巧用 Android 多进程，微信，微博等主流 App 都在用

阅读 4512 收藏 495 2017-06-16
原文链接：url.cn

巧用Android多进程，微信，微博等主流App都在用

Original *2017-06-16 陈建维* 码个蛋 码个蛋 **码个蛋**

WeChat ID codeegg

Intro 每天更新优质文章：Android、职场干货，不定期大神语音分享。

作者博客

http://cjw-blog.net/

文章目录

1. 前言
2. 为什么要使用多进程？

5. 使用AIDL实现一个多进程消息推送
6. 实现思路
7. 例子具体实现
8. 知其然，知其所以然。
9. 跨进程的回调接口
10. DeathRecipient
11. 权限验证
12. 根据不同进程，做不同的初始化工作
13. 总结
14. 结语

1

前言

对于进程的概念，来到这里的都是编程修仙之人，就不再啰嗦了，相信大家倒着、跳着、躺着、各种姿势都能背出来。

2

程，还可能需要编写额外的进程通讯代码，还可能带来额外的Bug，这无疑加大了开发的工作量，在很多创业公司中工期也不允许，这导致了整个app都在一个进程中。

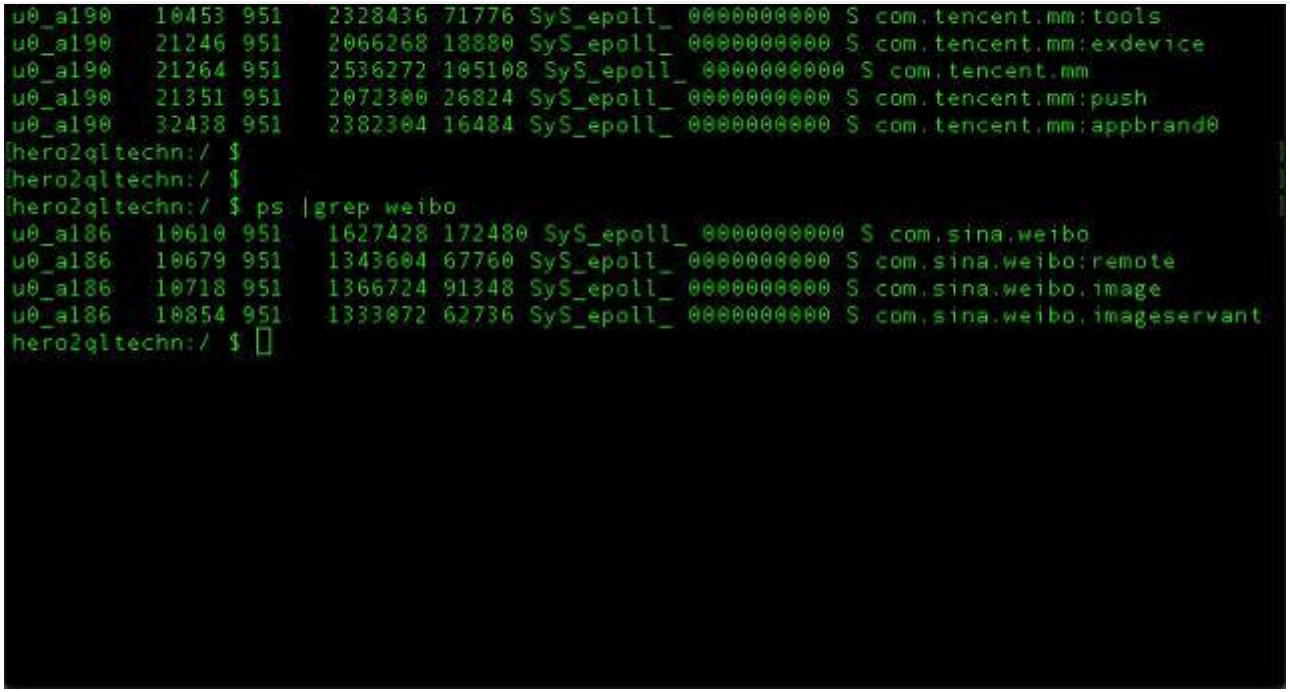
整个app都在一个进程有什么弊端？

在Android中，虚拟机分配给各个进程的运行内存是有限制值的（这个值可以是32M，48M，64M等，根据机型而定），试想一下，如果在app中，增加了一个很常用的图片选择模块用于上传图片或者头像，加载大量Bitmap会使app的内存占用迅速增加，如果你还把查看过的图片缓存在了内存中，那么OOM的风险将会大大增加，如果此时还需要使用WebView加载一波网页，我就问你怕不怕！

