在Android中，应用对数据连接的操作需要通过framework提供的以下两个服务类frameworks/base/services/java/com/android/server/ConnectivityService数据连接的系统级服务类，frameworks/base/core/java/android/net/ConnectivityManager数据连接的应用层管理类。下面就ConnectivityService的建立和彩信请求建立数据连接的过程做简单描述，欢迎有兴趣的朋友一起交流讨论。

# ConnectivityService的启动和驻留系统进程。

系统启动时，在SystemServer中调用ConnectivityService.getInstance(Context context)获取实例，再将其加入ServiceManager，常驻系统进程。

connectivity = ConnectivityService.getInstance(context);

ServiceManager.addService(Context.CONNECTIVITY\_SERVICE, connectivity);

获取ConnectivityService实例时，调用的是内部的一个Thread。这里使用thread来创建ConnectivityService实例，可想而知实例化时做了很多耗时操作。

public static ConnectivityService getInstance(Context context) {

return ConnectivityThread.getServiceInstance(context);

}

在该thread的run中来实例化ConnectivityService，为此线程还添加了looper循环，提供给ConnectivityService实例化时的消息队列使用，消息队列需要looper循环的支持。

private static class ConnectivityThread extends Thread {

private Context mContext;

private ConnectivityThread(Context context) {

super("ConnectivityThread");

mContext = context;

}

@Override

public void run() {

Looper.prepare();

synchronized (this) {

sServiceInstance = new ConnectivityService(mContext);

notifyAll();

}

Looper.loop();

}

public static ConnectivityService getServiceInstance(Context context) {

ConnectivityThread thread = new ConnectivityThread(context);

thread.start();

synchronized (thread) {

while (sServiceInstance == null) {

try {

// Wait until sServiceInstance has been initialized.

thread.wait();

} catch (InterruptedException ignore) {

Slog.e(TAG, "Unexpected InterruptedException while waiting"+" for ConnectivityService thread");

}

}

}

return sServiceInstance;

}

}

在构造方法中首先创建放置网络跟踪器，无线属性和网络属性的容器。mNetTrackers中存放的是系统支持的所有网络的跟踪器，之后对网络的操作都要依靠mNetTrackers中的网络跟踪器来执行。NetworkStateTracker是一个继承Handler的抽象类，其中定义了对网络功能的连接和停止等接口方法，实现它的子类有WifiStateTracker，MobileDataStateTracker分别完成对移动网络和wifi网络的操作功能。这些类都在frameworks/base/core/java/android/net/目录下。RadioAttributes和NetworkAttributes是ConnectivityService中定义的内部类，两个类中分别定义了无线和网络的相关属性，这些容器通过资源文件cofig.xml获取系统配置的值来实例化。

mNetTrackers = new NetworkStateTracker[ConnectivityManager.MAX\_NETWORK\_TYPE+1];

mRadioAttributes = new RadioAttributes[ConnectivityManager.MAX\_RADIO\_TYPE+1];

mNetAttributes = new NetworkAttributes[ConnectivityManager.MAX\_NETWORK\_TYPE+1];

cofig.xml文件的路径frameworks/base/core/res/res/values/,其中对无线和网络属性的配置如下：

--网络属性，从左到右的属性依次是网络名称，网络类型，使用的无线技术类型，优先级

<string-array translatable="false" name="networkAttributes">

<item>"wifi,1,1,1"</item>

<item>"mobile,0,0,0"</item>

<item>"mobile\_mms,2,0,2"</item>

<item>"mobile\_supl,3,0,2"</item>

<item>"mobile\_hipri,5,0,3"</item>

<item>"mobile\_dm,10,0,3"</item>

<item>"mobile\_mms0,11,0,2"</item>

<item>"mobile\_mms1,12,0,2"</item>

</string-array>

--无线属性,从左到右的属性依次是无线类型，并发类型

<string-array translatable="false" name="radioAttributes">

<item>"1,1"</item>

<item>"0,1"</item>

</string-array>

接着实例化了mNetRequestersPids存放请求网络进程的容器，可以看见它是一个存放List的数组，初始化时使用的是mPriorityList中的值作为下标，mPriorityList中的内容是在上一步按照mNetAttributes中NetAttributes的优先级存放的网络类型，这样mNetRequestersPids每一个下标就代表了一种网络类型，每一个下标在mNetRequestersPids指定的内容就是请求这一网络类型的所有进程id.

