再论Android中的广播和RPC

邓凡平

内容：

- All is about information
- 广播
- IPC和RPC-Theory
- 解密Binder
- commonRPC展示

All is about information

The Information: A History, a Theory, a Flood
—by James Gleick

All is about information

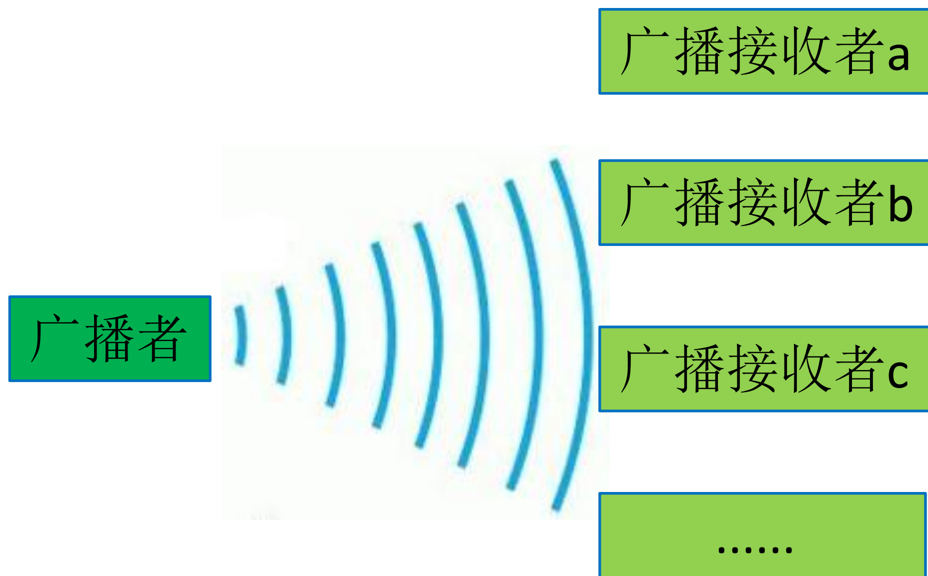
Take a closer look at:

① “贾君鹏，你妈妈喊你回家吃饭”

② “@贾君鹏，你妈妈喊你回家吃饭”

广播——原理

① “贾君鹏，你妈妈喊你回家吃饭”



广播系统的特点：
- 发送者只管发送消息，不考虑谁接收
- 接收者只关注自己感兴趣的消息

Android平台里的实现方式
发送者：
- Context.sendBroadcast系列API

接收者：
- AndroidManifest.xml中编写静态BroadcastReceiver标签，设置广播过滤条件
- 程序通过registerReceiver动态注册广播接收者

广播——Android平台示例

A： 广播者发送广播

```
sendBroadcast(Intent)
sendBroadcast(Intent, String)
sendOrderedBroadcast(Intent, String)
sendStickyBroadcast(Intent)
sendStickyOrderedBroadcast(Intent, BroadcastReceiver, Handler, int, String, Bundle)
```

B： 广播接收者的处理——注册广播接收者

静态注册者

```
<receiver android:name="com.tesla.tunguska.cpossetupwizard.UninstallBCReceiver"
          android:enabled="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
```

对什么广播感兴趣

动态注册者

```
static public MerchantConfig registerMerchantInfoListener(Context context,
                                                          BroadcastReceiver br){
    checkPermission(context,Key.PERMISSION_GETMERCHANTINFO);
    IntentFilter filter = new IntentFilter(
            Key.ACTION_GETMERCHANTINFO);
    merchantConfigBr = br;
    Intent info = context.registerReceiver(br,filter,
            Key.PERMISSION_SENDMERCHANTINFO,scheduler: null);
    if(info != null && info.hasExtra(Key.EXTRA_MERCHANTCONFIG)){
        MerchantConfig config = MerchantConfig.parse(
                info.getStringExtra(Key.EXTRA_MERCHANTCONFIG));
        return config;
    }
    return null;
}
```

