Android Phone 框架介绍（GsmPhone 为例）

---------- 青岛海信移动技术公司 软件设计二部 陈泽元 20130710

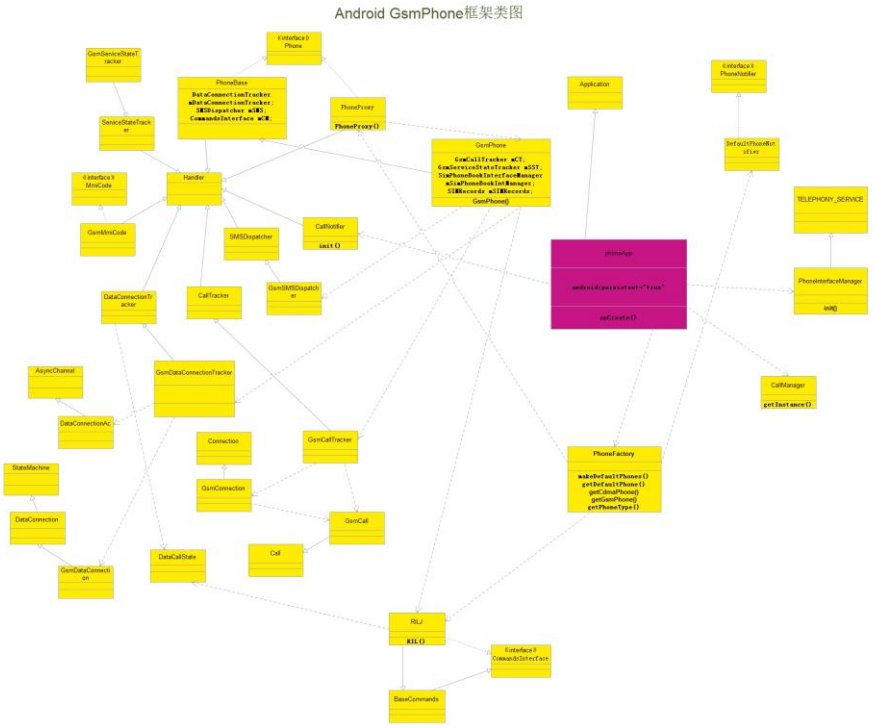
Android Phone 模块在代码结构中虽然作为一个 app 应用存在，但它提供了手机通信功能与用户交互的各种操作，并在开机时启动 phone 相关的各种服务，注册相关事件，为手机通信做好准备。

本篇将从 phoneApp 的启动，telephony framework 相关服务的启动及与 modem 的交互来分析 phone 框架（以 Android 4.1 原平台为例）。

首先，我们必须了解的是，phoneApp 作为一个独立的 app 存在，也是一个进程，启动起来后，就运行在自己独立的进程空间中，而与之交互的模块（如 TelephonyRegistry service，RILD 等）却运行在各自的进程空间中，它们之间交互必然涉及到进程间通讯（aidl、socket）。

（注： 本篇是在刚接触该模块时的学习总结，难免有很多的错漏，发现错漏时请及时联系我，以便修正）

Android phone 框架类图： 参考**《Android GsmPhone 框架类图.pdf》**



该类图包含了 phone 模块相关的各个类及其关系（暂未列出 phoneApp 应用、SMS 及 SIM 相关的类），从这张图中，我们可以清晰的了解 phone 框架相关的各个类之间是如何继承与实现的，对我们了解整个框架有个全局的概念。

我们知道，phoneApp 作为一个独立的 apk 应用，管理通话及 phone 状态相关的所有信息，这个应用是开机启动的（依赖于属性 android:persistent="true"，设置为该属性的 Application，将会被 AMS---ActivityManagerService 开机加载并启动）。但需要注意的是，它并没有启动 phoneApp 的任何一个 activity，而是启动了与 phone 相关的各个 service 及获取相关各模块的实例、注册相关事件，为后续 phone 的相关功能的处理做好准备。

phoneApp 开机启动后，将常驻内存中，各 service，注册的事件等一直处于监控运行状态，待用户有拨号或底层有来电、phone 状态或网络状态等事件到达时，将触发相关的事件处理流程（当然这些事件其它模块注册后也可以收到，比如 SystemUI 上的信号、数据图标的更新，待机界面运营商、SIM 卡状态的更新等），并在 UI 上做更新。

phone 的各功能通过 GsmPhone 来进行交互，一方面，它承接上层的各种请求，并将这些请求通过 RIL 传递给 modem，modem 处理完后的结果再返回给 GsmPhone；另外一方面，它也通过其它的处理模块（如 GsmServiceStateTracker、 GsmCallTracker、GsmDataConnectionTracker 等）监听处理 modem 的主动上报事件（如 modem 重启、来电、短信、信号、网络状态、SIM 卡状态等），并将其更新到相关对象实体中，必要的信息，会通过 intent 或 notify 的方式告知（注册过相关事件的）上层模块，以让上层根据上报信息及时更新 UI 显示。

TelephonyManager 作为第三方应用（此处的第三方应用包含系统内置的一些应用，如 Settings， MMS 及其它请求 phone 服务的中间层代码，也包含第三方开发的 apk）在请求 TELEPHONY_SERVICE (字符串定义为“phone”)时的对象，提供了一系列与 telephony framework 交互的接口，为第三方应用获取 phone 相关信息。作为进程间通讯的支持，ITelephony.aidl 中就集合了相关的接口，这些接口都在 PhoneInterfaceManager.java 中实现。第三方应用通过调用 TelephonyManager 中的接口，实际上就调用到了 PhoneInterfaceManager 中。

PhoneInterfaceManager 作为 TELEPHONY_SERVICE 的远程代理，提供了 phone 框架进程间通讯服务端功能的支持。TELEPHONY_SERVICE 并非在 SystemServer 中加载的，而是在 phoneApp 启动后，在 onCreate()中加载并注册进 ServiceManager 中去的。所以说，TELEPHONY_SERVICE 应该是运行在 phone 进程中的。

TelephonyRegistry 是一个 Android 系统级别的 service，开机后在 SystemServer 中加载的，用于注册与 phone 相关的各个状态或事件，当 modem 主动上报相关信息变化时，通过 TelephonyRegistry 注册了相关事件的模块都将能够收到这些信息的变化，根据需要通知其它模块来更新 UI 或作出相应的反应。

一、 TelephonyRegistry 的加载

TelephonyRegistry 是开机由 System Server 加载的，通过 addService()将其添加到 ServiceManager 中，交由 ServiceManager 统一管理，代码位于 SystemService.java 中：

```
class ServerThread extends Thread {
    public void run() {
        ……
        Slog.i(TAG, "Telephony Registry");
        ServiceManager.addService("telephony.registry", new TelephonyRegistry(context));
        ……
    }
}
```

TelephonyRegistry 注册到 ServiceManager 中后，就意味着它常驻内存中了，等待着其它模块（进程）的请求（请求通常就是一些事件的注册）。从代码处理流程中，我们会发现，TelephonyRegistry 实例对象并非直接被其它进程所使用，它所提供的接口在中间被进行了一层封装，使用到 TelephonyRegistry 实例对象的模块有两个：一个是 DefaultPhoneNotifier，

另外一个是 TelephonyManager。

为什么这些操作会被封装在两个处理模块中呢？从这两个模块的代码路径我们可以知道，它们的使用范围肯定是有区别的，DefaultPhoneNotifier 位于路径 frameworks/base/telephony/java/com/android/internal/telephony，显然位于 internal 下的代码只能够在 telephony 内部使用，上层应用是看不到的，不管它是 phoneApp 应用，还是需要用到 phone 相关特性的 Android 内置应用（如 Settings），都是不可用直接调用的，当然，第三方基于 SDK 开发的 apk 应用就更不可能使用了。但在开发中，上层应用或第三方 apk 可能需要用到 phone 相关的特性（如信号状态、网络状态、数据状态、SIM 卡状态及通话状态等）怎么办呢？这就由 TelephonyManager 来提供。TelephonyManager 通过将 phone 相关信息进行封装，将上层应用需要用到的相关接口暴露在 SDK 中，这样这些应用程序就能够得到 phone 相关信息了。TelephonyManager 代码路径位于 frameworks/base/telephony/java/android/telephony

从前面的 GsmPhone 框架类图中，我们可以看到，DefaultPhoneNotifier 继承自 interface PhoneNotifier，其实例对象开机在 PhoneFactory()中创建的(PhoneFactory.java)：

```
public static void makeDefaultPhone(Context context) {
    ……
    sPhoneNotifier = new DefaultPhoneNotifier();
    ……
    sProxyPhone = new PhoneProxy(new GSMPhone(context,
                                    sCommandsInterface, sPhoneNotifier));
}
```

并在创建 GsmPhone 时，被传递到 GsmPhone 中，这样当 modem 侧上报 phone 相关信息时，就可以通过这个 sPhoneNotifier 来捕获，并（根据谁注册该事件谁收到通知或得以回调的方式）告知上层应用。（该对象在 GsmPhone 构造函数中就被传递给了其基类 PhoneBase，在 PhoneBase 的 notifyXXXXX 接口中有引用该对象）

而上层应用想要及时得知 phone 相关特性的变化时，则借助 TelephonyManager 的 listen()并配合 PhoneStateListener 来实现事件的实时监听。例如，在 SystemUI 中，监控信号、数据状态变化时的代码(NetworkController.java)：

```
public NetworkController(Context context) {
    ……
    mPhone=
    (TelephonyManager)context.getSystemService(Context.TELEPHONY_SERVICE);
    mPhone.listen(mPhoneStateListener,
        PhoneStateListener.LISTEN_SERVICE_STATE
        | PhoneStateListener.LISTEN_SIGNAL_STRENGTHS
        | PhoneStateListener.LISTEN_CALL_STATE
        | PhoneStateListener.LISTEN_DATA_CONNECTION_STATE
        | PhoneStateListener.LISTEN_DATA_ACTIVITY);
    ……
}
```

通过 listen()接口，就可以做到实时监听自己感兴趣的事件，这样当底层有相关事件变化时，上层应用就能够实时收到，并通过上层重写的 mPhoneStateListener 的回调函数来响应这些事件。从 TelephonyManager 的 listen()接口实现，我们可以看到，它实际上是借助 TelephonyRegistry 的 listen()来实现的（TelephonyManager.java）：

```
public void listen(PhoneStateListener listener, int events) {
        String pkgForDebug = sContext != null ? sContext.getPackageName() : "<unknown>";
        try {
            Boolean notifyNow = (getITelephony() != null);
            //借助 TelephonyRegistry 的 listen 来实现真正的监听功能
            sRegistry.listen(pkgForDebug, listener.callback, events, notifyNow);
        } catch (RemoteException ex) {
            // system process dead
        } catch (NullPointerException ex) {
            // system process dead
        }
}
```

底层 phone 相关特性的变化又是如何通知到上层应用的呢？举例说明：

当底层有 phone 相关特性的变化时，将会逐层调用 notifyXXXXX()形式的函数，以将事件传递上来。例如通话状态变化的调用流程：（GsmCallTracker.java → GsmPhone.java → DefaultPhoneNotifier.java → TelephonyRegistry.java → 注册了相关事件的上层应用）

updatePhoneState() → phone.notifyPhoneStateChanged(); → mNotifier.notifyPhoneState(this); → mRegistry.notifyCallState(convertCallState(sender.getState()), incomingNumber); → r.callback.onCallStateChanged(state, incomingNumber); → PhoneStateListener 的 onCallStateChanged() → 应用层重写过的 onCallStateChanged()接口，即将该事件发回给注册事件时，应用层传递过来的 Handler，下面内容会详细说明这个 Handler 是谁

接下来就回调上层应用注册事件时的回调函数，进行相应的处理。如 NetworkController.java 中的 mPhoneStateListener 重写过的接口 onCallStateChanged()来处理该事件，以在 UI 上做相应的信息更新。

那么，通过 TelephonyRegistry 究竟能够监控哪些事件呢？

在 PhoneStateListener 中一一做了列举，如下：

LISTEN_SERVICE_STATE ➔ 网络状态信息的变化

LISTEN_SIGNAL_STRENGTH ➔ 信号值的变化

LISTEN_MESSAGE_WAITING_INDICATOR

LISTEN_CALL_FORWARDING_INDICATOR ➔ 呼叫转移指示

LISTEN_CELL_LOCATION ➔ 小区位置更新

LISTEN_CALL_STATE ➔ 呼叫状态的变化

LISTEN_DATA_CONNECTION_STATE ➔ 数据连接状态的变化

LISTEN_DATA_ACTIVITY ➔ 数据上下行的指示

LISTEN_SIGNAL_STRENGTHS ➔ 信号值的变化

LISTEN_OTASP_CHANGED ➔ OTASP(Over The Air Service Provisioning)的变化

LISTEN_CELL_INFO ➔ 小区信息的变化

上层应用如果想要监听这些状态信息，则在 AndroidManifest.xml 中需要添加下面的权限信息：

```
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```

小区位置更新还得添加权限：

```
<uses-permission android:name="android.permission. ACCESS_COARSE_LOCATION" />
```

另外一点，在 TelephonyRegistry.java 的代码中，我们还会发现一个现象，在某些事件的通知中，除了在使用 callback 回调的方式来告知上层外，还使用了 intent 的方式，例如

service state changed 事件。这又是为什么呢？使用 callback 的方式不就可以达到将事件通知上层的目的吗？为什么同时还要使用 intent？

以 service state changed 为例，在 notifyServiceState()接口中，同时使用了 callback 回调和发送 ACTION_SERVICE_STATE_CHANGED 的 intent，以通知其它模块 service state changed。我的理解是这样的：(TelephonyRegistry.java)

```
    public void notifyServiceState(ServiceState state) {
        if (!checkNotifyPermission("notifyServiceState()")){
            return;
        }
        synchronized (mRecords) {
            mServiceState = state;
            for (Record r : mRecords) {
                if ((r.events & PhoneStateListener.LISTEN_SERVICE_STATE) != 0) {
                    try {
                        r.callback.onServiceStateChanged(new ServiceState(state));
                    } catch (RemoteException ex) {
                        mRemoveList.add(r.binder);
                    }
                }
            }
            handleRemoveListLocked();
        }
        //发送 ACTION_SERVICE_STATE_CHANGED 的 intent，以通知上层应用
        broadcastServiceStateChanged(state);
}
```

上层应用借助 TelephonyManager 和 PhoneStateListener 来 listen 自己感兴趣的事件，当底层有相关事件需要告知上层时，通过 callback 的方式固然是可以正常的通知到上层应用。但是，对于某些应用来说，它可能并不能使用这种方式。比如 phoneApp 应用。我们知道，phoneApp 是开机启动的应用，但开机启动的是整个 phone 相关的 service 和框架，并没有启动与 phone 相关的任何 activity，包括 TELEPHONY_SERVICE 也是在 phoneApp 开机启动后才注册到 ServiceManager 中去的（将在下面的<phoneApp 的启动及初始化>部分讲解 phoneApp 是如何启动的），因此，如果想在 phoneApp 启动时使用 TelephonyManager tm = (TelephonyManager)getSystemService(Context.**TELEPHONY_SERVICE**); 的代码来获取 TELEPHONY_SERVICE 服务对象，显然是什么也得不到（因为它还没有注册到 ServiceManager 中去），相关事件也就无法注册上，无法通过 TelephonyManager 和 PhoneStateListener 来 listen 相关事件。如果等 TELEPHONY_SERVICE 注册到 ServiceManager 中以后，再来注册相关事件，时机上也不好把握，还不如使用接收 intent 的方式来的简单。这就带来了在代码层面上看到的 callback 回调和 intent 通知同时操作的结果。

还有一种情况，就是借助 PhoneStateIntentReceiver 来接收相关事件（注：这种情况不被 Android 所推荐，在 PhoneStateIntentReceiver.java 源文件开始部分已有说明，Android 建议我们还是用 TelephonyManager 和 PhoneStateListener 来代替）。在源代码中，可以发现接收并处理 ACTION_SERVICE_STATE_CHANGED 的地方并没有几个（该 intent 定义在 TelephonyIntents.java 中，是属于 internal 的代码，第三方应用可能无法直接使用）。除

phoneApp、MMS 外，就是 PhoneStateIntentReceiver 了。从 PhoneStateIntentReceiver.java 中的各个 notifyXXXX()接口实现，如 notifyServiceState()为例：

