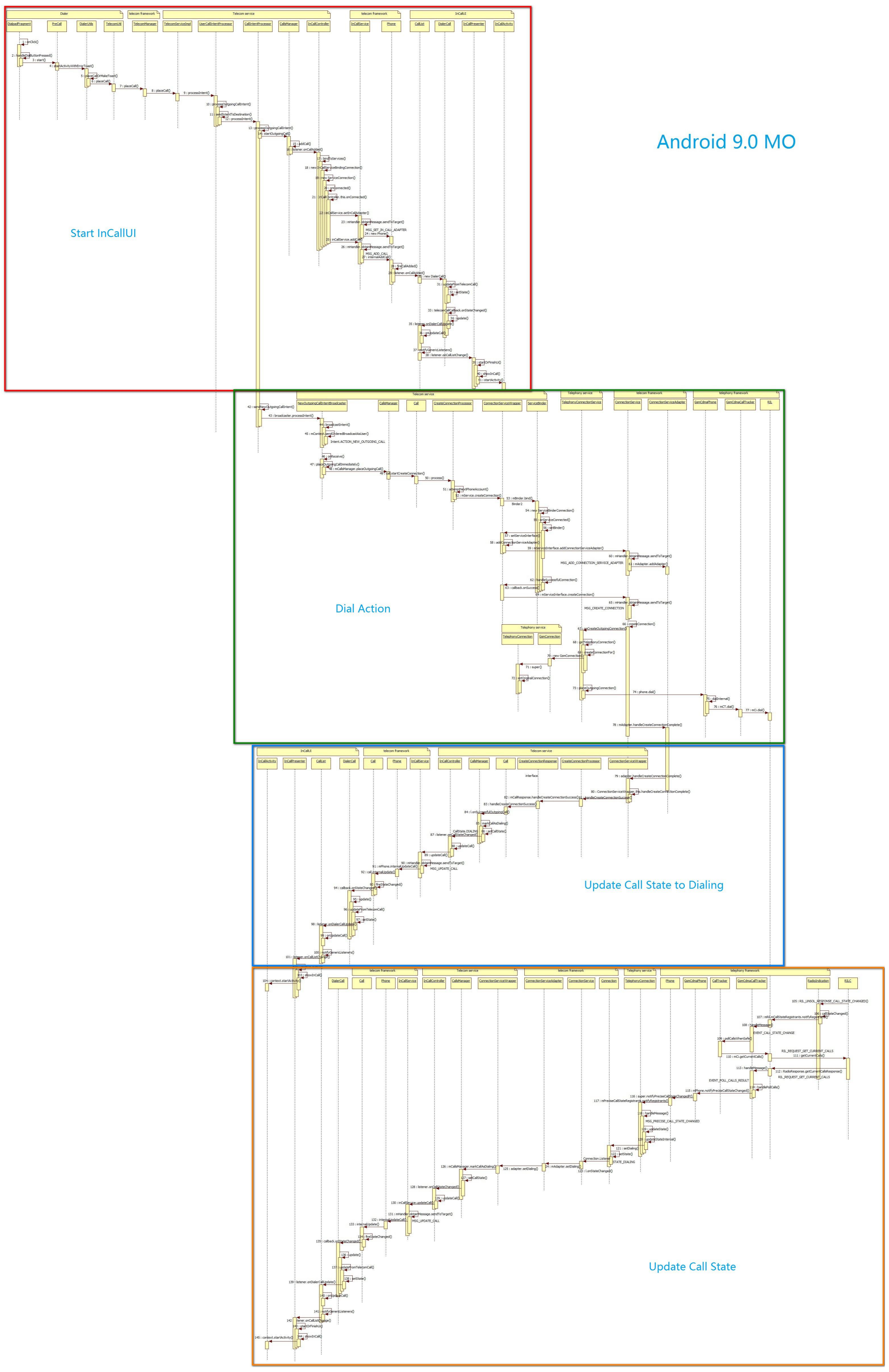
# Android 9.0 MO&MT 流程分析

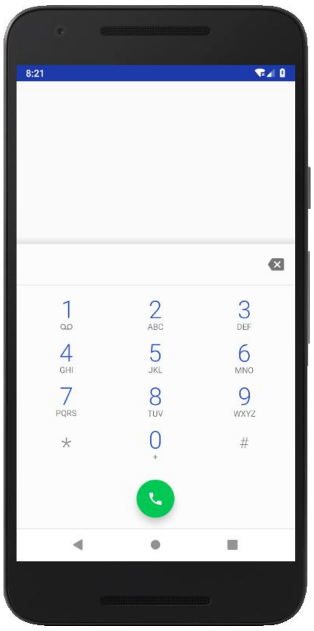
**Base on AOSP**

# Android 9.0 MO 流程



## 第一部分：Start InCallUI

拨号界面：DialpadFragment



当用户点击拨号按钮时，触发 **onClick**，继而执行 **handleDialButtonPressed**

```
public void onClick(View view) {

    int resId = view.getId();

    if (resId == R.id.dialpad_floating_action_button) {//拨号按钮资源
id



view.performHapticFeedback(HapticFeedbackConstants.VIRTUAL_KEY);

      handleDialButtonPressed();
```

**handleDialButtonPressed：**

**handleDialButtonPressed** 主要 handle 三种号码：

1.isDigitsEmpty 空号码的处理 handleDialButtonClickWithEmptyDigits

2.prohibitedPhoneNumberRegexp 禁止号码的处理 ErrorDialogFragment

3.PreCall.start 一般正常号码的处理

PreCall

PreCall 是 Android 9.0 新加入的，它是一个接口，用来 prepare 一个 CallIntentBuilder 在 telecom placecall 之前

PreCall.start：

```
DialerUtils.startActivityWithErrorToast(context, getIntent(context,

builder))
```