filter:IntentFilter，对什么广播感兴趣

广播——Android平台示例

B：广播接收者的处理——坐等广播

```
public class UninstallBCReceiver extends BroadcastReceiver {
    private final String TAG = "CPos-" + UninstallBCReceiver.class.getSimpleName();

    @Override
    public void onReceive(final Context context, Intent intent) {
        Log.e(TAG, "Receive " + intent.getAction());
        SharedPreferences pref = context.getSharedPreferences(name: "config",
                Context.MODE_PRIVATE);
        final boolean isCompleted = pref.getBoolean("cpos-initiliazed", false);
        Log.e(TAG, "check config is completed " + isCompleted);
        if(isCompleted){
            (Handler) handleMessage(msg) → {

                    Log.e(TAG, msg: "Uninstall myself");
                    SystemManager mSm = new SystemManager(context);
                    int nret = mSm.uninstall("com.tesla.tunguska.cpossetupwizard");
                    Log.e(TAG, "Uninstall myself return " + nret);
            }.sendEmptyMessageDelayed(what: 0, delayMillis: 3000);
        }

    }
}
```

区分广播信息

广播——广播发送方法的改进和其他

1 发送者的三种玩法
2 Subscriber/Publisher模式

A： 广播者发送广播

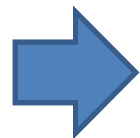
```
sendBroadcast(Intent)
sendBroadcast(Intent, String)
sendOrderedBroadcast(Intent, String)
sendStickyBroadcast(Intent)
sendStickyOrderedBroadcast(Intent, BroadcastReceiver, Handler, int, String, Bundle)
```

①群发，接收者无优先级，不能中断

②接收者可设置优先级，高优先级可中断广播继续发送

③广播发了，后来的接收者却接收不到，怎么处理？
发送者发送Sticky广播，该广播的内容由系统保存。当新的广播注册者到来，由系统将广播内容返给它

其他的广播机制？
- UDP组播？
- D-BUS？
- Uevent？

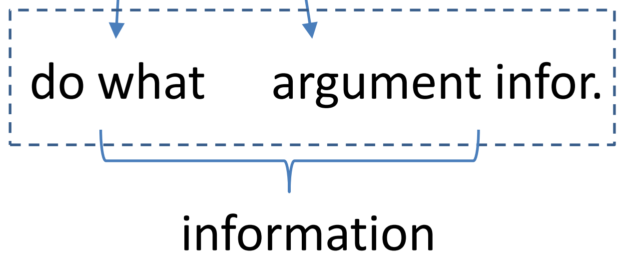


Publisher/Subscriber模型

IPC/RPC——Theory

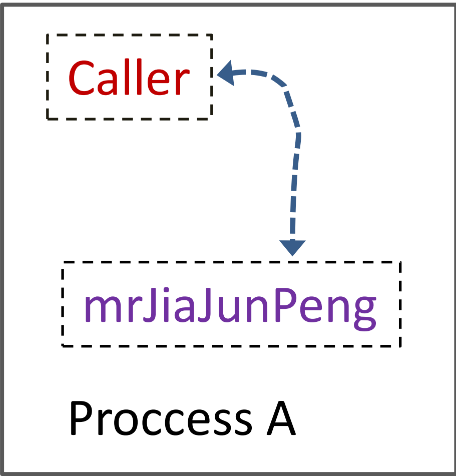
② “@贾君鹏，你妈妈喊你回家吃饭”

```
public sendMessage(){
  String msg = “Your Mother ...for dinner”;
   mrJiaJunPeng.tell(msg);
}
```



