# Telephony模块总结

|  |  |  |
| --- | --- | --- |
| Date  日期 | Version  版本 | Comments  备注 |
| 2018-02-18 | 0.1 | First version |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Function  职位 | Name  姓名 | Date  日期 | Signature  签名 |
| Written by  拟定 | SW | 曾灿炫 | 2018-03-04 | 曾灿炫 |
| Verified by  审核 |  |  |  |  |
| Verified by  审核 |  |  |  |  |
| Approved by  批准 |  |  |  |  |

目录

[Telephony模块总结 1](#_Toc2352992)

[1. 概述 3](#_Toc2352993)

[**1.1.** **本文档的目的** 3](#_Toc2352994)

[**1.2.** **背景** 3](#_Toc2352995)

[**1.3.** **缩略语清单** 3](#_Toc2352996)

[**1.4.** **参考文献** 3](#_Toc2352997)

[2. Telephony整体架构 4](#_Toc2352998)

[2.1. 简介 4](#_Toc2352999)

[2.2.1. 图解 4](#_Toc2353000)

[2.2.2. 架构组成 4](#_Toc2353001)

[2.2. Telephony Framework 5](#_Toc2353002)

[2.2.1. CatService 5](#_Toc2353003)

[2.2.2. RILJ 7](#_Toc2353004)

[2.2.3. 完整流程 8](#_Toc2353005)

[1.5. RIL层 9](#_Toc2353006)

[2.3.1. RILC运行机制 10](#_Toc2353007)

[2.3.2. 消息传递图 13](#_Toc2353008)

[2. 与Modem端的通信方式 14](#_Toc2353009)

[3.1. qcril+qmi 14](#_Toc2353010)

[3.2. 关键流程 14](#_Toc2353011)

[3.2.1. 监听信息 14](#_Toc2353012)

[3.2.2. 运作流程 14](#_Toc2353013)

[3. 总结 14](#_Toc2353014)

# 概述

* 1. **本文档的目的**

*本文档作为一个学习输出，通过分析STK不同层级间命令的传递方式，总结STK模块在Telephony部分的知识。*

* 1. **背景**

*STK即SIM Tool Kit，它提供一系列用于移动设备与SIM卡间交互的机制。通过这些机制，支持STK的手机可以操作SIM卡里的应用。其Framework层部分（主要涉及Telephony）负责消息的转化和传递，实现modem与应用间的交互。*

* 1. **缩略语清单**

*列出文中使用的术语的定义和缩略（语）词的英文全名和中文解释。*

| Term | Explanation |
| --- | --- |
| CAT | Universal Integrated Circuit Card |
| TR | Terminal Response |
| HIDL | HAL interface definition language |

* 1. **参考文献**

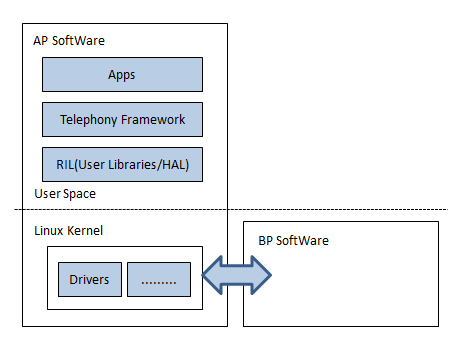
| Document | Explanation |
| --- | --- |
| Android消息机制1-Handler | http://gityuan.com/2015/12/26/handler-message-framework/ |
| Android Telephony 原理解析与开发指南 | 第二版 |
| 《STK模块架构和流程总结(Android M)》 | 洪仰宁 |
| UIM Multi-SIM Design Overview | 80-nh218-1\_c\_uim\_multi-sim\_design\_overview |

# Telephony整体架构

## 简介

### 2.2.1. 图解

目前的手机硬件结构大多采用双处理器架构，分为AP端与BP端。AP端运行开放式操作系统以及各种基于操作系统的应用，BP端负责无线通信基本能力，Telephony的业务跨越了AP与BP，是我们平时工作主要涉及的内容，具体的结构图如图1。