DialerUtils

DialerUtils.startActivityWithErrorToast：

startActivityWithErrorToast

先判断 Intent.ACTION_CALL.equals(intent.getAction())
然后进行 touch point 的检查，再进行 wps 高优先级电话的判断 shouldWarnForOutgoingWps，
如果是 wps 电话，那么 showing outgoing WPS dialog before placing call，再 placeCallOrMakeToast，一般电话直接 placeCallOrMakeToast。

placeCallOrMakeToast：

```
final boolean hasCallPermission = TelecomUtil.placeCall(context,

intent);//获得是否有 CallPermission 的布尔值，通过 TelecomUtil.placeCall
```

TelecomUtil

TelecomUtil.placeCall：

```
if (hasCallPhonePermission(context)) {

      getTelecomManager(context).placeCall(intent.getData(),

intent.getExtras());

      return true;

    }
```

获得 TelecomManager

再通过调用 **TelecomManager.placeCall**

```
1.ITelecomService service = getTelecomService()

2.service.placeCall(address, extras == null ? new Bundle() : extras,

                        mContext.getOpPackageName());
```

ITelecomService 的 **placeCall** 是在 TelecomServiceImpl 中实现的。

TelecomServiceImpl

placeCall 中分别处理 isSelfManaged 电话和 hasCallPrivilegedPermission

```
的电话
final Intent intent = new Intent(hasCallPrivilegedPermission ?
                                Intent.ACTION_CALL_PRIVILEGED :
Intent.ACTION_CALL, handle);
然后执行：
mUserCallIntentProcessorFactory.create(mContext, userHandle)
                                .processIntent(
                                        intent, callingPackage,
isSelfManaged ||
                                                (hasCallAppOp &&
hasCallPermission),
                                        true /* isLocalInvocation */);
```

UserCallIntentProcessorFactory.create 生成一个 UserCallIntentProcessor，因此执行的是 UserCallIntentProcessor.processIntent。

```
三种电话的 Intent：
if (Intent.ACTION_CALL.equals(action) ||
                Intent.ACTION_CALL_PRIVILEGED.equals(action) ||
                Intent.ACTION_CALL_EMERGENCY.equals(action)) {
            processOutgoingCallIntent(intent, callingPackageName,
canCallNonEmergency,
                    isLocalInvocation);
        }
```

**processOutgoingCallIntent：**

```
sendIntentToDestination(intent, isLocalInvocation);
```

**sendIntentToDestination：**

```
对 local 电话和非 local 电话的处理：

/**

     * Potentially trampolines the intent to the broadcast receiver that

runs only as the primary

     * user.  If the caller is local to the Telecom service, we send the

intent to Telecom without

     * rebroadcasting it.

     */

 if (isLocalInvocation)

TelecomSystem.getInstance().getCallIntentProcessor().processIntent(in

tent);

else

mContext.sendBroadcastAsUser(intent, UserHandle.SYSTEM);
```

用来对于 local 电话 TelecomSystem.getInstance().getCallIntentProcessor()得到了一个 CallIntentProcessor 对象，然后调用其 **processIntent**

CallIntentProcessor

**processIntent**

```
if (isUnknownCall) {

            processUnknownCallIntent(mCallsManager, intent);未知电话

的处理

        } else {

            processOutgoingCallIntent(mContext, mCallsManager,
intent);

        }
```

processOutgoingCallIntent

```
// Send to CallsManager to ensure the InCallUI gets kicked off before the

broadcast returns

        Call call = callsManager

                .startOutgoingCall(handle, phoneAccountHandle,

clientExtras, initiatingUser,

                        intent);



        if (call != null) {

            sendNewOutgoingCallIntent(context, call, callsManager,

intent);

        }
```

这里通过 CallsManager.startOutgoingCall start InCallUI，然后再 broadcast。

CallsManager

startOutgoingCall

```
先得到 PhoneAccount

PhoneAccount account = mPhoneAccountRegistrar.getPhoneAccount

Create a call with original handle

call = new Call

call.setIsSelfManaged(isSelfManaged)

判断设置 videoState   call.setVideoState(videoState)

call.setTargetPhoneAccount(phoneAccountHandle);

判断是否需要 if (needsAccountSelection)   The outgoing call can be placed, go
```

forward.

```
call.setState(
                    CallState.CONNECTING,
                    phoneAccountHandle == null ? "no-handle" :
phoneAccountHandle.toString());
```

RTT 的设置

```
call.createRttStreams();
```

最后　addCall(call);

addcall

```
for (CallsManagerListener listener : mListeners) {
listener.onCallAdded(call);
```

InCallController 是其一 listener

```
public void onCallAdded(Call call) {
        if (!isBoundAndConnectedToServices()) {
            Log.i(this, "onCallAdded: %s; not bound or connected.",
call);
            // We are not bound, or we're not connected.
            bindToServices(call);
```

bindToServices：

```
if (defaultDialerComponentInfo != null &&
                    !defaultDialerComponentInfo.getComponentName().eq
uals(
                            mSystemInCallComponentName)) {
                dialerInCall = new
InCallServiceBindingConnection(defaultDialerComponentInfo);
```

```
            }
```

InCallServiceBindingConnection：

```
private class InCallServiceBindingConnection extends

InCallServiceConnection {



        private final ServiceConnection mServiceConnection = new

ServiceConnection() {

            @Override

            public void onServiceConnected(ComponentName name, IBinder

service) {

              ...

                        if (mIsConnected) {

                            // Only proceed if we are supposed to be

connected.

                            onConnected(service);

                   ...

                }

            }
```

onConnected

```
IInCallService inCallService =

IInCallService.Stub.asInterface(service);

        mInCallServices.put(info, inCallService);
        try {
            inCallService.setInCallAdapter(
                    new InCallAdapter(
                            mCallsManager,
                            mCallIdMapper,
                            mLock,

info.getComponentName().getPackageName()));


for (Call call : calls) {
            try {
                if ((call.isSelfManaged()
&& !info.isSelfManagedCallsSupported()) ||
                        (call.isExternalCall()
&& !info.isExternalCallsSupported())) {
                    continue;
                }


                // Only send the RTT call if it's a UI in-call service
                boolean includeRttCall =
info.equals(mInCallServiceConnection.getInfo());
```

```
                // Track the call if we don't already know about it.
                addCall(call);
                numCallsSent += 1;

inCallService.addCall(ParcelableCallUtils.toParcelableCall(
                        call,
                        true /* includeVideoProvider */,
                        mCallsManager.getPhoneAccountRegistrar(),
                        info.isExternalCallsSupported(),
                        includeRttCall));
```