mNetRequestersPids = new ArrayList[ConnectivityManager.MAX\_NETWORK\_TYPE+1];

for (int i : mPriorityList) {

mNetRequestersPids[i] = new ArrayList();

}

还要实例化mFeatureUsers，存放用户请求的网络功能。mMmsFeatureRequest，存放用户请求的彩信网络功能。它们中存放的都是FeatureUser对象，

FeatureUser中包含了mNetworkType，mFeature，IBinder mBinder，mPid，int mUid，mCreateTime这些属性，还有binderDied方法，当内部的

IBinder对象死亡时调用，停用此FeatureUser表示的网络功能，还有expire()方法，当此功能失效时调用，停用代表的网络功能。

mFeatureUsers = new ArrayList();

mMmsFeatureRequest = new ArrayList();

接下来就是根据网络类型创建对应的跟踪器，来管理此类网络。

boolean noMobileData = !getMobileDataEnabledByPhoneId(TelephonyManager

.getDefaultDataPhoneId(mContext));

for (int netType : mPriorityList) {

switch (mNetAttributes[netType].mRadio) {

case ConnectivityManager.TYPE\_WIFI:

if (DBG) Slog.v(TAG, "Starting Wifi Service.");

WifiStateTracker wst = new WifiStateTracker(context, mHandler);

WifiService wifiService = new WifiService(context, wst);

ServiceManager.addService(Context.WIFI\_SERVICE, wifiService);

wifiService.startWifi();

mNetTrackers[ConnectivityManager.TYPE\_WIFI] = wst;

wst.startMonitoring();

break;

case ConnectivityManager.TYPE\_MOBILE:

mNetTrackers[netType] = new MobileDataStateTracker(context, mHandler,

netType, mNetAttributes[netType].mName);

mNetTrackers[netType].startMonitoring();

if (noMobileData) {

if (DBG) Slog.d(TAG, "tearing down Mobile networks due to setting");

mNetTrackers[netType].teardown();

}

break;

case ConnectivityManager.TYPE\_WIMAX:

NetworkStateTracker nst = makeWimaxStateTracker();

if (nst != null) {

nst.startMonitoring();

}

mNetTrackers[netType] = nst;

if (noMobileData) {

if (DBG) Slog.d(TAG, "tearing down WiMAX networks due to setting");

mNetTrackers[netType].teardown();

}

break;

default:

Slog.e(TAG, "Trying to create a DataStateTracker for an unknown radio type " +

mNetAttributes[netType].mRadio);

continue;

}

}

noMobileData获取的手机默认sim卡的数据连接开关的状态，如果为关闭，在创建了使用移动网络的跟踪器后，会通过跟踪器断开该网络的数据连接。 按照mPriorityList中网络的优先级，逐个使用其中存放的netType在mNetAttributes找到对应的网络属性，根据网络属性的mRadio无线类型判断需要创建的无线网络类型，我们看到cofig.xml文件中无线类型只有两个，分别是1和0，表示wifi和mobile无线信号，而在cofig.xml文件中配置的网络属性的无线类型也只有这两种，其中除了wifi网络,其它都需要使用移动网络。TYPE\_WIMAX是无线城域网，当前我国还没有此种网络，国内的手机在配置文件中都不会使用此种网络。使用wifi信号的只有一个网络属性，只会创建一个WifiStateTracker。使用移动网络的网络类型有多个，它们会分别被创建。MobileDataStateTracker创建时会根据传入的netType设置相应的apn,之后如果需要使用此MobileDataStateTracker连接网络，就会调用PhoneService的方法启用此apn，完成连接网络的操作。

