Android 4.4 SIM卡的管理流程

Date 日期	Version 版本	Comments 备注
2014-03-19	0.1	First version

	Function 职位	Name 姓名	Date 日期	Signature 签名
Written by 拟定	Telecom Team Leader	谢瀚武	2014-03-19	
Verified by 审核				
Verified by 审核				
Approved by 批准				

1. 前言

随着手机的普及，大家从非智能机到智能机的转变，从没有操作系统的定制机到智能手机，但唯一没有变的是，手机中的 SIM，今天我们就来谈谈手机中 SIM 卡相关的内容。在日常生活中，SIM 卡就是一张很小的卡片，但这个卡片上却存储了很重要的信息。

现有的手机中使用的 SIM，USIM，UIM 卡等统称为 UICC，即 Universal Integrated Circuit Card。

Android 在经过几次更新后，与 UICC 相关的管理出现了的重大的改变。与前面几个版相比，变化最大的就是 IccCard.java 这个文件，在 4.2 以前的版本中，该 IccCard 是一个类，而在 4.2 中，它却是一个接口，所以就不会存在以前版本中的 SimCard, UsimCard, 现在统一用 IccCardProxy 来替代他们所有功能。

本文基于 Android 4.4 的源码进行分析。

2. UICC 框架

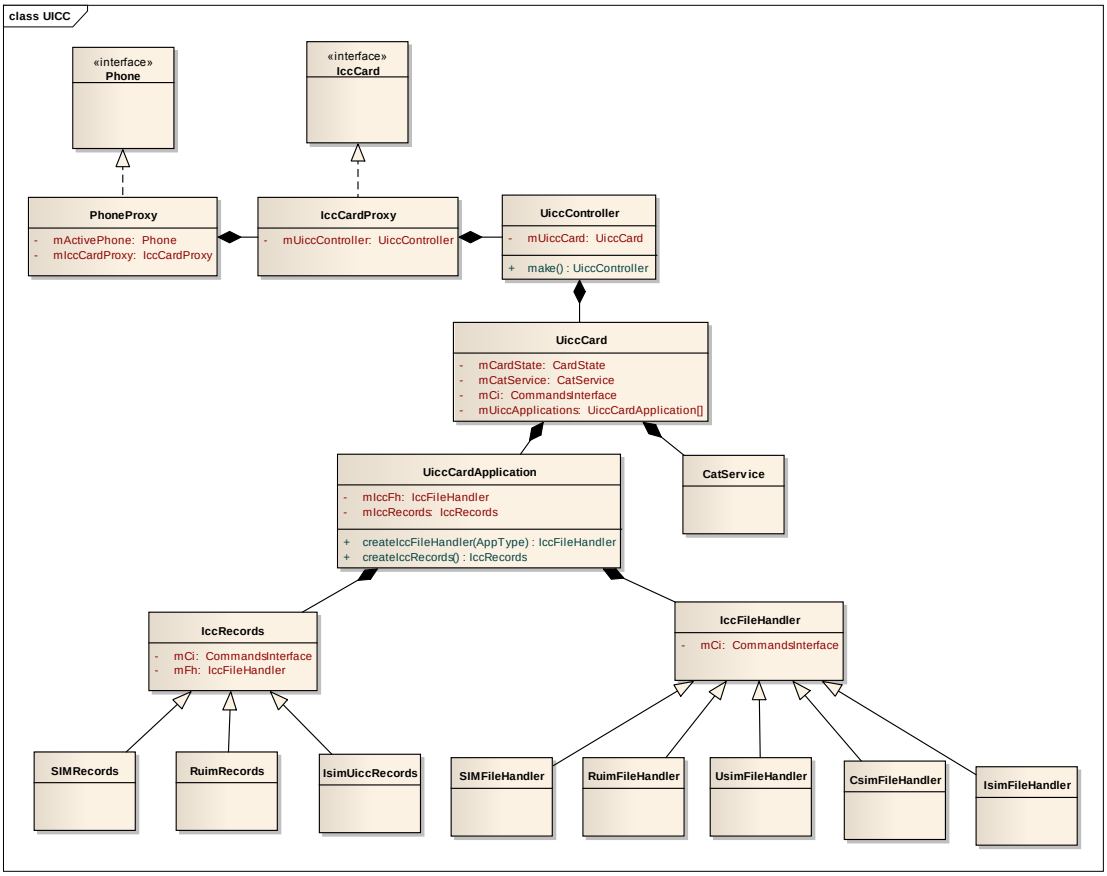
1) UICC 框架代码接口

在 Android 的源码中，SIM 卡相关的操作，都封装在 framework 中，存放在 Frameworks/base/telephony/java/com/android/internal/telephony 这个目录下。

这个目录中存储了所有与手机通讯业务相关的类文件，其中也包括了 SIM，STK，CALL，PS 数据业务。此目录下大家可以看到 GSM 和 CDMA 两个文件夹，这也是 SIM 卡相关的，如果插入的是 CDMA 卡片，就使用 CDMA 文件夹中的源码，如果是 2G SIM 卡，或是 3G 联通卡（即 GSM/WCDMA）都是使用 GSM 文件夹。当打开 GSM 文件夹，可以看到有一个 STK 文件夹，里面装的就是上面那个链接里面的源码。如果是 2G SIM 卡，或是 3G 联通卡（即 GSM/WCDMA）都是使用 GSM 文件夹。

Android 中是管理 UICC 的框架代码位于：
frameworks\opt\telephony\src\java\com\android\internal\telephony\uicc\UiccController.java
frameworks\opt\telephony\src\java\com\android\internal\telephony\gsm
frameworks\opt\telephony\src\java\com\android\internal\telephony\cdma

2) UICC 框架类结构



图：UICC 框架类结构图

对于不同的卡会有不同的类与之对应，这些类的作用如下表所示：

表：UICC 关键类

类	描述
UiccController	整个 UICC 相关信息的控制接口，用来控制所有与卡相关的操作；监控 SIM 状态变化，并通过 UiccCard 来更新 SIM 卡的信息
UiccCard	实际 UICC 卡的抽象，用来更新 SIM 卡的信息
IccCardStatus	维护 UICC 卡的状态：CardState & PinState
UiccCardApplication	描述 UICC 卡应用的信息；负责 Pin Puk 密码设置解锁，数据的读取，存储
CatService	负责 SIM Toolkit 相关的处理
IccConstants	SIM File Address；存储不同数据在 SIM 卡上的字段地址
SIMRecords/RuimRecords	记录 SIM 卡上的数据/文件内容
IccFileHandler	读取 SIM 数据以及接收读取的结果

3. UICC 源码分析

2.1. PhoneProxy

Phone 是接口基类；PhoneProxy 是 Phone 的实现类，并继承于 Handler。PhoneProxy 的主要作用是，在 Phone 创建后，用 phoneproxy 来替换 phone(包括 GSMPHONE，CDMAPHONE 等等)，这样做的好处是为了屏蔽这些 PHONE 之间的差异点。对外来说，都是一样的接口。好，既然这样的话，我要去看看 phoneproxy 怎么初始化的。代码如下：

```
1.  static private Phone sProxyPhone = null;
2.
3.  public static void makeDefaultPhone(Context context)
4.  {
5.      ...
6.      int phoneType = TelephonyManager.getPhoneType(networkMode);
7.      if (phoneType == PhoneConstants.PHONE_TYPE_GSM) {
8.          Log.i(LOG_TAG, "Creating GSMPhone");
9.          sProxyPhone = new PhoneProxy(new GSMPhone(context,
10.                 sCommandsInterface, sPhoneNotifier));
11.     } else if (phoneType == PhoneConstants.PHONE_TYPE_CDMA) {
12.         switch (TelephonyManager.getLteOnCdmaModeStatic()) {
13.             case PhoneConstants.LTE_ON_CDMA_TRUE:
14.                 Log.i(LOG_TAG, "Creating CDMALTEPhone");
15.                 sProxyPhone = new PhoneProxy(new CDMALTEPhone(context,
16.                     sCommandsInterface, sPhoneNotifier));
17.                 break;
18.             case PhoneConstants.LTE_ON_CDMA_FALSE:
19.             default:
20.                 Log.i(LOG_TAG, "Creating CDMAPhone");
21.                 sProxyPhone = new PhoneProxy(new CDMAPhone(context,
22.                         sCommandsInterface, sPhoneNotifier));
23.                 break;
24.         }
25. }
26.     ...
27. }
```