先执行 inCallService.setInCallAdapter

```
public void setInCallAdapter(IInCallAdapter inCallAdapter) {
            mHandler.obtainMessage(MSG_SET_IN_CALL_ADAPTER,
inCallAdapter).sendToTarget();
        }

case MSG_SET_IN_CALL_ADAPTER:
                    String callingPackage =
getApplicationContext().getOpPackageName();
                    mPhone = new Phone(new
InCallAdapter((IInCallAdapter) msg.obj), callingPackage,
getApplicationContext().getApplicationInfo().targetSdkVersion);

                    mPhone.addListener(mPhoneListener);

                    onPhoneCreated(mPhone);

                    break;

```
会 new 一个 Phone

再执行 inCallService.**addCall**

```
@Override

        public void addCall(ParcelableCall call) {

            mHandler.obtainMessage(MSG_ADD_CALL,

call).sendToTarget();

        }



case MSG_ADD_CALL:

                    mPhone.internalAddCall((ParcelableCall) msg.obj);

                    break;
```

调用刚刚创建的 Phone.internalAddCall

```
final void internalAddCall(ParcelableCall parcelableCall) {

        Call call = new Call(this, parcelableCall.getId(),

mInCallAdapter,

                parcelableCall.getState(), mCallingPackage,

mTargetSdkVersion);

        mCallByTelecomCallId.put(parcelableCall.getId(), call);

        mCalls.add(call);
        checkCallTree(parcelableCall);

        call.internalUpdate(parcelableCall, mCallByTelecomCallId);

        fireCallAdded(call);

     }
```

New 一个 Call，然后 **mCalls**.add(call);最后 **fireCallAdded**

```
private void fireCallAdded(Call call) {

        for (Listener listener : mListeners) {

            listener.onCallAdded(this, call);

        }

    }
```

CallList 是其一 listener

```
onCallAdded

 final DialerCall call =

        new DialerCall(context, this, telecomCall, latencyReport, true

/* registerCallback */);
```

**new** 了一个 DialerCall

DialerCall 初始化好后，CallList 收到 onDialerCallUpdate

```
@Override

    public void onDialerCallUpdate() {

      Trace.beginSection("CallList.onDialerCallUpdate");

      onUpdateCall(call);

      notifyGenericListeners();

      Trace.endSection();
    }
notifyGenericListeners
listener.onCallListChange(this);
```

InCallPresenter 注册了 **onCallListChange**

```
newState = startOrFinishUi(newState);
/**
   * When the state of in-call changes, this is the first method to get
called. It determines if the
   * UI needs to be started or finished depending on the new state and
does it.
   */
  private InCallState startOrFinishUi(InCallState newState)

if (showCallUi || showAccountPicker) {
      LogUtil.i("InCallPresenter.startOrFinishUi", "Start in call UI");
      showInCall(false /* showDialpad */, !showAccountPicker /*
newOutgoingCall */);

public void showInCall(boolean showDialpad, boolean newOutgoingCall) {
    LogUtil.i("InCallPresenter.showInCall", "Showing InCallActivity");
    context.startActivity(
        InCallActivity.getIntent(context, showDialpad,
newOutgoingCall, false /* forFullScreen */));
```

```
  }
```

这样就启动了 InCallActivity，至此 start InCallUI 完毕

## 第二部分 Dial Action

第二部分主要是把 dial 下传到 RIL 及以下

在之前的 CallIntentProcessor 的 processOutgoingCallIntent 中

```
// Send to CallsManager to ensure the InCallUI gets kicked off before the
broadcast returns
        Call call = callsManager
                .startOutgoingCall(handle, phoneAccountHandle,
clientExtras, initiatingUser,
                        intent);

        if (call != null) {    //现在 call ! = null
            sendNewOutgoingCallIntent(context, call, callsManager,
intent);
        }
```

**sendNewOutgoingCallIntent**

```
NewOutgoingCallIntentBroadcaster broadcaster = new
NewOutgoingCallIntentBroadcaster(
                context, callsManager, call, intent,
callsManager.getPhoneNumberUtilsAdapter(),
                isPrivilegedDialer);
final int result = broadcaster.processIntent();
```

**New** 一个 NewOutgoingCallIntentBroadcaster 再调用其 **processIntent**

NewOutgoingCallIntentBroadcaster **processIntent**

```
/**

     * Processes the supplied intent and starts the outgoing call

broadcast process relevant to the

     * intent.

     *

     * This method will handle three kinds of actions:

     *处理三种 Call Action

     * - CALL (intent launched by all third party dialers)

     * - CALL_PRIVILEGED (intent launched by system apps e.g. system

Dialer, voice Dialer)

     * - CALL_EMERGENCY (intent launched by lock screen emergency dialer)

if (sendNewOutgoingCallBroadcast) {

            UserHandle targetUser = mCall.getInitiatingUser();

            Log.i(this, "Sending NewOutgoingCallBroadcast for %s to %s",

mCall, targetUser);

            broadcastIntent(intent, number, !callImmediately,

targetUser);

        }
```

