# Android Contacts总结

## Android 中的Contacts模块的结构

|  |  |  |
| --- | --- | --- |
| 模块 | 源码路径 | 说明 |
| UI模块 | /packages/apps/Contacts/ | 通讯录的UI层，包含了通讯录，calllog，dialer等activity，以及新增的多选操作，SIM卡操作等界面。对通讯录的修改，除了要改数据库，基本都在这里面 |
| 数据模块 | /packages/providers/ContactsProvider/ | 主要是SQLITE的使用，对通讯录的数据库操作。由于contentprovider机制，可以说把数据与UI完全分开了。 |
| 系统模块（没有特定的模块，只列出涉及到的几个部分） | /frameworks/base/core/java/android/database | Android的数据库模块，重要的类：Cursor，重要的包：sqlite |
| /frameworks/base/core/java/android/pim | 封装了对vcard的操作 |
| /frameworks/base/core/java/android/provider | 封装了Contentprovider的辅助类，主要是对数据库中的表和URI等的定义，提供给app调用，基本上系统应用涉及到数据库的都有定义，重点关注下ContactsContract，CallLog |
| /frameworks/base/core/java/android/accounts | 账户相关，通讯录里涉及到账户 |
| /frameworks/base/telephony/java/com/android/internal/telephony | SIM卡相关，卡上联系人的读写流程，与RIL层紧密关联 |

## 数据模块

### SQLite数据库

常用语句：

|  |  |
| --- | --- |
| 创建表 | CREATE TABLE [表名] [列名] |
| 查询 | SELECT [列名] FROM [表名] WHERE [条件] |
| 插入 | INSERT [列名] INTO [表名] VALUES [列名对应的值] |
| 删除 | DELETE FROM[表名] WHERE [条件] |
| 更新 | UPDATE [表名] SET [更新项] WHERE [条件] |

更多关于SQLite的知识，如TABLE（表），INDEX（索引），VIEW（视图）等，请参考



### Android Contacts数据

ContactsProvider2继承自ContentProvider，ContentProvider将应用的数据提供给其他应用程序，其他应用可通过ContentResolver的接口来查询或读写本应用的数据。ContactsDatabaseHelper继承SQLiteOpenHelper，是一个SQLite辅助类，主要用来辅助ContentProvider，负责创建数据库，以及数据库的版本控制，初次创建会调用onCreate，可在里面创建数据库的表，版本变化会调用onUpgrade，可在里面更新相应的表，通过getWritableDatabase可以创建或者打开一个可读写的数据库，调用close关闭已打开的数据库。

关于ContentProvider和SQLiteOpenHelper，具体可参考SDK文档的说明。

需要注意的是，数据库或者Cursor不用了，一定要关闭，否则再次调用会报错。

下面是ContactsDatabaseHelper中定义的几个重要的表：

|  |  |
| --- | --- |
| public static final String CONTACTS = "contacts"; | 联系人表，每一条对应一条实际呈现的联系人，注意是合并后的 |
| public static final String RAW\_CONTACTS = "raw\_contacts"; | 实际的联系人表，这是合并前的，真正的联系人信息，包括显示名，账户，是否有号码，定制铃声，等等，比contacts里面含有的内容要多 |
| public static final String DATA = "data"; | 数据表，联系人的phone，email，name等数据实际是存在这里的，通过MIMETYPES区分数据类型，通过raw\_contact\_id与RAW\_CONTACTS表对应，一般查询联系人数据都是将这两个表联合查询得到结果 |
| public static final String MIMETYPES = "mimetypes"; | DATA中会存储的数据类型，这个表一般不用更改 |
| public static final String PHONE\_LOOKUP = "phone\_lookup"; | 用于通过号码进行查询的辅助表，查询得到data\_id和raw\_contact\_id |
| public static final String NAME\_LOOKUP = "name\_lookup"; | 用于通过各种名字进行查询的辅助表 |