. Android Telephony框架结构

从这个结构图可以看出：

1. Telephony主要运行在Linux Kernel上的用户空间。
2. **与Android系统分层对应**。
3. BP端是处理无线通信部分，即modem部分。

### 2.2.2. 架构组成

**1.** RIL层指的是工程目录下hardware/ril部分，又称RILC，由C代码组成，与上层java进行通信，具体的消息传递方式在2.3部分讲述。

**2.** Telephony Framework分为telecom和telephony，相关目录为:

.../frameworks/base/telephony/java/com/android/internal/telephony/

.../frameworks/base/telecomm/java/com/android/internal/telecom/

.../frameworks/opt/telephony/src/java/com/android/internal/telephony/

Stk模块相关代码为第三个目录下的CatService.java以及RIL.java，其中CatService.java负责RIL层与应用层间的消息传递；RIL.java为RIL层的上层结构，又称RILJ，与Telephony做最直接的交互，作为RIL的消息入口。

**3.**Apps层存放在目录/packages/services/Telephony/与/packages/services/Telecom/，不涉及STK模块。

接下来根据Telephony的分层结构进行单独的分析。

## Telephony Framework

这一部分实现了STK APP与RIL层的信息同步，在modem端上传SIM卡命令后，通过广播的方式将命令发送至APP处理，同时将APP下发的TR传至RILC，最终实现消息的转换与传递。

### CatService

#### 类图



.CatService

CatService继承Handler,实现了AppInterface接口。通过Handler的消息队列原理来接收来自应用层以及RILJ的消息。Handler的运作原理见[图3](#_3._Handler)。

1．STK APP通过调用接口函数onCmdResponse()发送消息，具体代码实现：

public synchronized void onCmdResponse(CatResponseMessage resMsg) {

if (resMsg == null) {

return;

}

// queue a response message.

Message msg = obtainMessage(MSG\_ID\_RESPONSE, resMsg);

msg.sendToTarget();

}

可以看到最终通过sendToTarget将消息发至队列。（这里用法与直接new Message 再调用sendMessage不同，跟进前者代码发现最终走的还是sendMessage，区别在于没有新构造一个消息对象，根据注释解释，前者更有效率）

2．回到CatService的构建函数，可以看到：

// Register ril events handling.

mCmdIf.setOnCatSessionEnd(this, MSG\_ID\_SESSION\_END, null);

mCmdIf.setOnCatProactiveCmd(this, MSG\_ID\_PROACTIVE\_COMMAND, null);

mCmdIf.setOnCatEvent(this, MSG\_ID\_EVENT\_NOTIFY, null);

mCmdIf.setOnCatCallSetUp(this, MSG\_ID\_CALL\_SETUP, null);

//mCmdIf.setOnSimRefresh(this, MSG\_ID\_REFRESH, null);

这里调用了CommandInterface的函数去注册实现观察者模式，具体实现在RILJ的父类BaseCommands当中：

public void setOnCatProactiveCmd(Handler h, int what, Object obj) {

mCatProCmdRegistrant = new Registrant (h, what, obj);

}

这里的Hnalder即为CatService自身。



3. Handler

#### 2.2.1.2. 关键函数

1. getInstance:：向外提供CatService的单例。

2. handlemessage：处理来自APP以及RILJ的消息。

3. sendStartDecodingMessageParams：接口函数，将RIL Message解码成CommandParams Objects。

4. handleRilMsg：处理解码器转换后的RIL层消息。

5. handleCommand：处理具体的命令。

6. handleCmdResponse：处理APP发来的Repsonse。

7. sendTerminalResponse：发送TR。

#### 2.2.1.3. 消息处理

那么CatService的具体消息处理流程图下图：



. CAT流程图

### RILJ

#### 类图



. RILJ

#### 关键函数

**BaseCommands**: 实现CommandsInterface的大部分接口函数，涉及多个RegistrantList与Registrant，向外提供多个事件的观察者模式注册函数。

查看Registrant文件，发现就是对Handler的封装，由注册的一方指定一个需要通知的事件，另一方只需要调用notifyRegistrant()来实现message的创建、填充、发送过程。RegistrantList实现多个注册对象的同时通知。