**对三种 call Action 进行处理后,发送广播 broadcastIntent**

```
/**

     * Sends a new outgoing call ordered broadcast so that third party
```

```
apps can cancel the
     * placement of the call or redirect it to a different number.
 private void broadcastIntent
 mContext.sendOrderedBroadcastAsUser(
                broadcastIntent,
                targetUser,
                android.Manifest.permission.PROCESS_OUTGOING_CALLS,
                AppOpsManager.OP_PROCESS_OUTGOING_CALLS,
                receiverRequired ? new
NewOutgoingCallBroadcastIntentReceiver() : null,
                null,  // scheduler
                Activity.RESULT_OK,  // initialCode
                number,  // initialData: initial value for the result
data (number to be modified)
                null);  // initialExtras
发送广播 Intent.ACTION_NEW_OUTGOING_CALL
当收到广播时 onReceive
 placeOutgoingCallImmediately(mCall, resultHandleUri, gatewayInfo,
                            mIntent.getBooleanExtra(

TelecomManager.EXTRA_START_CALL_WITH_SPEAKERPHONE, false),
mIntent.getIntExtra(TelecomManager.EXTRA_START_CALL_WITH_VIDEO_STATE,
                                    VideoProfile.STATE_AUDIO_ONLY));
                }
placeOutgoingCallImmediately 会调用 CallsManager 的 placeOutgoingCall
mCallsManager.placeOutgoingCall(call, handle, gatewayInfo,
speakerphoneOn, videoState);
```

CallsManager.placeOutgoingCall

```
if (call.getTargetPhoneAccount() != null || call.isEmergencyCall()) {
            // If the account has been set, proceed to place the outgoing
call.
            // Otherwise the connection will be initiated when the
account is set by the user.
            if (call.isSelfManaged() && !isOutgoingCallPermitted) {

notifyCreateConnectionFailed(call.getTargetPhoneAccount(), call);
            } else {
                if (call.isEmergencyCall()) {
                    // Drop any ongoing self-managed calls to make way
for an emergency call.
                    disconnectSelfManagedCalls("place emerg call" /*
reason */);
                }
                call.startCreateConnection(mPhoneAccountRegistrar);
            }
```

调用 Call.startCreateConnection 开始创建 Connection

```
void startCreateConnection(PhoneAccountRegistrar
phoneAccountRegistrar) {
        if (mCreateConnectionProcessor != null) {
            Log.w(this, "mCreateConnectionProcessor in
startCreateConnection is not null. This is" +
                    " due to a race between
NewOutgoingCallIntentBroadcaster and " +
                    "phoneAccountSelected, but is harmlessly resolved
by ignoring the second " +
                    "invocation.");
            return;
        }
        mCreateConnectionProcessor = new
CreateConnectionProcessor(this, mRepository, this,
                phoneAccountRegistrar, mContext);
        mCreateConnectionProcessor.process();
    }
```

New 一个 CreateConnectionProcessor，再调用其 process

```
if (mCall.getTargetPhoneAccount() != null) {
            mAttemptRecords.add(new CallAttemptRecord(
                    mCall.getTargetPhoneAccount(),
mCall.getTargetPhoneAccount()));
        }
 mAttemptRecordIterator = mAttemptRecords.iterator();
        attemptNextPhoneAccount();
```

**attemptNextPhoneAccount：**

```
 mService.createConnection(mCall,CreateConnectionProcessor.this);
 private ConnectionServiceWrapper mService;
```

调用 ConnectionServiceWrapper 的 createConnection

```
mBinder.bind(callback, call);
```

ServiceBinder

```
ServiceConnection connection = new ServiceBinderConnection(call);
public void onServiceConnected

if (binder != null) {
                        mServiceDeathRecipient = new
ServiceDeathRecipient(componentName);
                        try {

binder.linkToDeath(mServiceDeathRecipient, 0);
```

```
                            mServiceConnection = this;

                            setBinder(binder);

                            handleSuccessfulConnection();
```

**setBinder**

```
 mBinder = binder;setServiceInterface(binder);
```

**setServiceInterface 在 ConnectionServiceWrapper 中实现**

```
@Override

    protected void setServiceInterface(IBinder binder) {

        mServiceInterface =

IConnectionService.Stub.asInterface(binder);

        Log.v(this, "Adding Connection Service Adapter.");

        addConnectionServiceAdapter(mAdapter);

    }


private void addConnectionServiceAdapter(IConnectionServiceAdapter

adapter) {

        if (isServiceValid("addConnectionServiceAdapter")) {

            try {

                logOutgoing("addConnectionServiceAdapter %s",

adapter);
mServiceInterface.addConnectionServiceAdapter(adapter,

Log.getExternalSession());

            } catch (RemoteException e) {

            }

        }

    }
```

会调用 **ConnectionService.addConnectionServiceAdapter**

```
 mHandler.obtainMessage(MSG_ADD_CONNECTION_SERVICE_ADAPTER,

args).sendToTarget();


case MSG_ADD_CONNECTION_SERVICE_ADAPTER:

  mAdapter.addAdapter(adapter);

mAdapter  : private final ConnectionServiceAdapter mAdapter = new

ConnectionServiceAdapter();

ConnectionServiceAdapter.addAdapter :

 if (mAdapters.add(adapter)) {

            try {

                adapter.asBinder().linkToDeath(this, 0);

            } catch (RemoteException e) {

                mAdapters.remove(adapter);

            }

        }
```

至此 ServiceBinder 的 **setBinder** 执行完成
然后执行 ServiceBinder 的 **handleSuccessfulConnection**

```
private void handleSuccessfulConnection() {

        for (BindCallback callback : mCallbacks) {

            callback.onSuccess();

        }

        mCallbacks.clear();

    }
```

回调 ConnectionServiceWrapper 的 createConnection 的 BindCallback()

的 onSuccess() ，会执行：

```
 mServiceInterface.createConnection(

                            call.getConnectionManagerPhoneAccount(),

                            callId,

                            connectionRequest,

                            call.shouldAttachToExistingConnection(),

                            call.isUnknown(),

                            Log.getExternalSession());
```

又转到 ConnectionService，调用其 **createConnection**

```
mHandler.obtainMessage(MSG_CREATE_CONNECTION, args).sendToTarget();

case MSG_CREATE_CONNECTION:

 createConnection(

                                connectionManagerPhoneAccount,

                                id,
                                request,

                                isIncoming,

                                isUnknown);
```

