# STK Framework模块架构

|  |  |  |
| --- | --- | --- |
| Date  日期 | Version  版本 | Comments  备注 |
| 2018-12-20 | 0.1 | First version |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Function  职位 | Name  姓名 | Date  日期 | Signature  签名 |
| Written by  拟定 | SW | 曾灿炫 | 2018-12-20 | 曾灿炫 |
| Verified by  审核 |  |  |  |  |
| Verified by  审核 |  |  |  |  |
| Approved by  批准 |  |  |  |  |

目录

[STK Framework模块架构 1](#_Toc533264066)

[1. 概述 3](#_Toc533264067)

[**1.1.** **本文档的目的** 3](#_Toc533264068)

[**1.2.** **背景** 3](#_Toc533264069)

[**1.3.** **缩略语清单** 3](#_Toc533264070)

[**1.4.** **参考文献** 3](#_Toc533264071)

[2. UICC框架 4](#_Toc533264072)

[2.1. 简介 4](#_Toc533264073)

[2.2. 关键类 4](#_Toc533264074)

[2.3. CatService初始化过程 5](#_Toc533264075)

[2.3.1 UiccController实例化： 5](#_Toc533264076)

[2.3.2 UiccCard实例化： 5](#_Toc533264077)

[2.3.3 CatService实例化： 6](#_Toc533264078)

[3. Framework流程 7](#_Toc533264079)

[3.1 关键函数 7](#_Toc533264080)

[3.2 关键流程 8](#_Toc533264081)

[3.2.1 监听信息 8](#_Toc533264082)

[3.2.2 运作流程 8](#_Toc533264083)

[4. 总结 9](#_Toc533264084)

# 概述

* 1. **本文档的目的**

*本文档作为一个学习输出，总结STK模块在Framework层（CAT）的架构与运作流程。*

* 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层负责消息的转化和传递，实现RIL层与应用层间的交互。*

* 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 |
| --- | --- |
| UICC | Universal Integrated Circuit Card |
| TR | Terminal Response |
|  |  |

* 1. **参考文献**

| Document | Explanation |
| --- | --- |
| Android中的Uicc框架 | https://blog.csdn.net/u010961631/article/category/2430627 |
|  |  |
|  |  |

# UICC框架

## 简介

UICC包括多种逻辑应用，例如用户标识模块（SIM，Subscriber Identity Module）、通用用户标识模块（USIM，Universal Subscriber Identity Module）、IP多媒体业务标识模块（ISIM，IP Multimedia Service Identity Module）。即手机使用的SIM卡和USIM卡，对于这些不同类型的卡，Android提供了一套系统来抽象管理，详情可见/frameworks/opt/telephony/src/java/com

/android/internal/telephony/uicc目录下的UICC框架。

## 关键类

在UiccController.java中，可以看到该框架的类图，如下：

\* Following is class diagram for uicc classes:

\*

\* UiccController

\* #

\* |

\* UiccCard

\* # #

\* | ------------------

\* UiccCardApplication CatService

\* # #

\* | |

\* IccRecords IccFileHandler

\* ^ ^ ^ ^ ^ ^ ^ ^

\* SIMRecords---- | | | | | | ---SIMFileHandler

\* RuimRecords----- | | | | ----RuimFileHandler

\* IsimUiccRecords--- | | -----UsimFileHandler

\* | ------CsimFileHandler

\* ----IsimFileHandler

根据源码内容及相关注释，其主要部分如下：

1. UiccController.java:

----Uicc的总控制接口，向外提供UiccCard、UiccCardApplication、IccRecords、IccFileHandler对象

2. UiccCard.java:

----向外提供UiccCardApplication对象，并完成**CatService的初始化**

3. UiccCardApplication.java:

----创建并向外提供IccFileHandler、IccRecords对象

4. IccFileHandler.java:

----负责SIM卡文件系统的读写

5. IccRecords.java:

----保存SIM卡常用信息，包括SIM联系人、语音信箱等

6. CatService.java:

----**负责STK业务**

## CatService初始化过程

### **2.3.1** UiccController实例化：

在Phone模块的初始化过程中，调用了UiccController.java的静态方法make(Context c, CommandsInterface[] ci):