在上面这段代码，请大家注意 **new PhoneProxy()**的时候，会根据 PHONE 的类型创建不同的 PHONE，然后再用 PhoneProxy 把所有 PHONE 之间的区别不一样的给屏蔽掉，以便对外面的接口来说，都是一致的。PhoneProxy 的初始化，如下：

```
1.  public PhoneProxy(PhoneBase phone) {
2.      mActivePhone = phone;
3.      mResetModemOnRadioTechnologyChange = SystemProperties.getBoolean(
4.              TelephonyProperties.PROPERTY_RESET_ON_RADIO_TECH_CHANGE, false);
5.      mIccSmsInterfaceManagerProxy = new IccSmsInterfaceManagerProxy(
6.              phone.getIccSmsInterfaceManager());
7.      mIccPhoneBookInterfaceManagerProxy = new IccPhoneBookInterfaceManagerProxy(
8.              phone.getIccPhoneBookInterfaceManager());
```

```
9.      mPhoneSubInfoProxy = new PhoneSubInfoProxy(phone.getPhoneSubInfo());
10.     mCommandsInterface = ((PhoneBase)mActivePhone).mCM;
11.
12.     mCommandsInterface.registerForRilConnected(this, EVENT_RIL_CONNECTED, null);
13.     mCommandsInterface.registerForOn(this, EVENT_RADIO_ON, null);
14.     mCommandsInterface.registerForVoiceRadioTechChanged(
15.                             this, EVENT_VOICE_RADIO_TECH_CHANGED, null);
16.     mIccCardProxy = new IccCardProxy(phone.getContext(), mCommandsInterface);
17.     if (phone.getPhoneType() == PhoneConstants.PHONE_TYPE_GSM) {
18.         // For the purpose of IccCardProxy we only care about the technology family
19.         mIccCardProxy.setVoiceRadioTech(ServiceState.RIL_RADIO_TECHNOLOGY_UMTS);
20.     } else if (phone.getPhoneType() == PhoneConstants.PHONE_TYPE_CDMA) {
21.         mIccCardProxy.setVoiceRadioTech(ServiceState.RIL_RADIO_TECHNOLOGY_1xRTT);
22.     }
23. }
```

2.2. IccCardProxy

IccCard 是接口基类；IccCardProxy 是 IccCard 的实现类，并继承于 Handler，它主要有如下几个功能。

```
1.      public IccCardProxy(Context context, CommandsInterface ci) {
2.          log("Creating");
3.          mContext = context;
4.          mCi = ci;
5.          mCdmaSSM = CdmaSubscriptionSourceManager.getInstance(context,
6.                  ci, this, EVENT_CDMA_SUBSCRIPTION_SOURCE_CHANGED, null);
7.          mUiccController = UiccController.getInstance();
8.         //通过 UiccController 监听 SIM 卡的变化信息
9.          mUiccController.registerForIccChanged(this, EVENT_ICC_CHANGED, null);
10.         ci.registerForOn(this,EVENT_RADIO_ON, null);
11.         //监听 Radio 的状态，间接获取 SIM 卡的状态
12.         ci.registerForOffOrNotAvailable(this, EVENT_RADIO_OFF_OR_UNAVAILABLE, null);
13.         setExternalState(State.NOT_READY);
14.     }
```

如果有状态变化，其做了下面两件事情：

1) 通过 UiccController,获取 UiccCard,UiccCardApplication，IccRecords 的最新状态。
2) 通过 CdmaSubscriptionSourceManager，管理 CDMA 网络订阅信息，从 RUIM 或者 NV 获取 CDMA 的订阅信息

我们通过源码来分析下，如果有状态变化，则会调用 handleMessage()函数进行处理：

```
@Override
public void handleMessage(Message msg) {
        switch (msg.what) {
            case EVENT_RADIO_OFF_OR_UNAVAILABLE:
                mRadioOn = false;
                break;
            case EVENT_RADIO_ON:
                mRadioOn = true;
                if (!mInitialized) {
                    updateQuietMode();
                }
                break;
            case EVENT_ICC_CHANGED:
                if (mInitialized) {
                    updateIccAvailability();
                }
                break;
           ...
            case EVENT_CDMA_SUBSCRIPTION_SOURCE_CHANGED:
                updateQuietMode();
                break;
            default:
                loge("Unhandled message with number: " + msg.what);
                break;
        }
    }

 private void updateIccAvailability() {
        synchronized (mLock) {
            UiccCard newCard = mUiccController.getUiccCard();
            CardState state = CardState.CARDSTATE_ABSENT;
            UiccCardApplication newApp = null;
            IccRecords newRecords = null;
            if (newCard != null) {
                state = newCard.getCardState();
                newApp = newCard.getApplication(mCurrentAppType);
                if (newApp != null) {
                    newRecords = newApp.getIccRecords();
                }
            }

       if (mIccRecords != newRecords || mUiccApplication != newApp || mUiccCard != newCard){
                if (DBG) log("Icc changed. Reregestering.");
                unregisterUiccCardEvents();
```

```
59.                 mUiccCard = newCard;
60.                 mUiccApplication = newApp;
61.                 mIccRecords = newRecords;
62.                //注册成为 UiccCard、UiccCardApplication、IccRecords 的监听器，
63.                //可以从他们获取 SIM 卡的相关信息
64.                 registerUiccCardEvents();
65.             }
66.
67.             updateExternalState();
68.         }
69.     }
70.
71.
72. 1.通过 UiccController 监听 SIM 卡的变化信息
73.
74. // 通过这个方法，当 SIM 卡被锁定或者 SIM 卡的信息加载完毕时，IccCardProxy 都可以收到
75. private void registerUiccCardEvents() {
76.         if (mUiccCard != null) mUiccCard.registerForAbsent(this, EVENT_ICC_ABSENT, null);
77.         if (mUiccApplication != null) {
78.             mUiccApplication.registerForReady(this, EVENT_APP_READY, null);// SIM 卡 Ready
79.             mUiccApplication.registerForLocked(this, EVENT_ICC_LOCKED, null);// SIM 卡被
    PIN 锁
80.             mUiccApplication.registerForNetworkLocked(this, EVENT_NETWORK_LOCKED,
    null);//SIM 卡被网络锁
81.         }
82.         if (mIccRecords != null) {
83.             mIccRecords.registerForImsiReady(this, EVENT_IMSI_READY, null);// IMS 卡准备
    好
84.             mIccRecords.registerForRecordsLoaded(this, EVENT_RECORDS_LOADED, null);//
    SIM 卡的记录加载完毕
85.         }
86.     }
87.
88. 2.监听 Radio 的状态，间接获取 SIM 卡的状态。
89.  private void updateQuietMode() {
90.         synchronized (mLock) {
91.            ...
92.             if (mCurrentAppType == UiccController.APP_FAM_3GPP) {
93.                 newQuietMode = false;
94.             } else {
95.                 if (isLteOnCdmaMode) {
96.                     mCurrentAppType = UiccController.APP_FAM_3GPP;
97.                 }
98.                 cdmaSource = mCdmaSSM != null ?
```

```
                    mCdmaSSM.getCdmaSubscriptionSource() :
    Phone.CDMA_SUBSCRIPTION_UNKNOWN;

                newQuietMode = (cdmaSource == Phone.CDMA_SUBSCRIPTION_NV)
                        && (mCurrentAppType == UiccController.APP_FAM_3GPP2)
                        && !isLteOnCdmaMode;
            }

            if (mQuietMode == false && newQuietMode == true) {
                // Last thing to do before switching to quiet mode is
                // broadcast ICC_READY
                log("Switching to QuietMode.");
                setExternalState(State.READY);
                mQuietMode = newQuietMode;
            } else if (mQuietMode == true && newQuietMode == false) {
                if (DBG) {
                    log("updateQuietMode: Switching out from QuietMode."
                            + " Force broadcast of current state=" + mExternalState);
                }
                mQuietMode = newQuietMode;
                setExternalState(mExternalState, true);
            }
            if (DBG) {
                log("updateQuietMode: QuietMode is " + mQuietMode + " (app_type="
                    + mCurrentAppType + " isLteOnCdmaMode=" + isLteOnCdmaMode
                    + " cdmaSource=" + cdmaSource + ")");
            }
            mInitialized = true;
            sendMessage(obtainMessage(EVENT_ICC_CHANGED));
        }
    }

     private void setExternalState(State newState, boolean override) {
        synchronized (mLock) {
            if (!override && newState == mExternalState) {
                return;
            }
            mExternalState = newState;
            SystemProperties.set(PROPERTY_SIM_STATE, mExternalState.toString());
            broadcastIccStateChangedIntent(getIccStateIntentString(mExternalState),
                    getIccStateReason(mExternalState));
        }
}
```

```
142.    3.获得 SIM 卡的最新的状态信息后，将 SIM 卡的信息广播除去.(ACTION_SIM_STATE_CHANGED)
143.    private void broadcastIccStateChangedIntent(String value, String reason) {
144.        synchronized (mLock) {
145.            if (mQuietMode) {
146.                log("QuietMode: NOT Broadcasting intent ACTION_SIM_STATE_CHANGED " +  value
147.                        + " reason " + reason);
148.                return;
149.            }
150.
151.            Intent intent = new Intent(TelephonyIntents.ACTION_SIM_STATE_CHANGED);
152.            intent.addFlags(Intent.FLAG_RECEIVER_REPLACE_PENDING);
153.            intent.putExtra(PhoneConstants.PHONE_NAME_KEY, "Phone");
154.            intent.putExtra(IccCardConstants.INTENT_KEY_ICC_STATE, value);
155.            intent.putExtra(IccCardConstants.INTENT_KEY_LOCKED_REASON, reason);
156.
157.            if (DBG) log("Broadcasting intent ACTION_SIM_STATE_CHANGED " +  value
158.                    + " reason " + reason);
159.            ActivityManagerNative.broadcastStickyIntent(intent, READ_PHONE_STATE,
160.                    UserHandle.USER_ALL);
161.        }
162.    }
```