为了方便读写，android还创建了一些VIEW，VIEW也是SQLite中的一部分，与TABLE不同的是，表中存的是实际数据，而VIEW是对表的查询，如：

|  |  |
| --- | --- |
| public static final String DATA\_ALL = "view\_data"; | 查询所有的data，实际上是联合了data，raw\_contacts，mimetypes等表进行查询 |
| public static final String CONTACTS\_ALL = "view\_contacts"; | 联合了raw\_contacts和contacts表进行查询 |

数据创建后通过ContactsProvider2提供给外界调用。ContactsProvider2使用URI与UriMatcher机制匹配针对不同的数据的各种不同的操作，如：

|  |  |
| --- | --- |
| matcher.addURI(ContactsContract.AUTHORITY, "contacts", CONTACTS); | 匹配contacts |
| matcher.addURI(ContactsContract.AUTHORITY, "contacts/#", CONTACTS\_ID); | 通过指定id的uri匹配contacts |
| matcher.addURI(ContactsContract.AUTHORITY, "contacts/#/data", CONTACTS\_DATA); | 通过指定id的uri匹配contacts的数据 |

通过UriMatcher获取到相应的操作码之后，在query，insert，delete等方法中通过判断操作码进行相应的操作。

具体的流程还需要参考代码。这里要注意的几个细节：

1. 插入或者更新一条号码记录，会同时更新PHONE\_LOOKUP表，将号码转化成NORMALIZED\_NUMBER存入，同时存入最小匹配位数等信息，如果要匹配号码，尽量使用这个表进行查询，而不是直接查询data表。
2. 存储姓名记录与号码类似，不过比号码更复杂点。姓名有很多类型，另外还有一个显示名，显示名顾名思义就是要显示的联系人名字，如果没有姓名项就显示号码，如果没有号码就显示email等。Android提供一个NameLookupBuilder类，用来根据不同的语言等规则创建针对一个名字的各种lookupkey，往往一个名字会产生很多lookupkey，比如中文，有全拼查询，也有首字母查询，这些可以根据需求在NameLookupBuilder进行定制，比如拨号盘的数字转拼音匹配，中文拼音有一个辅助类HanziToPinyin，它使用的字库在\external\icu4c中。匹配姓名尽量通过NAME\_LOOKUP表，而不是直接通过姓名字符串比对。
3. ContactsProvider2实现了applyBatch方法，可以将一系列数据库操作打包到一个List中一起操作，在编辑联系人数据库时很有必要，因为更新联系人数据库往往要同时更新好几个表，而且每个表不一定只更新一条数据。

## UI模块

主要就是几个activity，要注意的是，入口activity（也就是在launcher里面显示的activity）有两个：DialtactsActivity和DialtactsContactsEntryActivity，分别是拨号盘和通讯录，而DialtactsContactsEntryActivity的targetActivity实际上也是DialtactsActivity，它是一个TabActivity，源代码里都是显示4个tab：拨号盘，通话记录，通讯录，收藏，我们定制的代码在onCreate的时候会进行判断，然后决定显示哪几个Tab， 如果启动的是通讯录，显示的是通讯录，群组和收藏。当然这是2.3的，4.0上这块代码做了很大改变，用到了Fragment。

ContactsListActivity是很重要的一个activity，大部分的通讯录界面都可以用这个activity来显示，它有很多模式，例如通讯录界面是MODE\_DEFAULT，收藏界面就是其中的MODE\_STARRED，每种模式又有很多子模式，通过组合可以得到各种定制需求，一般的需求都可以实现，如无必要，不需要建新的activity。

关于创建联系人，这里涉及到账户的东西。Android中创建不同的账户联系人是不一样的，比如google账户的联系人的email拥有几种类型，而默认的exchange账户联系人有另外几种；某种账户联系人是否允许同步和合并等等。所以在创建联系人时，会先查询已创建的账户，让用户选择将联系人创建在哪个账户；如果没查询到账户，则将联系人创建在手机内存中（无账户属性）。选择了Account之后，作为参数通过getInflatedSource方法传入Sources类，获取相应的ContactsSource，android中每种账户对应一种ContactsSource：

|  |  |
| --- | --- |
| 默认 | FallbackSource |
| Exchange账户 | ExchangeSource |
| Google账户 | GoogleSource |