**createConnection 方法中执行 onCreateOutgoingConnection，onCreateOutgoingConnection 的实现在其子类 TelephonyConnectionService 中：**

```
 connection = isUnknown ? onCreateUnknownConnection(callManagerAccount,

request)

                    : isIncoming ?

onCreateIncomingConnection(callManagerAccount, request)

                    : onCreateOutgoingConnection(callManagerAccount,

request);
```

**TelephonyConnectionService 的 onCreateOutgoingConnection**

```
// Get the right phone object from the account data passed in.

            final Phone phone =

getPhoneForAccount(request.getAccountHandle(), isEmergencyNumber);

            Connection resultConnection =

getTelephonyConnection(request, numberToDial,

                isEmergencyNumber, handle, phone);

//getTelephonyConnection 返回的 resultConnection

if (resultConnection instanceof TelephonyConnection) {

                if (request.getExtras() != null &&

request.getExtras().getBoolean(
                         TelecomManager.EXTRA_USE_ASSISTED_DIALING,

false)) {

                     ((TelephonyConnection)

resultConnection).setIsUsingAssistedDialing(true);

                 }

                 placeOutgoingConnection((TelephonyConnection)

resultConnection, phone, request);

             }

return resultConnection;
```

用 **getTelephonyConnection 得到** resultConnection ：

```
 final TelephonyConnection connection =

                 createConnectionFor(phone, null, true /* isOutgoing */,

request.getAccountHandle(),

                         request.getTelecomCallId(),

request.getAddress(), request.getVideoState());


connection.setAddress(handle, PhoneConstants.PRESENTATION_ALLOWED);

        connection.setInitializing();

        connection.setVideoState(request.getVideoState());

        connection.setRttTextStream(request.getRttTextStream());

        return connection;
```

通过 **createConnectionFor 获得** connection

```
if (phoneType == TelephonyManager.PHONE_TYPE_GSM) {
            returnConnection = new GsmConnection(originalConnection,
telecomCallId, isOutgoing);
```

New 一个 GsmConnection， **return** returnConnection;

GsmConnection 是 TelephonyConnection 的子类，GsmConnection 调用 super 方法得到一个 TelephonyConnection，至此 **getTelephonyConnection 完成。**

**然后调用 placeOutgoingConnection**

```
 if (phone != null) {
                originalConnection = phone.dial(number, new
ImsPhone.ImsDialArgs.Builder()
                        .setVideoState(videoState)
                        .setIntentExtras(extras)
                        .setRttTextStream(connection.getRttTextStream
())
                        .build());
            }
```

phone.dial，虽然这里的 phone 是 Phone 类型，但 Phone 中并没有 dial 方法，GsmCdmaPhone 是 Phone 的子类，其中有 dial 方法，因此会调用 GsmCdmaPhone 的 dial 方法，GsmCdmaPhone 的 dial 方法会调用 CarrierConfigManager 获得一些配置 useImsForCall、useImsForEmergency、isUt、useImsForUt

```
if (isPhoneTypeGsm()) {
            return dialInternal(dialString, new DialArgs.Builder<>()
                    .setIntentExtras(dialArgs.intentExtras)

                    .build());

dialInternal：

return mCT.dial(newDialString);

public GsmCdmaCallTracker mCT;

因此调用 GsmCdmaCallTracker 的 dial

 mCi.dial

即调用 RIL 的 dial
```

经过复杂的 bind 过程，ConnectionServiceWrapper.createConnection bind 为 ConnectionService.createConnection,从 Telecom service 到 telecom framework，ConnectionService 在 createConnection 后调用 TelephonyConnectionServiceon 的 CreateOutgoingConnection，创建出 GsmConnection，TelephonyConnection，然后 TelephonyConnectionServiceon 执行 placeOutgoingConnection，调用了 GsmCdmaPhone 的 dial，
再 GsmCdmaCallTracker.dial (mCT.dial),最后调 mCi.dial，将拨号下传到 RIL 底层。至此第二部分完毕。

## 第三部分 Update Call State to Dialing

当创建 Connection 完成后会通知上层 InCallUI 更新。

当 Connection 创建完成后 ConnectionService 的 **createConnection** 方法里的

**mAdapter**.handleCreateConnectionComplete 将被调用

即执行 ConnectionServiceAdapter 的 handleCreateConnectionComplete

```
 adapter.handleCreateConnectionComplete(id, request, connection,

                              Log.getExternalSession());

回到 ConnectionServiceWrapper 内部类 Adapter 执行

handleCreateConnectionComplete：
```

```
 ConnectionServiceWrapper.this
                              .handleCreateConnectionComplete(callId,
request, connection);
```

**可知执行的是** ConnectionServiceWrapper 自己的 handleCreateConnectionComplete：

```
// Successful connection
            if (mPendingResponses.containsKey(callId)) {
                mPendingResponses.remove(callId)
                            .handleCreateConnectionSuccess(mCallIdMapper,
connection);
```

搜索 handleCreateConnectionSuccess，发现被定义在三个地方：

CreateConnectionResponse.java

CreateConnectionProcessor.java

Call.java

CreateConnectionResponse 是接口，CreateConnectionProcessor 和 Call 都实现了这个接口

发现调用的是 CreateConnectionProcessor 的 handleCreateConnectionSuccess：

```
 mCallResponse.handleCreateConnectionSuccess
```

而 private CreateConnectionResponse mCallResponse;

因此传到 Call，调用 Call 的 handleCreateConnectionSuccess：

```
 case CALL_DIRECTION_OUTGOING:
```

```
                for (Listener l : mListeners) {
                    l.onSuccessfulOutgoingCall(this,


getStateFromConnectionState(connection.getState()));
                }
```

CallsManager 是其监听者

CallsManager#onSuccessfulOutgoingCall：

```
 markCallAsDialing(call)
```

markCallAsDialing 做了如下处理：

设置 CallState，是否开启扬声器、是否关闭了声音

```
void markCallAsDialing(Call call) {
        setCallState(call, CallState.DIALING, "dialing set
explicitly");
        maybeMoveToSpeakerPhone(call);
        maybeTurnOffMute(call);
        ensureCallAudible();
    }
```

setCallState 为 CallState.DIALING 后 listener.onCallStateChanged(call, oldState, newState);