//PhoneFactory->makeDefaultPhone

// Instantiate UiccController so that all other classes can just

// call getInstance()

sUiccController = UiccController.make(context, sCommandsInterfaces);

make()函数具体如下：

//UiccController

public static UiccController make(Context c, CommandsInterface[] ci) {

synchronized (mLock) {

if (mInstance != null) {

throw new RuntimeException("MSimUiccController.make() should only be called once");

}

mInstance = new UiccController(c, ci);

return (UiccController)mInstance;

}

}

可见UiccController采用的是单例模式，当存在实例时再次调用make()将报错，只能通过调用getInstance()获得该对象。

UiccController实际是一个Handler,在其构造函数当中，注册了EVENT\_ICC\_STATUS\_CHANGED，EVENT\_SIM\_REFRESH等事件，用于监听底层上传的卡状态更新与卡刷新。

### 2.3.2 UiccCard实例化：

当UiccController接收到上文所说的两种事件时，都会去注册监听一个EVENT\_GET\_ICC\_STATUS\_DONE事件，一旦收到该消息就会进入onGetIccCardStatusDone (AsyncResult ar, Integer index)函数：

//UiccCard

private synchronized void onGetIccCardStatusDone(AsyncResult ar, Integer index) {

if (ar.exception != null) {

Rlog.e(LOG\_TAG,"Error getting ICC status. "

+ "RIL\_REQUEST\_GET\_ICC\_STATUS should "

+ "never return an error", ar.exception);

return;

}

if (!isValidCardIndex(index)) {

Rlog.e(LOG\_TAG,"onGetIccCardStatusDone: invalid index : " + index);

return;

}

IccCardStatus status = (IccCardStatus)ar.result;

if (mUiccCards[index] == null) {

//Create new card

mUiccCards[index] = new UiccCard(mContext, mCis[index], status, index);

} else {

//Update already existing card

mUiccCards[index].update(mContext, mCis[index] , status);

}

if (DBG) log("Notifying IccChangedRegistrants");

mIccChangedRegistrants.notifyRegistrants(new AsyncResult(null, index, null));

}

这里使用了synchronized，避免同时对UiccCard进行操作。

从中可看到，当对应的mUiccCards[]中无对象时就实例化一个新的UiccCard, 若存在就调用它的update(Context c, CommandsInterface ci, IccCardStatus ics)。而在UiccCard的构造函数当中，同样调用了该函数。

### 2.3.3 CatService实例化：

进入UiccCard.java的update函数，该函数先是根据条件更新或者实例化一个新的CardApplication对象， 继而实例化IccRecords.java,IccFileHandler.java, 不做具体介绍 。

接下来调用了createAndUpdateCatServiceLocked()，当不在mCatService对象时调用CatService的getInstance(mCi, mContext, this, mPhoneId)：

//UiccCard

private void createAndUpdateCatServiceLocked() {

if (mUiccApplications.length > 0 && mUiccApplications[0] != null) {

// Initialize or Reinitialize CatService

if (mCatService == null) {

mCatService = CatService.getInstance(mCi, mContext, this, mPhoneId);

} else {

mCatService.update(mCi, mContext, this);

}

} else {

if (mCatService != null) {

mCatService.dispose();

}

mCatService = null;

}

}

进入到CatService的getInstance函数，可见：

//CatService->getInstance

if (sInstance == null) {

int simCount = TelephonyManager.getDefault().getSimCount();

sInstance = new CatService[simCount];

for (int i = 0; i < simCount; i++) {

sInstance[i] = null;

}

}

即当前不存在CatService的实例化对象sInstance时，调用构建函数实例化一个新的对象。

由此，CatService的初始化过程结束。

注：缺少一个完整流程的时序图。

# Framework流程

## 3.1 CatService

### 3.1.1 关键函数

1. handleMessage(Message msg)：CatService继承了Handler,因此该函数是了解整个类的关键入口。

2. getInstance(CommandsInterface ci,Context context, UiccCard ic, int slotId)：单例模式，通过该函数对外提供对象。

