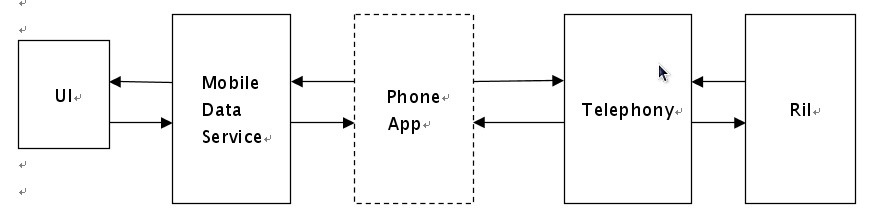
# Android数据业务

关键词：Android，GPRS, APN, Service，Client, Connection, Tracke

数据连接的整体框架简单如下图所示：



## 第一部分：UI部分

用户发起数据流程的连接或断开，最终都是要将结果反馈给用户，让用户明白当前的数据连接的状态。这部分一个主要的对象是Settings对象(com.android.phone.Settings)，向ConnectivityManager发起数据连接请求。

**1. Settings.java-> onPreferenceTreeClick():**

public boolean onPreferenceTreeClick(…)

{

……

if (preference == mButtonDataEnabled)

{

ConnectivityManager cm =

(ConnectivityManager)getSystemService(Context.CONNECTIVITY\_SERVICE);

if (TelephonyManager.getPhoneCount() > 1)

{

cm.setMobileDataEnabledByPhoneId(mSubId, mButtonDataEnabled.isChecked()); //🡺**2**

}

else

{

cm.setMobileDataEnabled(mButtonDataEnabled.isChecked());

}

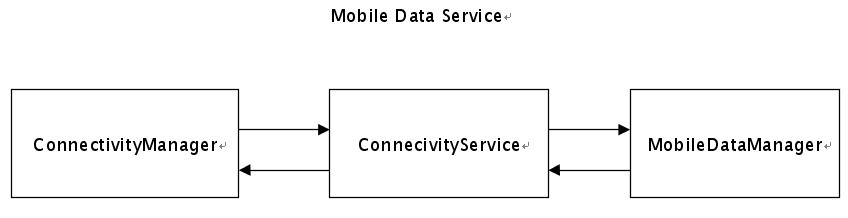
return true;

}

……

}

## 第二部分：Mobile Data Service部分



这部分是相当于数据业务的中转站了，向上响应用户的请求，向下向telephony派发用户请求并处理来自telephony的状态变化。这里面很重要的对象，包括：

（1）ConnectivityService：ConnectivityManager是它的客户端，两者通过著名的binder机制进行通信，调用ConnectivityManager这里等同于调用到ConnectivityService。

（2）MobileDataStateTracker，这是移动数据业务的"追踪器"，又向telephony进行交互，它的对象组合在 ConnectivityService对象里面，当然ConnectivityService对象还管理着其它的数据连接类型，包括 WifiStateTracker。

**2. ConnectivityManager.java-> setMobileDataEnabledByPhoneId():**

public void setMobileDataEnabledByPhoneId(int phoneId, boolean enabled)

{

try{

mService.setMobileDataEnabledByPhoneId(phoneId, enabled); //🡺**3**

} catch (RemoteException e){

}

}

**3. ConnectivityService.java-> setMobileDataEnabledByPhoneId():**

*private NetworkStateTracker mNetTrackers[];*

public synchronized void setMobileDataEnabledByPhoneId(int phoneId, boolean enabled)

{

……

int defaultDataPhoneId = TelephonyManager.getDefaultDataPhoneId(mContext);

if (enabled)

{

if (phoneId == defaultDataPhoneId)

{

if (mNetTrackers[ConnectivityManager.TYPE\_MOBILE] != null)

{

mNetTrackers[ConnectivityManager.TYPE\_MOBILE].reconnect(); //🡺**4**

}

}

}

else

{

……

}

}

由于MobileDataStateTracker extends NetworkStateTracker，所以这里的mNetTrackers是一个MobileDataStateTracke，包含各种移动数据连接，包裹MMS，SUPL,DUN。

**4. MobileDataStateTracker.java-> reconnect():**

public boolean reconnect()

