

Android 的电话部分



Android 的电话部分

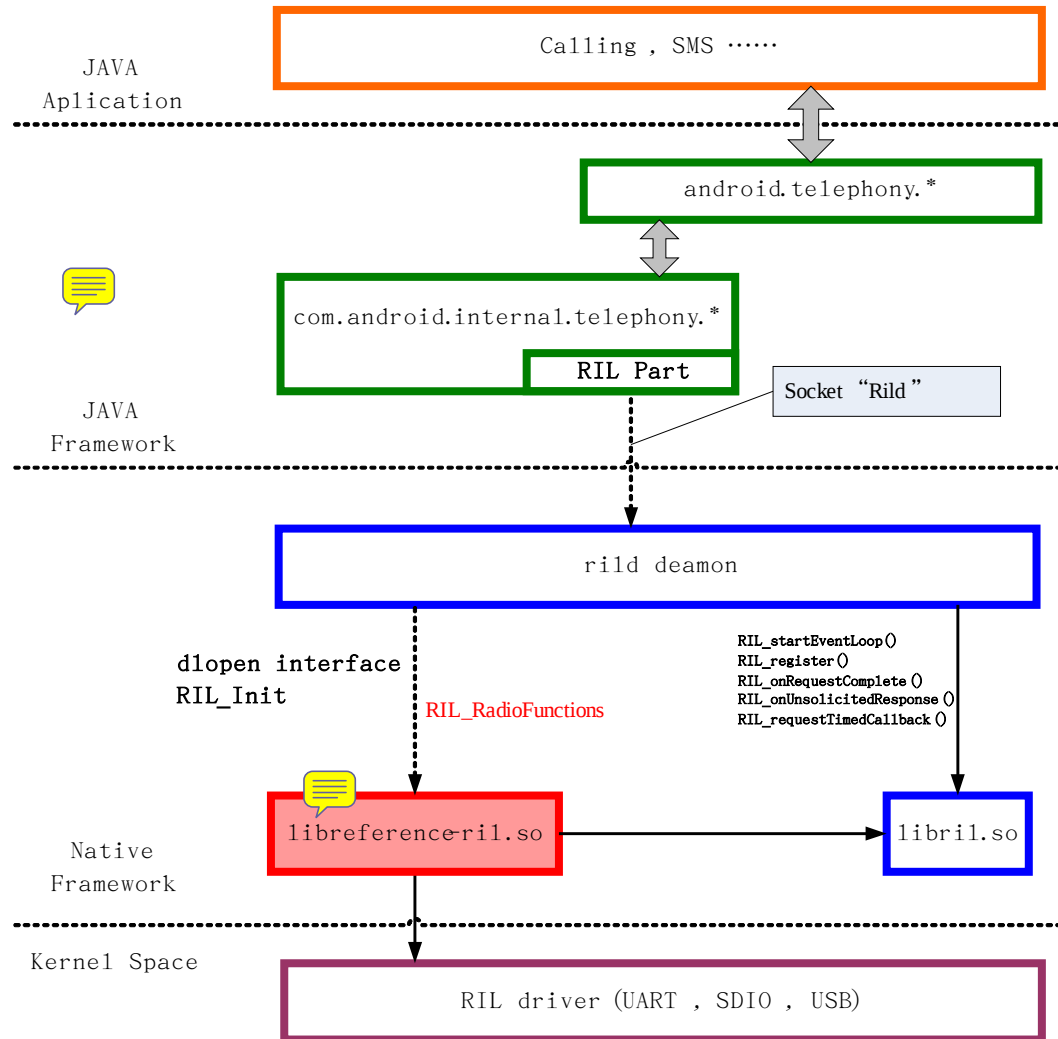
- 第一部分 Android 电话部分的综述
- 第二部分 Modem 驱动
- 第三部分 Android 电话的本地 RIL 层
- 第四部分 电话部分 JAVA 框架及应用

第一部分 Android 电话部分的结构

Android 作为一个智能手机平台，电话（**Telephony**）部分的功能自然是重点。电话部分的主要功能是呼叫（**Call**）、短信（**Sms**）、数据连接（**Data Connection**）以及**SIM**卡、电话本等功能。本章将介绍绝大多数功能的实现框架。