微信，微博等主流app是如何解决这些问题的？

微信移动开发团队在 《Android内存优化杂谈》 一文中就说到：“对于webview，图库等，由于存在内存系统泄露或者占用内存过多的问题，我们可以采用单独的进程。微信当前也会把它们放在单独的tools进程中”。

下面我们使用adb查看一下微信和微博的进程信息（Android 5.0以下版本可直接在“设置 -> 应用程序”相关条目中查看）：



进入adb shell后，使用 “ps | grep 条目名称” 可以过滤出想要查看的进程。

可以看到，微信的确有一个tools进程，而新浪微博也有image相关的进程，而且它们当中还有好些其它的进程，比如微信的push进程，微博的remote进程等，这里可以看出，他们不单单只是把上述的WebView、图库等放到单独的进程，还有推送服务等也是运行在独立的进程中的。一个消息推送服务，为了保证稳定性，可能需要和UI进程分离，分离后即使UI进程退出、Crash或者出现内存消耗过高等情况，仍不影响消息推送服务。

可见，合理使用多进程不仅仅是有多大好处的问题，我个人认为而且是很有必要的。

采解决问题，又或者，在面试的时候，跟面试官聊到这方面的知识时候也不至于尴尬。

3

为什么需要“跨进程通讯”？

Android的进程与进程之间通讯，有些不需要我们额外编写通讯代码，例如：把选择图片模块放到独立的进程，我们仍可以使用startActivityForResult方法，将选中的图片放到Bundle中，使用Intent传递即可。（看到这里，你还不打算把你项目的图片选择弄到独立进程么？）

但是对于把“消息推送Service”放到独立的进程，这个业务就稍微复杂点了，这个时候可能会发生Activity跟Service传递对象，调用Service方法等一系列复杂操作。

由于各个进程运行在相对独立的内存空间，所以它们是不能直接通讯的，因为程序里的变量、对象等初始化后都是具有内存地址的，举个简单的例子，读取一个变量的值，本质是找到变量的内存地址，取出存放的值。不同的进程，运行在相互独立的内存（其实就可以理解为两个不同的应用程序），显然不能直接得知对方变量、对象的内存地址，这样的话也自然不能访问对方的变量，对象等。此时两个进程进行交互，就需要使用跨进程通讯的方式去实现。简单说，跨进程通讯就是一种让进程与进程之间可以进行交互的技术。

4

跨进程通讯的方式有哪些？

1. 四大组件间传递Bundle;
2. 使用文件共享方式，多进程读写一个相同的文件，获取文件内容进行交互；

Google一下就能解决的问题，就不啰嗦了）；

4. 使用AIDL(Android Interface Definition Language)，Android接口定义语言，用于定义跨进程通讯的接口；
5. 使用ContentProvider，常用于多进程共享数据，比如系统的相册，音乐等，我们也可以通过ContentProvider访问到；
6. 使用Socket传输数据。

接下来本文将重点介绍使用AIDL进行多进程通讯，因为AIDL是Android提供给我们的标准跨进程通讯API，非常灵活且强大（貌似面试也经常会问到，但是真正用到的也不多…）。上面所说的Messenger也是使用AIDL实现的一种跨进程方式，Messenger顾名思义，就像是一种串行的消息机制，它是一种轻量级的IPC方案，可以在不同进程中传递Message对象，我们在Message中放入需要传递的数据即可轻松实现进程间通讯。但是当我们需要调用服务端方法，或者存在并发请求，那么Messenger就不合适了。而四大组件传递Bundle，这个就不需要解释了，把需要传递的数据，用Intent封装起来传递即可，其它方式不在本文的讨论范围。

下面开始对AIDL的讲解，各位道友准备好渡劫了吗？

5

使用AIDL实现一个多进程消息推送

像图片选择这样的多进程需求，可能并不需要我们额外编写进程通讯的代码，使用四大组件传输Bundle就行了，但是像推送服务这种需求，进程与进程之间需要高度的交互，此时就绕

1. UI和消息推送的Service分两个进程；
2. UI进程用于展示具体的消息数据，把用户发送的消息，传递到消息Service，然后发送到远程服务器；
3. Service负责收发消息，并和远程服务器保持长连接，UI进程可通过Service发送消息到远程服务器，Service收到远程服务器消息通知UI进程；
4. 即使UI进程退出了，Service仍需要保持运行，收取服务器消息。