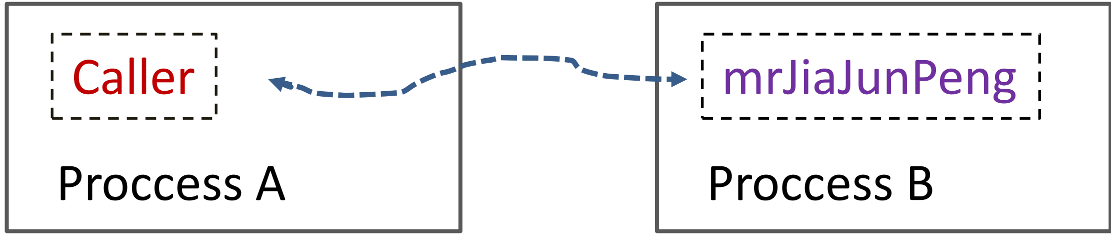
IPC/RPC——Theory

①IPC



mrJiaJunPeng和Caller在一个进程里。普通的函数调用

②RPC

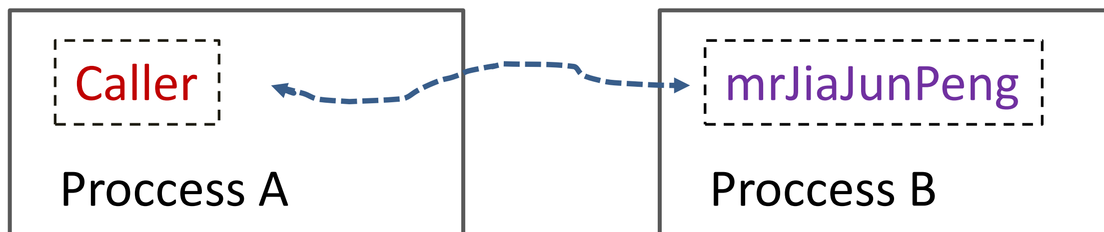


mrJiaJunPeng在另外一个进程里....，如何通信？

三个关键问题：
1 发给谁？
2 发什么内容？
3 怎么发？

IPC/RPC——Theory

②RPC



三个关键问题之怎么发？

通信方法：

- socket
- pipe
- shared memory

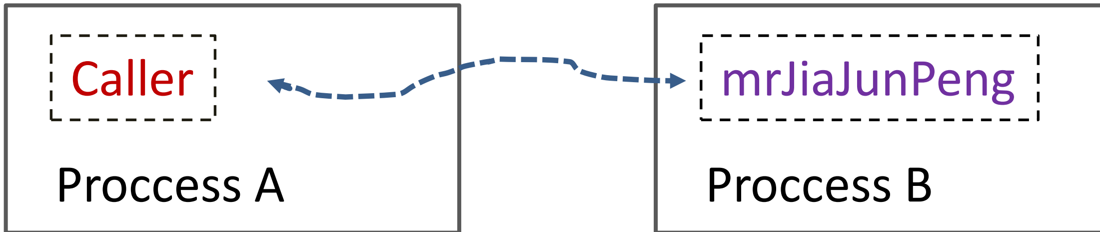
- http?
- webservice?

Binder:

1. Kernel里有个binder设备，纯软件层 (drivers/staging/android/binder.c)
2. 需要binder通信的app都会打开这个设备
3. 剩下就是角色（交互协议）的问题了

IPC/RPC——Theory

②RPC



三个关键问题之
发给谁？
发什么内容？

发给谁:

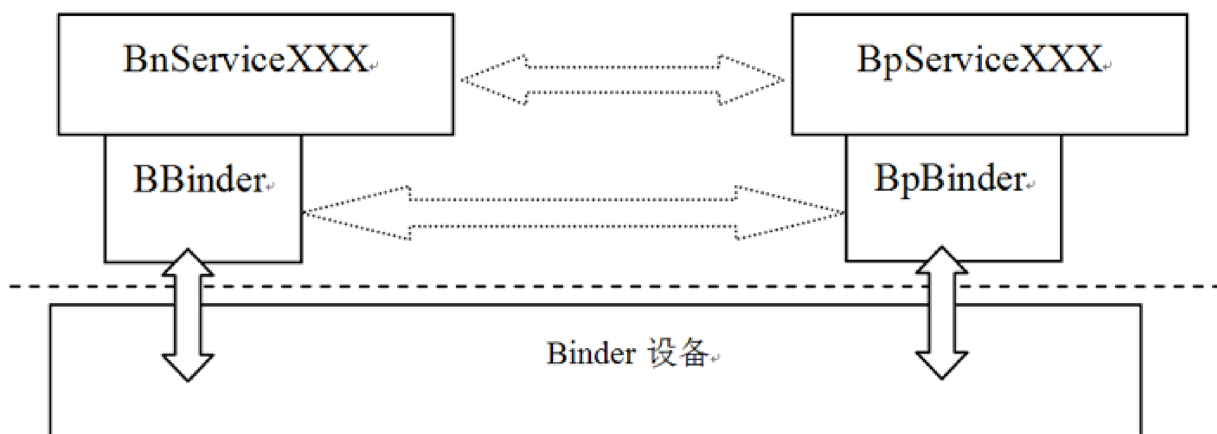
- socket/http/webservice：地址+端口号/url
- pipe：fd
- shared memory：/soname(shm_open)

发什么内容:
1 完全自己组织数据
2 基于http等标准协议之上的协议

Binder：
1 封装了binder的协议

This is Binder,Nothing new

Why everybody think Binder is difficult?