2.3. IccCardStatus

IccCardStatus 用于描述手机所插入的手机卡的信息，包括卡的状态，PIN 码锁定的状态，所包含的应用的状态信息。RIL.java 中获取得到 SIM 卡的信息后，将信息封装成 IccCardStatus 返回给 UiccController.

具体的成员变量如下:

```
1.      //CardState ,描述 SIM 卡的状态，
2.      public enum CardState {
3.          CARDSTATE_ABSENT,  //表示掉卡
4.          CARDSTATE_PRESENT, //表示卡正常
5.          CARDSTATE_ERROR; //表示卡出现了错误
6.
7.          boolean isCardPresent() {
8.              return this == CARDSTATE_PRESENT;
9.          }
10.
11. 。。。
12.
13.     };
14.     PinState ,描述 PIN 码锁定信息
```

```
    public enum PinState {
        PINSTATE_UNKNOWN,// PIN 码状态不确定
        PINSTATE_ENABLED_NOT_VERIFIED, // PIN 码锁定，用户输入的 PIN 码错误，无法进入手机
        PINSTATE_ENABLED_VERIFIED,// PINM 码锁定，用户输入正确的 PIN 码，进入手机
        PINSTATE_DISABLED,// 没有进行 PIN 码锁定
        PINSTATE_ENABLED_BLOCKED,//PIN 码解锁失败，提示输入 PUK 码
        PINSTATE_ENABLED_PERM_BLOCKED;// PUK 码解锁失败后，永久锁定

        boolean isPermBlocked() {
            return this == PINSTATE_ENABLED_PERM_BLOCKED;
        }

        boolean isPinRequired() {
            return this == PINSTATE_ENABLED_NOT_VERIFIED;
        }

        boolean isPukRequired() {
            return this == PINSTATE_ENABLED_BLOCKED;
        }
    }
    IccCardApplicationStatus，描述 UiccCardApplication 的信息，每一个 IccCardApplicationStatus,对应于一个 UiccCardApplication.
    IccCardStatus 用于创建 UiccCard,一个 UiccCard 中可能包含多个 UiccCardApplication.
    public class IccCardApplicationStatus {
    //描述 UiccCardApplication 的类型，和卡的类型对应
    public enum AppType{
        APPTYPE_UNKNOWN,
        APPTYPE_SIM,  //GSM 卡
        APPTYPE_USIM, // WCDMA 卡
        APPTYPE_RUIM, // CDMA 卡
        APPTYPE_CSIM,
        APPTYPE_ISIM
    };
    //描述 UiccCardApplication 的状态，和卡的状态对应
    public enum AppState{
        APPSTATE_UNKNOWN, //卡不存在
        APPSTATE_DETECTED,// 卡已经检测到
        APPSTATE_PIN,// 卡已经被 PIN 码锁定
        APPSTATE_PUK,// 卡已经被 PUK 码锁定
        APPSTATE_SUBSCRIPTION_PERSO, //卡已经被网络锁定
        APPSTATE_READY;//卡已经准备好了

```

```
56.         boolean isPinRequired() {
57.             return this == APPSTATE_PIN;
58.         }
59.
60.         boolean isPukRequired() {
61.             return this == APPSTATE_PUK;
62.         }
63.
64.         boolean isSubscriptionPersoEnabled() {
65.             return this == APPSTATE_SUBSCRIPTION_PERSO;
66.         }
67.
68.         boolean isAppReady() {
69.             return this == APPSTATE_READY;
70.         }
71.
72.         boolean isAppNotReady() {
73.             return this == APPSTATE_UNKNOWN  ||
74.                    this == APPSTATE_DETECTED;
75.         }
76.     };
77.     //描述卡被网络锁定的信息，对国内用户来说意义不大
78.     public enum PersoSubState{
79.         PERSOSUBSTATE_UNKNOWN,
80.         PERSOSUBSTATE_IN_PROGRESS,
81.         PERSOSUBSTATE_READY,
82.         PERSOSUBSTATE_SIM_NETWORK,
83.         PERSOSUBSTATE_SIM_NETWORK_SUBSET,
84.         PERSOSUBSTATE_SIM_CORPORATE,
85.         PERSOSUBSTATE_SIM_SERVICE_PROVIDER,
86.         PERSOSUBSTATE_SIM_SIM,
87.         PERSOSUBSTATE_SIM_NETWORK_PUK,
88.         PERSOSUBSTATE_SIM_NETWORK_SUBSET_PUK,
89.         PERSOSUBSTATE_SIM_CORPORATE_PUK,
90.         PERSOSUBSTATE_SIM_SERVICE_PROVIDER_PUK,
91.         PERSOSUBSTATE_SIM_SIM_PUK,
92.         PERSOSUBSTATE_RUIM_NETWORK1,
93.         PERSOSUBSTATE_RUIM_NETWORK2,
94.         PERSOSUBSTATE_RUIM_HRPD,
95.         PERSOSUBSTATE_RUIM_CORPORATE,
96.         PERSOSUBSTATE_RUIM_SERVICE_PROVIDER,
97.         PERSOSUBSTATE_RUIM_RUIM,
98.         PERSOSUBSTATE_RUIM_NETWORK1_PUK,
99.         PERSOSUBSTATE_RUIM_NETWORK2_PUK,
```

```
100.         PERSOSUBSTATE_RUIM_HRPD_PUK,
101.         PERSOSUBSTATE_RUIM_CORPORATE_PUK,
102.         PERSOSUBSTATE_RUIM_SERVICE_PROVIDER_PUK,
103.         PERSOSUBSTATE_RUIM_RUIM_PUK;
104.
105.         boolean isPersoSubStateUnknown() {
106.             return this == PERSOSUBSTATE_UNKNOWN;
107.         }
108.     };
109.
110.     public AppType       app_type;
111.     public AppState      app_state;
112.     // applicable only if app_state == RIL_APPSTATE_SUBSCRIPTION_PERSO
113.     public PersoSubState  perso_substate;
114.     // null terminated string, e.g., from 0xA0, 0x00 -> 0x41, 0x30, 0x30, 0x30
    */
115.     public String        aid;
116.     // null terminated string
117.     public String        app_label;
118.     // applicable to USIM and CSIM
119.     public int           pin1_replaced;
120.     public PinState           pin1;//卡的 pin1 码
121.     public PinState           pin2;//卡的 pin2 码
122.     ....
123.     }
```