Android 电话部分主要分为：**Modem**驱动、**RIL**（**Radio Interface Layer**）、电话服务框架、应用 4 层结构。

第一部分 Android 电话部分的结构



第二部分 Modem 驱动

实现电话功能的主要硬件是通信模块（**Modem**），**Modem** 通过与通信网络进行沟通传输语音及数据，完成呼叫、短信等相关电话功能。

对于大部分目前的独立通信模块而言，无论是 **2G** 还是 **3G** 都已经非常成熟，模块化相当完善，硬件接口非常简单，也有着相对统一的软件接口。一般的 **Modem** 模块装上 **SIM** 卡，直接上电即可工作，自动完成初始的找网、网络注册等工作，完成之后即可打电话、发短信等。但独立模块因为体积问题，在手机设计中较少使用，而是使用 **chip-on-board** 的方式。另外也有不少 **Modem** 基带与应用处理器共存。

第三部分 Android 电话的本地 RIL 层

3.1 简介

3.2 RILD 守护进程

3.3 libril 库

3.4 RIL 的实现库 Reference RIL

3.5 Request（请求）流程

3.6 Response（响应）流程

3.7 RIL 的移植工作

第一部分 Android 电话部分的结构

Radio Interface Layer (RIL) 提供了电话服务和的 **radio** 硬件之间的抽象层。 **RIL** 负责数据的可靠传输、 **AT** 命令的发送以及 **response** 的解析。应用处理器通过 **AT** 命令集与带 **GPRS** 功能的无线通讯模块通信。

AT command 由 **Hayes** 公司发明，是一个调制解调器制造商采用的一个调制解调器命令语言，每条命令以字母 "**AT**" 开头。

第三部分 Android 电话的本地 RIL 层

本地代码：

RIL 支持的本地代码包括 `ril` 库和守护进程：

[hardware/ril/include](#)

[hardware/ril/libril](#)

[hardware/ril/rild](#)

[hardware/ril/reference-ril](#)

编译结果是

`/system/bin/rild`：守护进程

`/system/lib/libril.so`：RIL 的库

`/system/lib/libreference-ril.so`：RIL 参考库

3.1 简介

[hardware/ril/include](#) 目录中的 `ril.h` 头文件是 Android 的 RIL 框架的结构和接口，包括各种数据结构，枚举值，定义各种以 `RIL_` 开头的命令整数值。宏 `RIL_SHLIB` 用于区分这个头文件在不同地方的定义。

```
#ifndef RIL_SHLIB
struct RIL_Env {
    void (*OnRequestComplete)(RIL-Token t, RIL_Errno e,
                              void *response, size_t responselen);
    void (*OnUnsolicitedResponse)(int unsolResponse, const void *data,
                                   size_t datalen);
    void (*RequestTimedCallback) (RIL_TimedCallback callback,
                                   void *param,
                                   const struct timeval *relativeTime);
};
const RIL_RadioFunctions *RIL_Init(const struct RIL_Env *env,
                                    int argc, char **argv);

#else /* RIL_SHLIB */
void RIL_register (const RIL_RadioFunctions *callbacks);
void RIL_onRequestComplete(RIL-Token t, RIL_Errno e,
                           void *response, size_t responselen);
void RIL_onUnsolicitedResponse(int unsolResponse, const void *data,
                               size_t datalen);
void RIL_requestTimedCallback (RIL_TimedCallback callback,
                               void *param,
                               const struct timeval *relativeTime);

#endif /* RIL_SHLIB */
```

3.2 RILD 守护进程

在 **android** 的 **RIL** 结构中，**libril.so** 是一个辅助功能的库，它为 **libreference-ril.so** 和 **rild** 使用。

rild 是一个守护进程，它动态打开（**dlopen**），**RIL** 的功能库，进入事件循环。

libreference-ril.so 是实现 **RIL** 功能的库，其主要的功能是 **AT** 命令的解析，当使用不同的硬节接口或者不同功能的硬件的时候，主要需要重写本库。