1. setOnCatProactiveCmd：主动式命令的监听注册，RILJ在接收到RILC的主动式命令消息后，由Registrant向注册时指定的Handler（这里指CatService）发送消息通知。

**RILJ:** 将Telephony的业务封装成RILRequest并发至RILC。

1. sendTerminalResponse：将主动式命令的处理结果封装后发送至RILC。
2. getRadioProxy: 初始化IRadio相关服务。
3. obtainRequest:创建RILRequest对象。
4. processResponse: 处理RILC请求返回的结果。
5. processIndication: 处理RILC上报的消息。

### 2.2.3. 完整流程

以下为主动式命令消息的处理时序图：



主动式命令上报



发送TR

## RIL层

进入RILC的目录，可以看到代码结构分为四部分，分别是include、libril、reference-ril、rild。

Include：

ril.h定义了RIL\_Init、RIL\_register等函数和其他结构体，以及RIL消息RIL\_REQUEST\_XXX和RIL\_UNSOL\_XXX。

libril：

ril\_commands.h、ril\_unsol\_commands.h、ril.cpp、ril\_service.cpp。编译输出libril.so。

reference\_ril：

编译输出libreference-ril.so，实现了RIL AT命令的交互机制。

rild：

主要关注点，编译生成可执行文件rild。

### **2.3.1.** RILC运行机制

#### 2.3.1.1. RIL消息分类

RIL中的消息类型，按照发起方分为Solicited以及UnSolicited。

1. Solicited是由AP端发起的消息，又根据传递方向分为Solicited Request和Solicited Response。
2. UnSolicited是由BP侧主动发起的消息。

这两种消息命名方式为RIL\_REQUEST\_ XXX、RIL\_REQUEST\_XXX，具体的消息定义查看[frameworks](http://10.92.35.16:8080/source/xref/Benz/frameworks/)/[base](http://10.92.35.16:8080/source/xref/Benz/frameworks/base/)/[telephony](http://10.92.35.16:8080/source/xref/Benz/frameworks/base/telephony/)/[java](http://10.92.35.16:8080/source/xref/Benz/frameworks/base/telephony/java/)/[com](http://10.92.35.16:8080/source/xref/Benz/frameworks/base/telephony/java/com/)/[android](http://10.92.35.16:8080/source/xref/Benz/frameworks/base/telephony/java/com/android/)/[internal](http://10.92.35.16:8080/source/xref/Benz/frameworks/base/telephony/java/com/android/internal/)/[telephony](http://10.92.35.16:8080/source/xref/Benz/frameworks/base/telephony/java/com/android/internal/telephony/)/[RILConstants.java](http://10.92.35.16:8080/source/xref/Benz/frameworks/base/telephony/java/com/android/internal/telephony/RILConstants.java)。

通过Radio log可以查看具体的消息类型与传递方向，其中’<’，’>’表示发送的方向。：

**Solicitesd:**

RILJ : [4041]> RIL\_REQUEST\_REPORT\_STK\_SERVICE\_IS\_RUNNING [SUB0]

RILJ : [4041]< RIL\_REQUEST\_REPORT\_STK\_SERVICE\_IS\_RUNNING [SUB0]

**UnSolicited:**

RILJ : [UNSL]< UNSOL\_STK\_PROACTIVE\_COMMAND [SUB0]