{

……

switch (setEnableApn(mApnType, true)) //🡺**5**

{

……

}

}

**5. MobileDataStateTracker.java-> setEnableApn ():**

private int setEnableApn(String apnType, boolean enable)

{

getPhoneService(false); //🡺**6**

……

try {

if (enable)

{

return mPhoneService.enableApnType(apnType); //🡺**7**

}

else

{

return mPhoneService.disableApnType(apnType);

}

} catch (RemoteException e) {

if (retry == 0) getPhoneService(true);

}

}

**6. MobileDataStateTracker.java-> getPhoneService ():**

mPhoneService是电话的服务的客户端，它的server端实际上是PhoneInterfaceManager对象(com.android.phone.PhoneInterfaceManager)，MobileDataStateTracker通过如下方式调用获取ITelephony接口的服务端。

private void getPhoneService(boolean forceRefresh)

{

mPhoneService = ITelephony.Stub.asInterface(ServiceManager.getService(PhoneFactory

.getServiceName("phone", getPhoneId())));

}

**7. PhoneInterfaceManager.java-> enableApnType():**

public int enableApnType(String type)

{

enforceModifyPermission();

return mPhone.enableApnType(type); //🡺**8**

}

其中，mPhone是PhoneProxy对象。这样，就将连接apn的请求发送到telephony框架层下去了。apn在设置应用里面有里面指定了，一般在工程目录下的system/etc/apns-conf.xml文件。

## 第三部分：Telephony部分

**8. PhoneProxy.java-> enableApnType():**

public int enableApnType(String type)

{

return mActivePhone.enableApnType(type); //🡺**9**

}

其中，mActivePhone是GSMPhone或者CDMAPhone的上溯接口PhoneBase对象。

**9.PhoneBase.java-> enableApnType():**

public DataConnectionTracker mDataConnection;

public int enableApnType(String type)

{

return mDataConnection.enableApnType(type); //🡺**10**

}

**10. DataConnectionTracker.java-> enableApnType():**

public synchronized int enableApnType(String type)

{

……

setEnabled(id, true); //🡺**11**

……

}

**11. DataConnectionTracker.java-> setEnabled ():**

protected void setEnabled(int id, boolean enable)

{

Message msg = obtainMessage(EVENT\_ENABLE\_NEW\_APN);

msg.arg1 = id;

msg.arg2 = (enable ? ENABLED : DISABLED);

sendMessage(msg); //🡺**12**

}

**12. DataConnectionTracker.java-> handleMessage ():**

public void handleMessage (Message msg)

{

switch (msg.what)

{

case EVENT\_ENABLE\_NEW\_APN:

onEnableApn(msg.arg1, msg.arg2); //🡺**13**

break;

……

}

}

**13. DataConnectionTracker.java-> onEnableApn():**

protected synchronized void onEnableApn(int apnId, int enabled)

{

……

onEnableNewApn(); //🡺**14**

……

}

onEnableNewApn()方法在DataConnectionTracker的派生类GsmDataConnectionTracker和CdmaDataConnectionTracker中实现，从而区别不同类型PHONE的数据连接流程。以GSM为例。

**14. GsmDataConnectionTracker.java->onEnableNewApn():**

protected void onEnableNewApn()

{

……

cleanUpConnection(true, Phone.REASON\_APN\_SWITCHED); //🡺**15**

……

}

**15. GsmDataConnectionTracker.java-> cleanUpConnection ():**

protected ArrayList<DataConnection> pdpList;

protected void cleanUpConnection(boolean tearDown, String reason)

{

……

for (DataConnection conn : pdpList)

{

……

conn.disconnect(obtainMessage(EVENT\_DISCONNECT\_DONE, reason)); //🡺**16**

……

}

……

}

conn是DataConnection对象，标识一种数据连接，可以看出这里实际上实现了一个数据连接的状态机。在DataConnection对象里面数据连接的状态分为以下几种：

