2dpInterface: %p", \_\_FUNCTION\_\_, sBluetoothA2dpInterface);

**if** (!sBluetoothA2dpInterface) **return** JNI\_FALSE;

addr = env->GetByteArrayElements(address, NULL);

btAddr = (bt\_bdaddr\_t \*) addr;

**if** (!addr) {

jniThrowIOException(env, EINVAL);

**return** JNI\_FALSE;

}

**if** ((status = sBluetoothA2dpInterface->connect((bt\_bdaddr\_t \*)addr)) != BT\_STATUS\_SUCCESS) {

ALOGE("Failed HF connection, status: %d", status);

}

env->ReleaseByteArrayElements(address, addr, 0);

**return** (status == BT\_STATUS\_SUCCESS) ? JNI\_TRUE : JNI\_FALSE;

}

到这里蓝牙耳机的连接过程就完成了

# ６、设备配对

# 

接收远程设备配对的请求BluetoothDevice.ACTION\_PAIRING\_REQUEST，然后向BluetoothPairingDialog.java请求创建“蓝牙配对请求”对话框

Settings/src/com/android/settings/bluetooth/BluetoothPairingRequest.java

配对首先接收点击事件，然后判断设备的绑定状态

/packages/apps/Settings/src/com/android/settings/bluetooth/BluetoothDevicePreference.java

void onClicked() {

int bondState = mCachedDevice.getBondState();

if (mCachedDevice.isConnected()) {

askDisconnect();

} else if (bondState == BluetoothDevice.BOND\_BONDED) {

mCachedDevice.connect(true);

} else if (bondState == BluetoothDevice.BOND\_NONE) {

pair();

}

}

/Settings/src/com/android/settings/bluetooth/BluetoothDevicePreference.java

private void pair() {

if (!mCachedDevice.startPairing()) {

Utils.showError(getContext(), mCachedDevice.getName(),

R.string.bluetooth\_pairing\_error\_message);

}

}

/Settings/src/com/android/settings/bluetooth/CachedBluetoothDevice.java

boolean startPairing() {

// Pairing is unreliable while scanning, so cancel discovery

if (mLocalAdapter.isDiscovering()) { mLocalAdapter.cancelDiscovery(); }

if (!mDevice.createBond()) { return false; }

mConnectAfterPairing = true; // auto-connect after pairing

return true;

}

/frameworks/base/core/java/android/bluetooth/BluetoothDevice.java

public boolean createBond() {

if (sService == null) {

Log.e(TAG, "BT not enabled. Cannot create bond to Remote Device");

return false;

}

try {

return sService.createBond(this);

} catch (RemoteException e) {Log.e(TAG, "", e);}

return false;

}

/packages/apps/Bluetooth/src/com/android/bluetooth/btservice/AdapterService.java

**public** **boolean** createBond(BluetoothDevice device) {

**if** (!Utils.*checkCaller*()) {

Log.*w*(*TAG*,"createBond(): not allowed for non-active user");

**return** **false**;

}

AdapterService service = getService();

**if** (service == **null**) **return** **false**;

**return** service.createBond(device);

}

StableState

**boolean** createBond(BluetoothDevice device) {

enforceCallingOrSelfPermission(*BLUETOOTH\_ADMIN\_PERM*,

"Need BLUETOOTH ADMIN permission");

DeviceProperties deviceProp = mRemoteDevices.getDeviceProperties(device);

**if** (deviceProp != **null** && deviceProp.getBondState() != BluetoothDevice.*BOND\_NONE*) {

**return** **false**;

}

Message msg = mBondStateMachine.obtainMessage(BondStateMachine.*CREATE\_BOND*);

msg.obj = device;

mBondStateMachine.sendMessage(msg);

**return** **true**;

}

PendingCommandState

/packages/apps/Bluetooth/src/com/android/bluetooth/btservice/BondStateMachine.java

**private** **boolean** createBond(BluetoothDevice dev, **boolean** transition) {

**if** (dev.getBondState() == BluetoothDevice.*BOND\_NONE*) {

infoLog("Bond address is:" + dev);

**byte**[] addr = Utils.*getBytesFromAddress*(dev.getAddress());

**if** (!mAdapterService.createBondNative(addr)) {

sendIntent(dev, BluetoothDevice.*BOND\_NONE*,

BluetoothDevice.UNBOND\_REASON\_REMOVED);