# 应用调用数据连接服务，启动数据连接（彩信调用数据连接过程）

彩信应用点击发送按键后，最终会调用TrancationService来处理彩信数据上传。TrancationService是彩信transaction包中的一个service，负责彩信收发时的数据传送。

彩信发送时，TrancationService在beginMmsConnectivity()方法中调用ConnectivityManager的startUsingNetworkFeature()方法来开启指定的网络连接。

protected int beginMmsConnectivity() throws IOException {

............................

int result = mConnMgr.startUsingNetworkFeature(ConnectivityManager.TYPE\_MOBILE, PhoneFactory.getFeature(Phone.FEATURE\_ENABLE\_MMS, mPhoneId));

..........................

}

调用startUsingNetworkFeature方法时传入的ConnectivityManager.TYPE\_MOBILE为指定要启用的网络类型，此网络类型是mNetAttributes从config.xml中初始化的第二个值，也做为了mNetAttributes的下标，来存放对应的网络属性。其值为0，表示的就是要使用mobile移动网络。第二个参数是要使用的彩信功能，返回的是类似mobile\_mms0这样的功能名称字符串，如果是单卡就直接会传入mobile\_mms。使用这样的名称是为了确定移动网络的接入方式，也就是apn。

String[] naStrings = context.getResources().getStringArray(

com.android.internal.R.array.networkAttributes);

for (String naString : naStrings) {

try {

NetworkAttributes n = new NetworkAttributes(naString);

..........................

mNetAttributes[n.mType] = n;

mNetworksDefined++;

} catch(Exception e) {

// ignore it - leave the entry null

}

}

ConnectivityManager中会通过ConnectivityService的本地代理IConnectivityManager调用ConnectivityService的startUsingNetworkFeature方法。调用service时，除了彩信应用传来的networkType, feature，还新new了一个Binder。此Binder如果死亡的话，会回调ConnectivityService的停止网络功能的方法，将和此Binder相连接的网络功能断开。

public int startUsingNetworkFeature(int networkType, String feature) {

try {

return mService.startUsingNetworkFeature(networkType, feature,

new Binder());

} catch (RemoteException e) {

return -1;

}

}

ConnectivityService首先使用传进来的networkType, feature, binder创建一个FeatureUser。FeatureUser是ConnectivityService的一个内部类，除了传入的参数，还保存有请求此功能的Uid和Pid,以及此FeatureUser的创建时间mCreateTime。当此FeatureUser过期或者内部的Binder死亡皆会回调stopUsingNetworkFeature方法来停止此FeatureUser请求的网络。如果之前有请求过彩信网络，会存在mMmsFeatureRequest中。此次请求的FeatureUser和mMmsFeatureRequest中首个请求如果相同，且没有断开，直接至mMmsFeatureState为CONNECTING。如果之前没有同类请求，就加入mMmsFeatureRequest中。前一种情况和已存在请求但没连接，都将返回APN\_REQUEST\_STARTED的状态码，表示请求已发出。mMmsFeatureRequest中的FeatureUser会在网络断开时移除。

FeatureUser f = new FeatureUser(networkType, feature, binder);

synchronized(this) {

if (isMmsFeature(feature)) {

if (mMmsFeatureRequest.isEmpty()) {

// current data connection is default

mMmsFeatureRequest.add(f);

mMmsFeatureState = FeatureState.CONNECTING;

} else {

// current data connection is mms

FeatureUser k = (FeatureUser)mMmsFeatureRequest.get(0);

if (TextUtils.equals(k.mFeature, feature) && mMmsFeatureState != FeatureState.DISCONNECTING) {

mMmsFeatureState = FeatureState.CONNECTING;