DcDefaultState，默认状态。

DcInactiveState，非激活状态。

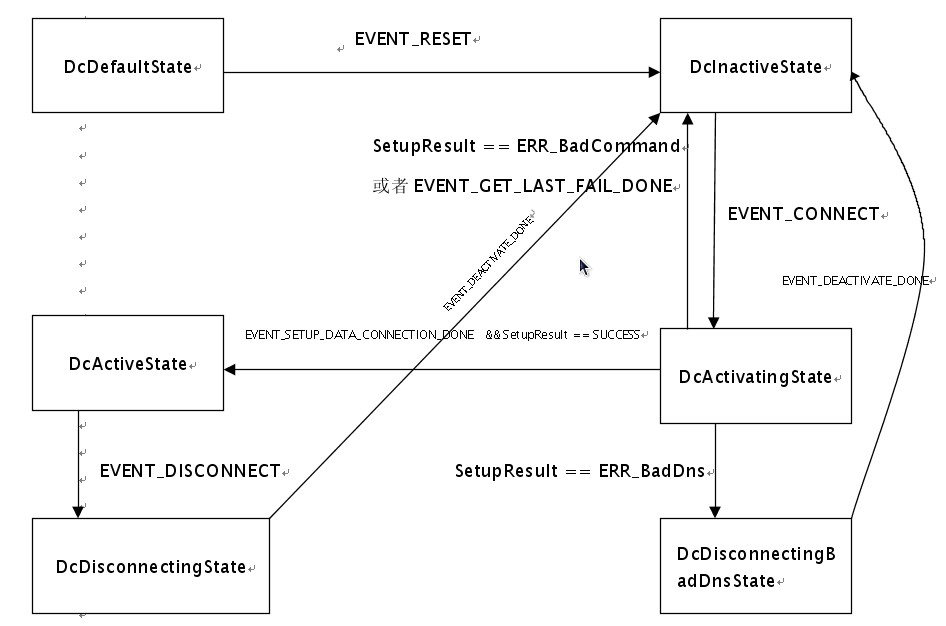
DcActivatingState，正在激活状态

DcActiveState，激活状态

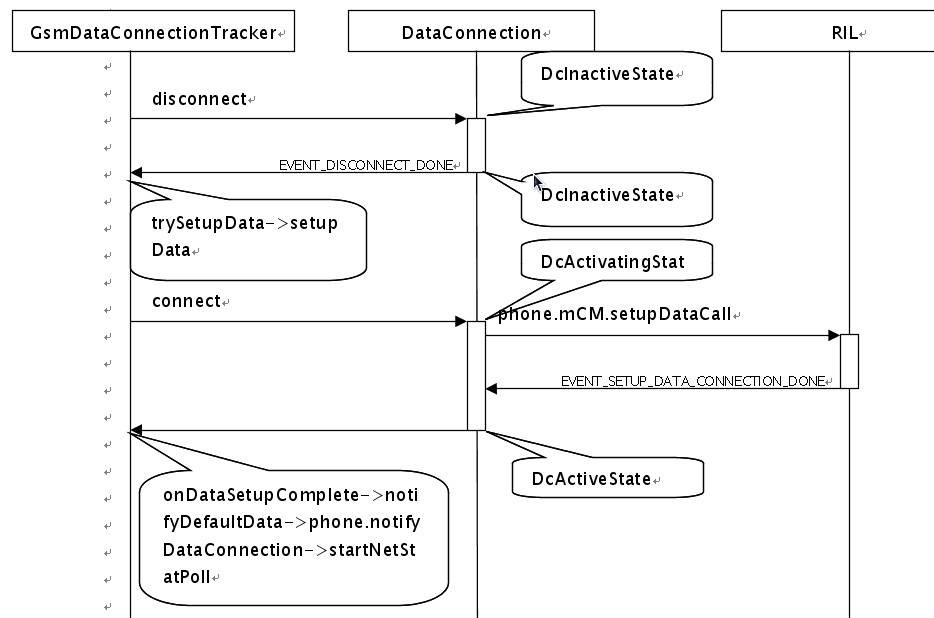
DcDisconnectingState，正在断开状态

DcDisconnectingBadDnsState，断开状态（因为错误的DNS）

状态转换如下图所示：



而动态图如下图所示：



数据连接包含几种状态，构造函数里面必须添加这几种状态，并且指定一个初始状态，根据DataConnection类的构造函数：

**DataConnection.java-> DataConnection():**

protected DataConnection(PhoneBase phone, String name)