**return** **false**;

} **else** **if** (transition) {

transitionTo(mPendingCommandState);

}

**return** **true**;

}

**return** **false**;

}

**static** jboolean **createBondNative**(JNIEnv\* env, jobject obj, jbyteArray address) {

ALOGV("%s:",\_\_FUNCTION\_\_);

jbyte \*addr;

jboolean result = JNI\_FALSE;

**if** (!sBluetoothInterface) **return** result;

addr = env->GetByteArrayElements(address, NULL);

**if** (addr == NULL) {

jniThrowIOException(env, EINVAL);

**return** result;

}

**int** ret = sBluetoothInterface->create\_bond((bt\_bdaddr\_t \*)addr);

env->ReleaseByteArrayElements(address, addr, NULL);

result = (ret == BT\_STATUS\_SUCCESS) ? JNI\_TRUE : JNI\_FALSE;

**return** result;

}

private void sendIntent(BluetoothDevice device, int newState, int reason) {

DeviceProperties devProp = mRemoteDevices.getDeviceProperties(device);

int oldState = BluetoothDevice.BOND\_NONE;

if (devProp != null) {

oldState = devProp.getBondState();

}

if (oldState == newState) return;

mAdapterProperties.onBondStateChanged(device, newState);

Intent intent = new Intent(BluetoothDevice.ACTION\_BOND\_STATE\_CHANGED);

intent.putExtra(BluetoothDevice.EXTRA\_DEVICE, device);

intent.putExtra(BluetoothDevice.EXTRA\_BOND\_STATE, newState);

intent.putExtra(BluetoothDevice.EXTRA\_PREVIOUS\_BOND\_STATE, oldState);

if (newState == BluetoothDevice.BOND\_NONE)

intent.putExtra(BluetoothDevice.EXTRA\_REASON, reason);

mAdapterService.sendBroadcastAsUser(intent, UserHandle.ALL,

AdapterService.BLUETOOTH\_PERM);

infoLog("Bond State Change Intent:" + device + " OldState: " + oldState

+ " NewState: " + newState);

}

void bondStateChangeCallback(int status, byte[] address, int newState) {

BluetoothDevice device = mRemoteDevices.getDevice(address);

。。

Message msg = obtainMessage(BONDING\_STATE\_CHANGE);

msg.obj = device;

if (newState == BOND\_STATE\_BONDED)

msg.arg1 = BluetoothDevice.BOND\_BONDED;

else if (newState == BOND\_STATE\_BONDING)

msg.arg1 = BluetoothDevice.BOND\_BONDING;

else

msg.arg1 = BluetoothDevice.BOND\_NONE;

msg.arg2 = status;

sendMessage(msg);

}

void aclStateChangeCallback(int status, byte[] address, int newState) {

BluetoothDevice device = getDevice(address);

。。。

Intent intent = null;

if (newState == AbstractionLayer.BT\_ACL\_STATE\_CONNECTED) {

intent = new Intent(BluetoothDevice.ACTION\_ACL\_CONNECTED);

debugLog("aclStateChangeCallback: State:Connected to Device:" + device);

} else {

intent = new Intent(BluetoothDevice.ACTION\_ACL\_DISCONNECTED);

debugLog("aclStateChangeCallback: State:DisConnected to Device:" + device);

}

intent.putExtra(BluetoothDevice.EXTRA\_DEVICE, device);

intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT);

mAdapterService.sendBroadcast(intent, mAdapterService.BLUETOOTH\_PERM);

}

void sspRequestCallback(byte[] address, byte[] name, int cod, int pairingVariant,

int passkey) {

//TODO(BT): Get wakelock and update name and cod

。。。

int variant;

boolean displayPasskey = false;

if (pairingVariant == AbstractionLayer.BT\_SSP\_VARIANT\_PASSKEY\_CONFIRMATION) {

variant = BluetoothDevice.PAIRING\_VARIANT\_PASSKEY\_CONFIRMATION;

displayPasskey = true;

} else if (pairingVariant == AbstractionLayer.BT\_SSP\_VARIANT\_CONSENT) {

variant = BluetoothDevice.PAIRING\_VARIANT\_CONSENT;

} else if (pairingVariant == AbstractionLayer.BT\_SSP\_VARIANT\_PASSKEY\_ENTRY) {

variant = BluetoothDevice.PAIRING\_VARIANT\_PASSKEY;