#### 消息处理

1. Solicited消息

以2.2.3发送TR为例，RILJ最终调用的IRadio. sendTerminalResponseToSim()，该函数最终在

hardware/ril/libril/ril\_service.cpp中实现：

Return<void> RadioImpl::sendTerminalResponseToSim(int32\_t serial,

const hidl\_string& commandResponse) {

#if VDBG

RLOGD("sendTerminalResponseToSim: serial %d", serial);

#endif

dispatchString(serial, mSlotId, RIL\_REQUEST\_STK\_SEND\_TERMINAL\_RESPONSE,

commandResponse.c\_str());

return Void();

}

其中dispatchString函数：

bool dispatchString(int serial, int slotId, int request, const char \* str) {

RequestInfo \*pRI = android::addRequestToList(serial, slotId, request);

if (pRI == NULL) {

return false;

}

char \*pString;

if (!copyHidlStringToRil(&pString, str, pRI)) {

return false;

}

CALL\_ONREQUEST(request, pString, sizeof(char \*), pRI, slotId);

memsetAndFreeStrings(1, pString);

return true;

}

在RadioImpl中可看到所有的Solicited消息处理最终都调用了dispatchString()。

主要关注addRequestToList与CALL\_ONREQUEST

**addRequestToList:**

RequestInfo \*

addRequestToList(int serial, int slotId, int request) {

RequestInfo \*pRI;

int ret;

RIL\_SOCKET\_ID socket\_id = (RIL\_SOCKET\_ID) slotId;

/\* Hook for current context \*/

/\* pendingRequestsMutextHook refer to &s\_pendingRequestsMutex \*/

pthread\_mutex\_t\* pendingRequestsMutexHook = &s\_pendingRequestsMutex;

/\* pendingRequestsHook refer to &s\_pendingRequests \*/

RequestInfo\*\* pendingRequestsHook = &s\_pendingRequests;

......

pRI = (RequestInfo \*)calloc(1, sizeof(RequestInfo));

if (pRI == NULL) {

RLOGE("Memory allocation failed for request %s", requestToString(request));

return NULL;

}

pRI->token = serial;

**pRI->pCI = &(s\_commands[request]);**

pRI->socket\_id = socket\_id;

ret = pthread\_mutex\_lock(pendingRequestsMutexHook);

assert (ret == 0);

pRI->p\_next = \*pendingRequestsHook;

\*pendingRequestsHook = pRI;

ret = pthread\_mutex\_unlock(pendingRequestsMutexHook);

assert (ret == 0);

return pRI;

}

其中s\_commands[]定义为ril\_commands.h，该头文件定义了RIL请求类型和回调函数，如{RIL\_REQUEST\_STK\_SEND\_TERMINAL\_RESPONSE, radio::sendTerminalResponseToSimResponse}

即RILC会调用sendTerminalResponseToSimResponse作为Solicited Request消息的Response。

**CALL\_ONREQUEST：**

ril\_service.cpp的宏定义为：

#define CALL\_ONREQUEST(a, b, c, d, e) s\_vendorFunctions->onRequest((a), (b), (c), (d))

即CALL\_ONREQUEST将调用s\_vendorFunctions->onRequest, 进入该函数，发现没有对RIL\_REQUEST\_STK\_SEND\_TERMINAL\_RESPONSE做具体的处理，只调用RIL\_onRequestComplete。

进入该函数的定义：

extern "C" void

RIL\_onRequestComplete(RIL\_Token t, RIL\_Errno e, void \*response, size\_t responselen) {

RequestInfo \*pRI;

int ret;

RIL\_SOCKET\_ID socket\_id = RIL\_SOCKET\_1;

pRI = (RequestInfo \*)t;

if (!checkAndDequeueRequestInfoIfAck(pRI, false)) {

RLOGE ("RIL\_onRequestComplete: invalid RIL\_Token");

return;

}

socket\_id = pRI->socket\_id;

appendPrintBuf("[%04d]< %s",

pRI->token, requestToString(pRI->pCI->requestNumber));

if (pRI->cancelled == 0) {

int responseType;