{

……

clearSettings();

setDbg(false);

addState(mDefaultState);

addState(mInactiveState, mDefaultState);

addState(mActivatingState, mDefaultState);

addState(mActiveState, mDefaultState);

addState(mDisconnectingState, mDefaultState);

addState(mDisconnectingBadDnsState, mDefaultState);

setInitialState(mInactiveState); //🡺

}

设置初始化状态为mInactiveState，意思是未激活的连接。

*private DcInactiveState mInactiveState = new DcInactiveState();*

所有的状态类都是继承自HierarchicalState，例如：

*private class DcInactiveState extends HierarchicalState*

**16. DataConnectionTracker.java-> handleMessage():**

public void handleMessage (Message msg)

{

switch (msg.what)

{

……

case EVENT\_DISCONNECT\_DONE:

onDisconnectDone((AsyncResult) msg.obj); //🡺**17**

break;

……

}

}

**17. GsmDataConnectionTracker.java-> onDisconnectDone():**

protected void onDisconnectDone(AsyncResult ar)

{

……

trySetupData(reason); //🡺**18**

……

}

**18. GsmDataConnectionTracker.java-> trySetupData ():**

protected boolean trySetupData(String reason)

{

……

return setupData(reason); //🡺**19**

……

}

**19. GsmDataConnectionTracker.java-> setupData ():**

protected boolean setupData(String reason)

{

……

Message msg = obtainMessage();

msg.what = EVENT\_DATA\_SETUP\_COMPLETE;

msg.obj = reason;

pdp.connect(msg, apn); //🡺**20**

……

}

**20. GsmDataConnection.java-> onConnect():**

protected void onConnect(ConnectionParams cp)

{

……

Message msg = obtainMessage(EVENT\_SETUP\_DATA\_CONNECTION\_DONE, cp);

msg.obj = cp;

……

phone.mCM.setupDataCall(

Integer.toString(RILConstants.SETUP\_DATA\_TECH\_GSM),

Integer.toString(RILConstants.DATA\_PROFILE\_DEFAULT),

apn.apn, apn.user, apn.password, Integer.toString(authType),

protocol, msg); 🡺**21**

}

其中，phone的类型为GSMPhone，GSMPhone继承PhoneBase，PhoneBase的成员变量mCM的类型为接口CommandsInterface，并且RIL类实现了接口CommandsInterface，所以，实际上是调用RIL. setupDataCall()。

## 第四部分：RIL层（java部分）

**21.RIL.java-> setupDataCall():**

public void setupDataCall(…)

{

RILRequest rr = RILRequest.obtain(RIL\_REQUEST\_SETUP\_DATA\_CALL, result);

……

send(rr); //🡺**22**

}

**22. RIL.java-> send ():**

protected void send(RILRequest rr)

{

Message msg;

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

acquireWakeLock();

msg.sendToTarget(); //🡺**23**

}

**23. RIL.java-> RILSender. handleMessage():**

class RILSender extends Handler implements Runnable{

……

public void handleMessage(Message msg)

{

RILRequest rr = (RILRequest)(msg.obj);

RILRequest req = null;

switch (msg.what)

{

case EVENT\_SEND:

……

LocalSocket s;

s = mSocket;

……

s.getOutputStream().write(dataLength);

s.getOutputStream().write(data); //🡺**24**

……

break;

……

}

}

}

## 第五部分：RIL层（c/c++部分）

**24. reference-ril.c-> onRequest():**

static void onRequest (int request, void \*data, size\_t datalen, RIL\_Token t)

{

switch (request)

{

……

case RIL\_REQUEST\_SETUP\_DATA\_CALL:

requestSetupDataCall(data, datalen, t);

break;

}

}

至此，则通过串口向Ril库发送建立数据连接的AT命令。这样，把请求发送到RIL层之后，转换到mActivatingState状态，接下来就是等待RIL层回应了。

## 第六部分：接收Ril层建立数据连接完成应答

**25.RIL.java-> RILReceiver. run():**

class RILReceiver implements Runnable{

……

public void run()

{

……

processResponse(p); //🡺**26**

……

}

}

**26. RIL.java-> processResponse ():**

private void processResponse (Parcel p)

{

int type;

type = p.readInt();

if (type == RESPONSE\_UNSOLICITED)

{

processUnsolicited (p);

}

else if (type == RESPONSE\_SOLICITED)

{

processSolicited (p); //🡺**27**

}

……

}

**27. RIL.java-> processSolicited ():**

protected void processSolicited (Parcel p)