1 代码比较复杂，派生和继承关系较多，又使用了模板类，宏等
2 业务和通信混在一起。这和binder之前的RPC手段都不相同

回顾：非binder的RPC是怎样的呢？
1 建立通信连接
2 组织业务层数据/协议，发送和接收

Binder：二合一
1 像本地函数一样调用

解密Binder——Native Binder（服务端）（2）

通讯和业务怎么结合？

①通讯层收到请求，调用onTransact
②每个函数调用都有一个code，根据code，调用不同的业务函数

```
status_t BnMediaPlayerService::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch (code) {
        case CREATE: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            sp<IMediaPlayerClient> client =
                interface_cast<IMediaPlayerClient>(data.readStrongBinder());
            audio_session_t audioSessionId = (audio_session_t) data.readInt32();
            sp<IMediaPlayer> player = create(client, audioSessionId);
            reply->writeStrongBinder(IInterface::asBinder(player));
            return NO_ERROR;
        } break;
        case CREATE_MEDIA_RECORDER: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            const String16 opPackageName = data.readString16();
            sp<IMediaRecorder> recorder = createMediaRecorder(opPackageName);
            reply->writeStrongBinder(IInterface::asBinder(recorder));
            return NO_ERROR;
        } break;
        case CREATE_METADATA_RETRIEVER: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            sp<IMediaMetadataRetriever> retriever = createMetadataRetriever();
            reply->writeStrongBinder(IInterface::asBinder(retriever));
            return NO_ERROR;
```

Java层的Binder

代码实践

1 编写AIDL文件，类似java的interface

IPrinter.aidl

```
package com.tesla.tunguska.cpos.devic
import com.tesla.tunguska.cpos.device

interface IPrinter{
    int open();
    int close();
    int begin();
    int end();
    int printContent(in PrintContent data);
    int printData(in byte[] data);
    int queryStatus();
```

业务函数

参数问题：

1 string、int等基本数据类型可直接传递

2 复杂数据类型需要编写单独的aidl文件和java文件

3 参数前的修饰in/out/inout要小心

PrintContent.aidl

```
package com.tesla.tunguska.cpos.device.protocol;
parcelable PrintContent;
```

复杂参数：

1单独编写一个aidl文件

2 编写一个类。数据的打包/解包在该类中完成

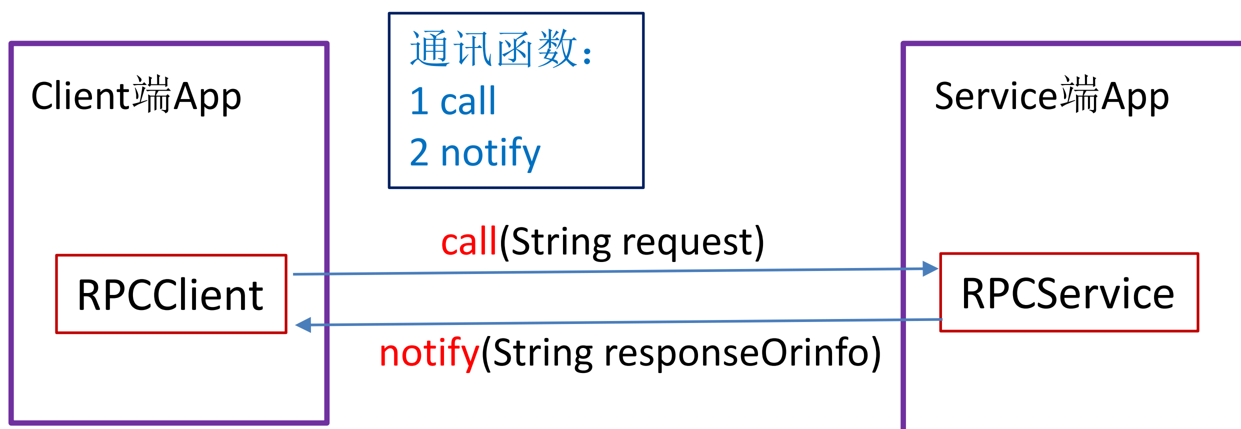
PrintContent.java

```
public class PrintContent implements Parcelable {
    /**...*/
    static public class SectionContent implements Parcelable {...}

    /**...*/
    static public class LineContent implements Parcelable {...}

    public ArrayList<LineContent> mLines;

    public void addLine(LineContent content) {...}

    public static final Parcelable.Creator<PrintContent> CREATOR =
            new Parcelable.Creator<PrintContent>() {...};

    public static class Builder {...}

    public PrintContent() {...}

    public PrintContent(Parcel source) {...}

    @Override
    public int describeContents() {...}

    @Override
    public void writeToParcel(Parcel dest, int flags) {...}
}
```