......

int rwlockRet = pthread\_rwlock\_rdlock(radioServiceRwlockPtr);

assert(rwlockRet == 0);

ret = pRI->pCI->responseFunction((int) socket\_id,

responseType, pRI->token, e, response, responselen);

rwlockRet = pthread\_rwlock\_unlock(radioServiceRwlockPtr);

assert(rwlockRet == 0);

}

free(pRI);

}

以上的主要处理流程是：调用checkAndDequeueRequestInfoIfAck找到RIL请求对应的RequestInfo，通过pRI->pCI->responseFunction发起XXXResponse的调用，即上一步**addRequestToList**指定的Solicited消息对应的Response函数。

跟进Telephony framework中RadioResponse里的sendTerminalResponseToSimResponse，可看到最终调用了RILJ的processResponseDone，完成response的发送。

1. UnSolicited消息

UnSolicited类型的消息不确定发起点，从RILJ往下跟进，在RadioIndication.java中可看到函数stkProactiveCommand，用于通知CAT并调用RILJ进行处理。

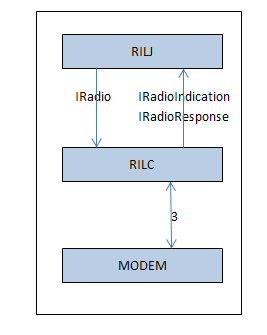
RadioIndication继承IRadioIndication.Stub，进入ril\_service.cpp，radio::stkProactiveCommandInd调用了对应的stkProactiveCommand，实现了消息的传递。

### 2.3.2. 消息传递图

下图即为RIL的消息处理机制，其中RILJ通过rild提供的IRadio服务发送消息，RILC通过phone提供的IRadioResponse和IRadioIndication发送消息，这一部分统称为HIDL，是基于Binder通信实现的。（AIDL，HIDL运作原理还需要进一步学习）

以IRadio为例：IRadio.hal中定义了多个函数接口，如sendTerminalResponseToSim。在ril\_servise中RadioImpl再实现该函数。RILJ通过调用IRadio的接口实现消息的传递。

图中的3是RILC与Modem的交互方式，是由厂商实现，下文了解一下高通的实现方式。



RIL消息传递

# 与Modem端的通信方式

## 3.1. 结构图

下图为高通的双卡结构图，其中RILC分成RIL daemon与vendor RIL(QCRIL)。

即高通通过QCRIL+QMI实现RILC与MODEM的通信。相关文件在路径vendor/qcom/proprietary/。



. RIL结构图

## QCRIL+QMI

### vendor RIL 初始化

rild.c主函数main中获取了vendor RIL(QCRIL)的库以及初始化入口方法，并进行初始化与注册：

int main(int argc, char \*\*argv) {

.......

int i;

const char \*clientId = NULL;

RLOGD("\*\*RIL Daemon Started\*\*");

RLOGD("\*\*RILd param count=%d\*\*", argc);

umask(S\_IRGRP | S\_IWGRP | S\_IXGRP | S\_IROTH | S\_IWOTH | S\_IXOTH);

**//1.获取QCRIL路径**

for (i = 1; i < argc ;) {

if (0 == strcmp(argv[i], "-l") && (argc - i > 1)) {

rilLibPath = argv[i + 1];

i += 2;

} else if (0 == strcmp(argv[i], "--")) {

i++;

hasLibArgs = 1;

break;

} else if (0 == strcmp(argv[i], "-c") && (argc - i > 1)) {

clientId = argv[i+1];

i += 2;

} else {

usage(argv[0]);

}

}

.......

**//2.打开vendor ril库**

dlHandle = dlopen(rilLibPath, RTLD\_NOW);

**//3.启动消息循环**

RIL\_startEventLoop();

**//4.获取vendor RIL的初始化入口方法**

rilInit =

(const RIL\_RadioFunctions \*(\*)(const struct RIL\_Env \*, int, char \*\*))

dlsym(dlHandle, "RIL\_Init");

......