{

……

RILRequest rr;

rr = findAndRemoveRequestFromList(serial);

……

if (rr.mResult != null

{

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

rr.mResult.sendToTarget(); //🡺**28**

}

rr.release();

}

这里，会形成一个异步结果，绑定到消息里面发上来。从第20步可以知道，上面的mActivationState状态会收到消息EVENT\_SETUP\_DATA\_CONNECTION\_DONE。

## 第七部分：应用层状态和UI更新

**28. DataConnection.java-> DcActivatingState. processMessage ():**

private class DcActivatingState extends HierarchicalState{

protected boolean processMessage(Message msg)

{

……

switch (msg.what)

{

……

case EVENT\_SETUP\_DATA\_CONNECTION\_DONE:

ar = (AsyncResult) msg.obj;

cp = (ConnectionParams) ar.userObj;

SetupResult result = onSetupConnectionCompleted(ar);

switch (result)

{

case SUCCESS:

……

mActiveState.setEnterNotificationParams(cp, FailCause.NONE); //

transitionTo(mActiveState);

break;

case ERR\_BadCommand:

mInactiveState.setEnterNotificationParams(cp, result.mFailCause);

transitionTo(mInactiveState);

break;

……

}

……

}

}

}

从第19步可以知道，接下来会处理消息EVENT\_DATA\_SETUP\_COMPLETE。

**29. DataConnectionTracker.java-> handleMessage():**

public void handleMessage (Message msg)

{

switch (msg.what)

{

……

case EVENT\_DATA\_SETUP\_COMPLETE:

cidActive = msg.arg1;

if (((AsyncResult) msg.obj).exception == null)

{

addActiveCid(cidActive);

}

onDataSetupComplete((AsyncResult) msg.obj); //🡺**30**

break;

……

}

}

**30. GsmDataConnectionTracker.java-> onDataSetupComplete():**

protected void onDataSetupComplete(AsyncResult ar)

{

……

notifyDefaultData(reason); //🡺**31**

……

}

**31. GsmDataConnectionTracker.java-> notifyDefaultData ():**

protected void notifyDefaultData(String reason)

{

setState(State.CONNECTED);

phone.notifyDataConnection(reason); //🡺**32**

startNetStatPoll();

……

}

phone为GsmDataConnectionTracker基类DataConnectionTracker的成员变量，类型为PhoneBase，所以调用PhoneBase. notifyDataConnection()。

**32. PhoneBase.java-> notifyDataConnection():**

*protected PhoneNotifier mNotifier;*

public void notifyDataConnection(String reason)

{

boolean supportMpdp = SystemProperties.getBoolean("persist.telephony.mpdp", true);

if (supportMpdp)

{

……

}

else

{

mNotifier.notifyDataConnection(this, reason); //🡺**33**

}

}

mNotifier的类型为接口PhoneNotifier，而DefaultPhoneNotifier实现了接口PhoneNotifier，所以实际调用DefaultPhoneNotifier. notifyDataConnection()。

**33. DefaultPhoneNotifier.java-> notifyDataConnection():**

*private ITelephonyRegistry mRegistry;*

public void notifyDataConnection(Phone sender, String reason)

{

TelephonyManager telephony = TelephonyManager.getDefault(sender.getPhoneId());

try {

mRegistry.notifyDataConnection(

convertDataState(sender.getDataConnectionState()),

sender.isDataConnectivityPossible(), reason,

sender.getActiveApn(),

sender.getActiveApnTypes(),

sender.getInterfaceName(null),

((telephony!=null) ? telephony.getNetworkType() :

TelephonyManager.NETWORK\_TYPE\_UNKNOWN),

sender.getGateway(null)); //🡺**34**

} catch (RemoteException ex) {

// system process is dead

}

}

mRegistry的类型为ItelephonyRegistry，DefaultPhoneNotifier是ITelephonyRegistry接口的客户端，其服务端是TelephonyRegistry（com.android.server.TelephonyRegistry）。所以，实际调用TelephonyRegistry. notifyDataConnection()。

**34. TelephonyRegistry.java-> notifyDataConnection():**

*private final ArrayList<Record> mRecords = new ArrayList();*

public void notifyDataConnection(…)