```
//注册 TelephonyIntents.ACTION_SERVICE_STATE_CHANGED
public void notifyServiceState(int eventWhat) {
    mWants |= NOTIF_SERVICE;
    mServiceStateEventWhat = eventWhat;
    mFilter.addAction(TelephonyIntents.ACTION_SERVICE_STATE_CHANGED);
}

//接收 TelephonyIntents.ACTION_SERVICE_STATE_CHANGED 并处理
public void onReceive(Context context, Intent intent) {
    ……
    else if (TelephonyIntents.ACTION_SERVICE_STATE_CHANGED.equals(action)) {
        mServiceState = ServiceState.newFromBundle(intent.getExtras());
        if (mTarget != null && getNotifyServiceState()) {
            Message message = Message.obtain(mTarget,
                                    mServiceStateEventWhat);
            mTarget.sendMessage(message); //将 message 发送回注册时的 Handler
        }
    }
    ……
}
```

可以看出，它们实际上就是将定义在 TelephonyIntents 中的各 intent（并非全部，而只是部分）注册到该 receiver 中，同时将其与应用层传递进来的 Handler 及 message event 进行绑定，当底层有相关事件变化时，在 TelephonyRegistry 中会发出相关 intent，PhoneStateIntentReceiver 就能及时接收到，并将它绑定的 message event 交回给 Handler 来处理。

示例代码如下：(Settings 模块的 RadioInfo.java 文件)

```
//创建 PhoneStateIntentReceiver 实例对象，并将其与 mHandler 绑定，后续 message 将交回给该 Handler 处理
mPhoneStateReceiver = new PhoneStateIntentReceiver(this, mHandler);
//注册几个自己感兴趣的事件，以后收到这些事件后，将交由本文件中的 mHandler 处理
mPhoneStateReceiver.notifySignalStrength(EVENT_SIGNAL_STRENGTH_CHANGED);
mPhoneStateReceiver.notifyServiceState(EVENT_SERVICE_STATE_CHANGED);
mPhoneStateReceiver.notifyPhoneCallState(EVENT_PHONE_STATE_CHANGED);
```

在 PhoneStateIntentReceiver()中，能够接收的 TelephonyIntents 有以下几个：

```
TelephonyIntents.ACTION_SERVICE_STATE_CHANGED
TelephonyIntents.ACTION_SIGNAL_STRENGTH_CHANGED
TelephonyManager.ACTION_PHONE_STATE_CHANGED
```

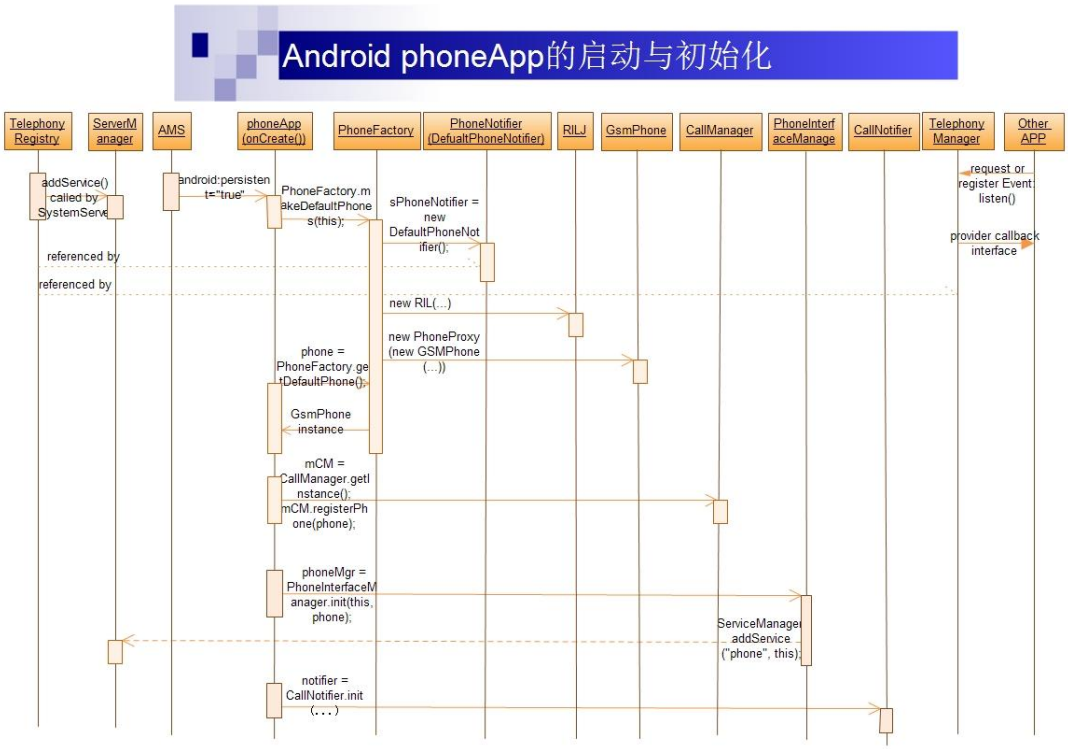
可以根据需要，增加其它 TelephonyIntents()的扩展。PhoneStateIntentReceiver 的方式也可以实现在应用层监控并处理 phone 相关事件，不过，Android 还是建议我们使用 TelephonyManager 与 PhoneStateListener 的组合来替代这种方式，所以我们可以看到，在 PhoneStateIntentReceiver 中也并没有处理几个 TelephonyIntents 中定义的 intent。当然，我们也可以直接监听广播出来的这些 intent，直接对它们进行处理，这在各模块中都有使用，比

如，NotificationManagerService.java 中监听 call state 的变化，以在状态栏上更新通话图标的代码：(NotificationManagerService.java)

```
private BroadcastReceiver mIntentReceiver = new BroadcastReceiver() {
    ……
    else if (action.equals(TelephonyManager.ACTION_PHONE_STATE_CHANGED)) {
        ……
    }
    ……
}
```

总结一下 TelephonyRegister：它是一个开机由 SystemService 加载的 Android service，提供 phone 相关各事件的注册，及后期事件处理时的回调。在应用层或其它模块中，可以使用 TelephonyManager 的 listen() 接口并配合 PhoneStateListener 来实现对（PhoneStateListener 中定义的）事件的监听，并在创建 PhoneStateListener 实例对象时重写相应的接口，以真正处理这些事件；当然也可以监听 TelephonyIntents.java 或 TelephonyManager.java 中定义的部分 intent，来响应这些事件的变化（能够接收的 intent 可以到源文件 TelephonyRegistry.java 中去查看）。

二、 phoneApp 的启动及初始化：参考《Android phoneApp 的启动与初始化.pdf》



从上图可以看出，phoneApp 由于在 AndroidManifest.xml 中设置了 android:persistent=”true”，所以会被 AMS 开机加载，执行 onCreate()接口，开始 phoneApp 的初始化。Phone 实例就是在这个过程中被创建的。从 phoneApp 的启动，来对该图所述流

程做一个说明。

phoneApp 应用开机启动后，将通过 PhoneFactory.java 的 makeDefaultPhones()来创建一个 phone 实例，本篇中创建的就是 GsmPhone 实例。在创建 GsmPhone 实例的过程中，就涉及到优选网络的配置，并创建并启用 RIL 通讯框架，以连接 rild socket 与 modem 交互。

在 GsmPhone 构造函数中，可以发现，它将 GsmCallTracker、GsmServiceStateTracker、GsmDataContionnectionTracker 及 SIMRecords 、GsmSMSDispatcher 、SimPhoneBookInterfaceManager、SimSmsInterfaceManager、PhoneSubInfo 都给实例化了，这样，不管是底层来的事件，还是上层来的请求，都将通过 GsmPhone 统一的管理起来了。结合前面的类图可以知道，这些实例中，大部分的基类都是 Handler，也就是说，它们都是用来处理相关 message 的，只不过 Android 将其按照处理业务进行了分类，便于维护和扩展，处理后的事件，将通过 intent 或 notify 的方式告知上层，以便上层根据需要作出响应（当然，对于上层传递下来的请求，相应的处理结果将回传给之前传递下来的 Hanlder，交回给上层处理）。

创建完 GsmPhone 实例后，就开始创建一系列与 phone 相关的实例对象（如去电、来电、Tone 音、Ringer 等），注册相关事件或 intent。TELEPHONY_SERVICE 也是在这个过程中注册到 ServiceManager 中去的。注册后，第三方应用就可以通过 TelephonyManager 中提供的接口来获取 phone 相关的状态或其它信息。

后续相关的事件的处理，就交由 GsmPhone 实例化时的各个 Handler 来进行。

根据 onCreate()接口处理流程可以知道，phoneApp 启动后做了以下工作：（部分内容在上图中未画出）

（1）是否支持语音功能：

```
sVoiceCapable =
        getResources().getBoolean(com.android.internal.R.bool.config_voice_capable);
```

这个变量用来标识当前产品是否支持语音功能，config_voice_capable 定义在路径 /frameworks/base/core/res/res/values/config.xml 中。为什么会有这个判断呢？这是根据不同的产品形态和需求来决定的。Android4.1 是支持平板产品的，平板产品有 WIFI 版和 3G 版，3G 版也可能支持语音或不支持语音，对于不支持语音的产品，该配置属性需要设置为 false，以屏蔽语音相关的所有操作功能。做下调试就可以发现，在手机产品中将该项设为 false 后（modem 侧依旧开启语音支持），对方来电时，对方能够听到振铃声，但该手机却一点反应都没有。

（2）创建 Phone 实例并获取实例对象：

```
// Initialize the telephony framework
PhoneFactory.makeDefaultPhones(this);
// Get the default phone
phone = PhoneFactory.getDefaultPhone();
```

接下来，就逐步启动了 Phone 管理的框架，为处理各种上层请求和底层上报事件做好准备。

（3）获取 CallManager 对象并将 phone 实例注册到 CallManager 中去，：

```
mCM = CallManager.getInstance();
mCM.registerPhone(phone);
```

（4）创建并初始化 NotificationMgr 实例：

```
notificationMgr = NotificationMgr.init(this);
```

处理与 phone 相关的通知栏、状态栏信息。（主要是通话相关的通知）

（5）创建并初始化 PhoneInterfaceManager 实例：

```
phoneMgr = PhoneInterfaceManager.init(this, phone);
```

（6）创建并初始化蓝牙通话相关实例和服务：

```
mBtHandsfree = BluetoothHandsfree.init(this, mCM);
startService(new Intent(this, BluetoothHeadsetService.class));
```

（7）创建并初始化 Ringer（通话时振铃和震动）实例：

```
ringer = Ringer.init(this);
```

（8）休眠唤醒的处理：

```
// before registering for phone state changes
PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
                mWakeLock = pm.newWakeLock(PowerManager.FULL_WAKE_LOCK
                                | PowerManager.ACQUIRE_CAUSES_WAKEUP,
                                LOG_TAG);
// lock used to keep the processor awake, when we don't care for the display.
mPartialWakeLock = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK
                                | PowerManager.ON_AFTER_RELEASE, LOG_TAG);
```

（9）传感器（与通话相关）的实例初始化：

```
mAccelerometerListener = new AccelerometerListener(this, this);
```

（10）创建并初始化主叫相关实例对象：

```
// Create the CallController singleton, which is the interface
// to the telephony layer for user-initiated telephony functionality
// (like making outgoing calls.)
callController = CallController.init(this);
// ...and also the InCallUiState instance, used by the CallController to
// keep track of some "persistent state" of the in-call UI.
inCallUiState = InCallUiState.init(this);
```

（11）创建并初始化与特定联系人相关的 Tone 音等相关的实例：

```
// Create the CallerInfoCache singleton, which remembers custom ring tone and
// send-to-voicemail settings.
//
// The asynchronous caching will start just after this call.
callerInfoCache = CallerInfoCache.init(this);
```

类 CallerInfoCache 用来处理与联系人中保存的特定的铃声、Voice mail 相关信息。当编辑联系人信息并保存时，该文件中的信息也将及时，它会及时的更新。Android 的大概流程是，编辑联系人信息时，CallerInfoCacheUtils.java 中将发送广播 com.android.phone.UPDATE_CALLER_INFO_CACHE，该广播将由文件 CallerInfoCacheUpdateReceiver.java 来接收，并在 onReceive() 中调用 PhoneApp.getInstance().callerInfoCache.startAsyncCache()来启动 CallerInfoCache 的更新。

（12）创建并初始化来电等底层事件上报的实例对象：

```
// Create the CallNotifer singleton, which handles
// asynchronous events from the telephony layer (like
// launching the incoming-call UI when an incoming call comes
// in.)
notifier = CallNotifier.init(this, phone, ringer, mBtHandsfree, new CallLogAsync());
```

CallNotifier 主要处理来电相关的事件，当然也包含一些 phone state 相关的事件（毕竟来电

不可能脱离 phone state 而存在）。

（13）注册与 SIM 卡网络锁相关的事件：

```
// register for ICC status
IccCard sim = phone.getIccCard();
if (sim != null) {
    if (VDBG) Log.v(LOG_TAG, "register for ICC status");
    sim.registerForNetworkLocked(mHandler, EVENT_SIM_NETWORK_LOCKED, null);
}
```

registerForNetworkLocked()用于注册网络锁事件。SIM 卡有 PIN 码、PUK 码、PIN2 码及 PUK2 码，这是大家周之的。但对于有些网络而言，SIM 卡功能受网络锁控制，直接控制该 SIM 卡能否接入网络（在国内，好像不支持）。

（14）注册 MMI/USSD 事件

```
// register for MMI/USSD
mCM.registerForMmiComplete(mHandler, MMI_COMPLETE, null);
```

MMI/USSD 在 UI 上也是通过拨号的方式来与网络交互的，只不过在 modem 侧或网络侧处理方式与普通语音拨号不同。如通过*21#查询呼叫转移设置，执行的就是 MMI 处理流程。相关内容可以阅读 TS 22.030 协议文档。

（15）将 CallManager 注册到 PhoneUtils.java 中，以统一管理呼叫相关内容：

```
// register connection tracking to PhoneUtils
PhoneUtils.initializeConnectionHandler(mCM);
```

（16）注册与 phone 相关的 intent，以便及时响应相关事件：

```
// Register for misc other intent broadcasts.
IntentFilter intentFilter = new IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED);
intentFilter.addAction(BluetoothHeadset.ACTION_CONNECTION_STATE_CHANGED);
intentFilter.addAction(BluetoothHeadset.ACTION_AUDIO_STATE_CHANGED);
intentFilter.addAction(TelephonyIntents.ACTION_ANY_DATA_CONNECTION_STATE_CHA
NGED);
intentFilter.addAction(Intent.ACTION_HEADSET_PLUG);
intentFilter.addAction(Intent.ACTION_DOCK_EVENT);
ntentFilter.addAction(TelephonyIntents.ACTION_SIM_STATE_CHANGED);
intentFilter.addAction(TelephonyIntents.ACTION_RADIO_TECHNOLOGY_CHANGED);
intentFilter.addAction(TelephonyIntents.ACTION_SERVICE_STATE_CHANGED);
intentFilter.addAction(TelephonyIntents.ACTION_EMERGENCY_CALLBACK_MODE_CHA
NGED);
if (mTtyEnabled) {
   intentFilter.addAction(TtyIntent.TTY_PREFERRED_MODE_CHANGE_ACTION);
}
intentFilter.addAction(AudioManager.RINGER_MODE_CHANGED_ACTION);
registerReceiver(mReceiver, intentFilter);
```

（17）初始化 audio mode 配置：

```
// Make sure the audio mode (along with some
// audio-mode-related state of our own) is initialized
// correctly, given the current state of the phone.
PhoneUtils.setAudioMode(mCM);
```

以上是 phoneApp 启动时，onCreate()中初始化的一些实例或注册的事件、广播，在创建各个对象过程中，会有很多的处理流程，上面未做细化介绍。另外，还有 AudioManager、Media 相关的初始化未做说明。通过上述初始化流程，与 phone 相关的框架就启动起来了，能够正常接收来自底层的各种事件并将用户的操作发送到底层，实现相关功能。

三、 PhoneFactory 与 GsmPhone 的创建

PhoneFactory 提供了创建各个 phone 的基本接口，其中最重要的是 makeDefaultPhone()。

根据前面的分析，GsmPhone 就是在 makeDefaultPhone()中创建出来的。

phone 类型默认的（优选网络）是 RILConstants.PREFERRED_NETWORK_MODE，但是，如果“设置”模块中做了修改的话，将使用设置模块中指定的网络类型，然后根据获取的网络类型来创建与之对应的 phone 实例：(PhoneFactory.java)