} else {

boolean found = false;

for (int i = 0; i < mMmsFeatureRequest.size(); i++) {

FeatureUser u = (FeatureUser)mMmsFeatureRequest.get(i);

if (TextUtils.equals(feature, u.mFeature)) {

found = true;

break;

}

}

if (!found) {

if (DBG) {

Slog.d(TAG, "startUsing Mms pending " + feature);

}

mMmsFeatureRequest.add(f);

} else {

if (DBG) {

Slog.d(TAG, "startUsing duplicate Mms pending " + feature);

}

}

return Phone.APN\_REQUEST\_STARTED;

}

}

}

接下来获取指定网络类型是否有效，判断方式是networkType在0到MAX\_NETWORK\_TYPE的范围内，或者此networkType在系统中并没有配置，在mNetAttributes没有初始化它的属性，就会直接返回网络请求失败标识APN\_REQUEST\_FAILED。

if (!ConnectivityManager.isNetworkTypeValid(networkType) ||

mNetAttributes[networkType] == null) {

return Phone.APN\_REQUEST\_FAILED;

}

默认usedNetworkType是我们传入的networkType的值，如果是请求移动网络连接，需要再根据请求的feature判断要使用的网络连接方式usedNetworkType。当指定卡上的数据连接开关没有打开，并且是MMS和WAP连接，将会返回请求无效标识APN\_TYPE\_NOT\_AVAILABLE。因为多卡手机，会指定彩信请求的卡，指定的卡如果数据开关关闭，是无法使用的。但其他方式的网络连接不会指定使用的卡。

boolean skipAvailableCheck = false;

// TODO - move this into the MobileDataStateTracker

int usedNetworkType = networkType;

if(networkType == ConnectivityManager.TYPE\_MOBILE) {

if (!getMobileDataEnabledByPhoneId(getPhoneIdByFeature(feature)) &&

!(feature.indexOf(Phone.FEATURE\_ENABLE\_MMS)!=-1) &&

!(feature.indexOf(Phone.FEATURE\_ENABLE\_WAP)!=-1)) {

return Phone.APN\_TYPE\_NOT\_AVAILABLE;

}

if (TextUtils.equals(feature, Phone.FEATURE\_ENABLE\_DM)) {

usedNetworkType = ConnectivityManager.TYPE\_MOBILE\_DM;

} else if (TextUtils.equals(feature.substring(0, Phone.FEATURE\_ENABLE\_MMS.length()), Phone.FEATURE\_ENABLE\_MMS)) {

skipAvailableCheck = true;

usedNetworkType = ConnectivityManager.getMmsTypeByPhoneId(getPhoneIdByFeature(feature));

} else if (TextUtils.equals(feature, Phone.FEATURE\_ENABLE\_SUPL)) {

usedNetworkType = ConnectivityManager.TYPE\_MOBILE\_SUPL;

} else if (TextUtils.equals(feature, Phone.FEATURE\_ENABLE\_DUN)) {

usedNetworkType = ConnectivityManager.TYPE\_MOBILE\_DUN;

} else if (TextUtils.equals(feature, Phone.FEATURE\_ENABLE\_HIPRI)) {

usedNetworkType = ConnectivityManager.TYPE\_MOBILE\_HIPRI;

} else if (TextUtils.equals(feature, Phone.FEATURE\_ENABLE\_WAP)) {

usedNetworkType = ConnectivityManager.getMmsTypeByPhoneId(getPhoneIdByFeature(feature));