**//5.调用vendor RIL的RIL\_Init方法，此方法会返回RIL\_RadioFunctions**

funcs = rilInit(&s\_rilEnv, argc, rilArgv);

RLOGD("RIL\_Init rilInit completed");

**//6.注册vendor RIL的处理方法**

RIL\_register(funcs);

......

}

#### 传入回调函数

上文中提到的**funcs = rilInit(&s\_rilEnv, argc, rilArgv)**实际上调用了QCRIL的RIL\_Init，传入了s\_rilEnv，具体结构为：

static struct RIL\_Env s\_rilEnv = {

RIL\_onRequestComplete,

RIL\_onUnsolicitedResponse,

RIL\_requestTimedCallback,

RIL\_onRequestAck

};;

即传入rilc的方法供QCRIL回调。

#### 获取QCRIL的函数接口

查看QCRIL的RIL\_Init，主要执行了以下几个函数：

qmi\_ril\_set\_thread\_name( pthread\_self() , QMI\_RIL\_QMI\_RILD\_THREAD\_NAME);

// Initialize the event thread \*/

qcril\_event\_init();

// Initialize QCRIL

qcril\_init();

// Start event thread

qcril\_event\_start();

// start bootup if applicable

qmi\_ril\_initiate\_bootup();

return &qcril\_request\_api[ QCRIL\_DEFAULT\_INSTANCE\_ID ];

即初始化以及返回接口函数。

1. 其中qcril\_event\_int为初始化EventLoop线程，通过查看qcril\_event\_main的注释可知EventLoop用于读取并处理QMI的消息。
2. 通过return &qcril\_request\_api[ QCRIL\_DEFAULT\_INSTANCE\_ID ]返回函数列表供RILC使用。

### 消息传递

#### 以RILC->QMI为例

在QCRIL初始化时提供的qcril\_request\_api具体定义如下：

static const RIL\_RadioFunctions qcril\_request\_api[] = {

{ QCRIL\_RIL\_VERSION, onRequest\_rid, currentState\_rid, onSupports\_rid, onCancel\_rid, getVersion\_rid }

};

当RIL层有请求时，RILC调用其中的onRequest\_rid：

static void onRequest\_rid

(

int request,

void \*data,

size\_t datalen,

RIL\_Token t

)

{

**onRequest( qmi\_ril\_process\_instance\_id, request, data, datalen, t );**

}

实际上调用了onRequest，查看onRequest发现调用qcril\_dispatch\_event将消息发至对应的handler:

/\* Dispatch the request to the appropriate handler \*/

(entry\_ptr->handler)(params\_ptr, &ret);

这里entry\_ptr指的是qcril\_dispatch\_table\_entry\_type，该结构定义了各种消息类型下对应的调用方法，以发送TR为例：

{ QCRIL\_REG\_ALL\_STATES( RIL\_REQUEST\_STK\_SEND\_TERMINAL\_RESPONSE, qcril\_gstk\_qmi\_request\_stk\_send\_terminal\_response ) }

查看该方法发现调用了QMI的qmi\_client\_send\_msg\_async，进一步调用qmi\_service\_msg发送消息至modem。

具体的时序图如下：



RILC->QMI

消息从QMI到QCRIL的处理过程与上图类似：将消息发至QCRIL的队列、QCRIL调用qcril\_dispatch\_event将消息发至对应的handler，qcril\_dispatch\_table\_entry\_type这个结构不仅定义了上层消息对应的处理方法，还涉及modem发至QCRIL的各种消息。

# 总结

Telephony层涉及到的STK部分基本都是消息的传递，包括广播，Handler，HIDL。

RILJ以上的部分源码相对来说不难阅读和理解，且资料多。QCRIL+QMI部分仅根据log跟进了消息的传递。

以一张图总结整体的消息传递方式：