```
// Get preferred network mode
//获取默认的优选网络类型
int preferredNetworkMode = RILConstants.PREFERRED_NETWORK_MODE;
//获取“设置”模块中设定的网络类型，如果设置模块中有设定，则使用设置模块中指定的
//类型；如果没有设定，则使用默认的优选网络类型
int networkMode = Settings.Secure.getInt(context.getContentResolver(),
        Settings.Secure.PREFERRED_NETWORK_MODE, preferredNetworkMode);
Log.i(LOG_TAG, "Network Mode set to " + Integer.toString(networkMode));
……
//根据获取的网络类型，得到对应的 phone 类型
int phoneType = getPhoneType(networkMode);
//根据 phone 类型，创建对应的 phone 实例
if (phoneType == Phone.PHONE_TYPE_GSM) {
      Log.i(LOG_TAG, "Creating GSMPhone");
      sProxyPhone = new PhoneProxy(new GSMPhone(context,
                  sCommandsInterface, sPhoneNotifier)); //创建 GsmPhone
} else if (phoneType == Phone.PHONE_TYPE_CDMA) {
      switch (BaseCommands.getLteOnCdmaModeStatic()) {
            case Phone.LTE_ON_CDMA_TRUE:
                  Log.i(LOG_TAG, "Creating CDMALTEPhone");
                  sProxyPhone = new PhoneProxy(new CDMALTEPhone(context,
                              sCommandsInterface, sPhoneNotifier)); //创建 CDMALTEPhone
            break;
            case Phone.LTE_ON_CDMA_FALSE:
            default:
                  Log.i(LOG_TAG, "Creating CDMAPhone");
                  sProxyPhone = new PhoneProxy(new CDMAPhone(context,
                              sCommandsInterface, sPhoneNotifier)); //创建 CDMAPhone
            break;
    }
}
```

四、 PhoneNotifier 的初始化与作用 （DefaultPhoneNotifier）

PhoneNotifier 是一个 interface，DefaultPhoneNotifier 是其具体实现。本身提供的接口用于被 GsmPhone 实例调用，从而将底层的相关状态、事件告知给上层。其实例是在 PhoneFactory.java 的 makeDefaultPhone()创建 GsmPhone 时创建的，并在创建 GsmPhone 时将其作为参数传递给 GsmPhone，来看 GsmPhone 的构造函数：

```
public
GSMPhone (Context context, CommandsInterface ci, PhoneNotifier notifier,
    boolean unitTestMode) {
    super(notifier, context, ci, unitTestMode);
    ……
}
```

在 GsmPhone 构造函数中，传递进来的 DefaultPhoneNotifier 实例没有更多的操作，而是通过 super()调用父类（PhoneBase）的构造函数，将其赋值给父类的成员变量 protected PhoneNotifier mNotifier：

```
protected PhoneBase(PhoneNotifier notifier, Context context, CommandsInterface ci,
    boolean unitTestMode) {
    this.mNotifier = notifier;
    ……
}
```

这样，Phone 实例就与 DefaultPhoneNotifier 建立了联系（而由于 DefaultPhoneNotifier 中创建了 TelephonyRegistry 对象，它们之间也早已建立了联系）。当底层有相关事件到达或状态更新时，就会调用 GsmPhone.java 中的形如 notifyXXXXXX()的接口，GsmPhone.java 中的 notifyXXXXXX()会通过以下几种方式继续该流程：

（1） 调用 PhoneBase.java 中的父接口，调用形式类似于：

```
super.notifyServiceStateChangedP(ss);
```

（2） 直接通知注册过该事件的模块，调用形式类似于：

```
mYYYYYYRegistrants.notifyResult(code);
```

（3） 直接通过 mNotifier 变量来调用 DefaultPhoneNotifier 中的 notifyXXXXX()接口，调用形式类似于：

```
mNotifier.notifyXXXXXX(this);
```

（有兴趣的话，可以以 GsmPhone()中的接口 notifyPhoneStateChanged()和 notifySignalStrength()为例来分析下处理流程，看看相关信息是如何从底层经 GsmPhone 到 DefaultPhoneNotifier，再到 TelephonyRegistry，最后通知到应用层的。后面在分析实例流程时会以流程图的形式再详细介绍）

在 DefaultPhoneNotifier.java 中能够 notify 的相关事件有（PhoneNotifier.java）：

```
public interface PhoneNotifier {
    public void notifyPhoneState(Phone sender);
    public void notifyServiceState(Phone sender);
    public void notifyCellLocation(Phone sender);
    public void notifySignalStrength(Phone sender);
    public void notifyMessageWaitingChanged(Phone sender);
    public void notifyCallForwardingChanged(Phone sender);
```

```
    /** TODO - reason should never be null */
    public void notifyDataConnection(Phone sender, String reason, String apnType,
            Phone.DataState state);
    public void notifyDataConnectionFailed(Phone sender, String reason, String apnType);
    public void notifyDataActivity(Phone sender);
    public void notifyOtaspChanged(Phone sender, int otaspMode);
    // TODO - trigger notifyCellInfo from ServiceStateTracker
    public void notifyCellInfo(Phone sender, CellInfo cellInfo);
}
```

五、 PhoneStateListener 的作用和使用

在分析 TelephonyRegistry 时，已经提到了 PhoneStateListener，而且说得比较详细。此处再补充一下。

PhoneStateListener 必须结合 TelephonyManager 使用，才能够达到我们所期望的目的。否则 PhoneStateListener 没有任何用武之地。

我们还是以一个实例来说明 PhoneStateListener 是如何配合 TelephonyManager 实现底层事件的监听和处理的。以 CallManager.java 中对呼叫转移、呼叫等待相关内容的监听处理（NetworkController.java 中也有应用，前面 TelephonyRegistry 中已经讲解过相应的流程，所以此处不再分析这部分代码）：

(1) 获取 TELEPHONY_SERVICE 实例对象，并开始监听呼叫转移、呼叫等待事件(CallManager.java)：

```
private CallNotifier(PhoneApp app, Phone phone, Ringer ringer,
                     BluetoothHandsfree btMgr, CallLogAsync callLog) {
    ……
    TelephonyManager telephonyManager = (TelephonyManager)app.getSystemService(
            Context.TELEPHONY_SERVICE);
    //通过 listen()监听 LISTEN_MESSAGE_WAITING_INDICATOR 和
     //LISTEN_CALL_FORWARDING_INDICATOR
    telephonyManager.listen(mPhoneStateListener,
            PhoneStateListener.LISTEN_MESSAGE_WAITING_INDICATOR
            | PhoneStateListener.LISTEN_CALL_FORWARDING_INDICATOR);
    ……
}
```

(2) 通过 TelephonyManager 的 listen()接口，就将上面两个事件监听起来了，来看看 TelephonyManager 中的 listen()是如何处理的（TelephonyManager.java）：

```
public void listen(PhoneStateListener listener, int events) {
        String pkgForDebug = sContext != null ? sContext.getPackageName() : "<unknown>";
        try {
            Boolean notifyNow = (getITelephony() != null);
            //通过 TelephonyRegistry 的 listen()来实现真正的监听功能
            sRegistry.listen(pkgForDebug, listener.callback, events, notifyNow);
        } catch (RemoteException ex) {
            // system process dead
```

```
            } catch (NullPointerException ex) {
                // system process dead
            }
        }
```

显然，TelephonyManager 的 listen()接口是通过 TelephonyRegistry 的 listen()来实现真正的监听功能的。并且在参数传递过程中，将 mPhoneStateListener 的 callback 给传递了进去，这是非常重要的，因为它事件上就是对应 events 的处理函数集（并不是一个 callback 函数，而是每个 event 都会有一个对应的 callback 函数，所以此处的参数 listener.callback 实际上是一个函数集）

(3) 再来看看 TelephonyRegistry 中的 listen()又做了哪些工作（TelephonyRegistry.java）：

```
public void listen(String pkgForDebug, IPhoneStateListener callback, int events,
                   boolean notifyNow) {
    ……
    /* Checks permission and throws Security exception */
    checkListenerPermission(events);

    synchronized (mRecords) {
        // register
        Record r = null;
        find_and_add: {
            IBinder b = callback.asBinder();
            final int N = mRecords.size();
            for (int i = 0; i < N; i++) {
                r = mRecords.get(i);
                if (b == r.binder) {
                    break find_and_add;//来自同一个应用的多次监听请求不重复添加??
                }
            }
        r = new Record();
        r.binder = b;
        r.callback = callback;
        r.pkgForDebug = pkgForDebug;
        mRecords.add(r); //加入监听的队列中
    }
    ……
    if (notifyNow) {
        …… //检查当前是否有 EVENT 更新
    }
}
```

(4) 上面三步已将监听事件注册完毕，接下来就等待事件触发时，回调 callback 进行处理。当底层有相关事件到达时，将依次调用形如 notifyXXXXXX()的接口，直到 TelephonyRegistry.java 中的 notifyXXXXXX()接口。在 TelephonyRegistry.java 中的 notifyXXXXXX()接口中，将回调之前注册的 callback，上层应用将对该事件进行处理(TelephonyRegistry.java)：（以 notifyCallState()为例）

```
public void notifyCallState(int state, String incomingNumber) {
    //权限检查
    if (!checkNotifyPermission("notifyCallState()")) {
        return;
    }
    synchronized (mRecords) {
        mCallState = state;
        mCallIncomingNumber = incomingNumber;
        for (Record r : mRecords) {
            //for 循环回调处理：保证所有注册过 LISTEN_CALL_STATE 事件的模块都能
            //够被通知到，以回调注册时传入的回调函数，对事件进行处理
            if ((r.events & PhoneStateListener.LISTEN_CALL_STATE) != 0) {
                try {
                    r.callback.onCallStateChanged(state, incomingNumber);
                } catch (RemoteException ex) {
                    mRemoveList.add(r.binder);
                }
            }
        }
        handleRemoveListLocked();
    }
    //以广播的方式进行通知（前面在 TelephonyRegistry 中以讲解过为何此处还发广播），
    //此处发送的广播为：TelephonyManager.ACTION_PHONE_STATE_CHANGED
    broadcastCallStateChanged(state, incomingNumber);
}
```

上述描述就是如何使用 TelephonyManager 和 PhoneStateListener 实现 phone 相关状态及事件的监听，以及在 TelephonyRegistry 中是如何实现真正的监听和事件回调过程的。除了 TelephonyManager 和 PhoneStateListener 配合实现事件监听外，部分事件还可以通过注册 intent 的方式来监听，此处不再详述，可参考 TelephonyRegistry 部分的说明。

实际上，notify 的功能的底层实现还是 Handler 处理 message，这属于系统级别的内容，这里不做说明。有兴趣可以研究下 RegistrantList 类、Registrant 类，这两个类又是基于下面的几个类来实现的：Handler 类、Runnable 类、Looper 类、MessageQueue 类、Message 类(文件路径均在：/frameworks/base/core/java/android/os/)。分析这些类，就可以知道消息的注册和回调处理是如何实现的。这部分内容很多也很复杂，它是整个系统消息处理机制的基础。

六、 PhoneInterfaceManager 与 TelephonyManager

PhoneInterfaceManager 是 ITelephony.Stub 的实现类，也就是 TELEPHONY_SERVICE 服务（应该来说，它只是 TELEPHONY_SERVICE 暴露出来的接口，真正的 TELEPHONY_SERVICE 应该理解为整个 telephony framework 底层处理框架）。它是开机时在 phoneApp 中实例化的，并将刚刚创建的 phone 实例作为参数传递了进去，从而将其与该 phone 实例关联了起来。自然，它也就运行在 phone 进程中，实现了 ITelephony.aidl 中的全部接口。ITelephony.aidl 提供进程间通讯的接口，为进程间通讯提供了机制上的保证

（Binder）。

TelephonyManager 则提供了上层应用访问 phone 相关信息的接口，也就是说，将上层应用可能需要用到的各接口（通常声明在各个 aidl 文件中）抽取并集合起来，统一提供给应用层使用。涉及到 ITelephony.aidl、IPhoneSubInfo.aidl。

应用层正是通过直接调用 TelephonyManager 中提供的接口来访问 phone 相关信息的：

```
TelephonyManager    telMgr =
        (TelephonyManager)context.getSystemService(Context.TELEPHONY_SERVICE);
```

七、 CallManager 对通话的管理

Framework 侧语音呼叫所涉及的主要文件有：

CallManager.java ➔ 呼叫管理，提供 UI 层与 framework 层呼叫相关管理的接口

GsmCallTracker.java ➔ 处理呼叫 message 的 Handler

GsmConnection.java ➔ 主要处理通话时长相关信息

GsmCall.java ➔ 继承自 Call.java

DriverCall.java ➔ 用于 RIL 中处理从 modem 返回的 CLCC 命令结果，将各参数解析成对应的 call 参数

phoneApp 侧控制呼叫流程的主要文件：

CallController.java ➔ 控制主叫流程

CallCard.java ➔ 呼叫 UI 的显示信息（如来电人图片，通话时长，归属地等）

CallTime.java ➔ 通话时长

InCallScreen.java ➔ 处理来电

CallManager 提供了 phoneApp 访问并控制呼叫相关的各种接口，实际上就是将底层关于呼叫相关的功能抽象了出来，并根据需要，对一些呼叫相关的状态进行了组合封装成新的接口，以供 phonApp 调用，比如：发起主叫与挂断、来电、呼叫等待、语音加密、MMI、三方通话、注册呼叫或网络相关的各种事件，对 foregroundCall、backgroundCall、ringingCall 的管理等。

CallManager 是在 phoneApp 开机启动时实例化的，这在前面讲解 phoneApp 的开机启动时已有说明。需要重点关注的是，在 CallManager 实例化后，就调用了 CallManager.java 中的 registerPhone()，并且将前面创建的 phone 实例作为参数传递了进去：

```
mCM = CallManager.getInstance();
mCM.registerPhone(phone);
```

这样，CallManager 就与该 phone（本篇对应的也就是 GsmPhone）关联了起来，用于处理 GsmPhone 呼叫相关的管理和控制。

在 CallManager.java 的 registerPhone()接口中：

```
public boolean registerPhone(Phone phone) {
    Phone basePhone = getPhoneBase(phone);

    if (basePhone != null && !mPhones.contains(basePhone)) {

        if (DBG) {
            Log.d(LOG_TAG, "registerPhone(" +
                        phone.getPhoneName() + " " + phone + ")");
        }
```

```
            if (mPhones.isEmpty()) {
                mDefaultPhone = basePhone;
            }
            //mPhones 中的对象，表示已纳入 CallManager 呼叫管理 phone 实例
            mPhones.add(basePhone);
            //将 mRingingCalls 与 GsmCallTracker 中的 RingingCall 关联起来
            mRingingCalls.add(basePhone.getRingingCall());
            //将 mBackgroundCalls 与 GsmCallTracker 中的 BackgroundCall 关联起来
            mBackgroundCalls.add(basePhone.getBackgroundCall());
            //将 mForegroundCalls 与 GsmCallTracker 中的 ForegroundCall 关联起来
            mForegroundCalls.add(basePhone.getForegroundCall());
            //注册呼叫及 phone 状态相关事件（与呼叫相关联的重要事件或状态）
            registerForPhoneStates(basePhone);
            return true;
        }
        return false;
}
```

最重要的接口调用是 registerForPhoneStates()，该接口中注册了一系列与 phone 呼叫有关联的事件：

```
private void registerForPhoneStates(Phone phone) {
        // for common events supported by all phones
        //phone 实际上是 PhoneBase 对象，所以此处的 precise call state change 事件将注册给
        //PhoneBase
        phone.registerForPreciseCallStateChanged(mHandler,
                        EVENT_PRECISE_CALL_STATE_CHANGED, null);
        phone.registerForDisconnect(mHandler, EVENT_DISCONNECT, null);
        phone.registerForNewRingingConnection(mHandler,
                        EVENT_NEW_RINGING_CONNECTION, null);
        phone.registerForUnknownConnection(mHandler,
                        EVENT_UNKNOWN_CONNECTION, null);
        phone.registerForIncomingRing(mHandler, EVENT_INCOMING_RING, null);
        phone.registerForRingbackTone(mHandler, EVENT_RINGBACK_TONE, null);
        phone.registerForInCallVoicePrivacyOn(mHandler,
                        EVENT_IN_CALL_VOICE_PRIVACY_ON, null);
        phone.registerForInCallVoicePrivacyOff(mHandler,
                        EVENT_IN_CALL_VOICE_PRIVACY_OFF, null);
        phone.registerForDisplayInfo(mHandler, EVENT_DISPLAY_INFO, null);
        phone.registerForSignalInfo(mHandler, EVENT_SIGNAL_INFO, null);
        phone.registerForResendIncallMute(mHandler, EVENT_RESEND_INCALL_MUTE, null);
        phone.registerForMmiInitiate(mHandler, EVENT_MMI_INITIATE, null);
        phone.registerForMmiComplete(mHandler, EVENT_MMI_COMPLETE, null);
        phone.registerForSuppServiceFailed(mHandler, EVENT_SUPP_SERVICE_FAILED, null);
        phone.registerForServiceStateChanged(mHandler,
```