6

实现思路

先来整理一下实现思路：

1. 创建UI进程（下文统称为客户端）；
2. 创建消息Service（下文统称为服务端）；
3. 把服务端配置到独立的进程(AndroidManifest.xml中指定process标签）；
4. 客户端和服务端进行绑定（bindService）；
5. 让客户端和服务端具备交互的能力。(AIDL使用）；

7

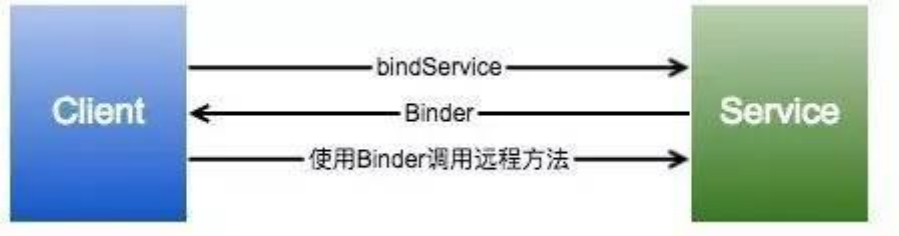
例子具体实现

https://github.com/V1sk/AIDL

Step0. AIDL调用流程概览

开始之前，我们先来概括一下使用AIDL进行多进程调用的整个流程：

1. 客户端使用bindService方法绑定服务端；
2. 服务端在onBind方法返回Binder对象；
3. 客户端拿到服务端返回的Binder对象进行跨进程方法调用；



整个AIDL调用过程概括起来就以上3个步骤，下文中我们使用上面描述的例子，来逐步分解这些步骤，并讲述其中的细节。

1. 创建客户端 -> MainActivity；
2. 创建服务端 -> MessageService;
3. 把服务端配置到另外的进程 -> android:process=":remote"

上面描述的客户端、服务端、以及把服务端配置到另外进程，体现在AndroidManifest.xml中，如下所示：

```
<manifest ...>
    <application ...>
        <activity android:name=".ui.MainActivity"/>
        <service
            android:name=".service.MessageService"
            android:enabled="true"
            android:exported="true"
            android:process=":remote" />
    </application>
</manifest>
```

开启多进程的方法很简单，只需要给四大组件指定android:process标签。

1.2 绑定MessageService到MainActivity

创建MessageService：

```
public class MessageService extends Service {
    public MessageService() {
    }
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```

客户端MainActivity调用bindService方法绑定MessageService

这一步其实是属于Service组件相关的知识，在这里就比较简单地说一下，启动服务可以通过以下两种方式：

1. 使用bindService方法 -> bindService(Intent service, ServiceConnection conn, int flags)；
2. 使用startService方法 -> startService(Intent service);

bindService & startService区别：使用bindService方式，多个Client可以同时bind一个Service，但是当所有Client unbind后，Service会退出，通常情况下，如果希望和Service交互，一般使用bindService方法，使用onServiceConnected中的IBinder对象可以和Service进行交互，不需要和Service交互的情况下，使用startService方法即可。

startService（比如像本例子中的消息服务，退出UI进程，Service仍需要接收到消息），代码如下：

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        setupService();
    }
    /**
     * unbindService
     */
    @Override
    protected void onDestroy() {
        unbindService(serviceConnection);
        super.onDestroy();
    }
    /**
     * bindService & startService
     */
    private void setupService() {
        Intent intent = new Intent(this, MessageService.class);
        bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);
        startService(intent);
    }
    ServiceConnection serviceConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            Log.d(TAG, "onServiceConnected");
        }
        @Override
        public void onServiceDisconnected(ComponentName name) {
            Log.d(TAG, "onServiceDisconnected");
        }
    };
}```
```

Stpe2.服务端在onBind方法返回Binder对象

过程调用机制的核心部分，该接口描述了与远程对象交互的抽象协议，而Binder实现了IBinder接口，简单说，Binder就是Android SDK中内置的一个多进程通讯实现类，在使用的时候，我们不用也不要去实现IBinder，而是继承Binder这个类即可实现多进程通讯。