{

……

synchronized (mRecords) {

mDataConnectionState = state;

mDataConnectionPossible = isDataConnectivityPossible;

mDataConnectionReason = reason;

mDataConnectionApn = apn;

mDataConnectionApnTypes = apnTypes;

mDataConnectionInterfaceName = interfaceName;

mDataConnectionNetworkType = networkType;

for (int i = mRecords.size() - 1; i >= 0; i--)

{

Record r = mRecords.get(i);

if ((r.events & PhoneStateListener.LISTEN\_DATA\_CONNECTION\_STATE) != 0)

{

try {

r.callback.onDataConnectionStateChanged(state, networkType); //🡺**35**

} catch (RemoteException ex) {

remove(r.binder);

}

}

}

}

broadcastDataConnectionStateChanged(state, isDataConnectivityPossible, reason, apn,

apnTypes, interfaceName, gateway); //🡺**36**

}

r是当前mRecords中的元素，包含有IPhoneStateListener接口的实现callback，TelephonyRegistry中的每个调用都会遍历mRecords中的元素，如果某个元素注册了对应接听，则调用callback的某个函数。客户端通过如下方式调用取得电话状态的监听， 以StatusBarPolicy.java中的mPhoneStateListener为例。

**StatusBarPolicy.java->StatusBarPolicy():**

*private PhoneStateListener[] mPhoneStateListener;*

public StatusBarPolicy(Context context)

{

……

for (int i = 0; i < numPhones; i++)

{

mSignalStrength[i] = new SignalStrength();

mSimState[i] = IccCard.State.READY;

mPhoneSignalIconId[i] = R.drawable.stat\_sys\_signal\_null;

mService.setIcon(mSignalIcon[i], mPhoneSignalIconId[i], 0);

mPhoneStateListener[i] = getPhoneStateListener(i);

mDataState[i] = TelephonyManager.DATA\_DISCONNECTED;

// register for phone state notifications.

((TelephonyManager) mContext

.getSystemService(PhoneFactory.getServiceName(Context.TELEPHONY\_SERVICE,i))).listen(

mPhoneStateListener[i],

PhoneStateListener.LISTEN\_SERVICE\_STATE

| PhoneStateListener.LISTEN\_SIGNAL\_STRENGTHS

| PhoneStateListener.LISTEN\_CALL\_STATE

| PhoneStateListener.LISTEN\_DATA\_CONNECTION\_STATE

| PhoneStateListener.LISTEN\_DATA\_ACTIVITY);

}

……

}

mPhoneStateListener是PhoneStateListener实例，PhoneStateListener实现了IPhoneStateListener接口，假如你继承PhoneStateListener子类，首先你要确定你感兴趣的监听事件，然后重写对应的方法。再像上面那样调用listen方法就可以了。

TelephonyRegistry的方法、监听动作、以及要重写的方法对应关系如下：

|  |  |  |
| --- | --- | --- |
| TelephonyRegistry的方法 | 监听动作 | PhoneStateListener子类中的中的回调 |
| notifyServiceState | PhoneStateListener  .LISTEN\_SERVICE\_STATE | public void  onServiceStateChanged  (ServiceState state) |
| notifySignalStrength | PhoneStateListener  .LISTEN\_SIGNAL\_STRENGTHS | public void  onSignalStrengthsChanged  (SignalStrength signalStrength) |
| notifyCallState | PhoneStateListener  .LISTEN\_CALL\_STATE | public void  onCallStateChanged  (int state, String incomingNumber) |
| notifyDataConnection | PhoneStateListener  .LISTEN\_DATA\_CONNECTION\_STATE | public void  onDataConnectionStateChanged  (int state, int networkType) |
| notifyDataActivity | PhoneStateListener  .LISTEN\_DATA\_ACTIVITY | public void  onDataActivity(int direction) |
| …… | …… | …… |

因此整个调用链是：

->DefaultPhoneNotifier. notifyDataConnection()

->TelephonyRegistry. notifyDataConnection()

->PhoneStateListener.callback. onDataConnectionStateChanged()

->PhoneStateListener子类的onDataConnectionStateChanged()

**35. StatusBarPolicy.java-> getPhoneStateListener ():**

private PhoneStateListener getPhoneStateListener(int subscription)