此时监听者 InCallController 收到了这个消息，然后 updateCall(call);

inCallService.updateCall(parcelableCall);继而调用 InCallService 的 updateCall：

```
 mHandler.obtainMessage(MSG_UPDATE_CALL, call).sendToTarget();

case MSG_UPDATE_CALL:

                    mPhone.internalUpdateCall((ParcelableCall)

msg.obj);
```

调用 Phone 的 internalUpdateCall：

```
call.internalUpdate(parcelableCall, mCallByTelecomCallId);
```

转到调用 Call 的 internalUpdate

```
if (stateChanged) {

            fireStateChanged(mState);

        }


private void fireStateChanged(final int newState) {

        for (CallbackRecord<Callback> record : mCallbackRecords) {

            final Call call = this;

            final Callback callback = record.getCallback();

            record.getHandler().post(new Runnable() {

                @Override

                public void run() {

                    callback.onStateChanged(call, newState);

                }

            });

        }
```

```
    }
DialerCall 定义了大量的 Call 的 CallBack：
 private final Call.Callback telecomCallCallback =
      new Call.Callback() {
        @Override
        public void onStateChanged(Call call, int newState) {
          LogUtil.v("TelecomCallCallback.onStateChanged", "call=" +
call + " newState=" + newState);
          update();
        }
update 中 updateFromTelecomCall(); setState(translatedState);
```

然后监听者 CallList 收到 onDialerCallUpdate，继而 onUpdateCall，然后通知注册者 InCallPresenter 做相应的改变。至此第三部分完毕。

## 第四部分 Update Call State

当拨号下发到 RIL 底层，底层会主动上报相关状态的改变，要求上层做一些响应，更新 InCallUI。

拨号下传到 RIL 底层后，底层主动上报 Call State 改变了，通知 RIL，RIL 通知其注册者 GsmCdmaCallTracker 会要求得到当前 call，getCurrentCalls，经 RIL 下发给底层，底层有请求的返回结果给 GsmCdmaCallTracker，GsmCdmaCallTracker 收到后调用 handlePollCalls：

```
 if (hasNonHangupStateChanged || newRinging != null ||
hasAnyCallDisconnected) {
            mPhone.notifyPreciseCallStateChanged();
            updateMetrics(mConnections);
        }
```

调用 GsmCdmaPhone notifyPreciseCallStateChanged：

```
 super.notifyPreciseCallStateChangedP();
```

从而调用父类 Phone 的 notifyPreciseCallStateChangedP

```
protected void notifyPreciseCallStateChangedP() {

        AsyncResult ar = new AsyncResult(null, this, null);

        mPreciseCallStateRegistrants.notifyRegistrants(ar);

        mNotifier.notifyPreciseCallState(this);

    }
```

通知注册者 TelephonyConnection

```
case MSG_PRECISE_CALL_STATE_CHANGED:

                    Log.v(TelephonyConnection.this,

"MSG_PRECISE_CALL_STATE_CHANGED");

                    updateState();

                    break;


void updateState() {

        if (mOriginalConnection == null) {

            return;

        }

        updateStateInternal();

        updateStatusHints();

        updateConnectionCapabilities();

        updateConnectionProperties();

        updateAddress();
        updateMultiparty();

        refreshDisableAddCall();

    }
```

**updateStateInternal：**

```
                case DIALING:

                case ALERTING:

                    if (mOriginalConnection != null &&
mOriginalConnection.isPulledCall()) {

                        setPulling();

                    } else {

                        setDialing();

                    }

                    break;
```

然后 setDialing，setDialing 没有在 TelephonyConnection 中实现，而是在其父类 Connection 中实现了：

```
public final void setDialing() {

        checkImmutable();

        setState(STATE_DIALING);

    }
```

setState：

```
 onStateChanged(state);
```

```
            for (Listener l : mListeners) {
                l.onStateChanged(this, state);
            }
```

ConnectionService 的 Connection.Listener 监听到 onStateChanged：

```
case Connection.STATE_DIALING:
                    mAdapter.setDialing(id);
                    break;
```

继而调用 ConnectionServiceAdapter 的 setDialing：

```
void setDialing(String callId) {
        for (IConnectionServiceAdapter adapter : mAdapters) {
            try {
                adapter.setDialing(callId, Log.getExternalSession());
            } catch (RemoteException e) {
            }
        }
    }
```

继而调用 ConnectionServiceWrapper 内部类 Adapter 的 **setDialing**：

```
 if (call != null) {
                        mCallsManager.markCallAsDialing(call);
```