```
                    EVENT_SERVICE_STATE_CHANGED, null);

        // for events supported only by GSM and CDMA phone
        if (phone.getPhoneType() == Phone.PHONE_TYPE_GSM ||
                phone.getPhoneType() == Phone.PHONE_TYPE_CDMA) {
                phone.setOnPostDialCharacter(mHandler,
                        EVENT_POST_DIAL_CHARACTER, null);
        }

        // for events supported only by CDMA phone
        if (phone.getPhoneType() == Phone.PHONE_TYPE_CDMA ){
                phone.registerForCdmaOtaStatusChange(mHandler,
                        EVENT_CDMA_OTA_STATUS_CHANGE, null);
                phone.registerForSubscriptionInfoReady(mHandler,
                        EVENT_SUBSCRIPTION_INFO_READY, null);
                phone.registerForCallWaiting(mHandler, EVENT_CALL_WAITING, null);
                phone.registerForEcmTimerReset(mHandler, EVENT_ECM_TIMER_RESET, null);
        }
}
```

以 registerForPreciseCallStateChanged()为例。我们会发现，在文件 CallManager.java 和 PhoneBase.java 中均有这个接口的定义，而且接口内部代码实现基本上是一样的，这是为什么呢？再仔细一看会发现，在 CallManager.java 的 registerForPhoneStates()中实际上调用的是 PhoneBase.java 的接口，这样的话，就将自己关注的 precise call state change 事件注册给了 PhoneBase.java，当底层有呼叫相关状态变化时，PhoneBase.java 中的 notifyPreciseCallStateChangedP()将得以调用，从而通过：

```
protected void notifyPreciseCallStateChangedP() {
        AsyncResult ar = new AsyncResult(null, this, null);
        mPreciseCallStateRegistrants.notifyRegistrants(ar);
}
```

将 precise call state change 事件通知给 CallManager（实际上，凡是通过注册过该事件给 PhoneBase 的模块都会收到该通知），在 CallManager 中将在 handleMessage()来处理该事件：

```
public void handleMessage(Message msg) {
        switch (msg.what) {
                ……
                case EVENT_PRECISE_CALL_STATE_CHANGED:
                        if (VDBG) Log.d(LOG_TAG, " handleMessage
                                (EVENT_PRECISE_CALL_STATE_CHANGED)");
                        mPreciseCallStateRegistrants.notifyRegistrants((AsyncResult) msg.obj);
                break;
                ……
        }
        ……
}
```

可以看到，在 handle precise call state change 消息时，也没有过多的操作，而是直接将该消

息通过 notifyRegistrants()又通知到其它模块（大部分是应用层）。谁曾经注册过该事件？搜索下会发现，注册该事件的应用层文件有以下几个：BluetoothHandsfree.java CallNotifier.java InCallScreen.java PhoneUtils.java
这些模块注册 precise call state change 消息，正是通过 CallManager.java 中的 registerForPreciseCallStateChanged()来实现的，所以它们直接接收到的 precise call state change 通知是由 CallManager 发出的，而不是接收 PhoneBase 发出的（虽然最终还是由 PhoneBase 发出的，但这些模块并不直接接收 PhoneBase 的通知，因为它们的 precise call state change 事件并没有直接注册给 PhoneBase），如 BluetoothHandsfree.java 中的注册代码：

```
//mCM 是 CallManager 的对象，因此此处的 precise call state change 将注册给 CallManager
mCM.registerForPreciseCallStateChanged(mStateChangeHandler,
                                       PRECISE_CALL_STATE_CHANGED, null);
```

其它事件的注册也是类似的。上层模块将事件注册给 CallManager，CallManager 又将同样的事件注册给 PhoneBase，这样当 PhoneBase 通知相关事件变化时，CallManager 再将这些事件转发给上层模块，实现了从上到下、从下到上整个流程的处理。为什么这么麻烦呢？为什么上层不直接将事件注册给 PhoneBase 呢？PhoneBase 是所有 phone 的基类，不负责与上层的直接交互，通过中间封装 CallManager 来管理，会使得架构设计更合理，避免 PhoneBase 涉及过多的上层业务操作导致过于复杂。

语音呼叫相关的状态有以下几种：（Call.java）

```
public enum State {
    IDLE, //idle 状态，无任何类型的呼叫
    ACTIVE, //激活状态，表示正在通话当中
    HOLDING, //呼叫保持状态，用于三方通话（需要自己开通呼叫保持功能和运营商的支持）
    DIALING, //正在发起语音呼叫的过程中，即拨号当中，暂时还没有接通对方
    ALERTING, //发起语音呼叫后，正在振铃状态，已接通对方，但对方还没有接听
    INCOMING, //来电（振铃）状态
    WAITING, //呼叫等待状态（需要被呼 SIM 开通呼叫等待功能）
    DISCONNECTED, //通话已完全结束，连接完全释放
    DISCONNECTING; //通话正在断开过程中，还没有完全断开
    //判断当前是否有语音通话存在（不管来电还是去电，也不管通话中还是振铃或呼叫
    //等待状态或呼叫保持状态，它们都意味着当前有语音通话存在）
    public boolean isAlive() {
        return !(this == IDLE || this == DISCONNECTED || this == DISCONNECTING);
    }
    //判断是否有来电处于（接通）振铃状态（不包括去电的 ALERTING），即：是否有来
    //电到达
    public boolean isRinging() {
        return this == INCOMING || this == WAITING;
    }
    //判断是否有去电处于拨号过程中（对方尚未接听），即：是否有正在往外拨打电话
    public boolean isDialing() {
        return this == DIALING || this == ALERTING;
```

```
    }
}
```

另外，还封装了其它接口供上层获取 call 的状态，如(Call.java)：

```
//判断某个 call 是否处于 IDLE 状态
public boolean isIdle() {
    return !getState().isAlive();
}
//判断某个 call 是否处于 DIALING 或 ALERTING 状态
public boolean isDialingOrAlerting() {
    return getState().isDialing();
}
//判断某个 call 是否处于 INCOMING 或 WAITING 状态
public boolean   isRinging() {
    return getState().isRinging();
}
```

根据 Android 侧的定义（GSM 协议有对应的规定），一个 GSM 接续中，最多允许 7 路连接接入；而每一个语音通话中，最多可以另外再添加 4 路通话（用于会议电话的情况），其变量定义在 GsmCallTracker.java 中：

```
static final int MAX_CONNECTIONS = 7;     // only 7 connections allowed in GSM
static final int MAX_CONNECTIONS_PER_CALL = 5; // only 5 connections allowed per call
```

CallManager 中一共管理着下列几类呼叫：

(1) mRingingCalls

对应 GsmCallTracker.java 中的 ringingCall 变量，用于管理处于 INCOMING 和 WAITING 状态的呼叫。

(2) mForegroundCalls

对应 GsmCallTracker.java 中的 foregroundCall 变量，用于管理处于 ACTIVE、DIALING、ALERTING 状态的呼叫。

(3) mBackgroundCalls

对应 GsmCallTracker.java 中的 backgroundCall 变量，用于管理处于 HOLDING 状态的呼叫。

各呼叫类型与状态之间的对应可以从下面接口看出来(GsmCallTracker.java)：

```
private GsmCall
    parentFromDCState (DriverCall.State state) {
        switch (state) {
            case ACTIVE:
            case DIALING:
            case ALERTING:
                return owner.foregroundCall; //foregroundCall 管理的 call 状态
            ……
            case HOLDING:
                return owner.backgroundCall; //backgroundCall 管理的 call 状态
            ……
```

```
            case INCOMING:
            case WAITING:
                return owner.ringingCall; //ringingCall 管理的 call 状态
            ……
        }
    }
```

其中 DriverCall 是提供与 RIL 上 get current call list 对接的接口，即解析来自 AT+CLCC 返回结果，以得到当前所有呼叫连接的详细信息（包括呼叫 ID、呼叫 direction、号码、号码类型、本路呼叫状态-----详细信息可以参考 3GPP 关于 CLCC 命令的定义）。

就使用而言，这些状态都可以通过 Call.java 中的 getState()来获取到。如 CallManager.java 中的判断方法：

```
//判断某个 call 的状态是否已经断开
if(Call.State.DISCONNECTED == call.getState())
```

当然，在 CallManager.java 中又做了一层封装，分别用 hasActiveFgCall()、hasActiveBgCall()、hasActiveRingingCall()来代替了。

在 GsmCallTracker.java 中的判断方法：

```
//判断某个 call 的状态是否处于 WAITING（呼叫等待）状态
if(ringingCall.getState() == GsmCall.State.WAITING)
//判断某个 call 的状态是否处于 INCOMING 或 WAITING 状态
if (ringingCall.getState().isRinging())
```

Phone 的状态有以下几个(Phone.java)：

```
enum State {
    IDLE, RINGING, OFFHOOK;
};
```

需要注意的是，phone 的状态与 call 的状态有关联，但不是一一对应的，phone 的状态一共只有 3 种，而 call 的状态有 9 种，它们之间的对应关系为(GsmCallTracker.java)：

```
private void
    updatePhoneState() {
        Phone.State oldState = state;

        if (ringingCall.isRinging()) {
            //phone 的 RINGING 状态对应 ringingCall 的 INCOMING 和 WAITING 状态
            state = Phone.State.RINGING;
        } else if (pendingMO != null ||
                !(foregroundCall.isIdle() && backgroundCall.isIdle())) {
            // pendingMO 表示发起的 MO call，即：发起的主叫，或称之为去电
            //整个判断条件表示：OFFHOOK 状态包括主叫摘机和挂机，被叫摘机和挂机
            //动作，对应的 call 状态有（包含前台和后台 call）：
            //DIALING、ALERTING、ACTIVE、HOLDING、DISCONNECTING。
            state = Phone.State.OFFHOOK;
        } else {
            //phone 的 IDLE 状态对应 call 的其它状态（没有去电，也没有来电，前后台
            //call 都处于）：DISCONNECTED 、IDLE 状态
```

```
                state = Phone.State.IDLE;
            }

            if (state == Phone.State.IDLE && oldState != state) {
                //新的 phone 状态为 IDLE，而之前的状态不是 IDLE，说明有语音通话断开
                voiceCallEndedRegistrants.notifyRegistrants(
                        new AsyncResult(null, null, null));
            } else if (oldState == Phone.State.IDLE && oldState != state) {
                //之前的 phone 状态为 IDLE，新的状态不是 IDLE，说明有语音通话存在
                voiceCallStartedRegistrants.notifyRegistrants (
                            new AsyncResult(null, null, null));
            }

            if (state != oldState) {
                //之前的 phone 状态和新的 phone 状态不一致，通知相关模块状态的变化
                phone.notifyPhoneStateChanged();
            }
        }
```

Phone 状态与 call 状态之间的对应关系可以通过 CallCard.java 的接口 updateState()所输出的 log 看出来。它会同时打印出三种 call 的详细信息及此时的 phone 信息，例如下面的 log：

```
去电后，对方接听，然后对方挂断时的 log:（对方挂断但尚未进入 IDLE 状态）
(注：对方挂断时，call 没有 DISCONNECTING 状态，只有主动挂断时才有，但无论谁挂断，都有 DISCONNECTED 状态)
D/CallCard(   557): updateState(CallManager {
D/CallCard(   557): state = IDLE //该 state 就是 phone 此时的状态
D/CallCard(   557): - Foreground: DISCONNECTED from Handler
                    (com.android.internal.telephony.gsm.GSMPhone) {42282b38}
D/CallCard(   557):      Conn: [ incoming: false state: DISCONNECTED post dial state:
                    COMPLETE] //foregroundCall 详细信息
D/CallCard(   557): - Background: IDLE from Handler
                    (com.android.internal.telephony.gsm.GSMPhone) {42282b38}
D/CallCard(   557):      Conn: [] //backgroundCall 详细信息
D/CallCard(   557): - Ringing: IDLE from Handler
                    (com.android.internal.telephony.gsm.GSMPhone) {42282b38} //ringingCall 信息
D/CallCard(   557): Phone: Handler (com.android.internal.telephony.gsm.GSMPhone)
                    {42282b38}, name = GSM, state = IDLE //当前 call 对应的 phone 信息
D/CallCard(   557): - Foreground: DISCONNECTED Background: IDLE Ringing: IDLE
D/CallCard(   557): })...
```

在 framework 的 Call 管理中，所有的呼叫都被放入 GsmCall.java 的 connections 队列中：

```
/*package*/ ArrayList<Connection> connections = new ArrayList<Connection>();
```

通过该文件中的 attach()、attachFake()和 detach()来负责 Call 队列的管理，处于队列中的每个 Call 状态的更新则通过 update()来完成。

RIL 侧从 modem 查询到的当前所有呼叫信息，通过 CLCC 命令返回，在 RIL.java 中的

responseCallList()解析后，封装在 DriverCall 对象中。Framework 将在 GsmCallTracker.java 的 handlePollCalls()完成更新 Call 队列 connections、各 call 状态及通知上层等工作：

```
//该接口处理代码很长，用来处理 CLCC 命令的返回结果，借助 connections（实际上就是
//一个 GsmConnection 的数组）队列来统一管理所有的 call，并通知各上层模块 call 信息的
//变化。
protected synchronized void
handlePollCalls(AsyncResult ar) {
    List polledCalls;

    if (ar.exception == null) {
        polledCalls = (List)ar.result; //modem 返回的 CLCC 查询结果
    } else if (isCommandExceptionRadioNotAvailable(ar.exception)) {
        // just a dummy empty ArrayList to cause the loop
        // to hang up all the calls
        polledCalls = new ArrayList();
    } else {
        // Radio probably wasn't ready--try again in a bit
        // But don't keep polling if the channel is closed
        pollCallsAfterDelay();
        return;
    }
    ……
    for (int i = 0, curDC = 0, dcSize = polledCalls.size()
            ; i < connections.length; i++) {
        //for 循环开始比较并处理 CLCC 返回结果和 connections 队列中缓存的原有的呼叫
        GsmConnection conn = connections[i];
        DriverCall dc = null;
        // polledCall list is sparse
        if (curDC < dcSize) {
            dc = (DriverCall) polledCalls.get(curDC);
            if (dc.index == i+1) {
                curDC++;
            } else {
                dc = null;
            }
        }
        if (conn == null && dc != null) {
            //当前索引 i 处的 call 为 null，但 CLCC 查询对应的索引处的 call 不为 null，
            //说明这是一路新的 call（可能是去电，也可能是来电。需要注意的是，不管
            //是来电还是去电，加入 connections 呼叫队列的操作都在该 for 循环中，也就
            //是说，去电并没有在发起呼叫时就加入了 connections 队列，因为那样操作
            //是不合理的，谁能保证该呼叫正常的发送给了 modem，而且 modem 正常的
            //进行了处理？如果在发起呼叫后，modem 处理该呼叫之前就将其加入
            //connections 队列，很容易给上层造成混乱，在 for 循环中统一处理更为合理）
```

```
                // Connection appeared in CLCC response that we don't know about
                if (pendingMO != null && pendingMO.compareTo(dc)) {
                    //发起的主叫 call：去电
                    if (DBG_POLL) log("poll: pendingMO=" + pendingMO);

                    // It's our pending mobile originating call
                    connections[i] = pendingMO;//发起的主叫，加入 call 队列
                    pendingMO.index = i;
                    pendingMO.update(dc);//更新 Call.java 的 state 的值
                    pendingMO = null;

                    // Someone has already asked to hangup this call
                    if (hangupPendingMO) {//处理主动挂断请求
                        hangupPendingMO = false;
                        try {
                            if (Phone.DEBUG_PHONE) log(
                                    "poll: hangupPendingMO, hangup conn " + i);
                            hangup(connections[i]);
                        } catch (CallStateException ex) {
                            Log.e(LOG_TAG, "unexpected error on hangup");
                        }

                        // Do not continue processing this poll
                        // Wait for hangup and repoll
                        return;
                    }
                } else {
                    //收到来电 call
                    connections[i] = new GsmConnection(phone.getContext(), dc, this, i);

                    // it's a ringing call
                    if (connections[i].getCall() == ringingCall) {
                        //来电 ringingCall
                        newRinging = connections[i];
                    } else {
                        //非去电，也非来电，未知的 call
                        // Something strange happened: a call appeared
                        // which is neither a ringing call or one we created.
                        // Either we've crashed and re-attached to an existing
                        // call, or something else (eg, SIM) initiated the call.