}

}

通过上一步获取的feature需要使用的网络类型usedNetworkType，获取系统中存在网络跟踪器NetworkStateTracker操作相关网络连接。如果获取不到需要使用的网络信息，将会返回APN\_TYPE\_NOT\_AVAILABLE表示所请求的APN无效。找到要使用的网络信息后，如果请求传入的networkType与请求功能需要的usedNetworkType网络类型相等，先增加此次请求的feature到mFeatureUsers中，将在网络断开时被清除。发送的EVENT\_RESTORE\_DEFAULT\_NETWORK的message会在指定时间后，在handler中调用f的expire方法，断开之前的连接，其实也就是设置了本次连接请求的超时时间，默认是5分钟。接着再调用network.startUsingNetworkFeature方法启动传入的请求网络功能feature，此方法返回的结果做为此次请求网络连接的结果返回。network就是通过usedNetworkType找到的NetworkStateTracker网络跟踪器。

NetworkStateTracker network = mNetTrackers[usedNetworkType];

if (network != null) {

if (usedNetworkType != networkType) {

..........................

} else {

synchronized(this) {

mFeatureUsers.add(f);

}

mHandler.sendMessageDelayed(mHandler.obtainMessage(EVENT\_RESTORE\_DEFAULT\_NETWORK,

f), getRestoreDefaultNetworkDelay());

return network.startUsingNetworkFeature(feature,

getCallingPid(), getCallingUid());

}

}

return Phone.APN\_TYPE\_NOT\_AVAILABLE;

根据调用ConnectivityService.startUsingNetworkFeature传入的networkType为TYPE\_MOBILE可知要连接的是移动网络，此NetworkStateTracker实际引用的是它的子类MobileDataStateTracker。从注释看应用直接调用此方法会返回-1表示请求网络失败。从上面根据feature获取需使用的网络类型usedNetworkType可以看出移动网络被分成多个子功能，每个功能都对应一个MobileDataStateTracker，当请求网络功能时，最后获取的usedNetworkType肯定不等于传入networkType的值TYPE\_MOBILE。TYPE\_MOBILE是默认给系统调用的数据连接，不会走startUsingNetworkFeature方法，所以当有其他应用调用请求网络功能走到此，肯定发生了异常，直接返回-1便是。wifi网络的WifiStateTracker调用此方法也会返回-1，同样wifi网络启用也不能被应用调用。

/\*\*

\* Tells the phone sub-system that the caller wants to

\* begin using the named feature. The only supported features at

\* this time are {@code Phone.FEATURE\_ENABLE\_MMS}, which allows an application

\* to specify that it wants to send and/or receive MMS data, and

\* {@code Phone.FEATURE\_ENABLE\_SUPL}, which is used for Assisted GPS.

\* @param feature the name of the feature to be used

\* @param callingPid the process ID of the process that is issuing this request

\* @param callingUid the user ID of the process that is issuing this request

\* @return an integer value representing the outcome of the request.

\* The interpretation of this value is feature-specific.

\* specific, except that the value {@code -1}

\* always indicates failure. For {@code Phone.FEATURE\_ENABLE\_MMS},

\* the other possible return values are

\*/

public int startUsingNetworkFeature(String feature, int callingPid, int callingUid) {

return -1;

}

正常情况会走到usedNetworkType != networkType的流程中。首先会取得请求网络的当前网络信息NetWorkInfo，如果是请求彩信功能连接，不会判断网络是否有效，网络是否有效类似于设备是否在网络范围内，漫游情况是否打开数据漫游开关等，彩信功能请求会在下一步进行连接尝试，其它网络功能的请求在网络无效时，直接返回APN\_TYPE\_NOT\_AVAILABLE当前网络不可用。需要请求的网络如果已经连接，直接返回APN\_ALREADY\_ACTIVE网络已激活；如果正在连接且没有发出断开请求，返回APN\_REQUEST\_STARTED连接请求已开始；如果没有连接，就调用network.reconnect()重新连接指定的网络。此处的network是MobileDataStateTracker。

if (usedNetworkType != networkType) {

Integer currentPid = new Integer(getCallingPid());

NetworkStateTracker radio = mNetTrackers[networkType];

NetworkInfo ni = network.getNetworkInfo();