2.2 其次，这个需要在onBind方法返回的Binder对象从何而来？

在这里就要引出本文中的主题了——AIDL多进程中使用的Binder对象，一般通过我们定义好的 .adil 接口文件自动生成，当然你可以走野路子，直接手动编写这个跨进程通讯所需的Binder类，其本质无非就是一个继承了Binder的类，鉴于野路子走起来麻烦，而且都是重复步骤的工作，Google提供了 AIDL 接口来帮我们自动生成Binder这条正路，下文中我们围绕AIDL 这条正路继续展开讨论（可不能把人给带偏了是吧 ）

2.3 定义AIDL接口

很明显，接下来我们需要搞一波上面说的Binder，让客户端可以调用到服务端的方法，而这个Binder又是通过AIDL接口自动生成，那我们就先从AIDL搞起，搞之前先看看注意事项，以免出事故：

AIDL支持的数据类型：

- Java 编程语言中的所有基本数据类型（如 int、long、char、boolean 等等）
- String和CharSequence
- Parcelable：实现了Parcelable接口的对象

- Map：其中的元素需要被AIDL支持，包括 key 和 value，另一端实际接收的具体类始终是 HashMap，但生成的方法使用的是 Map 接口

其他注意事项：

- 在AIDL中传递的对象，必须实现Parcelable序列化接口；
- 在AIDL中传递的对象，需要在类文件相同路径下，创建同名、但是后缀为.aidl的文件，并在文件中使用parcelable关键字声明这个类；
- 跟普通接口的区别：只能声明方法，不能声明变量；
- 所有非基础数据类型参数都需要标出数据走向的方向标记。可以是 in、out 或 inout，基础数据类型默认只能是 in，不能是其他方向。

下面继续我们的例子，开始对AIDL的讲解~

2.4 创建一个AIDL接口，接口中提供发送消息的方法（Android Studio创建AIDL：项目右键 -> New -> AIDL -> AIDL File），代码如下：

```
interface MessageSender {
    void sendMessage(in MessageModel messageModel);
}
```

一个比较尴尬的事情，看了很多文章，从来没有一篇能说清楚in、out、inout这三个参数方向的意义，后来在stackoverflow上找到能理解答案（https://stackoverflow.com/questions/4700225/in-out-inout-in-a-aidl-interface-parameter-value），我翻译一下大概意思：

被“in”标记的参数，就是接收实际数据的参数，这个跟我们普通参数传递一样的含义。在AIDL中，“out” 指定了一个仅用于输出的参数，换而言之，这个参数不关心调用方传递了什么数据过来，但是这个参数的值可以在方法被调用后填充（无论调用方传递了什么值过来，在方法执行的时候，这个参数的初始值总是空的），这就是“out”的含义，仅用于输出。而“inout”显然就是“in”和“out”的合体了，输入和输出的参数。区分“in”、“out”有什么用？这是非常重要的，因为每个参数的内容必须编组（序列化，传输，接收和反序列化）。in/out标签允许Binder跳过编组步骤以获得更好的性能。

上述的MessageModel为消息的实体类，该类在AIDL中传递，实现了Parcelable序列化接口，代码如下：

```
    private String to;
    private String content;
    ...
    Setter & Getter
    ...
    @Override
    public int describeContents() {
        return 0;
    }
    //...
    序列化相关代码
    //...
}
```

手动实现Parcelable接口比较麻烦，安利一款AS自动生成插件android-parcelable-intellij-plugin

创建完MessageModel这个实体类，别忘了还有一件事要做：”在AIDL中传递的对象，需要在类文件相同路径下，创建同名、但是后缀为.aidl的文件，并在文件中使用parcelable关键字声明这个类“。代码如下：

```
package com.example.aidl.data;

parcelable MessageModel;
```

对于没有接触过aidl的同学，光说就能让人懵逼，来看看此时的项目结构压压惊：

- MessageSender.aidl -> 定义了发送消息的方法，会自动生成名为MessageSender.Stub的Binder类，在服务端实现，返回给客户端调用
- MessageModel.java -> 消息实体类，由客户端传递到服务端，实现了Parcelable序列化
- MessageModel.aidl -> 声明了MessageModel可在AIDL中传递，放在跟MessageModel.java相同的包路径下