{

PhoneStateListener phoneStateListener = new PhoneStateListener(subscription)

{

……

public void onDataConnectionStateChanged(int state, int networkType)

{

mDataState[mSubscription] = state;

updateDataNetType(networkType); //更新数据连接网络类型

updateDataIcon(mSubscription); //更新数据连接图标

}

……

}

}

除此之外，TelephonyRegistry还发出一个广播消息：ACTION\_ANY\_DATA\_CONNECTION\_STATE\_CHANGED，包含数据连接的详细信息。

**36. TelephonyRegistry.java-> broadcastDataConnectionStateChanged():**

private void broadcastDataConnectionStateChanged(…)

{

Intent intent = new

Intent(TelephonyIntents.ACTION\_ANY\_DATA\_CONNECTION\_STATE\_CHANGED);

intent.addFlags(Intent.FLAG\_RECEIVER\_REPLACE\_PENDING);

intent.putExtra(Phone.STATE\_KEY, DefaultPhoneNotifier.convertDataState(state).toString());

if (!isDataConnectivityPossible)

{

intent.putExtra(Phone.NETWORK\_UNAVAILABLE\_KEY, true);

}

if (reason != null)

{

intent.putExtra(Phone.STATE\_CHANGE\_REASON\_KEY, reason);

}

intent.putExtra(Phone.DATA\_APN\_KEY, apn);

……

intent.putExtra(Phone.DATA\_APN\_TYPES\_KEY, types);

intent.putExtra(Phone.DATA\_IFACE\_NAME\_KEY, interfaceName);

……

intent.putExtra(Phone.DATA\_GATEWAY\_KEY, gatewayAddr);

intent.putExtra(Intents.EXTRA\_PHONE\_ID, mPhoneId);

mContext.sendStickyBroadcast(intent); //🡺**37**

}

而Mobile Data Service里面的MobileDataStateTracker会接收到这个动作，由它的BoadcastReceiver类MobileDataStateReceiver提取出数据连接的信息，然后设置好状态。

**37. MobileDataStateTracker.java-> MobileDataStateReceiver. onReceive():**

private class MobileDataStateReceiver extends BroadcastReceiver{

……

public void onReceive(Context context, Intent intent)

{

……

TelephonyManager tm = TelephonyManager.getDefault(getPhoneId());

setRoamingStatus(tm.isNetworkRoaming());

setSubtype(tm.getNetworkType(), tm.getNetworkTypeName());

setSubId(getPhoneId());

if (intent.getAction().equals(TelephonyIntents.

ACTION\_ANY\_DATA\_CONNECTION\_STATE\_CHANGED))

{

……

switch (state)

{

case DISCONNECTED:

……

case CONNECTED:

……

setDetailedState(DetailedState. CONNECTED, reason, apnName); //🡺**38**

break;

……

}

}

……

}

}

MobileDataStateTracker根据状态变化给ConnectivityService发送EVENT\_STATE\_CHANGED消息。

**38. MobileDataStateTracker.java-> setDetailedState():**

public void setDetailedState(NetworkInfo.DetailedState state, String reason, String extraInfo)

{

……

mNetworkInfo.setDetailedState(state, reason, extraInfo);

Message msg = mTarget.obtainMessage(EVENT\_STATE\_CHANGED, mNetworkInfo);

msg.sendToTarget(); //🡺**39**

}

ConnectivityService调用handleConnect去执行相关操作，包括关闭优先级比它低的数据连接，更新状态栏等等。

**39. ConnectivityService.java-> MyHandler. handleMessage():**

private class MyHandler extends Handler{

public void handleMessage(Message msg)

{

switch (msg.what)

{

case NetworkStateTracker.EVENT\_STATE\_CHANGED:

info = (NetworkInfo) msg.obj;

int type = info.getType();

NetworkInfo.State state = info.getState();

……

if (info.getDetailedState() == NetworkInfo.DetailedState.FAILED)

{

handleConnectionFailure(info);

}

else if (state == NetworkInfo.State.DISCONNECTED)

{

handleDisconnect(info);

}

else if (state == NetworkInfo.State.SUSPENDED)

{

handleDisconnect(info);

}

else if (state == NetworkInfo.State.CONNECTED)

{

handleConnect(info);

}

break;

……

}

}

}