// TODO: using serviceState to check is better for Msms

if (!skipAvailableCheck && ni.isAvailable() == false) {

if (DBG) Slog.d(TAG, "special network not available");

return Phone.APN\_TYPE\_NOT\_AVAILABLE;

}

..........................

if (DBG) Slog.d(TAG, "ni.isConnectedOrConnecting()="+ni.isConnectedOrConnecting());

if ((ni.isConnectedOrConnecting() == true) &&

!network.isTeardownRequested()) {

if (ni.isConnected() == true) {

// add the pid-specific dns

handleDnsConfigurationChange(networkType);

if (DBG) Slog.d(TAG, "special network already active");

return Phone.APN\_ALREADY\_ACTIVE;

}

if (DBG) Slog.d(TAG, "special network already connecting");

return Phone.APN\_REQUEST\_STARTED;

}

if (!isAlreadyConnecting) {

if (DBG) Slog.d(TAG, "reconnecting to special network");

synchronized(this) {

if(!network.reconnect()){

if (isMmsFeature(feature) && mMmsFeatureRequest.size() > 0) {

mMmsFeatureRequest.remove(0);

}

return Phone.APN\_TYPE\_NOT\_AVAILABLE;

}

}

} else {

if (DBG) Slog.d(TAG, "already reconnecting to special network");

}

return Phone.APN\_REQUEST\_STARTED;

}

MobileDataStateTracker中的reconnect,连接网络的操作调用的是setEnableApn(mApnType, true)，意思是开启指定的apn。此处传的mApnType是在MobileDataStateTracker创建时，根据netType确定的apn类型。移动网络不同的网络功能有不同的apn类型，此处传入的是"mms",其实所谓连接网络就是切换指定APN，APN的切换都需通过PhoneService来调用ril和modem通信来完成。

public boolean reconnect() {

setTeardownRequested(false);

if (DBG) Log.d(TAG, " reconnect :setEnableApn"+ mApnType );

switch (setEnableApn(mApnType, true)) {

case Phone.APN\_ALREADY\_ACTIVE:

..........................

case Phone.APN\_REQUEST\_STARTED:

..........................

case Phone.APN\_REQUEST\_FAILED:

..........................

case Phone.APN\_TYPE\_NOT\_AVAILABLE:

..........................

default:

..........................

}

return mEnabled;

}

先调用getPhoneService(false)获取mPhoneService，根据enable的值调用mPhoneService的enableApnType(apnType)启用指定的apn。

private int setEnableApn(String apnType, boolean enable) {

getPhoneService(false);

/\*

\* If the phone process has crashed in the past, we'll get a

\* RemoteException and need to re-reference the service.

\*/

for (int retry = 0; retry < 2; retry++) {

if (mPhoneService == null) {

Log.w(TAG,

"Ignoring feature request because could not acquire PhoneService");

break;

}

try {

if (enable) {

return mPhoneService.enableApnType(apnType);

} else {

return mPhoneService.disableApnType(apnType);

}

} catch (RemoteException e) {

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

}

}

Log.w(TAG, "Could not " + (enable ? "enable" : "disable")

+ " APN type \"" + apnType + "\"");

return Phone.APN\_REQUEST\_FAILED;

}

getPhoneService(false)返回的是PhoneInterfaceManager.java的实例，此类在packages/apps/Phone/src/com/android/phone包下，mPhoneService.enableApnType(apnType)实现时调用了PhoneInterfaceManager的成员变量mPhone，此mPhone在PhoneInterfaceManager创建时由PhoneFactory传建，依据网络模式分别有GsmPhone和CDMAPhone两种类型，分别在framework/telephony/java/com/android/internal/telephony/的gsm和cdma包下。此处只看GsmPhone，GsmPhone继承PhoneBase.java,enableApnType方法实现在父类PhoneBase中。在framework/telephony/java/com/android/internal/telephony/PhoneBase.java中，通过内部成员mDataConnection的enableApnType(type)方法启用apn。

public int enableApnType(String type) {

return mDataConnection.enableApnType(type);