OK，相信此时懵逼已解除~

2.5 在服务端创建MessageSender.aidl这个AIDL接口自动生成的Binder对象，并返回给客户端调用，服务端MessageService代码如下：

```
public class MessageService extends Service {
    private static final String TAG = "MessageService";
    public MessageService() {
    }
    IBinder messageSender = new MessageSender.Stub() {
        @Override
        public void sendMessage(MessageModel messageModel) throws RemoteException {
            Log.d(TAG, "messageModel: " + messageModel.toString());
        }
    };
    @Override
    public IBinder onBind(Intent intent) {
        return messageSender;
    }
}
```

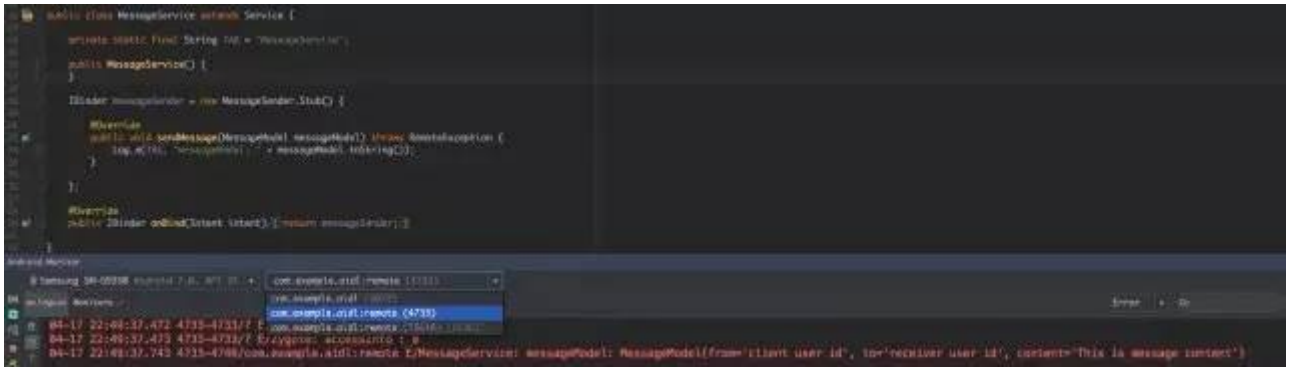
2.6 客户端拿到Binder对象后调用远程方法

调用步骤如下：

1. 在客户端的onServiceConnected方法中，拿到服务端返回的Binder对象；
2. 使用MessageSender.Stub.asInterface方法，取得MessageSender.aidl对应的操作接口；
3. 取得MessageSender对象后，像普通接口一样调用方法即可。

此时客户端代码如下：

在客户端中我们调用了MessageSender的sendMessage方法，向服务端发送了一条消息，并把生成的MessageModel对象作为参数传递到了服务端，最终服务端打印的结果如下：



这里有两点要说：

到这里，我们已经完成了最基本的使用AIDL进行跨进程方法调用，也是Step.0的整个细化过程，可以再回顾一下Step.0，既然已经学会使用了，接下来...全剧终。。。

如果写到这里全剧终，那跟咸鱼有什么区别...

8

知其然，知其所以然。

我们通过上述的调用流程，看看从客户端到服务端，都经历了些什么事，看看Binder的上层是如何工作的，至于Binder的底层，这是一个非常复杂的话题，本文不深究。（如果看到这里你又想问什么是Binder的话，请手动倒带往上看...）

我们先来回顾一下从客户端发起的调用流程：

1. MessageSender messageSender = MessageSender.Stub.asInterface(service);
2. messageSender.sendMessage(messageModel);

子目录，自己找，不爽你来打我啊 😊 ）

请看下方代码和注释，前方高能预警...