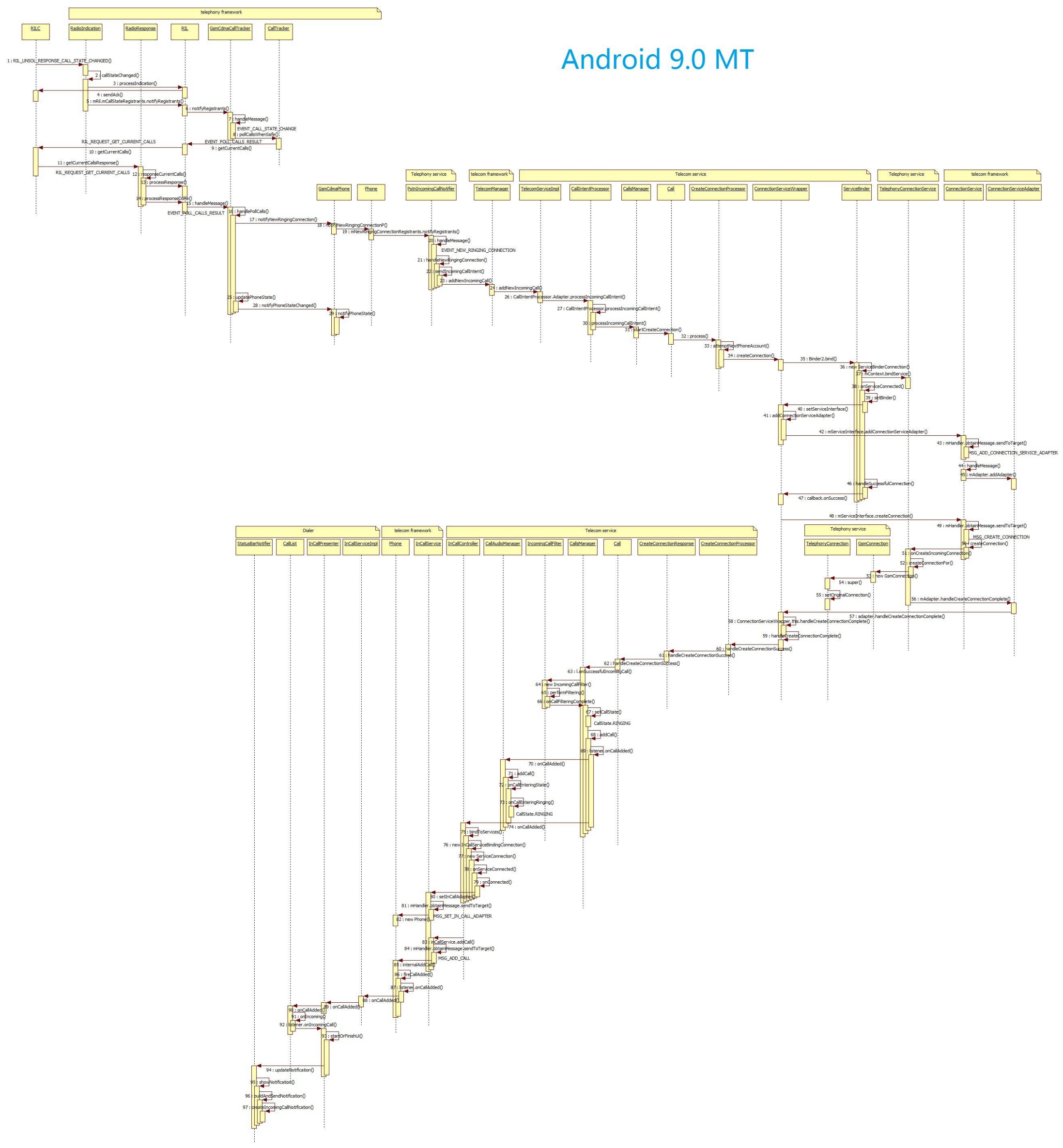
因此调用 **CallsManager** 的 markCallAsDialing

```
void markCallAsDialing(Call call) {
        setCallState(call, CallState.DIALING, "dialing set
explicitly");
```

```
        maybeMoveToSpeakerPhone(call);

        maybeTurnOffMute(call);

        ensureCallAudible();

    }
```

然后就跟之前第三部分一样了，根据底层上报的信息，上层做相应的改变，然后更新 InCallUI。

## Android 9.0 MT 流程



当下层收到来电时，下层向 RadioIndication 发送消息 RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED，RadioIndication 收到消息后，执行 callStateChanged

```
public void callStateChanged(int indicationType) {

        mRil.processIndication(indicationType);



        if (RIL.RILJ_LOGD)
mRil.unsljLog(RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED);


        mRil.mCallStateRegistrants.notifyRegistrants();

    }
```

调用 RIL 中 processIndication 向下层发送应答

```
void processIndication(int indicationType) {

        if (indicationType == RadioIndicationType.UNSOLICITED_ACK_EXP)

{

            sendAck();

            if (RILJ_LOGD) riljLog("Unsol response received; Sending ack

to ril.cpp");

        } else {

            // ack is not expected to be sent back. Nothing is required

to be done here.

        }

    }
```

然后 RIL 的 CallStateRegistrants 通知注册者电话状态改变 GsmCdmaCallTracker 在收到消息 EVENT_CALL_STATE_CHANGE 后调用 CallTracker 执行 pollCallsWhenSafe，pollCallsWhenSafe 调用 RIL 执行 getCurrentCalls，RIL 再向下层请求 getCurrentCalls，下层返回结果完成后调用 GsmCdmaCallTracker 的 handlePollCalls

```
 if (newRinging != null) {

            mPhone.notifyNewRingingConnection(newRinging);

        }
```

这里的 mPhone 是 GsmCdmaPhone，GsmCdmaPhone 其后调用的是 Phone 的

notifyNewRingingConnectionP 来通知注册者

然后 updatePhoneState()

Phone 通知注册者 PstnIncomingCallNotifier 事件 EVENT_NEW_RINGING_CONNECTION，PstnIncomingCallNotifier 处理后，调用 TelecomManager 的 addNewIncomingCall，TelecomManager 的 addNewIncomingCall 调用 TelecomServiceImpl 的 addNewIncomingCall

```
 Intent intent = new Intent(TelecomManager.ACTION_INCOMING_CALL);


intent.putExtra(TelecomManager.EXTRA_PHONE_ACCOUNT_HANDLE,

                                   phoneAccountHandle);


intent.putExtra(CallIntentProcessor.KEY_IS_INCOMING_CALL, true);

                           if (extras != null) {

                               extras.setDefusable(true);


intent.putExtra(TelecomManager.EXTRA_INCOMING_CALL_EXTRAS, extras);

                           }


mCallIntentProcessorAdapter.processIncomingCallIntent(

                                   mCallsManager, intent);
```

**mCallIntentProcessorAdapter** 是 CallIntentProcessor 中 interface Adapter

```
public interface Adapter {
        void processOutgoingCallIntent(Context context, CallsManager
callsManager,
                Intent intent);
        void processIncomingCallIntent(CallsManager callsManager,
Intent intent);
        void processUnknownCallIntent(CallsManager callsManager,
Intent intent);
    }


public void processIncomingCallIntent(CallsManager callsManager, Intent
intent) {


CallIntentProcessor.processIncomingCallIntent(callsManager, intent);
        }