}

mDataConnection是framework/telephony/java/com/android/internal/telephony/DataConnectionTracker.java的子类实例，DataConnectionTracker是一个抽象类，定义了enableApnType方法。每一个apnType对应一个id，enableApnType方法中获取的是APN\_MMS\_ID，id值为1，调用setEnabled来启用对应apnType的id，setEnabled(id, true)会通过handle的方式调用之后的流程， 到此返回状态APN\_REQUEST\_STARTED，apn切换的请求已开始，正确切换后，网络连接正常，系统会发送广播通知。

public synchronized int enableApnType(String type) {

int id = apnTypeToId(type);

..........................

// just because it's active doesn't mean we had it explicitly requested before

// (a broad default may handle many types). make sure we mark it enabled

// so if the default is disabled we keep the connection for others

setEnabled(id, true);

..........................

return Phone.APN\_REQUEST\_STARTED;

}

setEnabled方法再调用onEnableApn方法，onEnableApn方法实现APN切换要调用onEnableNewApn方法，此方法由子类完成。DataConnectionTracker针对CDMA和Gsm分别有CdmaDataConnectionTracker和GsmDataConnectionTracker两个子类，分别在telephony/cdma和telephony/gsm包下。

protected synchronized void onEnableApn(int apnId, int enabled) {

if (enabled == ENABLED) {

if (!dataEnabled[apnId]) {

dataEnabled[apnId] = true;

enabledCount++;

}

String type = apnIdToType(apnId);

if (!isApnTypeActive(type) || state == State.DISCONNECTING) {

log2("type:" + type+"mRequestedApnType:"+mRequestedApnType);

mRequestedApnType = type;

onEnableNewApn();

}

}

..........................

}

下面流程中打出的日志都会在radio相应中

GSM类型的网络管理数据连接是GsmDataConnectionTracker，它实现了onEnableNewApn()方法，这个方法切换APN时会调用ril的方法和Modem通信，完成断开或者建立数据连接的操作。RetryMgr.resetRetryCount()重置尝试连接次数，如果底层网络连接不成功，系统会重试连接指定的次数，超过指定次数给上层返回连接失败错误，每次上层请求连接时会先重置为0。isAllPdpDisconnectDone()判断当前是否所有数据连接都已断开，如果全部断开，调用trySetupData(Phone.REASON\_APN\_SWITCHED)连接指定的apn，否则先断开其他连接，再连接新的apn。

protected void onEnableNewApn() {

// change our retry manager to use the appropriate numbers for the new APN

if (mRequestedApnType.equals(Phone.APN\_TYPE\_DEFAULT)) {

mRetryMgr = mDefaultRetryManager;

} else {

mRetryMgr = mSecondaryRetryManager;

}

mRetryMgr.resetRetryCount();

log("onEnableNewApn REASON\_APN\_SWITCHED");

if (isAllPdpDisconnectDone()) {

trySetupData(Phone.REASON\_APN\_SWITCHED);

} else {

cleanUpConnection(true, Phone.REASON\_APN\_SWITCHED);

}

}

此处的log("onEnableNewApn REASON\_APN\_SWITCHED")，还有trySetupData下的log("\*\*\*trySetupData due to " + (reason == null ? "(unspecified)" : reason))，以及cleanUpConnection下的log("Clean up connection due to " + reason);

几条Log都可以做为底层建立数据连接的标识。

## 数据连接建立主要分三个过程：

1，上层应用通过ConnectivityMananger请求ConnecivityService对数据网络进行操作，包括建立，停止等操作。

2，ConnectivityService根据传入的网络类型，调用系统启动时创建的NetworkStateTracker建立数据连接，NetworkStateTracker调用Phone模块相关代码切换所请求数据连接的apn.

3，Phone模块再调用Framework下telephony模块的数据连接相关类，数据连接类再通过ril和Modem通信完成数据连接建立。