以展讯8810平台-android2.3代码为例：

1. 类SyncSimContactsReceiver：

这个类是一个广播接收器（broadcastReceiver），主要看它的onReceive()方法： 在接收到android.intent.action.BOOT\_COMPLETED这个intent的时候，会执行startService(),service对应的类是SyncSimContactsService

2. 类SyncSimContactsService：

这个类是一个服务（service），service被启动后，

2.1 先执行onCreate（），在onCreate()中会创建一个handler（mServiceHandler），

mServiceHandler = new ServiceHandler();

ServiceHandler是一个SyncSimContactsService的内部类，这个类实现了一个handleMessage方法.

创建Handler时默认没有传入参数，那么系统就会默认将当前线程的looper绑定到handler上，looper对象中维护着一个消息队列，handler发送的消息都会存储在这个消息队列中，looper不断的遍历这个消息队列，取出消息交给handleMessage（）来处理，因为looper属于当前线程，所以handleMessage（）就会在当前线程中执行。

2.2 再执行onStartCommand（），在onStartCommand()中主要执行doServicehandler（），在doServicehandler（）中，mServiceHandler会发送两个MESSAGE\_INIT消息，消息中的arg2参数会记录sim卡的index，表示要对两个sim卡都行初始化。

2.3 由于mServiceHandler绑定的是当前线程的消息队列，因此当前线程的消息队列收到MESSAGE\_INIT后会执行handleMessage方法，handleMessage对消息进行解析，消息中有4个主要参数：

1. what： 存储的值是messageId，在这里就是MESSAGE\_INIT。

2 arg1： 存储的值是startId， 这个参数是service在执行onStartCommand方法时，作为形参传入的，表示是由谁启动的service

3. arg2： 存储的值是phoneId， 表明是哪个sim卡。

4. obj： 存储的值是intent， 这个值也是service在执行onStartCommand方法时，作为形参传入的，表示是哪个intent来启动的service。

接下来就是具体处理MESSAGE\_INIT这个消息了，先要根据phoneId获取到对应的TelephonyManager的对象，然后通过调用getSimOperator方法来获取sim\_oper\_num（MCC+MNC)，如果这个值有效，则执行importDualSimAction方法来导入sim卡联系人的数据了。

在importDualSimAction中会创建一个线程，并启动这个线程：

DualSimcardImportThread simImport = new DualSimcardImportThread(serviceId,new ContactsAccount(accountName, Account.SIM\_ACCOUNT\_TYPE, uri), phoneId);

simImport.start();

DualSimcardImportThread是一个内部类， 它的构造函数的形参有3个：

1. serviceId：即前面提到的startId，

2. ContactsAccount：这个对象里面携带了访问sim卡的URI数据，如果是sim1，则对应的URI是"content://icc0/adn"，如果是sim2，对应的URI是"content://icc1/adn"

3. phoneId: 前面已经提到过。

这个线程类最主要的当然是实现了run方法，

在run方法中，先执行deleteSimAction方法，删除本地数据库"content://com.android.contacts/raw\_contacts"中account\_name=sim1 or sim2的数据，这些联系人不是本地联系人，所以开机后需要删除后重新加载。

接下来就是访问sim卡数据库了“content://icc0/adn”：

simCursor = mResolver.query(mAccount.getContactsAccountUri(),SIM\_COLUMN, null, null, null);

在phone.apk的manifest.xml中：

<provider android:name="MsmsIccProvider"

android:authorities="icc"

android:multiprocess="true"

android:readPermission="android.permission.READ\_CONTACTS"

android:writePermission="android.permission.WRITE\_CONTACTS" />

<provider android:name="MsmsIccProvider"

android:authorities="icc0"

android:multiprocess="true"

android:readPermission="android.permission.READ\_CONTACTS"

android:writePermission="android.permission.WRITE\_CONTACTS" />

<provider android:name="MsmsIccProvider"

android:authorities="icc1"

android:multiprocess="true"

android:readPermission="android.permission.READ\_CONTACTS"

android:writePermission="android.permission.WRITE\_CONTACTS" />

因此通过URI的匹配，我们可以访问到MsmsIccProvider这个类的Query方法

3. 类MsmsIccProvider

MsmsIccProvider的继续关系是MsmsIccProvider-》IccProvider-》ContentProvider，它是一个contentProvider，