3.2 RILD 守护进程

rild 是一个守护进程，在这里宏 **RIL_SHLIB** 被定义。执行的过程为获取参数 -> 打开功能库 -> 建立事件循环（线程） -> 执行 **RIL_Init** ->

```
int main(int argc, char **argv)
{
    /* 获取参数并解析 */
    dlHandle = dlopen(rilLibPath, RTLD_NOW);
    /* 启动线程，进入事件循环 */
    RIL_startEventLoop();
    rilInit = (const RIL_RadioFunctions (*)(const struct RIL_Env *, int, char
**))
        dlsym(dlHandle, "RIL_Init");
    /* 处理参数 */
    funcs = rilInit(&s_rilEnv, argc, rilArgv);
    RIL_register(funcs);
done:
    while(1) {
        sleep(0x00ffffff);
    }
}
```

init.rc 中，启动本守护进程，如果使用 **-l** 可以指定所使用的功能库

```
service ril-daemon /system/bin/rild
    socket rild stream 660 root radio
    socket rild-debug stream 660 radio system
    user root
    group radio cache inet misc
```

3.3 libril 库

libril.so 主要提供了用于注册的 `RIL_register()` 以及 `RIL_onRequestComplete()`, `RIL_onUnsolicitedResponse()`, `RIL_requestTimedCallback()` 三个函数在 RIL 处理过程中需要处理的函数, 它们作用相当于回调函数。

另外定义了下面的几个用于字符串转换的函数, 供 RIL 的功能库中使用。

```
extern "C" const char * requestToString(int request);  
extern "C" const char * failCauseToString(RIL_Errno);  
extern "C" const char * callStateToString(RIL_CallState);  
extern "C" const char * radioStateToString(RIL_RadioState);
```

3.4 RIL 的实现库 Reference RIL

`libreference-ril.so` 是 RIL 的功能库，在这里宏 `RIL_SHLIB` 被定义。其中，实现 `RIL_Init()` 函数，将 `RIL_Env` 传入环境的三个函数指针保留，在需要处理的场合调用，返回一个 `RIL_RadioFunctions` 类型的函数指针，用于注册。

```
static const RIL_RadioFunctions s_callbacks = {
    RIL_VERSION,
    onRequest,
    currentState,
    onSupports,
    onCancel,
    getVersion
};
static const struct RIL_Env *s_rilenv;
pthread_t s_tid_mainloop;
const RIL_RadioFunctions *RIL_Init(const struct RIL_Env *env,
                                   int argc, char **argv)
{
    /* ..... */
    pthread_attr_t attr;
    s_rilenv = env;
    /* 参数处理 */
    pthread_attr_init (&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    ret = pthread_create(&s_tid_mainloop, &attr, mainLoop, NULL);
    return &s_callbacks;
}
```

3.5 Request（请求）流程

对于请求流程，首先从 Java 层通过 Socket 将命令发送到 RIL 层的 RILD 守护进程，RILD 守护进程中，负责监听的 `ril_event_loop` 消息循环中的 `Select` 发现 RILD Socket 有了请求链接信号，会建立起一个 `record_stream`，打通与上层的数据通道并开始接收请求数据。数据通道的回调函数 `processCommandsCallback()` 会保证收到一个完整的 Request 后（Request 包的完整性由 `record_stream` 的机制保证），将其送达 `processCommandBuffer()` 函数。这是命令的下发流程。

对 Request 流程来说，它是由 Java 中的 `RIL.java` 发起的。

3.6 Response（响应）流程

Response（响应）有两类：**unsolicited** 表示主动上报的消息，如来电、来短信等；而 **solicited** 是 **AT** 命令的响应。判断是否是 **solicited** 的依据有两点：一是当前有 **AT** 命令正在等待响应；二是读取到的响应符合该 **AT** 命令的响应格式。对 **Response** 流程来讲，流程是从 **Modem** 设备发回响应数据开始的。

RILD 通过 **readerLoop()** 函数，利用 **readline** 逐行读取响应数据，随后通过 **processLine** 进行分析（与短信相关的响应比较独特，要特殊处理）。主动上报一般以 **+XXXX** 的形式出现，而 **AT** 命令的响应格式则有一行或多行之分，最终一定以 **OK** 或 **ERROR** 结尾。

3.6 Response（响应）流程

ProcessLine 有以下几种情形：

（a）没有 **AT** 命令正在等待响应或不符合 **AT** 响应格式，一般是主动上报行，由 **handleUnsolicited** 处理，**handleUnsolicited()**→**onUnsolicited()**→**RIL_onUnsolicitedResponse()**。