ContactsSource基于账户属性，通过管理一个HashMap<String, DataKind>，决定该账户联系人可存储项。DataKind指的就是每一种数据类型，如email，number等，它是动态长度的，下面有图。

每种账户联系人的不同，都体现在其对应的ContactsSource中，如果有定制的账户类型，你也要添加相应的ContactsSource子类，然后实现你的定制需求（一般定制的代码都会添加SIM卡联系人模块，实际上我们目前的代码中的做法并不好，是通过修改数据库来完成的，好的做法应该是通过添加一个适用于SIM卡联系人的ContactsSource，通过账户属性来控制）。

下面具体点说下，编辑和创建联系人这块，android设计了一个框架，主要涉及的类：

|  |  |
| --- | --- |
| EntityDelta | 一个数据模型，它实现了Parcelable接口，大家应该都知道，除了基本类型或者序列化外，只有实现了该接口的数据结构，才能通过Intent传递，或者IPC。该类型通过一个ValuesDelta 存储联系人存储在RawContacts表的相关信息，HashMap<String, ArrayList<ValuesDelta>>存储联系人存储在Data表非唯一属性的数据，如number，email等，具体看ValuesDelta |
| ValuesDelta | 也是实现了Parcelable接口，它实质就是存储了2个ContentValues类型的数据mBefore和mAfter，一个是编辑前，一个是编辑后的（新建联系人也有编辑前的，只不过里面是空的而已），ContentValues是一种android中方便和常用的数据类型，可以方便的通过Intent传递各种类型的数据，如不了解请看SDK文档 |
| EntityModifier | 用于对EntityDelta进行修改的辅助类，如parseExtras方法用来将Intent传过来的Bundle里面的数据添加到EntityDelta中 |
| EntitySet | EntityDelta的集合，因为一个EntityDelta实际上代表一个RawContacts记录，而编辑联系人是针对Contacts，它可能是合并了几个RawContacts记录 |
| ContactEditorView | 每一个实际联系人对应一个ContactEditorView |
| KindSectionView | 每一种数据对应一个KindSectionView |
| GenericEditorView | 每一条数据对应一个GenericEditorView |

下图为UI与数据结构的对应关系：



ContactsSource

DataKind

List<EditType>

List<EditField>

EntitySet

ContactEditorView

GenericEditorView

…

KindSectionView

…

HashMap<String, ArrayList<ValuesDelta>>

ValuesDelta

EntityDelta

回到创建联系人，EditContactActivity的oncreate里会进行判断：

1. 如果是创建新的联系人，首先弹出对话框让用户选择账户，然后将选择的Account数据传入ContentValues再将ContentValues传入EntityDelta中，然后再把Intent中可能含有的Bundle数据parse到该EntityDelta中，如短信那边传来的号码或者email，图库那边的Bitmap等等，最后把EntityDelta封装到EntitySet中。
2. 如果是编辑联系人，首先启动一个AsyncTask根据Intent传过来的数据去查询该联系人相关Data，得到一个EntitySet，然后再把Intent中可能含有的Bundle数据parse到该EntitySet中第一个EntityDelta中。AsyncTask是android的一个异步机制，其作用类似于Thread，具体可参考SDK文档。

后面的步骤都是一样的，调用bindEditor方法把数据绑定到Editor中，批量化产生UI控件：拿到mSate，也就是上面步骤得到的EntitySet变量，遍历其中的EntityDelta，每个EntityDelta生成一个ContactEditorView。首先从EntityDelta中得到accountType，根据accountType得到相应的ContactsSource，再把EntityDelta和得到的ContactsSource一起通过ContactEditorView的setState方法传递给ContactEditorView，下图为继承关系：



这块的框架用到了java中的几种设计模式，对学习java还是很有帮助的，具体还得看代码。Android在4.0上还是沿用了这套框架。

通讯录模块的代码看懂了之后再去修改，会简单很多，不会修改太多。比如803的代码的显示群组中联系人界面ContactsInGroup，完全可以直接在ContactsListActivity中添加一个模式来实现，或者不行就继承ContactsListActivity来实现其中的几个方法，完全没必要直接把ContactsListActivity的代码全部复制过去建个新的类。再比如SIM卡联系人编辑界面，完全可以通过Account和ContactsSource来控制，这样甚至有可能连数据库都不用修改。可惜种种原因，这块代码改动太多，已没法重新做了。