2.4. UiccController

1) UiccController 对象的创建

UiccController 是在 PhoneFactory 类的 makeDefaultPhone()方法中创建的，代码如下：

```
1.  public static void makeDefaultPhone(Context context)
2.  {
3.      // Instantiate UiccController so that all other classes can just call ge
        tInstance()
4.      UiccController.make(context, sCommandsInterface);
5.
6.      int phoneType = TelephonyManager.getPhoneType(networkMode);
7.      if (phoneType == PhoneConstants.PHONE_TYPE_GSM) {
8.          Log.i(LOG_TAG, "Creating GSMPhone");
9.          sProxyPhone = new PhoneProxy(new GSMPhone(context,
10.                 sCommandsInterface, sPhoneNotifier));
```

```
    } else if (phoneType == PhoneConstants.PHONE_TYPE_CDMA) {
        switch (TelephonyManager.getLteOnCdmaModeStatic()) {
            case PhoneConstants.LTE_ON_CDMA_TRUE:
                Log.i(LOG_TAG, "Creating CDMALTEPhone");
                sProxyPhone = new PhoneProxy(new CDMALTEPhone(context,
                    sCommandsInterface, sPhoneNotifier));
                break;
            case PhoneConstants.LTE_ON_CDMA_FALSE:
            default:
                Log.i(LOG_TAG, "Creating CDMAPhone");
                sProxyPhone = new PhoneProxy(new CDMAPhone(context,
                        sCommandsInterface, sPhoneNotifier));
                break;
        }
    }
    ...
}
```

可以看到，第一步先实例化 RIL，我们会得到一个 RIL 的实例，sCommandsInterface 变量实际指向的是 RIL 对象。然后，调用 UiccControl 的 make 函数创建 UiccControl 对象。

在上面这段代码，请大家注意 new PhoneProxy()的时候，会根据 PHONE 的类型创建不同的 PHONE，然后再用 PhoneProxy 把所有 PHONE 之间的区别不一样的给屏蔽掉，以便对外面的接口来说，都是一致的。

UiccController 的初始化函数代码如下：

```
    public static UiccController make(Context c, CommandsInterface ci) {
        synchronized (mLock) {
            if (mInstance != null) {
                throw new RuntimeException(
                      "UiccController.make() should only be called once");
            }
            mInstance = new UiccController(c, ci);
            return mInstance;
        }
    }

    public static UiccController getInstance() {
        synchronized (mLock) {
            if (mInstance == null) {
                throw new RuntimeException(
                "UiccController.getInstance can't be called before make()");
            }
            return mInstance;
```

```
        }
    }

    // UiccController 构造函数
    private UiccController(Context c, CommandsInterface ci) {
        mContext = c;
        mCi = ci;

        //注册 UICC 卡状态变化监听
        mCi.registerForIccStatusChanged(this, EVENT_ICC_STATUS_CHANGED, null);

        //注册 RADIO 状态变化监听
        mCi.registerForOn(this, EVENT_ICC_STATUS_CHANGED, null);
        mCi.registerForAvailable(this, EVENT_ICC_STATUS_CHANGED, null);
        mCi.registerForNotAvailable(this, EVENT_RADIO_UNAVAILABLE, null);
    }
```

这个函数会实例化 UiccController 中的 **mInstance** 成员变量，后面使用时，直接使用 getInstance 方向即可，从这个地方可以看，UiccController 采用了单例设计模式，只用创建一次。

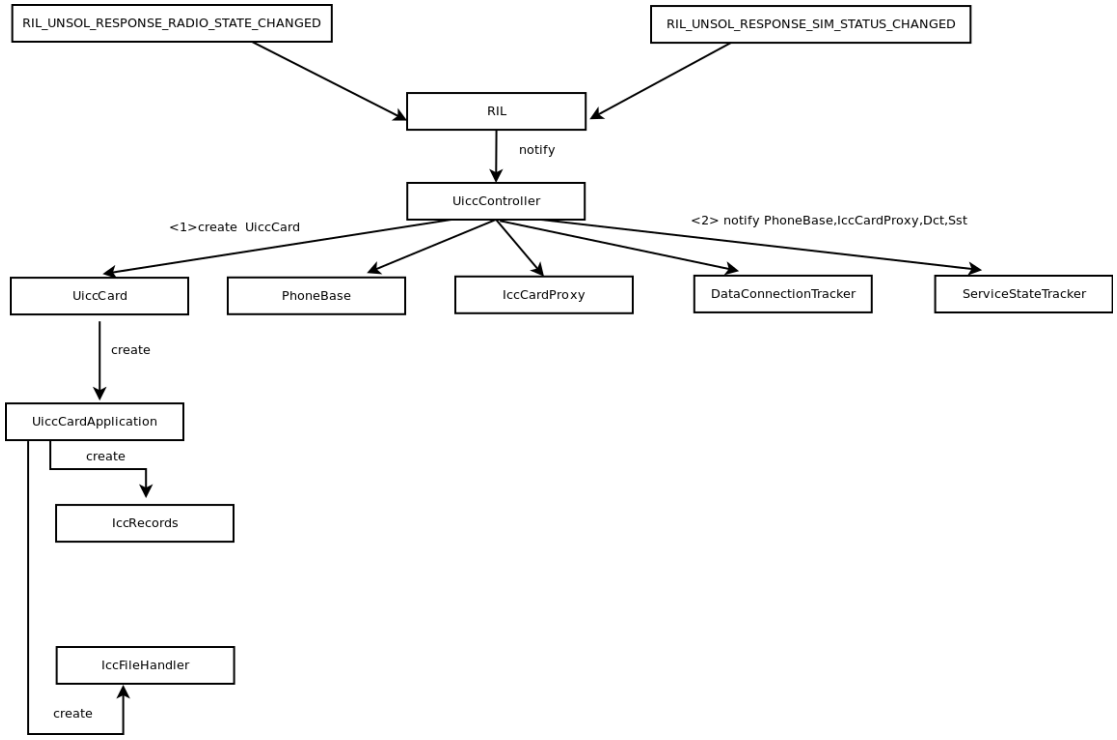
RIL 对象同时被保存到 UiccControl 类的成员变量 mCi 中。

从 UiccController 的构造函数中，我们可以看到该类还注册 EVENT_ICC_STATUS_CHANGED 和 EVENT_RADIO_UNAVAILABLE 监控事件，来监听 RIL 的消息。该事件是用来监控 SIM 卡状态和手机信号的变化。