（b）**isFinalResponseSuccess/isFinalResponseError** 是最终响应行，转到 **handleFinalResponse** 处理，**handleFinalResponse** 会发送线程同步信号，激活等待的发送线程。如前文提到的 **requestDial**，将从 **at_send_command** 返回，调用 **RIL_onRequestComplete** 处理响应。

（c）符合当前 **AT** 命令响应格式行，解析并获取数据，这是响应处理的中间过程，之后会继续收到最终响应行，进入（b）流程。

3.6 Response (响应) 流程

ProcessLine 有以下几种情形:

(a) 没有 **AT** 命令正在等待响应或不符合 **AT** 响应格式, 一般是主动上报行, 由 **handleUnsolicited** 处理, **handleUnsolicited()**→**onUnsolicited()**→**RIL_onUnsolicitedResponse()**。

(b) **isFinalResponseSuccess/isFinalResponseError** 是最终响应行, 转到 **handleFinalResponse** 处理, **handleFinalResponse** 会发送线程同步信号, 激活等待的发送线程。如前文提到的 **requestDial**, 将从 **at_send_command** 返回, 调用 **RIL_onRequestComplete** 处理响应。

(c) 符合当前 **AT** 命令响应格式行, 解析并获取数据, 这是响应处理的中间过程, 之后会继续收到最终响应行, 进入 (b) 流程。

3.7 RIL 的移植工作

根据各个系统的硬件差别，在移植的过程中主要的工作是实现一个类似 `libreference-ril.so` 的库或者应用。

当宏 **RIL_SHLIB** 被定义的时候，将使用库的形式；没有被定义的时候，将使用守护进程的方式（在这种情况下，将不需要 `rild`）。

RIL 移植主要需要考虑的问题：

- RIL 设备所使用的不同端口
- 在 `RIL_RadioFunctions` 的 `onRequest` 函数中需要处理的不同命令

第四部分 电话部分 JAVA 框架及应用

4.1 基本架构

4.2 呼叫

4.3 短信

4.4 数据连接

4.5 其他框架部分及其他应用

4.1 基本架构

Android 电话部分 JAVA 框架的代码路径在
[frameworks/base/telephony/java/](#)

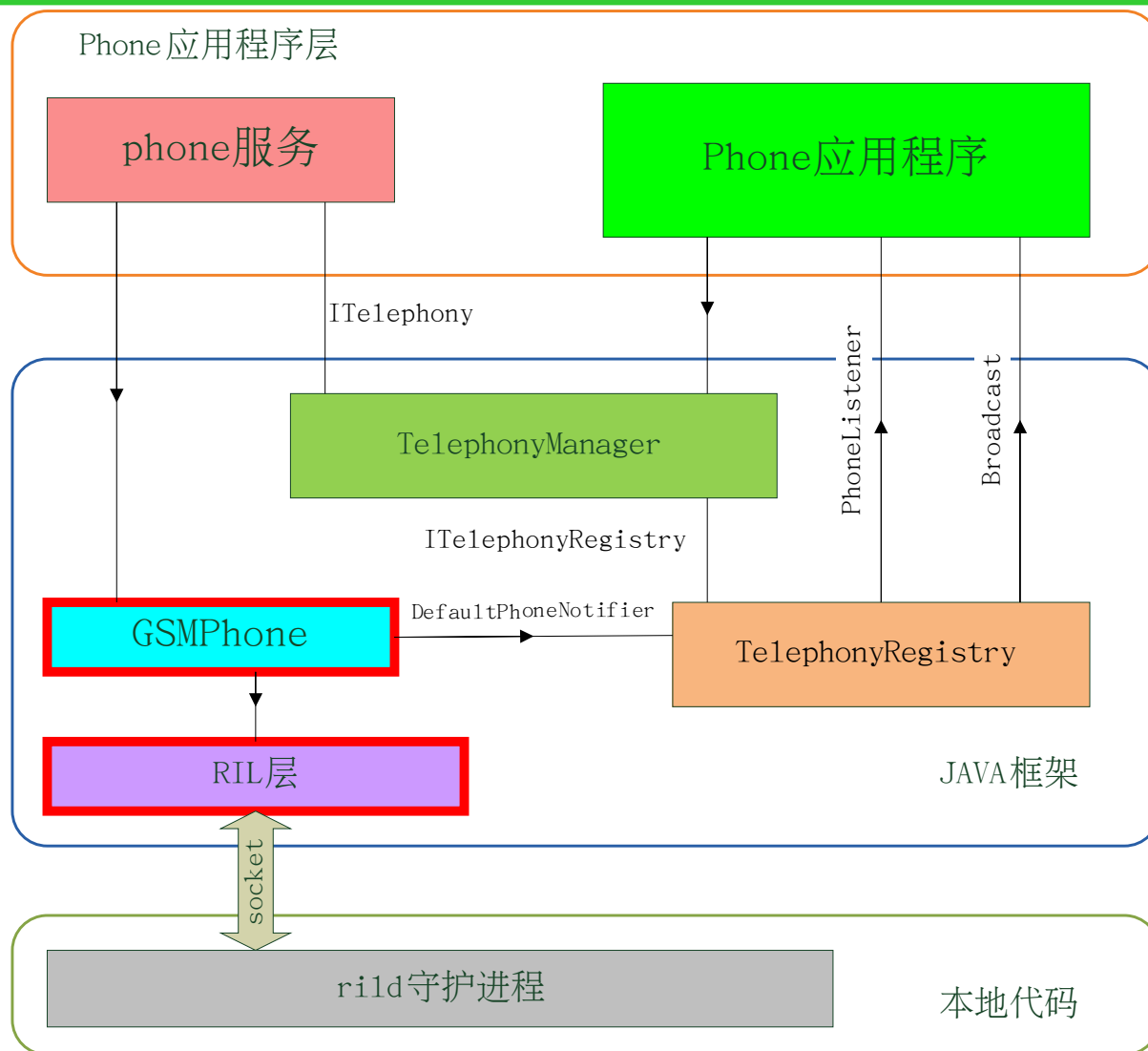
在这个目录 JAVA 类为:

[android/telephony](#)

[com/android/internal/telephony](#)

前者实现了 JAVA 类 `android.telephony` 以及 `android.telephony.gsm` (以及 `android.telephony.cdma`) , 后者实现了内部的类 `com.android.internal.telephony` 及 `com.android.internal.telephony.gsm` (以及 `com.android.internal.telephony.cdma`) , 带 `gsm` 的是 GSM 协议专用, 而不带的是通用部分。

4.1 基本架构



4.1 基本架构

Java 层与 RIL 本地代码的接口，是名为“rild”的 Socket，该 Socket 是电话服务命令的收发通道；而负责处理与 RIL 本地代码通信的是 RIL.java。

RIL.java 中实现了几个类来完成相应的操作，其中 RILRequest 代表一个电话服务命令请求，RIL.RILSender 负责处理命令的发送，RIL.RILReceiver 用于处理命令响应以及主动上报信息的接收，RILConstants.java 则定义了电话服务的具体命令。

而响应和主动上报消息的标准流程是：

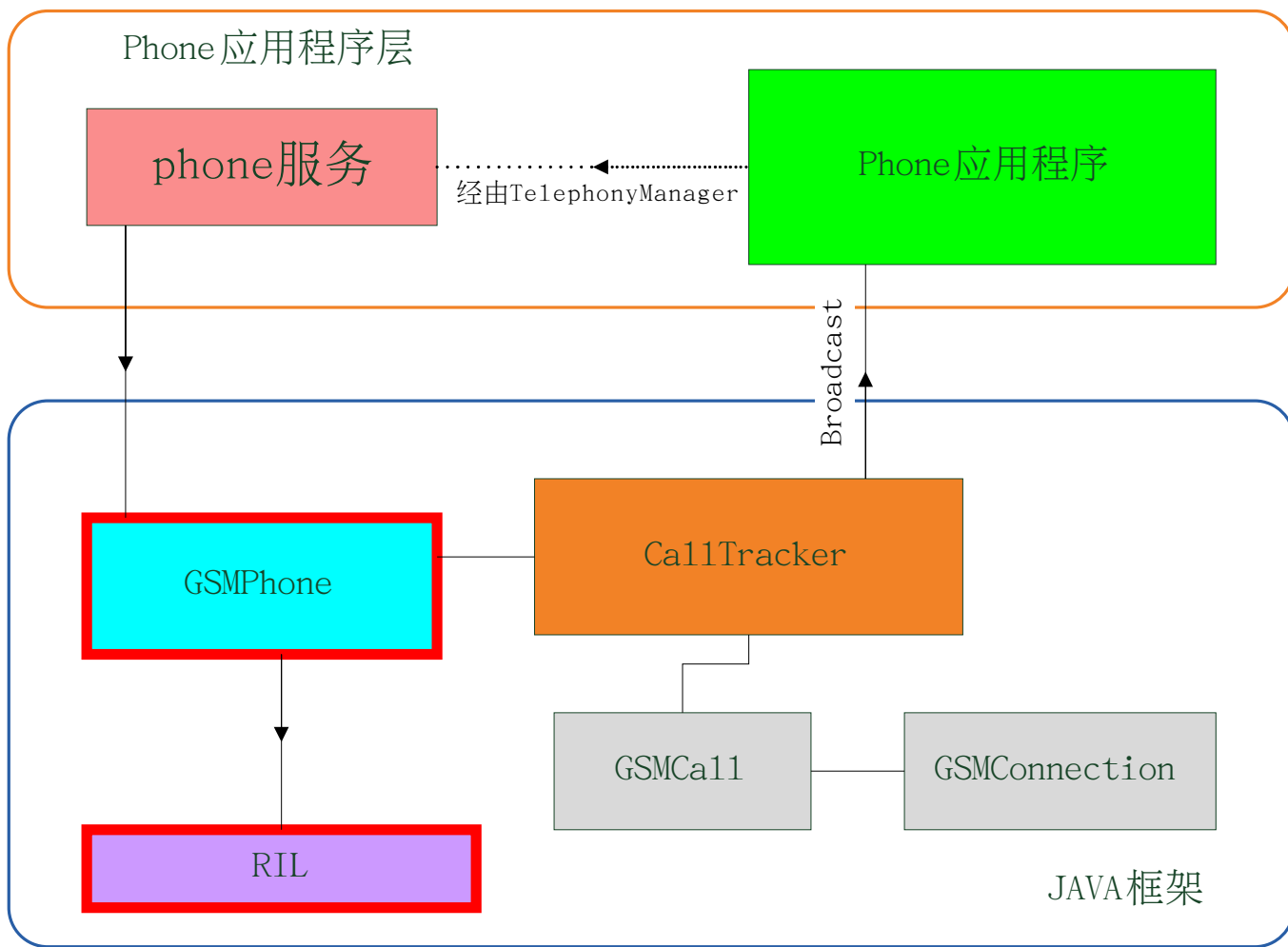
RILReceiver 线程监视 mSocket input

→ readRilMessage（读取完整响应）

→ processResponse

→ 分别处理 RESPONSE_UNSOLICITED 与 RESPONSE_SOLICITED（前者一般为主动上报，后者为命令响应）。

4.2 呼叫

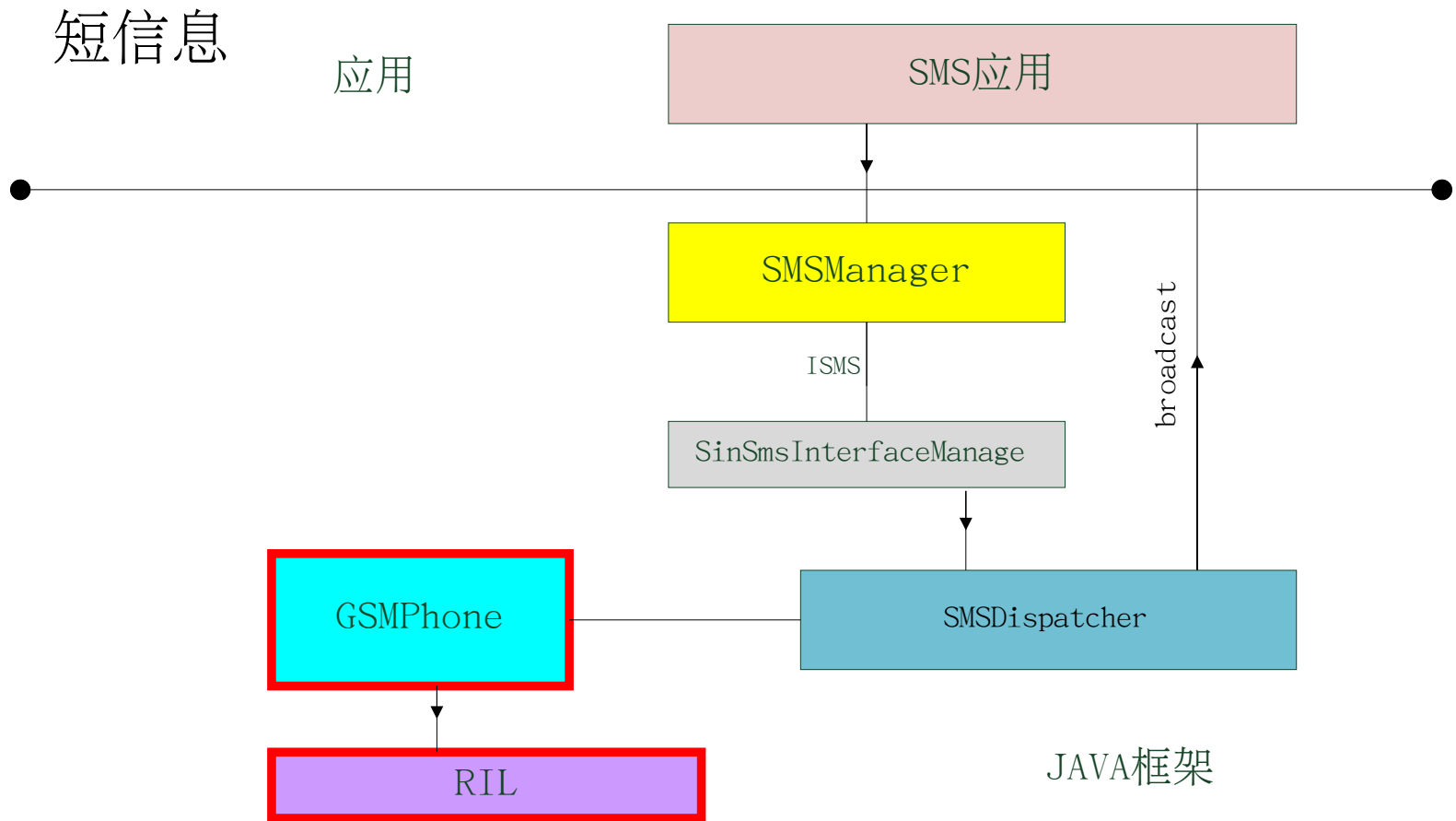


4.2 呼叫

呼叫 (Call) 构建于电话服务的基本架构之上。

与呼叫相关的主要用户接口，其实就是基于 ITelephony 接口实现在 Phone 应用中的 “phone” 服务，通过 TelephonyManager 提供访问接口。此服务内部通过 PhonyFactory 获取的 GSMPhone 来访问 RIL，提供诸如拨号、接通、挂断、保持通话等功能。