这里先看它实现的Query方法，

在Query方法中，会执行loadFromEf方法，入口参数是根据URI匹配得到的ADN/FDN/SDN，

ADN: Abbreviated dialing number, 就是常规的用户号码，用户可以存储/删除

FDN：Fixed dialer number，固定拨号，固定拨号功能让您设置话机的使用限制，当您开启固定拨号功能后，您只可以拨打存储的固定拨号列表中的号码。固定号码表存放在SIM卡中。能否使用固定拨号功能取决于SIM卡类型以及网络商是否提供此功能。

SDN：Service dialing number，系统拨叫号码，网络服务拨号，固化的用户不能编辑。

从以上的描述，我们可以看到，一般情况下都是访问ADN。

在loadFromEf中，要先得到一个IIccPhoneBook对象：

IIccPhoneBook iccIpb = IIccPhoneBook.Stub.asInterface(ServiceManager.getService(PhoneFactory.getServiceName("simphonebook", phoneId)));

这个对象是用AIDL接口来获取到的，然后调用getAdnRecordsInEf方法：

adnRecords = iccIpb.getAdnRecordsInEf(efType);

这是通过AIDL接口实现的方法调用，所以先要找到Stub的实体类，因为最终是调用到了Stub实体类的getAdnRecordsInEf方法.这个实体类的对象是通过ServiceManager.getService来获取的，那么找到addService的地方就可以发现它了。

4. 类IccPhoneBookInterfaceManagerProxy

IccPhoneBookInterfaceManagerProxy这是stub的实体类，继承了IIccPhoneBook.Stub，在它的构造函数中执行了addService方法：

String serviceName = PhoneFactory.getServiceName("simphonebook", phoneId);

if(ServiceManager.getService(serviceName) == null) {

ServiceManager.addService(serviceName, this);

}

addService方法传入参数为当前类的对象，因此，在PhoneFactory.getServiceName("simphonebook", phoneId)是获得的就是IccPhoneBookInterfaceManagerProxy类的对象。

那么在前面（3）中提到的MsmsIccProvider类中iccIpb.getAdnRecordsInEf方法实际就调用到了IccPhoneBookInterfaceManagerProxy类的getAdnRecordsInEf方法。

在getAdnRecordsInEf方法中，执行：

mIccPhoneBookInterfaceManager.getAdnRecordsInEf(efid);

mIccPhoneBookInterfaceManager是IccPhoneBookInterfaceManagerProxy的一个成员对象，它是何时被赋值的呢，注意在IccPhoneBookInterfaceManagerProxy的构造函数中：

mIccPhoneBookInterfaceManager = iccPhoneBookInterfaceManager;

iccPhoneBookInterfaceManager是构造函数传入的形参，这么看来，还要看IccPhoneBookInterfaceManagerProxy这个类是何时被实例化的？

5. 类PhoneProxy

在PhoneProxy的构造函数中，执行：

mIccPhoneBookInterfaceManagerProxy = new IccPhoneBookInterfaceManagerProxy(phone.getIccPhoneBookInterfaceManager());

mIccPhoneBookInterfaceManagerProxy是PhoneProxy的一个成员，在这里类IccPhoneBookInterfaceManagerProxy被实例化，之前我们提到，我们需要找到在实例化这个类时传入的形参，在这里我们看到这个形参是

通过phone.getIccPhoneBookInterfaceManager()来会获取的。这个获取到的对象就是（4）中的mIccPhoneBookInterfaceManager。

而phone是PhoneProxy的构造函数的形参传入的一个对象，要找到这个对象的出处，就要看类PhoneProxy是何时被实例化的？

6.PhoneFactory

在类PhoneFactory中有一个makeDefaultPhone方法，里面执行了：

sCommandsInterface[phone\_index] = new SprdRIL(context, networkMode, cdmaSubscription, phone\_index);

sProxyPhone[phone\_index] = new SprdPhoneProxy(new TDPhone(context,sCommandsInterface[phone\_index], sPhoneNotifier[phone\_index]));

先看类SprdPhoneProxy的继承关系： SprdPhoneProxy-》PhoneProxy-》Handler

也就是说，在这里，类PhoneProxy被实例化，当然实际上是它的子类SprdPhoneProxy被实例化，构造函数传入的形参是一个类TDPhone的对象，

类TDPhone的继承关系： TDPhone->GSMPhone->PhoneBase->Handler

前面（5）中要找的phone.getIccPhoneBookInterfaceManager方法的返回值就是类TDPhone的getIccPhoneBookInterfaceManager方法的返回值，getIccPhoneBookInterfaceManager方法在类TDPhone中没有实现，而是在它的父类GSMPhone中实现的。