## 系统模块

关于ContactsContract，有一个旧的Contacts类，android已经不用了，开发的话建议也不要再用了，新的ContactsContract已经涵盖了所有URI和字段，且更加完善。

关于SIM卡联系人模块，首先要了解下AIDL，这个资料有很多，然后关注下面几个类：

|  |  |
| --- | --- |
| \frameworks\base\telephony\java\com\android\internal\telephony\IccProvider.java | 提供给上层的接口，继承自contentprovider，也就是说上层可以通过contentprovider接口像操作数据库数据一样操作sim卡数据 |
| \frameworks\base\telephony\java\com\android\internal\telephony\IIccPhoneBook.aidl | 系统定义的SIM卡读写卡操作的AIDL接口，通过这些接口上层可以获取和更新卡中的信息 |
| \frameworks\base\telephony\java\com\android\internal\telephony\MSimIccPhoneBookInterfaceManagerProxy.java | 接口实现的类，这里面的方法与aidl文件定义的必须完全一致。在我们的双卡代码里面，这里并不是真正实现接口的地方，而是通过index获取对应的Phone，再获取Phone里面定义的IccPhoneBookInterfaceManagerProxy |
| \frameworks\base\telephony\java\com\android\internal\telephony\IccPhoneBookInterfaceManagerProxy.java | 其实通过双卡代码更改过后，这个类没有存在的必要了，MSimIccPhoneBookInterfaceManagerProxy已经代替了它，它现在的作用就是调用IccPhoneBookInterfaceManager |
| \frameworks\base\telephony\java\com\android\internal\telephony\IccPhoneBookInterfaceManager.java | 这个类是接口真正实现的地方，一些不需要考虑卡类型的方法可直接在这里面实现，其他的则需要在它的子类中实现，如gsm phone的SimPhoneBookInterfaceManager，cdma phone的RuimPhoneBookInterfaceManager |
| \frameworks\base\telephony\java\com\android\internal\telephony\gsm\SimPhoneBookInterfaceManager.java | IccPhoneBookInterfaceManager的子类，gsm实现 |
| \frameworks\base\telephony\java\com\android\internal\telephony\cdma\RuimPhoneBookInterfaceManager.java | IccPhoneBookInterfaceManager的子类，cdma实现 |
| \frameworks\base\telephony\java\com\android\internal\telephony\AdnRecordCache.java | IccPhoneBookInterfaceManager通过AdnRecordCache与下层沟通，AdnRecordCache本质上是个Handler，通过Message与AdnRecordLoader沟通，第一次读卡后会将数据存于AdnRecordCache中，后续再读直接从cache里读取，USIM卡特殊处理除外 |
| \frameworks\base\telephony\java\com\android\internal\telephony\AdnRecordLoader.java | AdnRecordLoader也是一个Handler，它接收到AdnRecordCache的请求之后，再去请求IccFileHandler |
| \frameworks\base\telephony\java\com\android\internal\telephony\IccFileHandler.java | IccFileHandler也是一个Handler，它会去调用RIL层的ICCIO接口，与底层进行沟通 |
| \frameworks\base\telephony\java\com\android\internal\telephony\gsm\UsimPhoneBookManager.java | UsimPhoneBookManager是gsm中对usim的特殊处理，AdnRecordCache会进行判断，如果是usim卡则不走上述流程，而转到UsimPhoneBookManager中进行处理 |
| \frameworks\base\telephony\java\com\android\internal\telephony\AdnRecord.java | 存储在卡上的And数据类，负责AND的格式化以及解析，具体要参考3GPP协议 |

这几个Handler间通过Message沟通，上层调用到AdnRecordCache后，AdnRecordCache进行判断，如果是USIM卡则走UsimPhoneBookManager，其它则走正常流程，如图：