3. handleRilMsg(RilMessage rilMsg):处理经过decode后的来自RIL层的消息，调用handleCommand。

4. handleCmdResponse(CatResponseMessage resMsg)：处理来自应用层的信息

5. handleCommand(CommandParams cmdParams, boolean isProactiveCmd)：处理主动式命令与非主动式命令。

### 3.1.2 关键流程

#### 3.1.2.1 监听信息

在CatService的构造函数当中，可看到：

//CatService->CatService

// 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);

mCmdIf.registerForIccRefresh(this, MSG\_ID\_ICC\_REFRESH, null);

mCmdIf.setOnCatCcAlphaNotify(this, MSG\_ID\_ALPHA\_NOTIFY, null);

这里是注册关于RIL层的相关信息的观察者，包括终结回话，主动式命令，卡更新等。

CatService实现了AppInterface接口，该接口用于Framework层与STK应用层的交互。接口中定义了各类主动式命令，CatService通过实现AppInterface的onCmdResponse

(CatResponseMessage resMsg)来注册相关的信息监听。

#### 3.1.2.1 运作流程

CatService本质是一个Handler，因此通过handleMessage(Message msg)来追溯其运作流程，

**首先是来自RIL层的消息：**

case MSG\_ID\_SESSION\_END:

case MSG\_ID\_PROACTIVE\_COMMAND:

case MSG\_ID\_EVENT\_NOTIFY:

case MSG\_ID\_REFRESH:

CatLog.d(this, "ril message arrived,slotid:" + mSlotId);

String data = null;

if (msg.obj != null) {

AsyncResult ar = (AsyncResult) msg.obj;

if (ar != null && ar.result != null) {

try {

data = (String) ar.result;

// Begin added by donghai.wu for XR6154319 telecomcode on 2018/09/28

TclPluginManager.getTclStk(mContext).judgeSetupMenu(mSlotId,data);

//End added by donghai.wu for XR6154319 telecomcode on 2018/09/28

} catch (ClassCastException e) {

break;

}

}

}

mMsgDecoder.sendStartDecodingMessageParams(new RilMessage(msg.what, data));

break;

case MSG\_ID\_CALL\_SETUP:

mMsgDecoder.sendStartDecodingMessageParams(new RilMessage(msg.what, null));

可看到当接收到对应信息时，函数会调用sendStartDecodingMessageParams(new RilMessage(msg.what, data))对信息进行解码。

当收到解码的消息，则调用handleRilMsg(RilMessage rilMsg)进行消息的处理。

case MSG\_ID\_RIL\_MSG\_DECODED:

handleRilMsg((RilMessage) msg.obj);

在handleRilMsg(RilMessage rilMsg)中，当消息为MSG\_ID\_SESSION\_END时则进入handleSessionEnd()，发送SESSION END消息至应用层;当消息为MSG\_ID\_EVENT\_NOTIFY 、MSG\_ID\_PROACTIVE\_COMMAND、 MSG\_ID\_REFRESH之一且解码成功时，调用handleCommand(CommandParams cmdParams, boolean isProactiveCmd)，对命令进行下一步处理并根据情况发送对应的TR。

**来自应用层的消息：**

当收到消息为MSG\_ID\_RESPONSE则进入handleCmdResponse((CatResponseMessage) msg.obj)进行上层消息的处理：

case MSG\_ID\_RESPONSE:

handleCmdResponse((CatResponseMessage) msg.obj);

上层传来的消息都是对于命令处理的response，这里会先调用validateResponse(resMsg)确定是否为RIL上传的上一个合法命令的response，不是则不做处理。根据注释里的解释，这里是为了避免“迷惑”基带和RIL层。

根据传来的result code，函数对传来的数据做出相对应的处理，这里不做介绍。在处理完消息后，函数最终调用sendTerminalResponse(CommandDetails cmdDet,ResultCode resultCode, boolean includeAdditionalInfo, int additionalInfo, ResponseData resp)发送TR给RIL层。

# 总结

以上便是STK在Framework 层的运作流程， 这个模块是进行一个消息的转换和传递作用，不涉及命令的细节处理。对这一模块的了解主要依靠阅读源码，根据log进行追溯。了解这一模块有助于理解STK整体运作流程与相关消息的传递和处理。