所以当手机信号不好或者 SIM 卡本身的状态发生变化的时候，消息都会集中到 UiccConroller 这里进行处理。

2) UiccController 对象发出通知

系统中其他对象对 SIM 卡的状态，SIM 卡的数据信息感兴趣，他们可以通过 UiccController 来监听 SIM 卡的状态变化。所以当 UiccCard 更新完成后，它就通知 ServiceStateTracker(Gsm or Cdma)， IccCardProxy， DataConnectionTracker， PhoneBase,SIM 卡的状态已经发生了变化。




```
1.  DataConnectionTracker
2.  GsmDataConnectionTracker
3.   protected void onUpdateIcc() {
4.         if (mUiccController == null ) {
5.             return;
6.         }
7.
8.         IccRecords newIccRecords =
    mUiccController.getIccRecords(UiccController.APP_FAM_3GPP);
9.
10.        IccRecords r = mIccRecords.get();
11.        if (r != newIccRecords) {
12.            if (r != null) {
13.                log("Removing stale icc objects.");
14.                r.unregisterForRecordsLoaded(this);
15.                mIccRecords.set(null);
16.            }
17.            if (newIccRecords != null) {
18.                log("New records found");
19.                mIccRecords.set(newIccRecords);
20.                newIccRecords.registerForRecordsLoaded(
21.                       this, DctConstants.EVENT_RECORDS_LOADED, null);
22.            }
23.        }
24.    }
```

```
25.   更新所持有的 mIccRecords 对象的数据
26. IccCardProxy
27.   private void updateIccAvailability() {
28.         synchronized (mLock) {
29.             UiccCard newCard = mUiccController.getUiccCard();
30.             CardState state = CardState.CARDSTATE_ABSENT;
31.             UiccCardApplication newApp = null;
32.             IccRecords newRecords = null;
33.             if (newCard != null) {
34.                 state = newCard.getCardState();
35.                 newApp = newCard.getApplication(mCurrentAppType);
36.                 if (newApp != null) {
37.                     newRecords = newApp.getIccRecords();
38.                 }
39.             }
40.
41.             if (mIccRecords != newRecords || mUiccApplication != newApp || mUiccCard !=
    newCard) {
42.                 if (DBG) log("Icc changed. Reregestering.");
43.                 unregisterUiccCardEvents();
44.                 mUiccCard = newCard;
45.                 mUiccApplication = newApp;
46.                 mIccRecords = newRecords;
47.                 registerUiccCardEvents();
48.             }
49.
50.             updateExternalState();
51.         }
52.     }
53.    更新所持有的 mUiccCard,mUiccApplication,mIccRecords 对象。
54.     当 sim 卡状态变化时，会发出广播
55.    private void broadcastIccStateChangedIntent(String value, String reason) {
56.         synchronized (mLock) {
57.             if (mQuietMode) {
58.                 log("QuietMode: NOT Broadcasting intent ACTION_SIM_STATE_CHANGED " +  value
59.                         + " reason " + reason);
60.                 return;
61.             }
62.
63.             Intent intent = new Intent(TelephonyIntents.ACTION_SIM_STATE_CHANGED);
64.             intent.addFlags(Intent.FLAG_RECEIVER_REPLACE_PENDING);
65.             intent.putExtra(PhoneConstants.PHONE_NAME_KEY, "Phone");
66.             intent.putExtra(IccCardConstants.INTENT_KEY_ICC_STATE, value);
67.             intent.putExtra(IccCardConstants.INTENT_KEY_LOCKED_REASON, reason);
```

```
68.
69.             if (DBG) log("Broadcasting intent ACTION_SIM_STATE_CHANGED " +  value
70.                     + " reason " + reason);
71.             ActivityManagerNative.broadcastStickyIntent(intent, READ_PHONE_STATE,
72.                     UserHandle.USER_ALL);
73.         }
74.     }
75.     如果应用程序或者系统服务对 SIM 卡的状态有兴趣，那么可以监听这个广播
76. PhoneBase
77. GsmPhone
78. protected void onUpdateIccAvailability() {
79.         if (mUiccController == null ) {
80.             return;
81.         }
82.
83.         UiccCardApplication newUiccApplication =
84.                 mUiccController.getUiccCardApplication(UiccController.APP_FAM_3GPP);
85.
86.         UiccCardApplication app = mUiccApplication.get();
87.         if (app != newUiccApplication) {
88.             if (app != null) {
89.                 if (LOCAL_DEBUG) log("Removing stale icc objects.");
90.                 if (mIccRecords.get() != null) {
91.                     unregisterForSimRecordEvents();
92.                     mSimPhoneBookIntManager.updateIccRecords(null);
93.                 }
94.                 mIccRecords.set(null);
95.                 mUiccApplication.set(null);
96.             }
97.             if (newUiccApplication != null) {
98.                 if (LOCAL_DEBUG) log("New Uicc application found");
99.                 mUiccApplication.set(newUiccApplication);
100.                mIccRecords.set(newUiccApplication.getIccRecords());
101.                registerForSimRecordEvents();
102.                mSimPhoneBookIntManager.updateIccRecords(mIccRecords.get());
103.            }
104.        }
105.    }
106.更新所持有的 mUiccApplication,mIccRecords，mSimPhoneBookIntManager 对象。
107.ServiceStateTracker
108.GsmServiceStateTracker
109.protected void onUpdateIccAvailability() {
110.        if (mUiccController == null ) {
111.            return;
```

```
112.        }
113.
114.        UiccCardApplication newUiccApplication =
115.                mUiccController.getUiccCardApplication(UiccController.APP_FAM_3GPP);
116.
117.        if (mUiccApplcation != newUiccApplication) {
118.            if (mUiccApplcation != null) {
119.                log("Removing stale icc objects.");
120.                mUiccApplcation.unregisterForReady(this);
121.                if (mIccRecords != null) {
122.                    mIccRecords.unregisterForRecordsLoaded(this);
123.                }
124.                mIccRecords = null;
125.                mUiccApplcation = null;
126.            }
127.            if (newUiccApplication != null) {
128.                log("New card found");
129.                mUiccApplcation = newUiccApplication;
130.                mIccRecords = mUiccApplcation.getIccRecords();
131.                mUiccApplcation.registerForReady(this, EVENT_SIM_READY, null);
132.                if (mIccRecords != null) {
133.                    mIccRecords.registerForRecordsLoaded(this, EVENT_SIM_RECORDS_LOADED,
    null);
134.                }
135.            }
136.        }
137.    }
138.更新所持有的 mIccRecords,mUiccApplication,等对象。
```

2.5. UICC Card 状态有变化时的处理

在 UiccController 创建完成后，由于其注册了 EVENT_ICC_STATUS_CHANGED 监控事件，当接收到这个消息后，UiccController 会调用 handleMessage 通过 RIL 给 MODEM 发送消息，查询下 SIM 卡的状态。

```
139.    public void handleMessage (Message msg) { //UICC Card 状态有变化时的处理
140.        synchronized (mLock) {
141.            switch (msg.what) {
142.                case EVENT_ICC_STATUS_CHANGED:
143.                    //UICC 状态变化，获取 UICC 状态
144.                    mCi.getIccCardStatus(obtainMessage(EVENT_GET_ICC_STATUS_DONE));
145.                    break;
146.                case EVENT_GET_ICC_STATUS_DONE:
147.                    //UICC 状态变化，获取 UICC 状态返回结果的处理
```

```
148.                     AsyncResult ar = (AsyncResult)msg.obj;
149.                     onGetIccCardStatusDone(ar);
150.                     break;
151.                 default:
152.                     Log.e(LOG_TAG, " Unknown Event " + msg.what);
153.             }
154.         }
155.     }
156.
157.     // UICC Card 有状态变化时的处理
158.     private synchronized void onGetIccCardStatusDone(AsyncResult ar) {
159.         if (ar.exception != null) {
160.             return;
161.         }
162.         //返回的数据结构 IccCardStatus
163.         IccCardStatus status = (IccCardStatus)ar.result;
164.
165.         //更新 Uicc Card 状态 ， 若 UiccCard 未创建，则新创建一个
166.         if (mUiccCard == null) {
167.             //Create new card
168.             mUiccCard = new UiccCard(mContext, mCi, status);
169.         } else {
170.             //Update already existing card
171.             mUiccCard.update(mContext, mCi , status);
172.         }
173.
174.         mIccChangedRegistrants.notifyRegistrants();
175.     }
```

当发送查询请求，待到查询结果上来后，会初始化 UiccController 的成员变量 mUiccCard，如果是 mUiccCard 是空，即还没有初始化过，就重新 NEW 一个 UiccCard 的实例。如果是实例化过的，就重新更新下 UiccCard 的信息。到这个时候，就可以算有 SIM 卡了，需要向其它注册了监控 SIM 卡状态的注册者通知。以便其它应用能做知道 SIM 卡已经 READY 了。