7.GSMPhone

在类GSMPhone中实现了getIccPhoneBookInterfaceManager方法

public IccPhoneBookInterfaceManager getIccPhoneBookInterfaceManager(){

return mSimPhoneBookIntManager;

}

mSimPhoneBookIntManager是一个成员，它在类GSMPhone的构造函数中被赋值：

public GSMPhone (Context context, CommandsInterface ci, PhoneNotifier notifier, boolean unitTestMode) {

。。。。

mSimPhoneBookIntManager = new SimPhoneBookInterfaceManager(this);

。。。。

}

mSimPhoneBookIntManager就是（6）中的类TDPhone的getIccPhoneBookInterfaceManager方法返回值

8. SimPhoneBookInterfaceManager

类SimPhoneBookInterfaceManager的继承关系：SimPhoneBookInterfaceManager-》IccPhoneBookInterfaceManager

（4）中的mIccPhoneBookInterfaceManager.getAdnRecordsInEf方法实际上就是类SimPhoneBookInterfaceManager的getAdnRecordsInEf方法，

而getAdnRecordsInEf在类SimPhoneBookInterfaceManager中没有实现，而是在它的父类IccPhoneBookInterfaceManager中实现的，

（3）中的iccIpb.getAdnRecordsInEf方法执行调用的就是IccPhoneBookInterfaceManager.getAdnRecordsInEf方法

9. IccPhoneBookInterfaceManager

类IccPhoneBookInterfaceManager中实现了getAdnRecordsInEf方法

在getAdnRecordsInEf方法中，执行：

adnCache.requestLoadAllAdnLike(efid, adnCache.extensionEfForEf(efid), response);

adnCache是类AdnRecordCache的对象

10. AdnRecordCache

在类AdnRecordCache中实现了requestLoadAllAdnLike方法，

在requestLoadAllAdnLike中，执行：

new AdnRecordLoader(mFh).loadAllFromEF(efid, extensionEf,obtainMessage(EVENT\_LOAD\_ALL\_ADN\_LIKE\_DONE, efid, 0))

这里实例化一个类AdnRecordLoader的对象，并且调用该对象的loadAllFromEF方法

11. AdnRecordLoader

在类AdnRecordLoader中实现了loadAllFromEF方法，

在loadAllFromEF方法中，执行：

mFh.loadEFLinearFixedAll(ef, obtainMessage(EVENT\_ADN\_LOAD\_ALL\_DONE));

mFh是类IccFileHandler的对象，实际上是它的子类TDUSIMFileHandler的对象，继承关系是：TDUSIMFileHandler-》SIMFileHandler-》IccFileHandler

12. IccFileHandler

在类IccFileHandler中实现了loadEFLinearFixedAll方法，

在loadEFLinearFixedAll方法中，执行：

phone.mCM.iccIO(COMMAND\_GET\_RESPONSE, fileid, getEFPath(fileid),0, 0, GET\_RESPONSE\_EF\_SIZE\_BYTES, null, null, response);

mCM是类SprdRIL的对象，SprdRIL的继承关系是：SprdRIL-》RIL

13. RIL

在类RIL中实现了iccIO方法，

在iccIO方法中，执行：

RILRequest rr = RILRequest.obtain(RIL\_REQUEST\_SIM\_IO, result);

创建一个RILRequest对象，requestId=RIL\_REQUEST\_SIM\_IO

然后把这个对象当作一个obj，放到message中，

msg = mSender.obtainMessage(EVENT\_SEND, rr);

messageId=EVENT\_SEND

mSender是一个handler对象，它是如何生成的呢，在RIL的构造函数中可以看到：

mSenderThread = new HandlerThread("RILSender");

mSenderThread.start();

Looper looper = mSenderThread.getLooper();

mSender = new RILSender(looper);

mSender在实例化时传入的looper是发送子线程的looper，因此，mSender发出的消息会发送到这个子线程的消息队列中，也就是mSenderThread这个线程中，handleMessage也会在这个子线程中执行。

接下来看类RILSender的handleMessage方法，

在handleMessage中，解析出来msg.what=EVENT\_SEND,从msg.obj中取出数据，写入socket

s = mSocket;

s.getOutputStream().write(dataLength);

s.getOutputStream().write(data);

mSocket是在另外一个子线程，即接收子线程中创建的，

s = new LocalSocket();

l = new LocalSocketAddress(SOCKET\_NAME\_RIL,LocalSocketAddress.Namespace.RESERVED);

s.connect(l);

mSocket = s;

至此，数据请求已经通过socket发送给rild了，rild会解析这些数据，再组装成AT命令，通过串口发送给modem，modem会返回AT数据给rild，rild再解析AT数据，然后打包通过socket发挥给RIL.java来接收。

实际上，前面这一句描述有些含糊，从（1）中我们可以看到，从SyncSimContactsReceiver接收到intent开始，一直到（13）整个代码执行都工作在Contacts.apk(android.process.acore)这个进程中，只不过有的是在主线程中，有的是在子线程中，这个应用进程通过socket与rild这个守护进程通信，将数据传送到rild这个进程中，rild得到串口返回的数据后，再通过socket发送给应用层进程Contacts.apk(android.process.acore).