commonRPC展示

AIDL的痛点：
1 每一类型的业务逻辑都需要写一个AIDL
2 如果需要修改API，就要重新编译SDK
3 如果参数复杂，又得写AIDL文件和打包/解包类

解决办法：
1 Binder是业务+通讯混在一起
2 如果上帝的归上帝，凯撒的归凯撒？



commonRPC展示

IRPC.aidl × IRPCClient.aidl ×

```
1  package com.cpos.rpc.aidl;
2
3  import com.cpos.rpc.aidl.IRPCClient;
4  import android.content.Intent;
5
6  interface IRPC{
7      int register(in String clientPkgName,in IRPCClient client);
8      oneway void call(int pid,String info,in Intent wrapData);
9      int unregister();
10 }
11
```

RPCClient.aidl ×

```
package com.cpos.rpc.aidl;

oneway interface IRPCClient{
    void notify(String info);
}
```

commonRPC展示——客户端

剩下就是业务+参数：
1 对外是对象
2 传输时转换成json字符串

sale：业务层函数
SaleRequest：消费请求

```
public int sale(SaleRequest request) {
    ....
    Log.e(TAG,"call sale:" + SaleRequest.parse(request));
    return mRpc.callAsyncWithName(
            name: "sale", SaleRequest.parse(request));
}
```

callAsyncWithName：是call的封装
1 name：服务端函数名
2 info：参数

```
CallContext  callInner(String name, String info,
                       Parcelable data, boolean bWait){
    CallContext callContext = new CallContext();
    callContext.setRequest(info);
    callContext.setRequestName(name);
    Intent wrapData = null;
    if(data != null){
        wrapData = new Intent();
        wrapData.putExtra(name,data);
    }
    putIntoCallRequest(callContext,bWait);
    try{
        //long begin = System.currentTimeMillis();
        mIRPC.call(myPid,CallContext.parse(callContext),wrapData);
        //...
        return callContext;
    }catch (Exception ex){
        ex.printStackTrace();
```

call函数的参数：
1 pid：客户端进程号
2 info：带函数名+原info的info

commonRPC展示——服务端

```java
public class ThirdPartyPaymentService extends BaseService implements
    RPCService.CallProcessor {
    private RPCService mRPC;
    ....
    @Override
    public void onCreate() {...}

    @Override
    public void onDestroy() {...}

    public IBinder onBind(Intent intent) {
        Loger.e("onBind");
        return mRPC.getIRPC();
    }

    @Override
    public String onCall(String name, String request, Parcelable data) {

        //name = sale.
        //request=

        return "0";
    }
```

name：client端调用的函数
request：json字符串，可还原回原对象，
比如SaleRequest
data：其他可序列化对象，比如bitmap

其他功能：
1 自动connect，reconnect
2 函数调用可阻塞，也可非阻塞
3 服务端可通知客户端

客户端死亡通知

```java
public void setup() {
    try {
        notifier.asBinder().linkToDeath(this, 0);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

开源地址：https://gitee.com/innost/commonRPC

谢谢！

謝謝！

Thanks！

Gracias！

ありがとうございます!

감사합니다!