1) 创建/更新 UiccCard 的信息

我们接下往下走，刚说到在初始化 UiccCard 的时候，会重新 NEW 一个实例，这个 NEW 的过程到底干了什么，请看下面的代码：

```
1.  @UiccCard.java
2.   public UiccCard(Context c, CommandsInterface ci, IccCardStatus ics) {
3.         if (DBG) log("Creating");
4.         mCardState = ics.mCardState;
```

```
5.          update(c, ci, ics);
6.      }
7.
8.      public void update(Context c, CommandsInterface ci, IccCardStatus ics) {
9.          synchronized (mLock) {
10.             if (mDestroyed) {
11.                 loge("Updated after destroyed! Fix me!");
12.                 return;
13.             }
14.
15.             CardState oldState = mCardState;
16.             mCardState = ics.mCardState;
17.             mUniversalPinState = ics.mUniversalPinState;
18.
19.             mGsmUmtsSubscriptionAppIndex = ics.mGsmUmtsSubscriptionAppIndex;
20.             mCdmaSubscriptionAppIndex = ics.mCdmaSubscriptionAppIndex;
21.             mImsSubscriptionAppIndex = ics.mImsSubscriptionAppIndex;
22.             mContext = c;
23.             mCi = ci;
24.
25.             //update applications
26.             for ( int i = 0; i < mUiccApplications.length; i++) {
27.                 if (mUiccApplications[i] == null) {
28.                     //Create newly added Applications
29.                     if (i < ics.mApplications.length) {
30.                         mUiccApplications[i] = new UiccCardApplication(this,
31.                                 ics.mApplications[i], mContext, mCi);
32.                     }
33.                 } else if (i >= ics.mApplications.length) {
34.                     //Delete removed applications
35.                     mUiccApplications[i].dispose();
36.                     mUiccApplications[i] = null;
37.                 } else {
38.                     //Update the rest
39.                     mUiccApplications[i].update(ics.mApplications[i], mContext, mCi);
40.                 }
41.             }
42.
43.             if (mUiccApplications.length > 0 && mUiccApplications[0] != null) {
44.                 // Initialize or Reinitialize CatService
45.                 mCatService = CatService.getInstance(mCi, mContext, this);
46.             } else {
47.                 if (mCatService != null) {
48.                     mCatService.dispose();
```

```
49.                 }
50.                 mCatService = null;
51.             }
52.             sanitizeApplicationIndexes();
53.
54.             RadioState radioState = mCi.getRadioState();
55.             // No notifications while radio is off or we just powering up
56.             if (radioState == RadioState.RADIO_ON
57.                 && mLastRadioState == RadioState.RADIO_ON) {
58.                 if (oldState != CardState.CARDSTATE_ABSENT &&
59.                         mCardState == CardState.CARDSTATE_ABSENT) {
60.                     if (DBG) log("update: notify card removed");
61.                     mAbsentRegistrants.notifyRegistrants();
62.                   mHandler.sendMessage(mHandler.obtainMessage(EVENT_CARD_REMOVED, null));
63.                 } else if (oldState == CardState.CARDSTATE_ABSENT &&
64.                         mCardState != CardState.CARDSTATE_ABSENT) {
65.                     if (DBG) log("update: notify card added");
66.                   mHandler.sendMessage(mHandler.obtainMessage(EVENT_CARD_ADDED, null));
67.                 }
68.             }
69.             mLastRadioState = radioState;
70.         }
71.     }
```

UiccCard 在 UiccController 中创建，对应实际的智能卡，当添加卡和移除卡的时候，在这个类中都会有所体现，它持有卡的状态的监听器。

从上面的代码看，最终都是调用 update()方法来更新状态。这个 update 的方法做了以下工作：

- 维护卡的状态和 PIN 码锁定状态。
- 创建或更新 mUiccApplications 的数组信息。
- 创建 CatService ，这个对象和 STK 有关。
- 通过状态判断，向外面发送 CARD REMOVE 还是 CARDADD 消息。

2) 创建/更新 UiccApplications 对象

UiccCardApplication,描述智能卡应用的信息，在 UiccCard 中创建。

```
//创建 Uicc Applications 对象
UiccCardApplication(UiccCard uiccCard,IccCardApplicationStatus as, Context c,
                            CommandsInterface ci) {
    mUiccCard = uiccCard;
    mSimId = mUiccCard.getMySimId();
    mAppState = as.app_state;
    mAppType = as.app_type;
```

```
    mPersoSubState = as.perso_substate;
    mAid = as.aid;
    mAppLabel = as.app_label;
    mPin1Replaced = (as.pin1_replaced != 0);
    mPin1State = as.pin1;
    mPin2State = as.pin2;

    mContext = c;
    mCi = ci;

    //根据 SIM 卡的类型创建相应的 IccFileHandler 对象和 IccRecord 对象
    mIccFh = createIccFileHandler(as.app_type);
    mIccRecords = createIccRecords(as.app_type, mContext, mCi);
    if (mAppState == AppState.APPSTATE_READY) {
        queryFdn();
        queryPin1State();
    }

    ci.registerForOffOrNotAvailable(mHandler, EVENT_RADIO_OFF_OR_UNAVAILABLE, null);
}
```

UiccCardApplciation 只会创建一次，在构造函数中，主要是获取 SIM 卡的一些状态信息，根据 SIM 卡的类型创建相应的 IccFileHandler 对象和 IccRecord 对象（用于读取卡的具体数据），并创建卡状态的数据监听器。

一旦卡的状态发生变化，UiccCardApplication 就会被更新。

```
//当 SIM 卡的状态发生变化时，update 的方法会被调用
//根据 UiccController 中得到的 IccCardStatus 的信息更新 Uicc Applications 对象
void update (IccCardApplicationStatus as, Context c, CommandsInterface ci) {
       synchronized (mLock) {
           //更新 type state pin ……
           AppType oldAppType = mAppType;
           AppState oldAppState = mAppState;

           mAppType = as.app_type;
           mAppState = as.app_state;
           ……

           //APP Type 变化更新
           if (mAppType != oldAppType) {
               if (mIccFh != null) { mIccFh.dispose();}
               if (mIccRecords != null) { mIccRecords.dispose();}
               mIccFh = createIccFileHandler(as.app_type);
               mIccRecords = createIccRecords(as.app_type, c, ci);
```

```
            }

            //APP State 变化更新
            if (mAppState != oldAppState) {
                // If the app state turns to APPSTATE_READY, then query FDN status,
                //as it might have failed in earlier attempt.
                if (mAppState == AppState.APPSTATE_READY) {
                    queryFdn();//FDN 查询
                    queryPin1State();//PIN 查询
                }

                notifyPinLockedRegistrantsIfNeeded(null); //PIN 状态通知
                notifyReadyRegistrantsIfNeeded(null);  //UICC 是否 Ready 状态通知
            }
        }
    }
```

这里会根据 UICC 的状态继续下一步的操作：

- 如果 UICC 需要 PIN 解锁，则会发出需要 Pin 码锁通知；进行 UICC pin 码输入解锁，然后状态变化，继续更新 UICC Card 和 Uicc Applications 直到 UICC 状态 Ready；
- 如果 UICC 已经 ready，则发出 UICC Ready 通知；

3) 创建 IccRecrods 对象

我们接下往下走，刚说到在初始化 UiccApplications 的时候，会创建 IccRecords 对象，IccRecords 对象用来记录 SIM 卡中的具体的信息。

```
private IccRecords createIccRecords(AppType type, Context c, CommandsInterface ci)
{
        if (type == AppType.APPTYPE_USIM || type == AppType.APPTYPE_SIM) {
            return new SIMRecords(this, c, ci);
        } else if (type == AppType.APPTYPE_RUIM || type == AppType.APPTYPE_CSIM){
            return new RuimRecords(this, c, ci);
        } else if (type == AppType.APPTYPE_ISIM) {
            return new IsimUiccRecords(this, c, ci);
        } else {
            // Unknown app type (maybe detection is still in progress)
            return null;
        }
    }
```

根据 SIM 卡的类型创建相应的 IccRecords 对象，下面以 SimRecors 为例进行分析。我们看下面其构造函数到底干了什么：

```
public SIMRecords(UiccCardApplication app, Context c, CommandsInterface ci) {
    super(app, c, ci);
    adnCache = new AdnRecordCache(mFh); //AND 记录缓冲区

    //注册监听事件
    mCi.registerForIccRefresh(this, EVENT_SIM_REFRESH, null);
    mCi.registerForPhbReady(this, EVENT_PHB_READY, null);
    mCi.registerForRadioStateChanged(this, EVENT_RADIO_STATE_CHANGED, null);
    mCi.registerForAvailable(this, EVENT_RADIO_AVAILABLE, null);
    mCi.registerForEfCspPlmnModeBitChanged(this, EVENT_EF_CSP_PLMN_MODE_BIT_CHANGED, null);

    //Start off by setting empty state
    resetRecords();

    //监听 UiccApplications 发出的 Sim Ready 通知
    mParentApp.registerForReady(this, EVENT_APP_READY, null);
}
```