这里，我们假设数据已经通过socket返回给应用层进程Conatacts.apk(android.process.acore)，那么RIL.java是怎么接收的呢？

在类RIL的构造函数中可以看到：

mReceiver = new RILReceiver();

mReceiverThread = new Thread(mReceiver, "RILReceiver");

mReceiverThread.start();

这里创建了一个接收子线程，这个线程用来接收rild进程通过socket返回来的数据，

我们看一下接收子线程的run方法，

在run方法中，执行：

InputStream is = mSocket.getInputStream();

length = readRilMessage(is, buffer);

p = Parcel.obtain();

p.unmarshall(buffer, offset, length - offset);

p.setDataPosition(0);

processResponse(p);

从socket接收到的数据进行解析，主要在processResponse方法中执行，

ret = responseICC\_IO(p);

AsyncResult.forMessage(rr.mResult, ret, null);

rr.mResult.sendToTarget();

将数据解析出来后，打包到message中，发送出去，那么发送到哪个消息队列中了呢？

rr.mResult存储的message对象是iccIO方法的形参传入的，而iccIO方法是（12）中的类IccFileHandler的loadEFLinearFixedAll方法调用的，

14. IccFileHandler

在IccFileHandler中实现了loadEFLinearFixedAll方法，

public void loadEFLinearFixedAll(int fileid, Message onLoaded) {

Message response = obtainMessage(EVENT\_GET\_RECORD\_SIZE\_DONE,

new LoadLinearFixedContext(fileid,onLoaded));

phone.mCM.iccIO(COMMAND\_GET\_RESPONSE, fileid, getEFPath(fileid),

0, 0, GET\_RESPONSE\_EF\_SIZE\_BYTES, null, null, response);

}

这里的response就是（13）中的rr.mResult，

IccFileHandler的继承关系：IccFileHandler-》Handler

这里我们也看一下IccFileHandler的构造函数，

protected IccFileHandler(PhoneBase phone) {

super();

this.phone = phone;

}

在调用父类构造函数时，没有传入参数，意味着这个handler是与主线程的消息队列绑定的，handleMessage方法就在主线程中执行，

那么（13）中的rr.mResult.sendToTarget()发送到了主线程中，主线程会调用IccFileHandler的handleMessage方法进行处理，

public void handleMessage(Message msg) {

。。。。。。。

case EVENT\_GET\_RECORD\_SIZE\_DONE:

。。。。。。。。

lc.countRecords = size / lc.recordSize;

phone.mCM.iccIO(COMMAND\_READ\_RECORD, lc.efid, getEFPath(lc.efid),

lc.recordNum,

READ\_RECORD\_MODE\_ABSOLUTE,

lc.recordSize, null, null,

obtainMessage(EVENT\_READ\_RECORD\_DONE, lc));

。。。。。。。。。

根据socket返回的数据，计算出联系人record的totalNumber，然后再一次通过调用类RIL的iccIO方法向socket发送数据请求读取第一个联系人record的具体内容，然后socket接收到返回数据后，发送message给消息队列，继续由类IccFileHandler的handleMessage方法处理，

case EVENT\_READ\_RECORD\_DONE：

。。。。。。

lc.results.add(result.payload);

lc.recordNum++;

。。。。。。

phone.mCM.iccIO(COMMAND\_READ\_RECORD, lc.efid, getEFPath(lc.efid),lc.recordNum，READ\_RECORD\_MODE\_ABSOLUTE,lc.recordSize, null, null,obtainMessage(EVENT\_READ\_RECORD\_DONE,0,pathNum,lc));

。。。。。。

把读取到record内容保存起来，然后num++，继续读取下一条record，依此类推，逐条的来发送请求和接收数据，sim的容量是250，则要发送250个数据请求给rild。

当最后一个数据接收到之后， 就要发送message给上层的消息队列了，

response = lc.onLoaded;

response.sendToTarget();