                        Log.i(LOG_TAG,"Phantom call appeared " + dc);

                        // If it's a connected call, set the connect time so that
```

```
                    // it's non-zero.    It may not be accurate, but at least
                    // it won't appear as a Missed Call.
                    if (dc.state != DriverCall.State.ALERTING
                            && dc.state != DriverCall.State.DIALING) {
                        connections[i].connectTime = System.currentTimeMillis();
                    }

                    unknownConnectionAppeared = true;
                }
            }
            hasNonHangupStateChanged = true;
        } else if (conn != null && dc == null) {
            //当前索引 i 处有 call 信息，但 CLCC 查询对应的索引没有，说明有 call 断开
            //连接了
            // Connection missing in CLCC response that we were
            // tracking.
            //断开连接的 call 放入 droppedDuringPoll 队列，待后续进一步处理
            droppedDuringPoll.add(conn);
            // Dropped connections are removed from the CallTracker
            // list but kept in the GsmCall list
            connections[i] = null;
        } else if (conn != null && dc != null && !conn.compareTo(dc)) {
            //当前索引 i 处有 call 信息，且 CLCC 查询对应的索引也有 call 信息，且两个
            //call 信息不相同，说明 conn 与 dc 是两个不同的 call
            // Connection in CLCC response does not match what
            // we were tracking. Assume dropped call and new call
            //将 conn 的这个 call 放入 droppedDuringPoll 队列，待后续移除。
            droppedDuringPoll.add(conn);
            //重新创建 dc 这个 GsmConnection，并将其放入 call 队列
            connections[i] = new GsmConnection (phone.getContext(), dc, this, i);

            if (connections[i].getCall() == ringingCall) {
                newRinging = connections[i];
            } // else something strange happened
            hasNonHangupStateChanged = true;
        } else if (conn != null && dc != null) { /* implicit conn.compareTo(dc) */
            //当前索引与 CLCC 查询的对应索引的 call 都不为空（且两个 call 是同一个
            //call），则更新该 call 的状态。
            boolean changed;
            changed = conn.update(dc);
            hasNonHangupStateChanged = hasNonHangupStateChanged || changed;
        }
        .....
    }
```

```
// This is the first poll after an ATD.
// We expect the pending call to appear in the list
// If it does not, we land here
if (pendingMO != null) {
    //刚发起主叫时，modem 侧还没有收到 ATD 命令，CLCC 返回结果肯定是没有这
    //路主叫信息的，此时 pendingMO 不可能在前面的 for 循环中得到处理，故它不为
    //null。
    Log.d(LOG_TAG,"Pending MO dropped before poll fg state:"
                        + foregroundCall.getState());
    //直接将其放入 droppedDuringPoll，待后续移除掉，因为接下来 CLCC 查询会得
    //到这路主叫 call 的详细信息，并再次进入 handlePollCalls()的 for 循环，该主叫 call
    //得以加入 connections 队列（放心：这路主叫不会被遗漏的），正因为能够用 for
    //循环来统一处理 call，所有主叫的 call 并没有在发起主叫时就放入 connections，
    //而是等待下一次 CLCC 查询返回结果后，才根据查询结果是否为 null，来决定是
    //否放入队列
    droppedDuringPoll.add(pendingMO);
    pendingMO = null;
    hangupPendingMO = false;
}

if (newRinging != null) {
    //新的 ringingCall，通知上层模块
    phone.notifyNewRingingConnection(newRinging);
}

// clear the "local hangup" and "missed/rejected call"
// cases from the "dropped during poll" list
// These cases need no "last call fail" reason
for (int i = droppedDuringPoll.size() - 1; i >= 0 ; i--) {
    //清空 droppedDuringPoll 中的 call 信息，并执行断开相关流程
    GsmConnection conn = droppedDuringPoll.get(i);

    if (conn.isIncoming() && conn.getConnectTime() == 0) {
        // Missed or rejected call
        Connection.DisconnectCause cause;
        if (conn.cause == Connection.DisconnectCause.LOCAL) {
            //本地主动挂断（disconnect cause 为 LOCAL）
            cause = Connection.DisconnectCause.INCOMING_REJECTED;
        } else {
            //非本地挂断
            cause = Connection.DisconnectCause.INCOMING_MISSED;
        }
```

```
                if (Phone.DEBUG_PHONE) {
                    log("missed/rejected call, conn.cause=" + conn.cause);
                    log("setting cause to " + cause);
                }
                droppedDuringPoll.remove(i);
                conn.onDisconnect(cause);
            } else if (conn.cause == Connection.DisconnectCause.LOCAL) {
                // Local hangup
                droppedDuringPoll.remove(i);
                conn.onDisconnect(Connection.DisconnectCause.LOCAL);
            } else if (conn.cause ==
                Connection.DisconnectCause.INVALID_NUMBER) {
                droppedDuringPoll.remove(i);
                conn.onDisconnect(Connection.DisconnectCause.INVALID_NUMBER);
            }
        }

        // Any non-local disconnects: determine cause
        if (droppedDuringPoll.size() > 0) {
            //未知原因的挂断，向 modem 查询挂断的原因
            cm.getLastCallFailCause(
                obtainNoPollCompleteMessage(EVENT_GET_LAST_CALL_FAIL_CAUSE));
        }

        if (needsPollDelay) {
            //下一次 CLCC 查询请求
            pollCallsAfterDelay();
        }

        // Cases when we can no longer keep disconnected Connection's
        // with their previous calls
        // 1) the phone has started to ring
        // 2) A Call/Connection object has changed state...
        //      we may have switched or held or answered (but not hung up)
        if (newRinging != null || hasNonHangupStateChanged) {
            //清空全部 ringingCall、foregroundCall、backgroundCall 信息
            internalClearDisconnected();
        }
        //更新 phone 状态并通知上层模块：将 phone 状态（共 3 种）与 call（共 9 种）状态对
        //应起来
        updatePhoneState();

        if (unknownConnectionAppeared) {
            phone.notifyUnknownConnection();
```

```
        }

        if (hasNonHangupStateChanged || newRinging != null) {
            //通知上层有 call 信息更新： precise call state changed
            phone.notifyPreciseCallStateChanged();
        }

        //dumpState();
    }
```

我们会发现，在 connections 中缓存的每个 call 都是一个 GsmConnection 对象，通过 GsmConnection.java 中的 attach 或 attachFake()将其放入 ringingCall 或 foregroundCall 或 backgroundCall 中进行管理，最终都放入了 Call.java 的 connections 中，它是一个 ArrayList 类型的变量。GsmConnection 主要负责通话时长相关的工作及状态的维护。

八、 CallNotifier 的初始化与作用

CallNotifier 的实例化也是在 phoneApp 启动时，在 onCreate()中进行的。在 CallNotifier 实例化中，会通过 registerForNotifications()向 CallManager 注册一系列 call 相关的事件。相同的事件注册在 PhoneBase.java 中也有一套，它们的引用上下文是有区别的，这一点在 CallManager 的讲解中有提到。

从 CallNotifier 类的实现看，它实际上就是一个 Hanlder，处理 call 相关的事件，同时实现了 CallerInfoAsyncQuery.OnQueryCompleteListener 接口，当监听到呼叫信息时，会在 onQueryComplete()中查询该联系人相关的铃声配置。

该类中还有一个内部类 InCallTonePlayer-----继承自 Thread，用于播放通话相关的 Tone 音（如呼叫等待音，挂断音，忙音等等）

九、 TelephonyProvider 的作用

TelephonyProvider.java 继承自 ContentProvider，通过内部类 DatabaseHelper 来实现数据库 telephony.db 中关于 APN 信息的增删改查操作。

Android 平台默认的 apn 配置文件位于：development\data\etc 路径下（各平台可能根据各自实现的不同，重新做了配置，可以通过在所有 mk 文件中搜索 apns-conf.xml 来查看当前使用的是 apn 配置文件）

```
private static final String PARTNER_APNS_PATH = "etc/apns-conf.xml";
```

xml 中的 APN 配置信息就是在 loadApns()中，通过解析 xml 文件，将 APN 配置信息写入 telephony.db 数据库。

需要注意的是：APN 数据库只会在刷机后，第一次开机，才会解析 apns-conf.xml 文件，并写入数据库，以后每次开机，都不会再解析该文件，所有的 APN 增删改查都将在数据库中进行。因此，修改了 apns-conf.xml 文件，如果想要看看修改后的效果，只更新 apns-conf.xml 文件，是没有用的，必须同时删除/data/data/com.android.telephony.provider 中的整个 telephony.db 数据库文件，然后重新开机，xml 中的数据才会重新解析并写入 telephony.db 中，否则，你会发现无论你如何修改 apns-conf.xml 文件，从 telephony.db 中读取的 APN 配置信息始终不发生任何变化。

十、 RILJ 和 RIL

RILJ 是在 phoneApp 应用启动后实例化的，作为 socket 的客户端，与 RILD 进行通讯。它的运行受 phone 应用生命周期的影响。当 phone 应用异常退出时，它也就不存在了（也就是 socket 的客户端连接断开了）。

RILD 是开机启动的进程，由 init.rc 加载，启动后，会创建 socket 通讯的服务器端，并等待客户端的连接。

从 RILJ 的初始化中，我们可以看到：

```
public RIL(Context context, int preferredNetworkType, int cdmaSubscription) {
    super(context);
    ….
    mRequestMessagesPending = 0;
    mRequestMessagesWaiting = 0;
    //启动一个新线程，用来发送来自上层的请求
    mSenderThread = new HandlerThread("RILSender");
    mSenderThread.start();

    Looper looper = mSenderThread.getLooper();
    mSender = new RILSender(looper);