```

实际上调用的还是 CallIntentProcessor.processIncomingCallIntent

CallIntentProcessor.processIncomingCallIntent

```
callsManager.processIncomingCallIntent(phoneAccountHandle,
clientExtras);
```

调用 CallsManager 的 processIncomingCallIntent

CallsManager 的 processIncomingCallIntent

```
if (!isHandoverAllowed || (call.isSelfManaged()
&& !isIncomingCallPermitted(call,
                call.getTargetPhoneAccount()))) {
```

```
            notifyCreateConnectionFailed(phoneAccountHandle, call);

        } else {

            call.startCreateConnection(mPhoneAccountRegistrar);

        }
```

调用 Call.startCreateConnection

```
 mCreateConnectionProcessor = new CreateConnectionProcessor(this,

mRepository, this,

                phoneAccountRegistrar, mContext);

        mCreateConnectionProcessor.process();
```

**new** CreateConnectionProcessor，然后 process

```
if (mCall.getTargetPhoneAccount() != null) {

            mAttemptRecords.add(new CallAttemptRecord(

                    mCall.getTargetPhoneAccount(),

mCall.getTargetPhoneAccount()));

        }


 mAttemptRecordIterator = mAttemptRecords.iterator();

        attemptNextPhoneAccount();
```

**attemptNextPhoneAccount**

```
 if (mCall.isIncoming()) {

                    mService.createConnection(mCall,

CreateConnectionProcessor.this);
```

```
private ConnectionServiceWrapper mService;
```

ConnectionServiceWrapper createConnection

```
mBinder.bind(callback, call);
```

bind 到 ConnectionService 的 createConnection

```
 connection = isUnknown ? onCreateUnknownConnection(callManagerAccount,
request)
                    : isIncoming ?
onCreateIncomingConnection(callManagerAccount, request)
                    : onCreateOutgoingConnection(callManagerAccount,
request);
```

当然这里是 onCreateIncomingConnection 在 TelephonyConnectionService 中实现

TelephonyConnectionService onCreateIncomingConnection

```
TelephonyConnection connection =
                createConnectionFor(phone, originalConnection, false
/* isOutgoing */,
                        request.getAccountHandle(),
request.getTelecomCallId(),
                        request.getAddress(), videoState);
```

**createConnectionFor** 中 **new** GsmConnection

```
if (phoneType == TelephonyManager.PHONE_TYPE_GSM) {
            returnConnection = new GsmConnection(originalConnection,
telecomCallId, isOutgoing);
```

然后 TelephonyConnectionService 向 ConnectionServiceAdapter 返回 mAdapter.handleCreateConnectionComplete

ConnectionServiceAdapter 再向 ConnectionServiceWrapper 返回 adapter.handleCreateConnectionComplete
ConnectionServiceWrapper 向 CreateConnectionProcessor 返回 handleCreateConnectionSuccess
CreateConnectionProcessor 再向 CreateConnectionResponse 返回 handleCreateConnectionSuccess
CreateConnectionResponse 再向 Call 返回 handleCreateConnectionSuccess
Call 通知其监听者 CallsManager onSuccessfulIncomingCall

```
@Override

    public void onSuccessfulIncomingCall(Call incomingCall) {

        Log.d(this, "onSuccessfulIncomingCall");

        if

(incomingCall.hasProperty(Connection.PROPERTY_EMERGENCY_CALLBACK_MODE

)) {

            Log.i(this, "Skipping call filtering due to ECBM");

            onCallFilteringComplete(incomingCall, new

CallFilteringResult(true, false, true, true));

            return;

        }



        List<IncomingCallFilter.CallFilter> filters = new

ArrayList<>();

        filters.add(new

DirectToVoicemailCallFilter(mCallerInfoLookupHelper));

        filters.add(new AsyncBlockCheckFilter(mContext, new

BlockCheckerAdapter(),
                mCallerInfoLookupHelper));
        filters.add(new CallScreeningServiceFilter(mContext, this,
mPhoneAccountRegistrar,
                mDefaultDialerCache, new
ParcelableCallUtils.Converter(), mLock));
        new IncomingCallFilter(mContext, this, incomingCall, mLock,
                mTimeoutsAdapter, filters).performFiltering();
    }
```

**new** IncomingCallFilter 用来过滤来电，比如黑名单拒接，完成后调用 IncomingCallFilter 的 onCallFilteringComplete

```
mListener.onCallFilteringComplete(mCall, mResult);
```

通知监听者 CallsManager

CallsManager

```
public void onCallFilteringComplete(Call incomingCall,
CallFilteringResult result)
if (incomingCall.getState() != CallState.DISCONNECTED &&
                incomingCall.getState() != CallState.DISCONNECTING) {
            setCallState(incomingCall, CallState.RINGING,
                    result.shouldAllowCall ? "successful incoming
call" : "blocking call");


if (result.shouldAllowCall) {
            if (hasMaximumManagedRingingCalls(incomingCall)) {
                if (shouldSilenceInsteadOfReject(incomingCall)) {
                    incomingCall.silence();
                } else {
                    Log.i(this, "onCallFilteringCompleted: Call
rejected! " +
                            "Exceeds maximum number of ringing
calls.");
                    rejectCallAndLog(incomingCall);
                }
            } else if (hasMaximumManagedDialingCalls(incomingCall)) {
                Log.i(this, "onCallFilteringCompleted: Call rejected!
Exceeds maximum number of " +
                        "dialing calls.");
                rejectCallAndLog(incomingCall);
            } else {
                addCall(incomingCall);
            }
```

CallsManager 设置好 setCallState 为 CallState.RINGING，然后 addCall(incomingCall)，然后通知监听者 listener.onCallAdded 监听者主要有 CallAudioManager InCallController

CallAudioManager 主要启动响铃相关操作

InCallController 会 bindToServices 到 InCallService,new 一个 phone，然后 fireCallAdded 通知监听者 listener.onCallAdded

然后到 InCallServiceImpl，InCallPresenter，CallList，最后显示来电。