SIMRecords 对象在创建的时候，会注册相应的监听事件，包括会向 UiccApplications 注册 EVENT_APP_READY 监听事件。

4) 创建 IccFileHandler 对象

我们接下往下走，刚说到在初始化 UiccApplications 的时候，会创建 IccFileHandler 对象，IccFileHandler 对象用来读取 SIM 卡中的数据。

```
    private IccFileHandler createIccFileHandler(AppType type) {
        switch (type) {
            case APPTYPE_SIM:
                return new SIMFileHandler(this, mAid, mCi);
            case APPTYPE_RUIM:
                return new RuimFileHandler(this, mAid, mCi);
            case APPTYPE_USIM:
                return new UsimFileHandler(this, mAid, mCi);
            case APPTYPE_CSIM:
                return new CsimFileHandler(this, mAid, mCi);
            case APPTYPE_ISIM:
                return new IsimFileHandler(this, mAid, mCi);
            default:
                return null;
        }
```

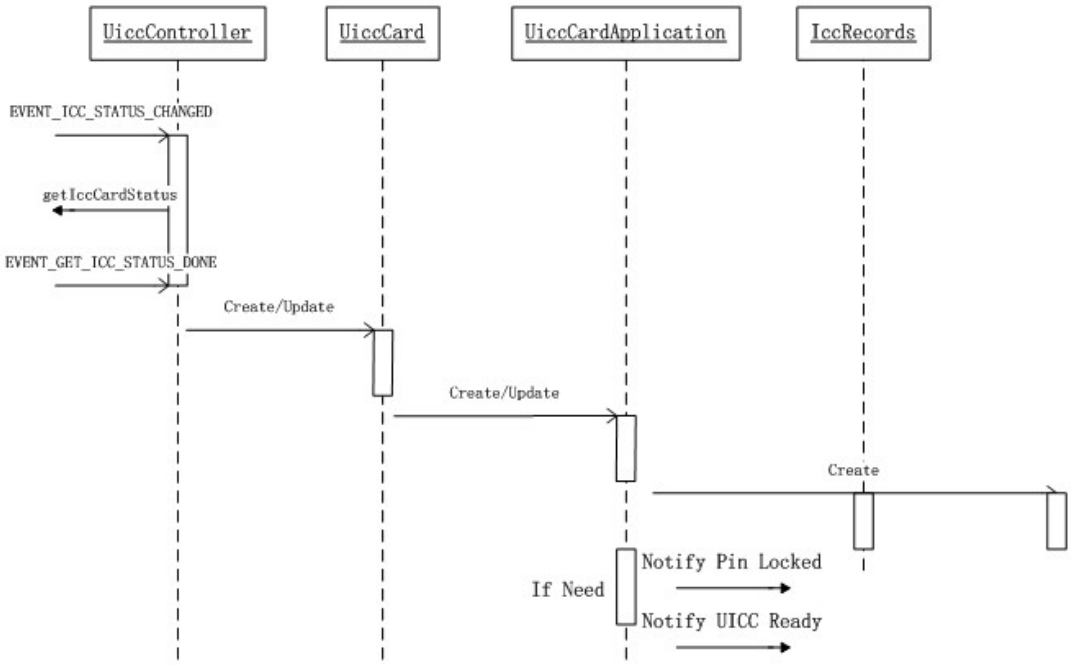
```
    }
```

IccFileHandler 类的构造函数中，根据 SIM 卡的类型创建相应的处理对象。以 SIMFileHandler 为例，其构造函数只是简单的调用父类的构造函数。

```
    public SIMFileHandler(UiccCardApplication app, String aid, CommandsInterface ci) {
        super(app, aid, ci);
    }
```

5)　状态有状态变化时的流程图

其完整的流程图如下所示：



2.6.　UICC 数据读取过程

手机启动时，根据 SIM 卡的类型，进入 SIMRecords, 开始探测 SIM 卡的状态，因为，有些 SIM 卡会设置有 PIN 码，如果 SIM 卡有 PIN 码的话，手机会弹出输入 PIN 码的框，等待用户进行解码，注意，这个时候，如果 PIN 码如果没有解的话，手机是不会去读 SIM 卡的，因为，读 SIM 卡时，必须通过 PIN 才能去读，只有一些比较特殊的字段，可以不用,比如 ECC 也就是紧急呼叫号码（一般存在卡上，运营商定制的）。同时，这 PIN 码未解的情况，手机中 SIM 卡的状态也是 PIN_REQURIED_BLOCK,

当解完 PIN 码，或是手机没有设置 PIN 码，这时，手机的会探测到 SIM 是 READY 的状态，手机只有检测到 SIM READY，才会发出读卡的请求。

UiccController 监听到 SIM 卡的状态变化，它就会会发出广播，通知 ServiceStateTracker(Gsm or Cdma)，IccCardProxy，UiccApplications 等 SIM 卡的状态已经发生了变化，并更新所持有的 mUiccCard,mUiccApplication,mIccRecords 对象。

接着，在 UiccApplications 中发出 UICC Ready 的通知，IccRecord 对象在接收到 UICC Ready 的通知后，就可以进行 SIM 卡数据的读写。

下面以 SimRecors 为例进行分析。

1) SIMRecords 监控事件的注册

SIMRecords 对象在创建的时候，会 UiccApplications 注册 EVENT_APP_READY 监听事件。

```
public SIMRecords(UiccCardApplication app, Context c, CommandsInterface ci) {
    ......

    //监听 UiccApplications 发出的 Sim Ready 通知
    mParentApp.registerForReady(this, EVENT_APP_READY, null);
}
```

2) SIMRecords 处理 SIM Ready 消息

在 UiccApplications 中发出 UICC Ready 的通知后，SIMRecords 对象便会调用 handleMessage()函数处理该消息。

```
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case EVENT_APP_READY:
                onReady();
                break;
            //IO events 通过 IccFileHandler 数据读取 SIM 数据，返回结果处理
            case EVENT_GET_IMSI_DONE:
                ……
                break;

            case EVENT_GET_MBI_DONE:
                ……
                break;
            ……
        }
    }
```

3) SIMRecords 数据的读取

在 onReady()方法中调用 fetchSimRecords()读取 SIM 中的数据。

```
    public void onReady() {
```

```
        fetchSimRecords(); //读取 record
    }

    //读取 SIM 中的数据
    protected void fetchSimRecords()
    {
        //通过 IccFileHandler 向 RIL 发送读取数据的消息
        mFh.loadEFTransparent(EF_ICCID, obtainMessage(EVENT_GET_ICCID_DONE));
        recordsToLoad++;

        // Record number is subscriber profile
        mFh.loadEFLinearFixed(EF_MBI, 1, obtainMessage(EVENT_GET_MBI_DONE));
        recordsToLoad++;

        mFh.loadEFTransparent(EF_AD, obtainMessage(EVENT_GET_AD_DONE));
        recordsToLoad++;
        ……
    }
```

具体的读取数据的操作是由 IccFileHandler 完成的，数据保存在 IccRecords 和相关系统属性中。

手机开始先读去 SIM 的 IMSI 数据，IMSI（International Mobile SubscriberIdentification Number）主要用来查找运营商的网络，里面有 MCC,MNC。接着读取 ICCID 数据，ICCID(Integrate circuit card identity)唯一标识一个移动用户。以读取 ICCID 过程为例，其调用的是 mFh.loadEFTransparent()函数接口。

4) IccFileHandler 数据读取

调用 mFh.loadEFTransparent 函数，这个就是调用 IccFileHandler 类的方法，读取 SIM 卡字段：

```
public void loadEFTransparent(int fileid, Message onLoaded) {
        Message response = obtainMessage(EVENT_GET_BINARY_SIZE_DONE,
                                         fileid, 0, onLoaded);
        mCi.iccIOForApp(COMMAND_GET_RESPONSE, fileid, getEFPath(fileid),
                        0, 0, GET_RESPONSE_EF_SIZE_BYTES, null, null, mAid, response);
    }

public void loadEFLinearFixed(int fileid, int recordNum, Message onLoaded) {
        Message response
            = obtainMessage(EVENT_GET_RECORD_SIZE_DONE,
                        new LoadLinearFixedContext(fileid, recordNum, onLoaded));

        mCi.iccIOForApp(COMMAND_GET_RESPONSE, fileid, getEFPath(fileid),
```

```
                    0, 0, GET_RESPONSE_EF_SIZE_BYTES, null, null, mAid, response);
    }
```