    ConnectivityManager cm = (ConnectivityManager)context.getSystemService(
            Context.CONNECTIVITY_SERVICE);
    //判断 wifi 版本和 3G 版本
    if (cm.isNetworkSupported(ConnectivityManager.TYPE_MOBILE) == false) {
        riljLog("Not starting RILReceiver: wifi-only");
    } else {
        //3G 版本，则启动一个新线程，用来读取来自 RIL 返回的 modem 处理结果
        riljLog("Starting RILReceiver");
        mReceiver = new RILReceiver();
        mReceiverThread = new Thread(mReceiver, "RILReceiver");
        mReceiverThread.start();
        ……
    }
}
```

RILSender 和 RILReceiver 是 RILJ 上实现 RIL REQUEST 处理的核心。RILSender 负责 RIL REQUEST 发送给 RIL，RIL 转换成对应的 AT 命令后再发给 modem，modem 处理完 AT 命令将返回处理结果给 RIL，RIL 再转发给 RILJ，在 RILJ 上就是通过 RILReceiver 来读取的。这中间，RILJ 和 RIL 之间是通过名为 rild 的 socket 进行通讯的，数据都通过 Parcel 序列化成了字节流数据，在两个不同进程间进行传输；而 RIL 和 modem 之间，则是直接通过往 modem 设备端口写 AT 命令以及从该端口读取返回结果来交互的，由 linux 的 write 和 read 系统调用做保证。

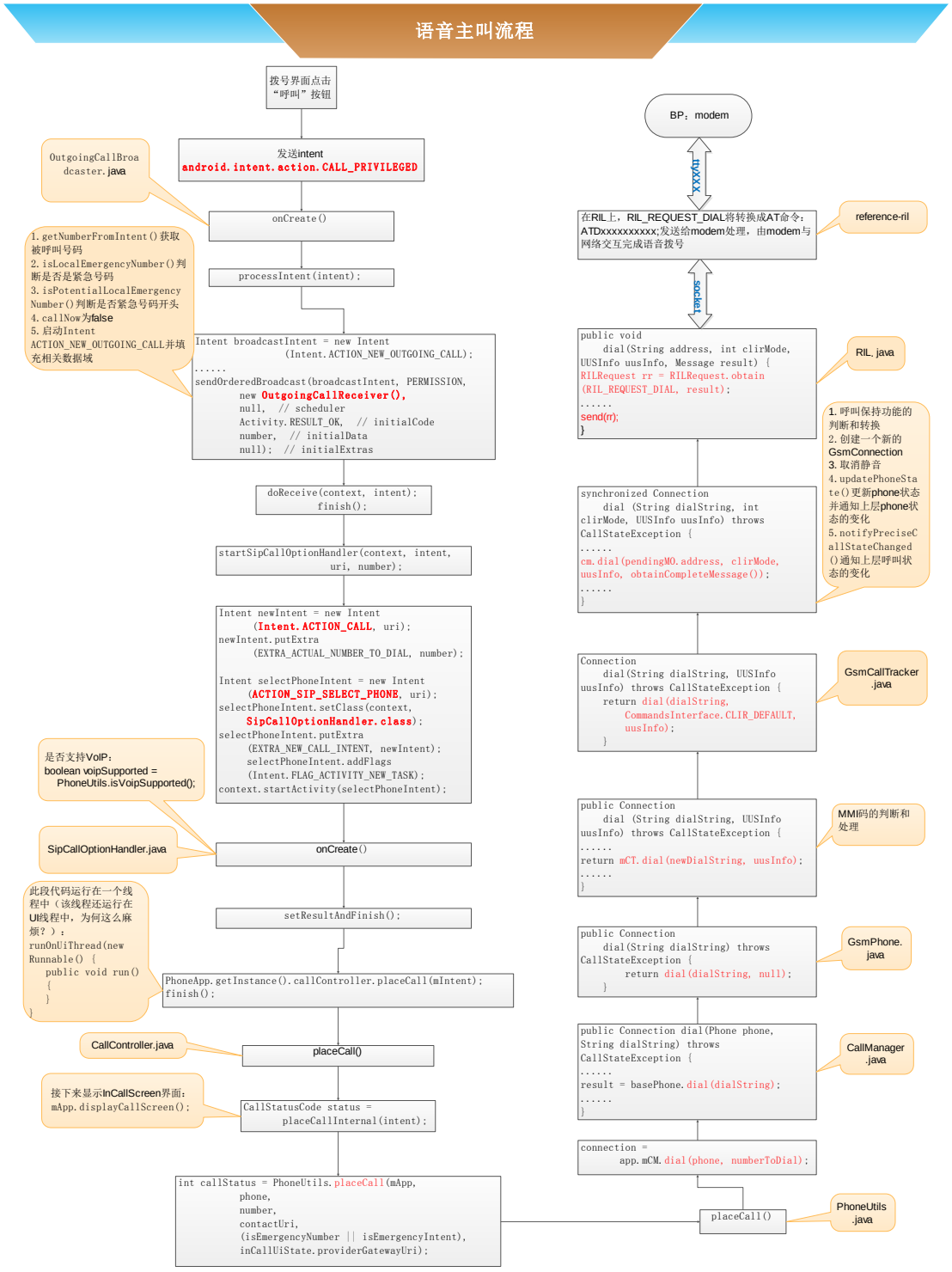
十一、 实例流程分析

首先需要说明以下几点：

(1) 下面的实例流程基本未考虑判断条件对流程的影响，都是按照成功实现相关功能的流程来做的描述，如果考虑判断条件、异常情况的处理，会比较复杂，具体部分请参考源代码。

(2) Android 平台本身是基于 Message 的异步非实时操作系统，所以里面的流程绝大部分都是基于 Message、callback(handler)，register、notify 方式来设计的。也就是说，一个功能的实现，它不一定是以同步的方式来线性执行的。下面的流程图均未描述出异步处理流程，但基本上都在流程图的注释部分做了说明，提醒处理某个事件时，之前在哪个地方对事件进行了注册，处理该事件实际上就是一个回调机制。

1、 语音主叫流程

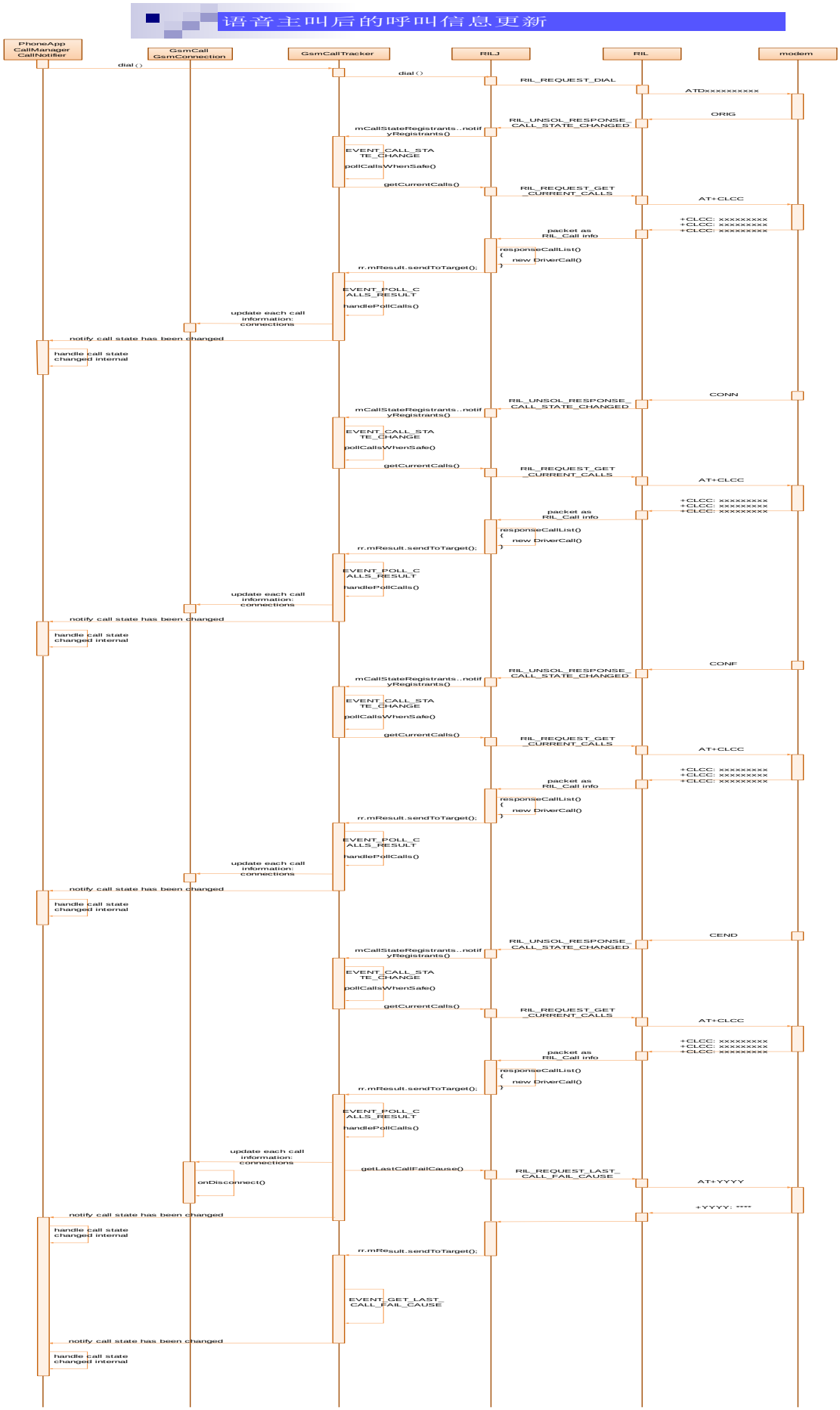


注：流程图中，未对 AT+CLCC 处理过程做描述

在语音通话中，无论去电还是来电，都会涉及到通话状态的更新。通话状态何时更新？如何更新到 UI 层的呢？这必定需要 modem 来告知我们。

Modem 告知通话状态的更新，也是基于 AT 命令的方式。不同制式、不同厂家所使用的 AT 命令会有所不同（如 RING、ORIG、CONN、CEND、NO CARRIER、%ILCC 等等），需要 refrence-ril 来分别解析，但 framework 及应用层不关心具体是哪个 AT 命令，因为 RIL 会将其解析后，转换成 RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED 报告上来。

举个简单的例子：（以 UI 拨打电话后，到对方接听再到挂断）


语音主叫后的呼叫信息更新
PhoneApp CallManager CallNotifier
GsmCall GsmConnection
GsmCallTracker
RILJ
RIL
modem
dial()
dial()
RIL_REQUEST_DIAL
ATDxxxxxxxxx
ORIG
RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED
mCallStateRegistrants..notifyRegistrants()
EVENT_CALL_STATE_CHANGE
pollCallsWhenSafe()
getCurrentCalls()
RIL_REQUEST_GET_CURRENT_CALLS
AT+CLCC
+CLCC: xxxxxxxx
+CLCC: xxxxxxxx
+CLCC: xxxxxxxx
packet as RIL_Call info
responseCallList()
{
new DriverCall()
}
rr.mResult.sendToTarget();
EVENT_POLL_CALLS_RESULT
handlePollCalls()
update each call information: connections
notify call state has been changed
handle call state changed internal
CONN
CONF
CEND
onDisconnect()
getLastCallFailCause()
RIL_REQUEST_LAST_CALL_FAIL_CAUSE
AT+YYYY
+YYYY: ****
EVENT_GET_LAST_CALL_FAIL_CAUSE

(1) UI 上拨打电话后，将调用到 GsmCallTracker 的 dial()接口，在 framework 上将其转换成 RIL_REQUEST_DIAL 请求，并发送给 RIL

(2) RIL 层会转换成 ATD 命令发送给 modem

(3) modem 收到 ATD 后，解析出语音电话号码，向网络侧发起语音呼叫相关信令，并向 RIL 返回当前正在发起呼叫的主动上报 AT 命令 ORIG （各 modem 平台不同，上报的 AT 命令也不同）

(4) RIL 收到始呼主动上报 AT 命令 ORIG 后进行解析，然后转换成 RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED，上报给 framework

(5) framework 收到后，在 processUnsolicited()中进行处理：通过 mCallStateRegistrants 通知到注册过该事件的上层模块

(6) GsmCallTracker 收到该通知后，将由 EVENT_CALL_STATE_CHANGE 来响应：调用 pollCallsWhenSafe()，在 pollCallsWhenSafe()中，将调用 getCurrentCalls()经 RIL 给 modem 发送 AT+CLCC 命令，查询当前所有存在的通话的状态（index、number、call direction、call state、mode 等）。上层各模块始呼状态更新，都是根据该查询结果来获取或判断的（状态更新为 dialing）

(7) （GSM 主叫前提下）当对方收到来电后，modem 将再次通过另外的主动上报 AT 命令 CONN 告知 RIL，重复(4)(5)(6)步骤 （状态更新为 alerting）（注：CDMA 模式下，没有这个状态）

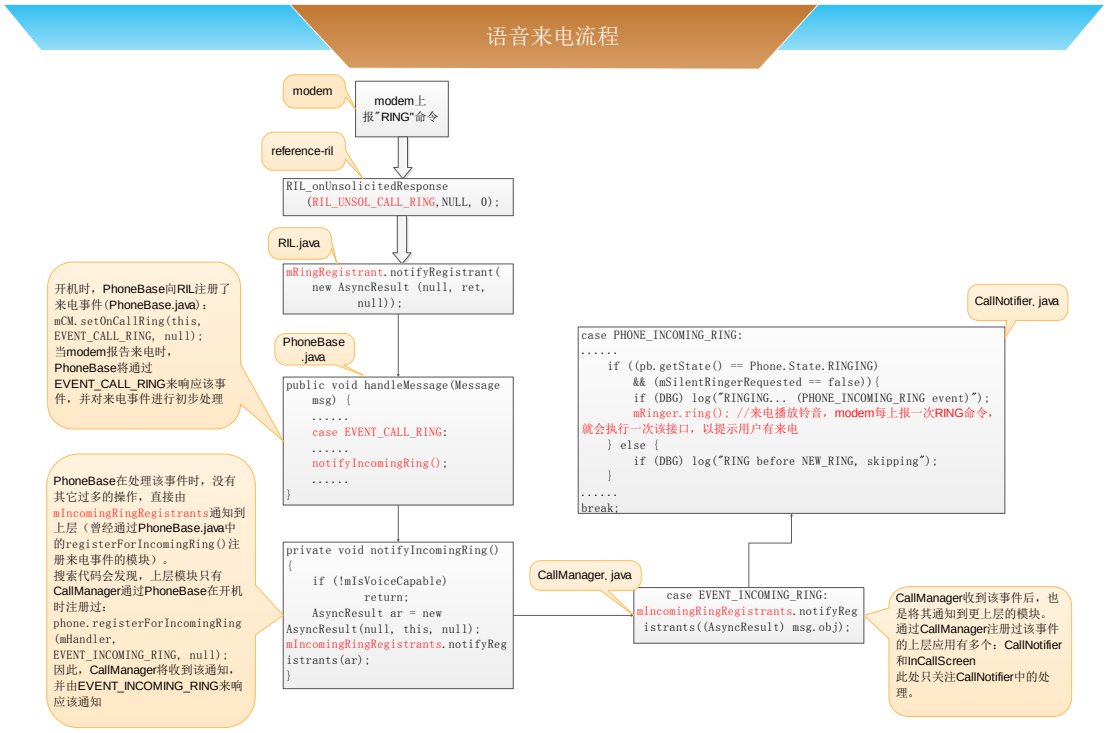
(8) 对方接听电话后，modem 还将通过一个主动上报 AT 命令 CONF 告知 RIL，重复(4)(5)(6)步骤 （状态更新为 active）

(9) 通话挂断后（本地挂断或对方挂断或网络异常端开），modem 将再次通过另外的主动上报 AT 命令 CEND 告知 RIL，重复(4)(5)(6)步骤 （状态更新为 idle）

(10) GsmCallTracker 收到语音挂断信息更新后，会再调用 getLastCallFailCause()来向 modem 查询电话挂断的原因，并在收到 modem 返回结果后，由 EVENT_GET_LAST_CALL_FAIL_CAUSE 来响应（通知到应用层）

由上面的过程可以知道，UI 上拨打电话后，从这个需求被传递给 modem 开始，以后的流程都将由 modem 来主导，AP 侧只是被动的接收并处理信息的更新。后续所有呼叫状态的更新，都由 modem 主动上报开始（告知有 call 状态发生了更新），framework 在收到后，并不立即判断谁发生了状态更新、更新到了什么状态，而是先通过 pollCallsWhenSafe()来向 modem 发送 AT+CLCC 来查询当前所有的通话，该命令获取到的信息是完整而全面的（当前存在的所有通话，无论主叫还是被叫，处于何状态，都会通过这个 AT 命令获取到），然后将查询到的结果更新到 GsmCall、GsmConnection 的各个状态中去。根据需要，在 handlePollCalls()的适当地方将通过 notify 的方式，将相关信息通知到（曾经注册过这些事件的）应用层，让 UI 做出对应的更新。

2、 语音来电流程



注：流程图中，未对 AT+CLCC 处理过程做描述

语音来电与去电有相似之处，即 AT+CLCC 查询当前手机所有通话信息的过程。查询出的结果，都是在 handlePollCalls()中来处理的。

在 handlePollCalls()中，针对通话信息的更新，会有不同的 notify 来通知上层模块：

(1) phone.notifyNewRingingConnection(newRinging); ➔ 将新的来电事件（包括 incoming call 和 waiting call）通知到上层。CallNotifier 收到该通知后，将由 PHONE_NEW_RINGING_CONNECTION 来响应。

(2) updatePhoneState; ➔ 将对 state 全局变量赋值（该变量维护着 framework 层 phone 当前的状态：IDLE 、RINGING、OFFHOOK）。该接口中，根据 phone 最近两次的状态的不同，会有 3 类不同的信息需要通知到上层：

a. notify call ended event:

```
if (state == Phone.State.IDLE && oldState != state) {
    voiceCallEndedRegistrants.notifyRegistrants(
        new AsyncResult(null, null, null)); ➔ 语音通话挂断后，通知相应模块，当前有通话结束。
}
```

b. notify call started event:

```
else if (oldState == Phone.State.IDLE && oldState != state) {
    voiceCallStartedRegistrants.notifyRegistrants (
        new AsyncResult(null, null, null)); ➔ 有语音通话启动，通知相应的模块
}
```

c. notify phone state changed:

```
if (state != oldState) {
    phone.notifyPhoneStateChanged(); ➔ 相邻两次 phone 状态不一致，通知上层 phone 状
```

态发生了变化 }

(3) phone.notifyUnknownConnection(); ➔ 未知的 connection

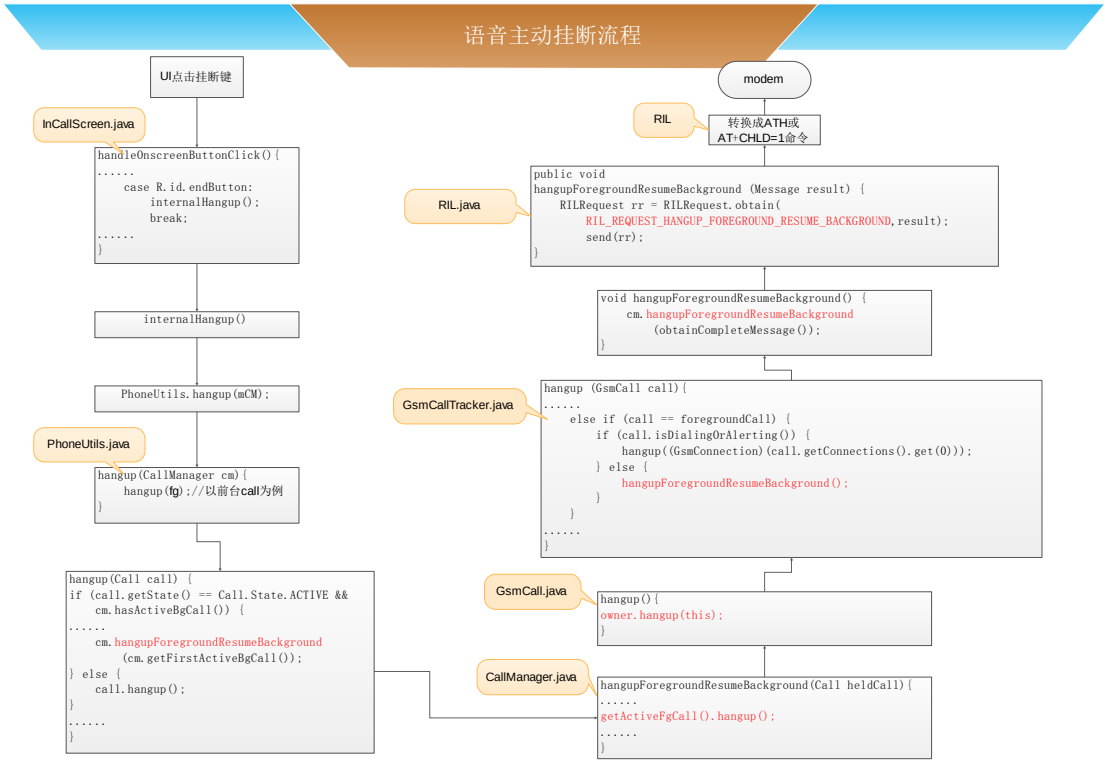
(4) phone.notifyPreciseCallStateChanged(); ➔ notify precise call state changed. 接收该 notify 的模块及响应该 notify 的事件为：

模块	（收到 notify 时的）回调事件	响应事件的处理接口
CallManager.java	EVENT_PRECISE_CALL_STATE_CHANGED	mPreciseCallStateRegistrants.notifyRegistrants()
BluetoothHandsfree.java	PRECISE_CALL_STATE_CHANGED	handlePreciseCallStateChange()
CallNotifier.java	PHONE_STATE_CHANGED	onPhoneStateChanged()
InCallScreen.java	PHONE_STATE_CHANGED	onPhoneStateChanged()
PhoneUtils.java	PHONE_STATE_CHANGED	handleMessage()

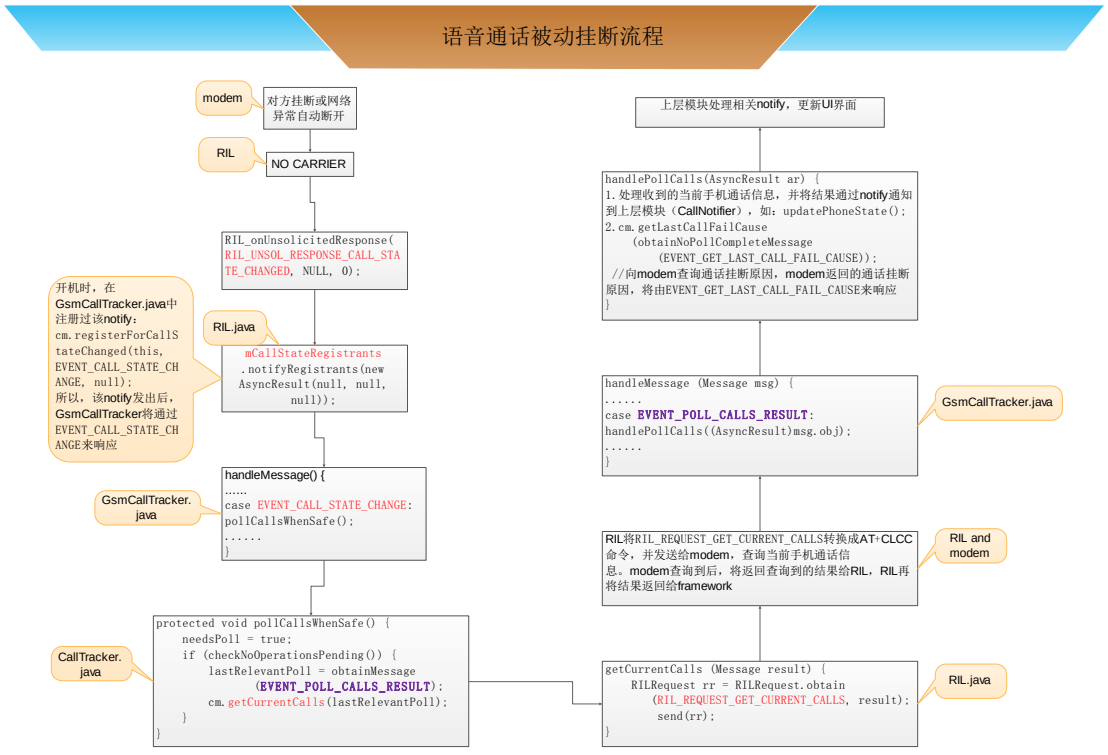
在上面的几个模块中，CallManager.java 模块实际上是起到承上启下的作用，一方面，它为应用层模块提供注册 precise call state changed 的接口 registerForPreciseCallStateChanged()，另外一方面，它自己又会向 PhoneBase.java 注册 precise call state changed。这样，GsmCallTracker 中调用 notifyPreciseCallStateChanged()来通知 precise call state changed 后，CallManager 会接收到（且也只有它会接收到，此时应用层模块并未接收到），CallManager 收到后，并不负责对事件的具体处理，而是直接调用 mPreciseCallStateRegistrants.notifyRegistrants()，将其通知到应用层模块（应用层模块是通过 CallManager 间接注册了 precise call state changed），具体工作由应用层来进行。

所以，当来电时，CallNotifier 会先后收到事件 PHONE_INCOMING_RING、PHONE_NEW_RINGING_CONNECTION 和 PHONE_STATE_CHANGED。其中，PHONE_INCOMING_RING 只会在 incoming call 时收到；PHONE_NEW_RINGING_CONNECTION 会在 incoming call 和 waiting call 时收到；PHONE_STATE_CHANGED 则在 phone 状态发生变化时收到（phone 的状态变化由 GsmCallTracker 的 updatePhoneState()维护）。

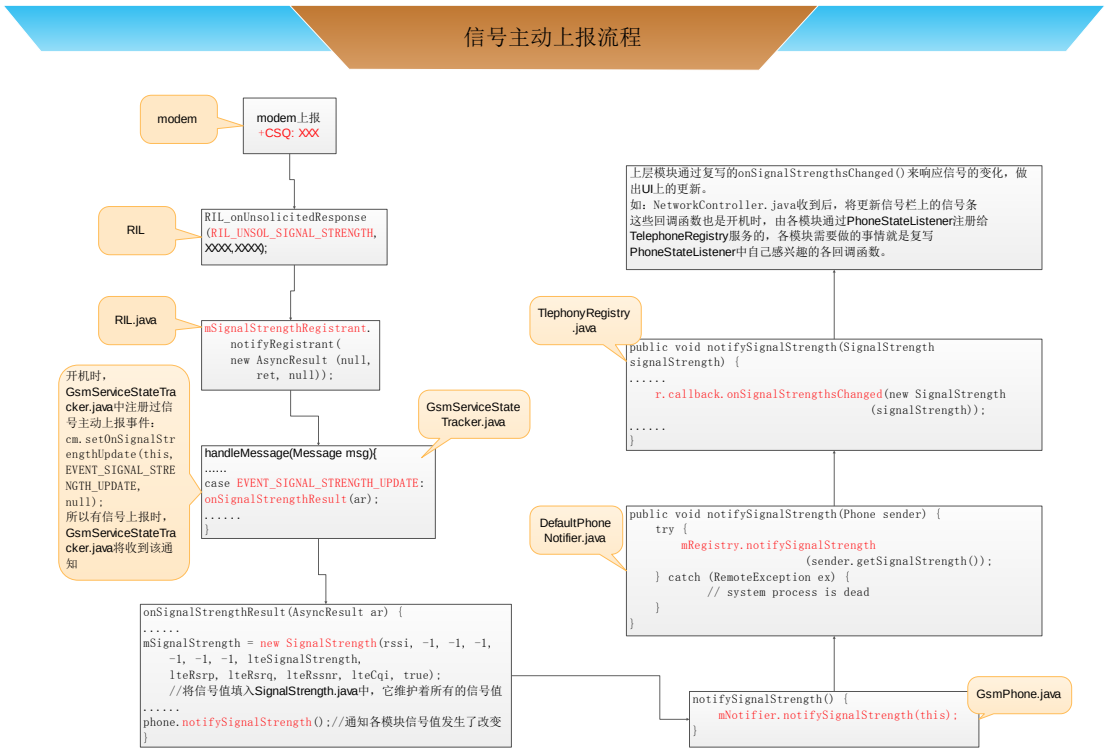
3、 语音通话主动挂断流程



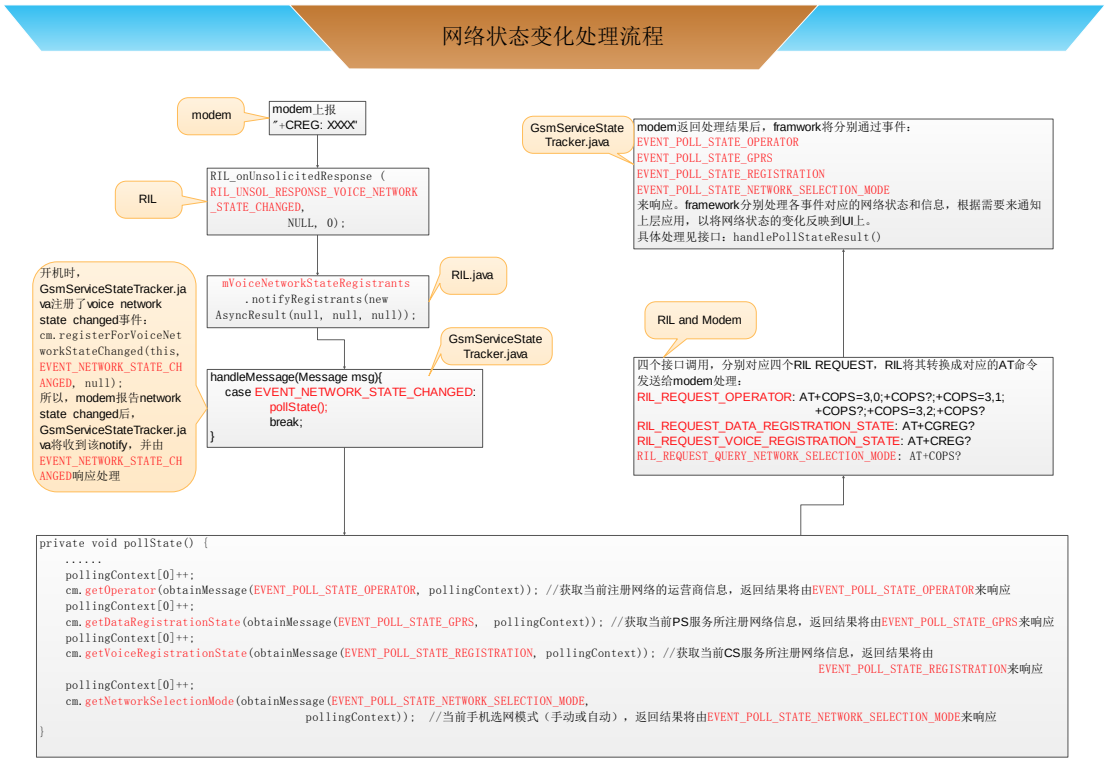
4、语音通话被动挂断流程



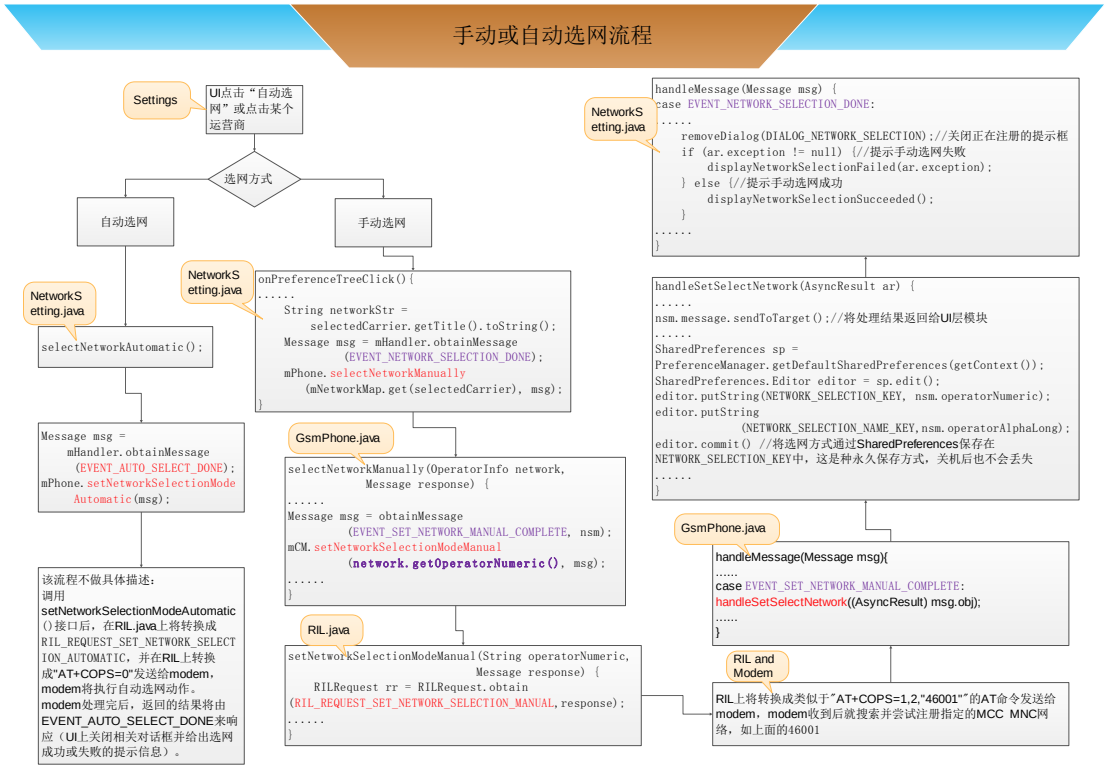
5、信号上报流程



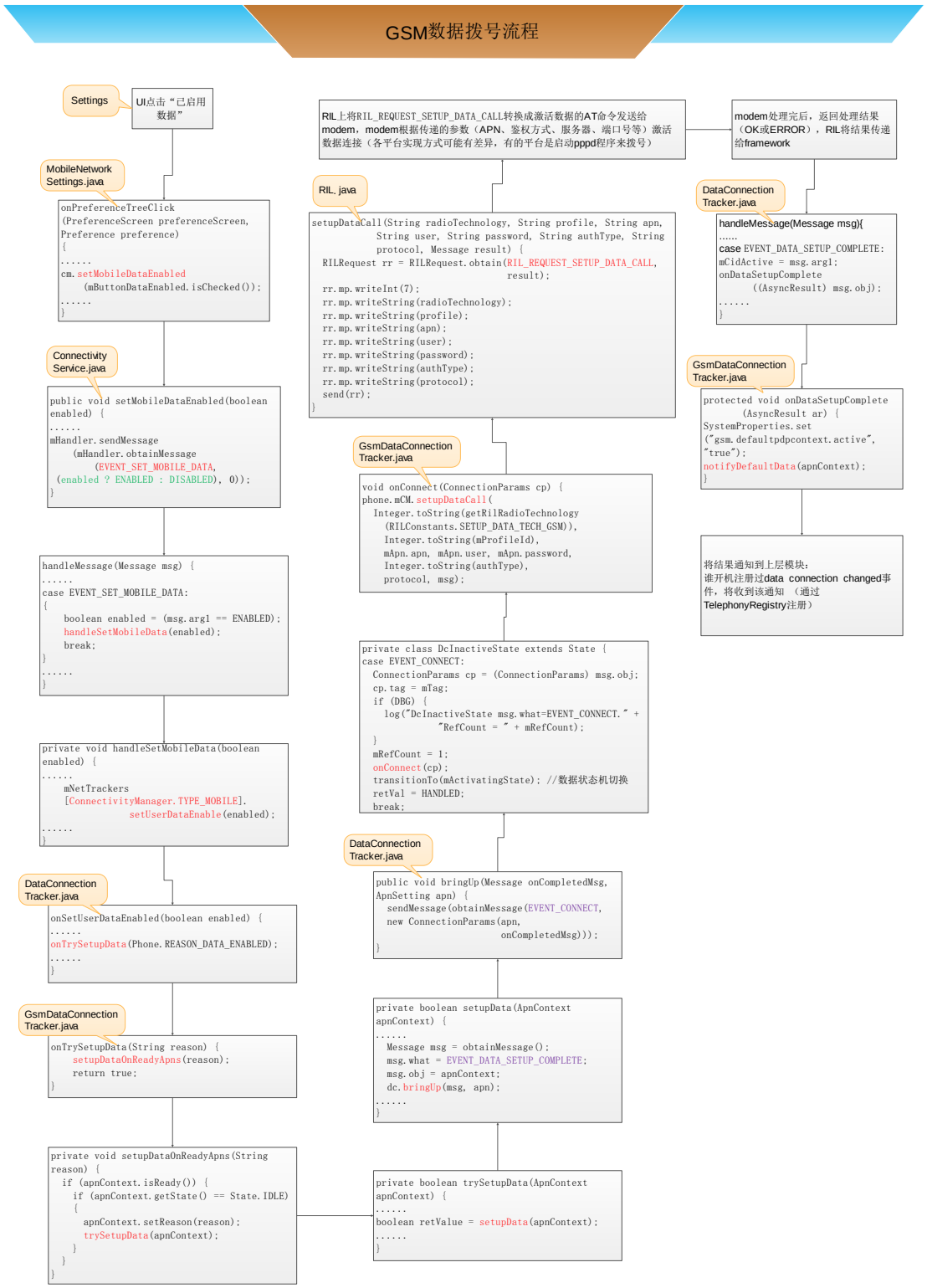
6、网络状态变化处理流程



7、手动或自动选网流程


手动或自动选网流程
Settings
UI点击“自动选网”或点击某个运营商
选网方式
自动选网
手动选网
NetworkSetting.java
selectNetworkAutomatic();
Message msg = mHandler.obtainMessage (EVENT_AUTO_SELECT_DONE); mPhone.setNetworkSelectionModeAutomatic(msg);
该流程不做具体描述：调用setNetworkSelectionModeAutomatic()接口后，在RIL.java上将转换成RIL_REQUEST_SET_NETWORK_SELECTION_AUTOMATIC，并在RIL上转换成"AT+COPS=0"发送给modem，modem将执行自动选网动作。modem处理完后，返回的结果将由EVENT_AUTO_SELECT_DONE来响应（UI上关闭相关对话框并给出选网成功或失败的提示信息）。
NetworkSetting.java
onPreferenceTreeClick(){ ..... String networkStr = selectedCarrier.getTitle().toString(); Message msg = mHandler.obtainMessage (EVENT_NETWORK_SELECTION_DONE); mPhone.selectNetworkManually (mNetworkMap.get(selectedCarrier), msg); }
GsmPhone.java
selectNetworkManually(OperatorInfo network, Message response) { ..... Message msg = obtainMessage (EVENT_SET_NETWORK_MANUAL_COMPLETE, nsm); mCM.setNetworkSelectionModeManual (network.getOperatorNumeric(), msg); ..... }
RIL.java
setNetworkSelectionModeManual(String operatorNumeric, Message response) { RILRequest rr = RILRequest.obtain (RIL_REQUEST_SET_NETWORK_SELECTION_MANUAL,response); ..... }
RIL and Modem
RIL上将转换成类似于"AT+COPS=1,2,"46001""的AT命令发送给modem，modem收到后就搜索并尝试注册指定的MCC MNC网络，如上面的46001
GsmPhone.java
handleMessage(Message msg){ ..... case EVENT_SET_NETWORK_MANUAL_COMPLETE: handleSetSelectNetwork((AsyncResult) msg.obj); ..... }
handleSetSelectNetwork(AsyncResult ar) { ..... nsm.message.sendToTarget();//将处理结果返回给UI层模块 ..... SharedPreferences sp = PreferenceManager.getDefaultSharedPreferences(getContext()); SharedPreferences.Editor editor = sp.edit(); editor.putString(NETWORK_SELECTION_KEY, nsm.operatorNumeric); editor.putString (NETWORK_SELECTION_NAME_KEY,nsm.operatorAlphaLong); editor.commit() //将选网方式通过SharedPreferences保存在NETWORK_SELECTION_KEY中，这是种永久保存方式，关机后也不会丢失 ..... }
NetworkSetting.java
handleMessage(Message msg) { case EVENT_NETWORK_SELECTION_DONE: ..... removeDialog(DIALOG_NETWORK_SELECTION);//关闭正在注册的提示框 if (ar.exception != null) {//提示手动选网失败 displayNetworkSelectionFailed(ar.exception); } else {//提示手动选网成功 displayNetworkSelectionSucceeded(); } ..... }

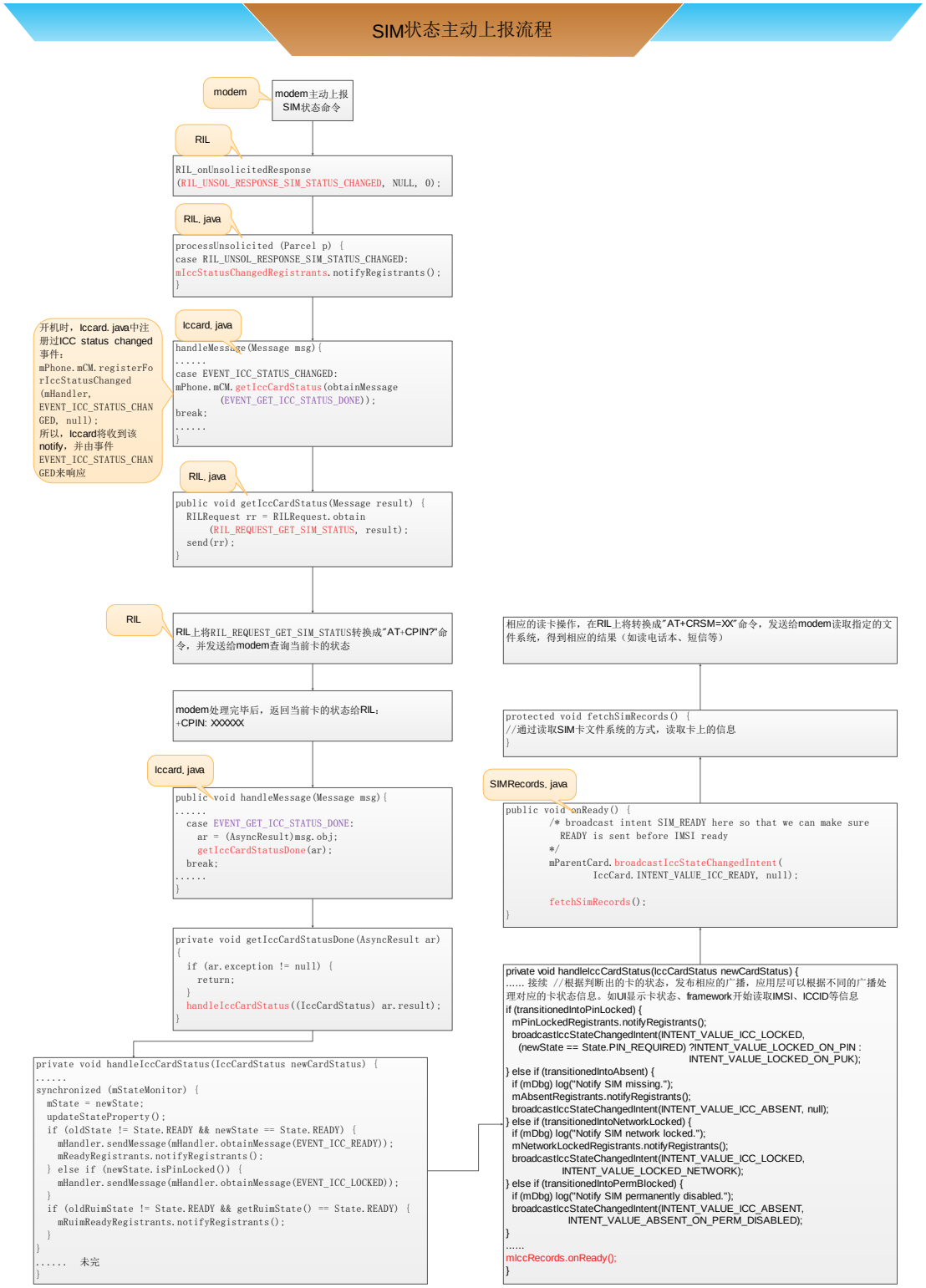
8、GSM 数据拨号流程



数据拨号流程中，会维护一个状态机，当 UI 开启数据开关到拨号成功的过程中，依次会经历以下状态转换：DcInactiveState → DcActivatingState → DcActiveState 而关闭数据开关到数据成功挂断，依次将经历以下数据状态转换：DcActiveState → DcDisconnectingState → DcInactiveState

数据的挂断流程，与上面相似，流程从 onSetUserDataEnabled()接口开始流程发生变化，数据挂断会调用 onCleanUpAllConnections(Phone.REASON_DATA_DISABLED)来执行断开所有数据连接的动作，具体流程不再作分析。

9、 SIM 卡状态主动上报流程



10、 短信发送流程

11、 短信接收流程

十二、 开发中的经验教训

1、 getGsmPhone

getGsmPhone()是 PhoneFactory 类的一个接口，从该接口实现：