} else if (pairingVariant == AbstractionLayer.BT\_SSP\_VARIANT\_PASSKEY\_NOTIFICATION) {

variant = BluetoothDevice.PAIRING\_VARIANT\_DISPLAY\_PASSKEY;

displayPasskey = true;

} else {

errorLog("SSP Pairing variant not present");

return;

}

BluetoothDevice device = getDevice(address);

if (device == null) {

warnLog("Device is not known for:" + Utils.getAddressStringFromByte(address));

addDeviceProperties(address);

device = getDevice(address);

}

Intent intent = new Intent(BluetoothDevice.ACTION\_PAIRING\_REQUEST);

intent.putExtra(BluetoothDevice.EXTRA\_DEVICE, device);

if (displayPasskey) {

intent.putExtra(BluetoothDevice.EXTRA\_PAIRING\_KEY, passkey);

}

intent.putExtra(BluetoothDevice.EXTRA\_PAIRING\_VARIANT, variant);

mAdapterService.sendBroadcast(intent, mAdapterService.BLUETOOTH\_ADMIN\_PERM);

}

接收配对的设备属于服务端，注册了静态的接收请求，当对方请求连接蓝牙设备的时候，在BluetoothPairingRequest.java中就会接收连接请求，并弹出连接请求对话框，同时给发送方发送aclStateChangeCallback进行连接，同时向发送方发送sspRequestCallback，发送方就会弹出连接对话框

/packages/apps/Settings/AndroidManifest.xml

<receiver android:name=".bluetooth.BluetoothPairingRequest">

<intent-filter>

<action android:name="android.bluetooth.device.action.PAIRING\_REQUEST" />

<action android:name="android.bluetooth.device.action.PAIRING\_CANCEL" />

</intent-filter>

</receiver>

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);

} else {

// Put up a notification that leads to the dialog

Resources res = context.getResources();

Notification.Builder builder = new Notification.Builder(context)

.setSmallIcon(android.R.drawable.stat\_sys\_data\_bluetooth)

.setTicker(res.getString(R.string.bluetooth\_notif\_ticker));

PendingIntent pending = PendingIntent.getActivity(context, 0,

pairingIntent, PendingIntent.FLAG\_ONE\_SHOT);

String name = intent.getStringExtra(BluetoothDevice.EXTRA\_NAME);

if (TextUtils.isEmpty(name)) {

name = device != null ? device.getAliasName() :

context.getString(android.R.string.unknownName);

}

。。。

}

}

在双方都配对成功之后状态机就会调用bondStateChangeCallback

void bondStateChangeCallback(int status, byte[] address, int newState) {

BluetoothDevice device = mRemoteDevices.getDevice(address);

if (device == null) { device = mAdapter.getRemoteDevice(Utils.getAddressStringFromByte(address)); }

Message msg = obtainMessage(BONDING\_STATE\_CHANGE);

msg.obj = device;

if (newState == BOND\_STATE\_BONDED)

msg.arg1 = BluetoothDevice.BOND\_BONDED;

else if (newState == BOND\_STATE\_BONDING)

msg.arg1 = BluetoothDevice.BOND\_BONDING;

else

msg.arg1 = BluetoothDevice.BOND\_NONE;

msg.arg2 = status;

sendMessage(msg);

}

/AdapterProperties.java

void onBondStateChanged(BluetoothDevice device, int state)

{

try {

byte[] addrByte = Utils.getByteAddress(device);

DeviceProperties prop = mRemoteDevices.getDeviceProperties(device);

if (prop == null) prop = mRemoteDevices.addDeviceProperties(addrByte);

prop.setBondState(state);

if (state == BluetoothDevice.BOND\_BONDED) {

if(!mBondedDevices.contains(device)) {

debugLog("Adding bonded device:" + device);

mBondedDevices.add(device);

}

} else if (state == BluetoothDevice.BOND\_NONE) {

if (mBondedDevices.remove(device)) debugLog("Removing bonded device:" + device);

else debugLog("Failed to remove device: " + device);

}

}

}

此时AdapterService就会更新显示的设备，调用getBondedDevices方法，状态机中的状态PendingCommandState就会进入 BONDING\_STATE\_CHANGING，同时接收方会给发送方发送aclStateChangeCallback断开连接。