读取 SIM 数据，用的是 CommandsInterface 接口提供的 loadEFTransparent 和 loadEFLinearFixed 方法，就是针对不同的文件格式，实际都是调用 RIL.java 中的 iccIOForApp() 方法，其向低层的 MODEM 发送一个读取 SIM 卡的命令，在 RIL.JAVA 中。Fileid 是字段的地址，如上面说的 AND（在这为 6F3A），FDN(在这为 6F3B)。

```
    void iccIOForApp (int command, int fileid, String path,
                int p1, int p2, int p3,
                String data, String pin2, String aid, Message result)
    {
        //Note: This RIL request has not been renamed to ICC,
        //       but this request is also valid for SIM and RUIM
        RILRequest rr = RILRequest.obtain(RIL_REQUEST_SIM_IO, result);

        rr.mp.writeInt(command);
        rr.mp.writeInt(fileid);
        rr.mp.writeString(path);
        rr.mp.writeInt(p1);
        rr.mp.writeInt(p2);
        rr.mp.writeInt(p3);
        rr.mp.writeString(data);
        rr.mp.writeString(pin2);
        rr.mp.writeString(aid);

        send(rr);
    }
```

iccIOForApp()方法函数的参数含义：

参数	描述

command	读写更新……
	操作命令：
	final int COMMAND_READ_BINARY = 0xb0;
	final int COMMAND_UPDATE_BINARY = 0xd6;
	final int COMMAND_READ_RECORD = 0xb2;
	final int COMMAND_UPDATE_RECORD = 0xdc;
	final int COMMAND_SEEK = 0xa2;
	final int COMMAND_GET_RESPONSE = 0xc0;
	……

fileid	数据字段在 SIM 文件系统中的地址 ： 例如 Plmn：0x6F30
path	此数据字段上级所有目录地址： 例如 Plmn 的 Path：MF + DF_GSM = "0x3F000x7F20" 地址字段都需要根据 UICC 文件系统结构，地址决定

p1	从 3GPP SIM 相关协议可以看到，P1，P2，P3 等这些参数的含义： S：stands for data sent by the ME R：stands for data received by the ME Offset is coded on 2 bytes where P1 gives thehigh order byte and P2 the low order byte. '00 00' means no offset and reading/updating starts with the first byte '00 01' means that reading/updating starts with the second byte.
p2	
p3	
data	
pin2	
aid	由 UICC 传递上来的
result	回调 Message

Table 9: Coding of the commands

COMMAND	INS	P1	P2	P3	S/R
SELECT	'A4'	'00'	'00'	'02'	S/R
STATUS	'F2'	'00'	'00'	lgth	R
READ BINARY	'B0'	offset high	offset low	lgth	R
UPDATE BINARY	'D6'	offset high	offset low	lgth	S
READ RECORD	'B2'	rec No.	mode	lgth	R
UPDATE RECORD	'DC'	rec No.	mode	lgth	S
SEEK	'A2'	'00'	type/mode	lgth	S/R
INCREASE	'32'	'00'	'00'	'03'	S/R
VERIFY CHV	'20'	'00'	CHV No.	'08'	S
CHANGE CHV	'24'	'00'	CHV No.	'10'	S
DISABLE CHV	'26'	'00'	'01'	'08'	S
ENABLE CHV	'28'	'00'	'01'	'08'	S
UNBLOCK CHV	'2C'	'00'	see note	'10'	S
INVALIDATE	'04'	'00'	'00'	'00'	-
REHABILITATE	'44'	'00'	'00'	'00'	-
RUN GSM ALGORITHM	'88'	'00'	'00'	'10'	S/R
SLEEP	'FA'	'00'	'00'	'00'	-
GET RESPONSE	'C0'	'00'	'00'	lgth	R
TERMINAL PROFILE	'10'	'00'	'00'	lgth	S
ENVELOPE	'C2'	'00'	'00'	lgth	S/R
FETCH	'12'	'00'	'00'	lgth	R
TERMINAL RESPONSE	'14'	'00'	'00'	lgth	S

当低层的 MODEM 读到字段结果后，会有一个返回结果，由于发送读取请求时，有一个事件信息 *EVENT_GET_BINARY_SIZE_DONE*，当有返回时，会直接交给 IccFileHandler，然后由 IccFileHandler 转发给 SIMRecords，最后进行处理该字段读完后应该执行的操作。由 RIL.JAVA 通知 IccFileHandler，处理如下：

```
1.  IccFileHandler.java
2.      @Override
3.      public void handleMessage(Message msg) {
```

```
4.  ...
5.              case EVENT_GET_BINARY_SIZE_DONE:
6.                  ar = (AsyncResult)msg.obj;
7.                  response = (Message) ar.userObj;
8.                  result = (IccIoResult) ar.result;
9.
10.                 if (processException(response, (AsyncResult) msg.obj)) {
11.                     break;
12.                 }
13.
14.                 data = result.payload;
15.
16.                 fileid = msg.arg1;
17.
18.                 if (TYPE_EF != data[RESPONSE_DATA_FILE_TYPE]) {
19.                     throw new IccFileTypeMismatch();
20.                 }
21.
22.                 if (EF_TYPE_TRANSPARENT != data[RESPONSE_DATA_STRUCTURE]) {
23.                     throw new IccFileTypeMismatch();
24.                 }
25.
26.                 size = ((data[RESPONSE_DATA_FILE_SIZE_1] & 0xff) << 8)
27.                        + (data[RESPONSE_DATA_FILE_SIZE_2] & 0xff);
28.
29.                 mCi.iccIOForApp(COMMAND_READ_BINARY, fileid, getEFPath(fileid),
30.                                 0, 0, size, null, null, mAid,
31.                                 obtainMessage(EVENT_READ_BINARY_DONE,
32.                                               fileid, 0, response));
33.             break;
34.
35.             case EVENT_READ_IMG_DONE:
36.             case EVENT_READ_RECORD_DONE:
37.
38.                 ar = (AsyncResult)msg.obj;
39.                 lc = (LoadLinearFixedContext) ar.userObj;
40.                 result = (IccIoResult) ar.result;
41.                 response = lc.mOnLoaded;
42.
43.                 if (processException(response, (AsyncResult) msg.obj)) {
44.                     break;
45.                 }
46.
47.                 if (!lc.mLoadAll) {
```

```
                    sendResult(response, result.payload, null);
                } else {
                    lc.results.add(result.payload);

                    lc.mRecordNum++;

                    if (lc.mRecordNum > lc.mCountRecords) {
                        sendResult(response, lc.results, null);
                    } else {
                        mCi.iccIOForApp(COMMAND_READ_RECORD, lc.mEfid, getEFPath(lc.mEfid),
                                lc.mRecordNum,
                                READ_RECORD_MODE_ABSOLUTE,
                                lc.mRecordSize, null, null, mAid,
                                obtainMessage(EVENT_READ_RECORD_DONE, lc));
                    }
                }

            break;
...
}
```

回到 SIMRecords，处理 ICCID 读完后相关操作，代码如下：

```
case EVENT_GET_ICCID_DONE:
                    isRecordLoadResponse = true;

                    ar = (AsyncResult)msg.obj;
                    data = (byte[])ar.result;

                    if (ar.exception != null) {
                        break;
                    }

                    iccid = IccUtils.bcdToString(data, 0, data.length);

                    Log.d(LOG_TAG, "iccid: " + iccid);

             break;
```

到此，一个完整的读取 ICCID 的过程就完成了。

PS：有可能有人会问，为什么有 iccFh.loadEFTransparent()和 iccFh.loadEFLinearFixed()两个不同的接口，这主要是跟所要读取 EF 的类型有关系，SIM 卡上的文件类型有 Elementary File,

Delicated File, Cyclic File，其中 EF 又分为 Linear fixed EF，Transparent EF，Cyclic EF，所以读取的方式是不一样的，可能参考 3GPP 11.11，3GPP 51.011.