4.3 短信



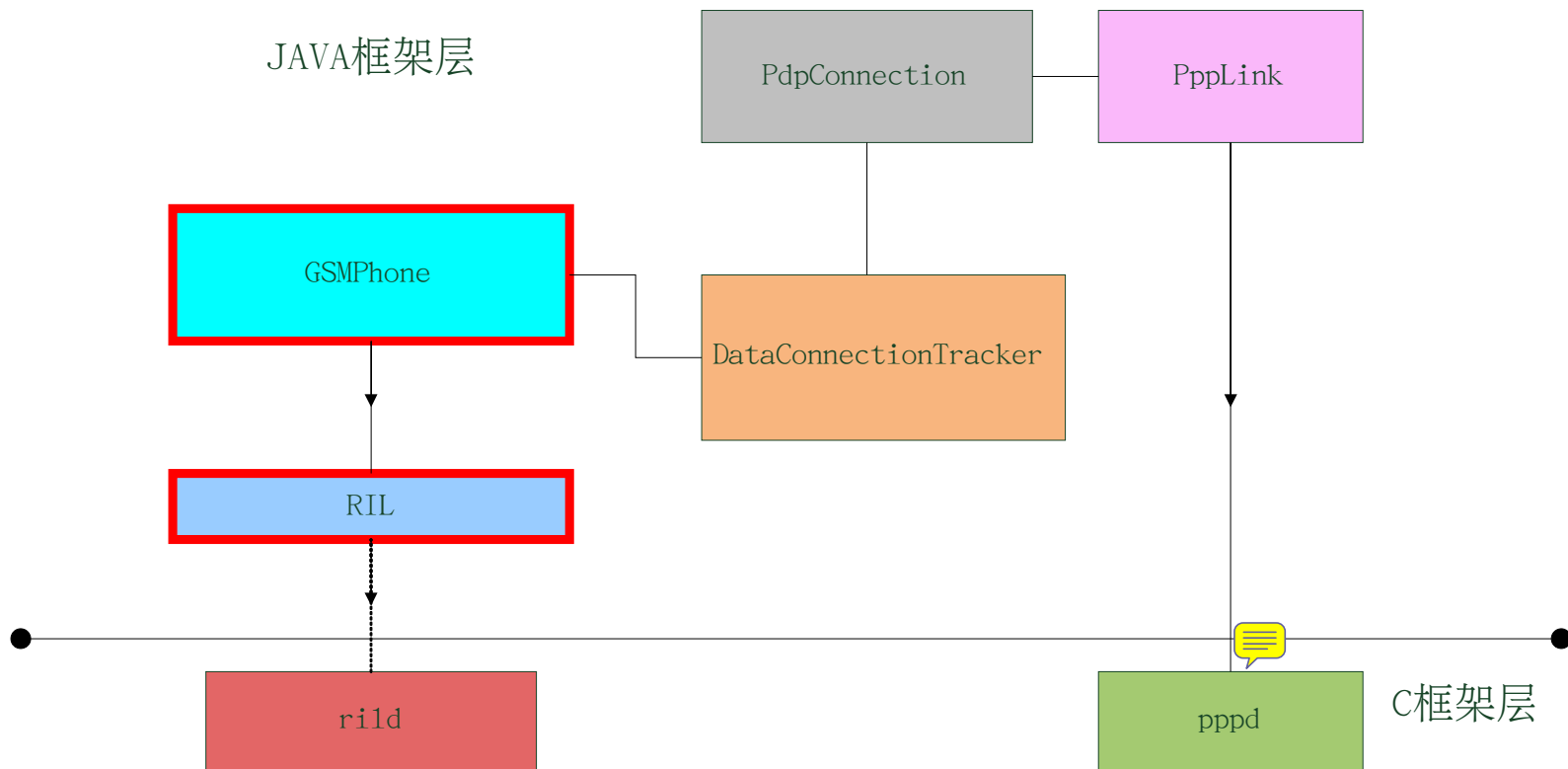
4.3 短信

SMSManager 实现短信发送以及与 SIM 卡短信相关的操作，通过 ISms 接口提供对应的实现。ISms 的服务器端实现是 SimSmsInterfaceManager（在 GSM 类下，如果是 CDMA 则使用 RuimSmsInterfaceManager），SimSmsInterfaceManager 中关于短信发送的重要部分主要由 SMSDispatcher 提供支持。

SMSDispatcher 是短信部分的核心，提供发送 SMS 等操作接口，同时也提供接收 SMS 和返回报告等接口，它同样被集成到 GSMPhone 中。

4.4 数据连接

数据业务



4.4 数据连接

Android 的数据连接通常使用 PPP 方式提供，Android 将 pppd 移植到 ARM 平台以支持此特性，生成 pppd 守护进程供需要时开启。

数据连接主要分两个步骤：首先通过 AT 命令激活 PDP 连接，再利用 pppd 通过数据端口完成拨号连接。

数据连接的核心控制类是

`DataConnectionTracker`，`GSMPhone` 拥有其实例。数据连接一般不需要用户太多干预，设置好 APN 后，在适当的情况下就会自动激活。

激活的入口点是：

`DataConnectionTracker.trySetupData`

→ `setupData`

→ `PdpConnection.connect`

→ `CommandsInterface.setupDefaultPDP`，通过

`PdpConnection` 访问 `GSMPhone` 中 RIL 层的 `setupDefaultPDP` 实现。

4.5 其他框架部分及其他应用

电话部分比较庞大，除了呼叫、短信和数据连接几种关键应用模式，还有 **SIM** 卡、电话本、联系人、**USSD**、**STK**、网络选择等众多功能。对于应用部分，除了 **Phone** 和 **Mms**，也广泛存在于 **Contacts**、**Settings** 乃至很多用到电话相关服务的应用当中。



谢谢！