```
public static Phone getGsmPhone() {
    synchronized(PhoneProxy.lockForRadioTechnologyChange) {
        Phone phone = new GSMPhone(sContext, sCommandsInterface, sPhoneNotifier);
        return phone;
    }
}
```

可以看出，调用该接口，将新创建一个 GSMPhone，所以，无论对于上层应用还是 telephonyframework 来讲，都一定要慎重调用该接口。如果你所做的产品是单卡项目，那么一定不会涉及到调用该接口（虽然从代码实现上你可以轻易的调用到），一旦调用，手机中将创建出两个 GSMPhone，你手机将发生意想不到的各种意外（如操作一次手机通讯相关功能，会发现该功能不间断的不停的在循环自动执行，这是由于两个 GSMPhone 交替处理 EVENT 导致的）。从 Android 平台代码实现看，整个代码中，调用 getGsmPhone()的地方只有一个，那就是出现在 Phone 切换时，CDMAPhone 切换到 GSMPhone。

2、 RIL 中处理主动上报命令流程中发送 AT 命令

在 RIL 上，处理主动上报 AT 命令 Unsolicited cmd 的线程中，一定不要直接调用下述任何一个接口来执行发送 AT 命令的动作：

at_send_command()
at_send_command_singleline()
at_send_command_numeric()
at_send_command_multiline()
at_send_command_sms()

因为处理 Unsolicited cmd 的线程与发送 AT 命令的线程，在 RIL 上是两个不同的线程，它们各自的工作是不可以交叉的，各有各的职责。虽然在处理 Unsolicited cmd 的线程中能够直接调用上面的接口来发送 AT 命令（语法上不会出错），但会无规律的出现 RIL 空指针的现象（at_send_command_full_nolock()接口中）。因为时机恰巧时，会发生两个线程同时操作该接口，导致该接口被重入，在其中一个线程中出现全局变量为 NULL 的情况（s_commandmutex 锁只能锁住发送 AT 命令线程中的 AT 处理流程，逐个逐个的进入该接口处理，但它对 Unsolicited cmd 线程起不到任何作用）。

3、 AT+COPS=?的超时设置

根据开发经验，各 modem 平台，在所有需要处理的 AT 命令中，AT+COPS=?是耗时最长的，所以 RIL 上的 AT 超时设置值一定要大于它执行的时间（各平台不同，该命令执行时间也有差异，一般 60s 差不多）。

4、 Android 侧 ro.ril.wake_lock_timeout 的设置

ro.ril.wake_lock_timeout 是 RILJ 上设置的超时时间，RILJ 上也维护着 RIL REQUEST

队列，它把 request 发送给 RIL 后，也在等待 RIL 返回处理结果，所以它也有个超时配置。它的作用是：保证在超时时间到达之前，telephony framework 不会休眠，以保证 RIL 返回处理结果时，RILJ 能够顺利的接收到。该配置值一定要大于 RIL 上设置的 AT 命令最长超时时间，否则，会出现 RIL 返回结果给 RILJ 时，RILJ 却休眠了，导致结果丢失的现象。

5、 插入中国移动卡，手动选择中国联通网络，关机后重新开机，手机无法注册任何一个网络，包括中国移动网络

这是 Android 平台默认的处理结果。当进行了手动选网后，会在 handleSetSelectNetwork() 中，通过 SharedPreferences 将选网方式记忆下来，并在开机时，在 GsmServiceStateTracker.java 中判断该方式，根据 AP 的配置，可能将该选网方式在开机时发送给 modem：

```
case EVENT_SIM_READY:
    // Set the network type, in case the radio does not restore it.
    cm.setCurrentPreferredNetworkType();

    // The SIM is now ready i.e if it was locked
    // it has been unlocked. At this stage, the radio is already
    // powered on.
    if (mNeedToRegForSimLoaded) {
        phone.mIccRecords.registerForRecordsLoaded(this,
                EVENT_SIM_RECORDS_LOADED, null);
        mNeedToRegForSimLoaded = false;
    }

    boolean skipRestoringSelection = phone.getContext().getResources().getBoolean(
                com.android.internal.R.bool.skip_restoring_network_selection);
    //查询 AP 配置，该配置用来标识是否需要开机恢复上次设置的选网模式。Android 默认配置是 false，也就是需要向 modem 下发上次的选网模式，以让 modem 根据上次选网模式进行网络的搜索和注册
    if (!skipRestoringSelection) {
        // restore the previous network selection.
        log("skipRestoringSelection=" + skipRestoringSelection + ", then restore saved network selection.");
        phone.restoreSavedNetworkSelection(null); //恢复上次选网模式
    }
    pollState();
    // Signal strength polling stops when radio is off
    queueNextSignalStrengthPoll();
    break;
```

skip_restoring_network_selection 的配置位于下面路径的文件中：
framework/base/core/res/res/values/config.xml

十三、Android 原生态 bug 汇总

在开发过程中，陆续发现了几个 Android 原生态的 bug（部分是 Android 设计的流程，

可能不一定是 bug，就看个人的见解了）

1. 拨号盘输入*43#开启呼叫等待功能失败 (Android 4.1)

原因是：Android 平台将功能请求类型设置为了 SERVICE_CLASS_NONE，导致发送的 AT 命令中，参数传递错误，无法开启语音呼叫等待功能。（见文件 GsmMmiCode.java 文件 processCode()，在调用 siToServiceClass()后，没有 43 码的处理，返回了 SERVICE_CLASS_NONE）

解决办法：当判断出请求类型都不匹配时，将类型设置为 SERVICE_CLASS_VOICE，而不是 SERVICE_CLASS_NONE

2. 成功注册了网络，但 UI 下拉菜单显示“无服务”

原因是：SIMRecords.java 中 getDisplayRule()中的 PLMN 显示判断条件不正确

解决方法：将 spn==null 的判断条件修改为 TextUtils.isEmpty(spn)

3. 进行下面操作步骤后，手机不注册网络：
   (1)插入中国联通卡
   (2)进行手动选网操作---选择“中国联通”
   (3)关机，取出联通卡，再插入中国移动卡
   (4)开机后，发现无法注册移动网络

原因是：Android 原平台上，设置手动搜网后，对以后开机都是生效的（这是 Android 的设计处理流程，不能称之为 bug，就看我们的需求是什么）。以后开机都会根据上次手动选网指定的网络（MCC MNC）来注册，如果注册不上，则手机无法使用 （对普通用户来说这种默认处理方式是不合理的），但状态栏会弹出提示信息，大致内容是所选择的 XXXX 网络无法注册。

解决办法：去掉原平台的这种默认处理。通过将/frameworks/base/core/res/res/config.xml 中 skip_restoring_network_selection 设置为 true，这样开机搜网方式就由 modem 来决定，AP 不会再指定特定的网络 (modem 侧一般都是自动搜网，但也有可能 modem 也是记忆上次的搜网方式，下次开机以此方式来搜网)

4. 数据包发送后无响应，当发送包连续达到门限值---10 个时，modem 将被重置，引起 UI 显示丢网及弹出 STK 菜单等问题

原因是：经从 kernel 底层调查，发现是由于 AP 侧发送了 IPV6 数据包， modem 侧默认关闭了 IPV6 功能，导致这类数据包发送后，AP 侧收不到任何应答，但 AP framework 会进行计数，当发现连续发送了 10 个包，中间未收到任何包，framework 将会重置 modem （类似于一次自动的开关飞行模式动作）。见文件 GsmDataConnectionTracker.java 中的调用流程：onDataStallAlarm() → sendMessage(obtainMessage(EVENT_DO_RECOVERY)) → case EVENT_DO_RECOVERY → doRecovery()

解决办法：由于 IPV6 功能当前没有测试条件且 Icera modem 也关闭了 IPV6 功能，故 AP 侧也采取关闭 IPV6 功能来解决该问题。通过在 init.rc 中添加下面内容：

```
write /proc/sys/net/ipv6/conf/all/disable_ipv6 1
```

5. 三方通话挂断时，界面显示通话时长错误。操作步骤：
   (1)A 与 B 通话 10s
   (2)配合机 C 给 A 拨打电话

(3)A 收到 C 的来电后不接听，等待 30s

(4)配合机 C 挂断

(5)观察测试机 A 界面与 C 通话时长显示

测试结果：A 界面显示与 C 通话时长 10s

预期结果：A 界面上不显示与 C 通话时长，因为来电 C 根本没有被 A 接听。

原因是：这是 Android 原生态的一个 bug，在处理界面通话时长时，测试机 A 对处于 waiting 状态的 call 没有及时去获取通话时长信息，导致界面显示刚收到来电时的时长。

解决方案：在 CallTime.java 的 updateCallStateWidgets()中添加对 WAITING 状态 call 的通话时长处理，同时修改 updateElaspedTimeWidget()中的显示判断，通话时长为 0 时，界面显示空信息。

6. 2G 网络下，通话过程中，关闭数据连接，当通话挂断后，发现数据开关已关闭，但数据一直不断开，能正常浏览网页

原因是：Android 平台设计流程上，在 2G 网络下，默认语音和数据是不可以并发的（见接口 isConCurrentVoiceAndDataAllowed()的实现）。所以在通话中，关闭数据连接，isDataAllowed()会返回 false，framework 不会将断开数据连接的请求发送给 modem，通话挂断后，也没有再向 modem 发送该请求，所以界面上看，数据开关已关闭，但实际上，modem 侧还一直连接着。

但通话中，开启数据连接，通话挂断后，数据连接能够正常建立。这是由于数据拨号有个重试机制，虽然通话中不能进行数据拨号，但通话挂断后，它会尝试重拨，把数据连接建立起来。

解决方案：可以考虑从 onVoiceCallEnded()中做修改。通话挂掉时，判断如果数据依旧处于连接状态，则再发送数据挂断请求给 modem，让 modem 断开数据连接。