这里的lc.onLoaded是在（11）中的mFh.loadEFLinearFixedAll(ef, obtainMessage(EVENT\_ADN\_LOAD\_ALL\_DONE))传入的，因此发送消息的handler是类AdnRecordLoader的对象，消息队列中接收到消息后

就要调用类AdnRecordLoader的handleMessage方法了

15. AdnRecordLoader

类AdnRecordLoader的继承关系：AdnRecordLoader-》Handler

在类AdnRecordLoader中实现了handleMessage方法，

case EVENT\_ADN\_LOAD\_ALL\_DONE:

。。。

for (int i = 0, s = datas.size(); i < s; i++) {

adn = new AdnRecord(ef, 1 + i, datas.get(i));

adns.add(adn);

}

userResponse.sendToTarget();

....

将数据打包到消息中，发送到消息队列中，那么userResponse这个消息又是和哪个handler关联的呢？

public void loadAllFromEF(int ef, int extensionEF, Message response) {

this.ef = ef;

this.extensionEF = extensionEF;

this.userResponse = response;

。。。

mFh.loadEFLinearFixedAll(ef, obtainMessage(EVENT\_ADN\_LOAD\_ALL\_DONE));

}

可以看到userResponse是loadAllFromEF方法的形参传入的，这是由（10）中的

new AdnRecordLoader(mFh).loadAllFromEF(efid, extensionEf,obtainMessage(EVENT\_LOAD\_ALL\_ADN\_LIKE\_DONE, efid, 0))调用的

这个message（EVENT\_LOAD\_ALL\_ADN\_LIKE\_DONE）是类AdnRecordCache的对象发出的，那么发出这个消息后，由类AdnRecordCache的handleMessage方法调用的

16. AdnRecordCache

类AdnRecordCache实现了HandleMessage方法，

case EVENT\_LOAD\_ALL\_ADN\_LIKE\_DONE:

waiters = adnLikeWaiters.get(efid);

adnLikeWaiters.delete(efid);

notifyWaiters(waiters, ar);

在notifyWaiters方法中，

Message waiter = waiters.get(i);

AsyncResult.forMessage(waiter, ar.result, ar.exception);

waiter.sendToTarget();

这里还是发送一个消息到消息队列中，这个消息是什么呢？ waiter消息是在requestLoadAllAdnLike方法中被赋值的，

public void requestLoadAllAdnLike(int efid, int extensionEf,Message response) {

。。。

waiters.add(response);

。。。

}

这个response是形参带入的，就是（9）中的adnCache.requestLoadAllAdnLike(efid, adnCache.extensionEfForEf(efid), response)调用的，

即类IccPhoneBookInterfaceManager的getAdnRecordsInEf方法调用的，

17.IccPhoneBookInterfaceManager

在类IccPhoneBookInterfaceManager中实现了getAdnRecordsInEf方法，

public synchronized List<AdnRecord> getAdnRecordsInEf(int efid) {

...

Message response = mBaseHandler.obtainMessage(EVENT\_LOAD\_DONE, status);

adnCache.requestLoadAllAdnLike(efid, adnCache.extensionEfForEf(efid), response);

waitForResult(status);

...

}

这里可以看到message是mBaseHandler发出的，由mBaseHandler的handleMessage方法处理，

case EVENT\_LOAD\_DONE:

notifyPending(ar);

这里执行解锁，那么getAdnRecordsInEf方法中的waitForResult就可以返回了，

至此，getAdnRecordsInEf方法才执行完毕，而这是个AIDL接口调用，它是由（3）中的iccIpb.getAdnRecordsInEf(efType)发起的，现在执行到了类MsmsIccProvider的loadFromEf方法中，

18. MsmsIccProvider

在类MsmsIccProvider中，实现了loadFromEf方法，

private ArrayList<ArrayList> loadFromEf(int efType, int phoneId) {

。。。

adnRecords = iccIpb.getAdnRecordsInEf(efType);

int N = adnRecords.size();

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

loadRecord(adnRecords.get(i), results);

}

这个方法执行完毕之后，走回到类MsmsIccProvider的query方法中，将results转换成cursor对象返回。

query方法执行完毕后，（2）中的mResolver.query(mAccount.getContactsAccountUri(),SIM\_COLUMN, null, null, null);也就返回了

19. SyncSimContactsService

类SyncSimContactsService的DualSimcardImportThread这个线程的run方法中，query语句就执行完毕了

至此，从sim卡上就读取到全部的联系人数据了，接下来就是将这些数据存储到本地数据库中了，当然要做一下标记，account\_name=sim1 or sim2。