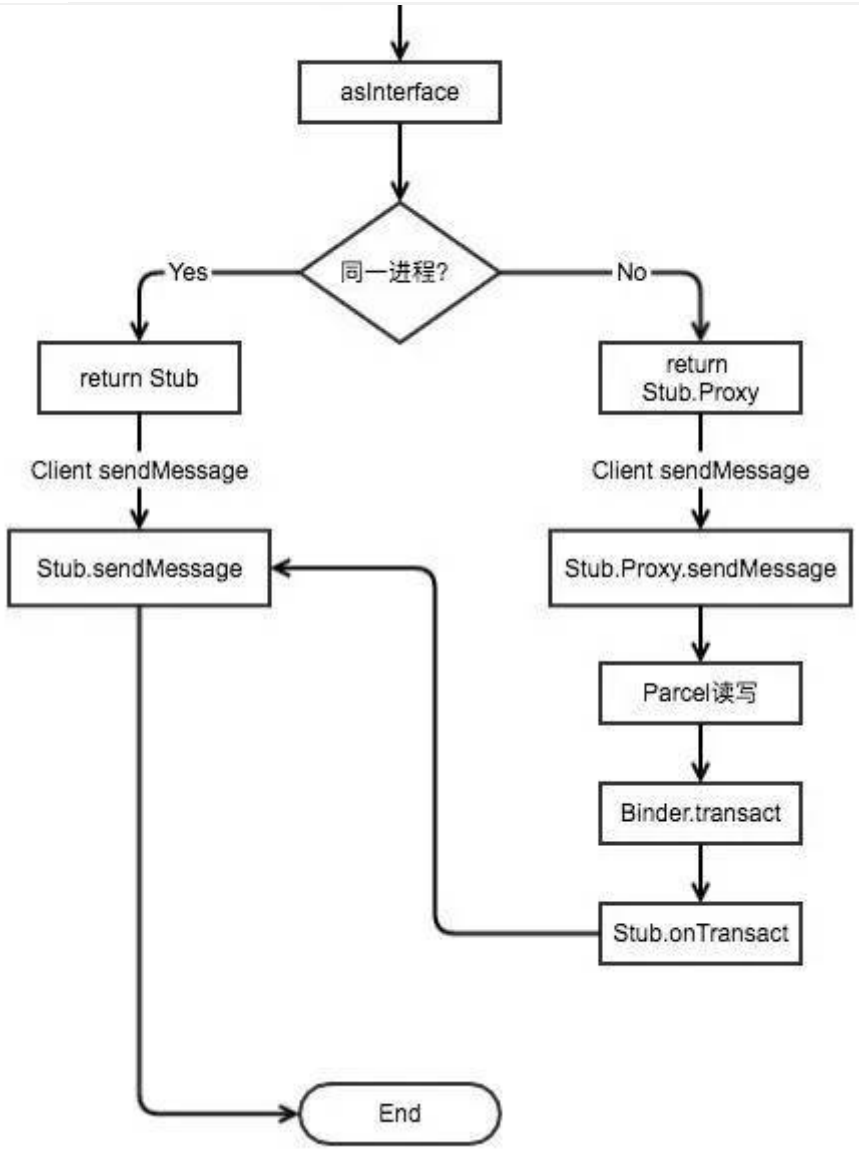
```
     * 实质是调用了 Stub.Proxy 的 sendMessage 方法，从而触发跨进程数据传递，
     * 最终Binder底层将处理好的数据回调到此方法，并调用我们真正的sendMessage方法
     */
    @Override
    public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags)
    throws android.os.RemoteException {
        switch (code) {
            case INTERFACE_TRANSACTION: {
                reply.writeString(DESCRIPTOR);
                return true;
            }
            case TRANSACTION_sendMessage: {
                data.enforceInterface(DESCRIPTOR);
                com.example.aidl.data.MessageModel _arg0;
                if ((0 != data.readInt())) {
                    _arg0 = com.example.aidl.data.MessageModel.CREATOR.createFromParcel(data);
                } else {
                    _arg0 = null;
                }
                this.sendMessage(_arg0);
                reply.writeNoException();
                return true;
            }
        }
        return super.onTransact(code, data, reply, flags);
    }
```

```
    private static class Proxy implements com.example.aidl.MessageSender {
        private android.os.IBinder mRemote;
        Proxy(android.os.IBinder remote) {
            mRemote = remote;
        }
        /**
         * Proxy中的sendMessage方法，并不是直接调用我们定义的sendMessage方法，而是经过一顿的Parcel读写，
         * 然后调用mRemote.transact方法，把数据交给Binder处理，transact处理完毕后会调用上方的onTransact方法
         * onTransact拿到最终得到的参数数据，调用由我们真正的sendMessage方法
         */
        @Override
        public void sendMessage(com.example.aidl.data.MessageModel messageModel)
```

```
        try {
            _data.writeInterfaceToken(DESCRIPTOR);
            if ((messageModel != null)) {
                _data.writeInt(1);
                messageModel.writeToParcel(_data, 0);
            } else {
                _data.writeInt(0);
            }
            //调用Binder的transact方法进行多进程数据传输，处理完毕后调用上方的onTransact方法
            mRemote.transact(Stub.TRANSACTION_sendMessage, _data, _reply, 0);
            _reply.readException();
        } finally {
            _reply.recycle();
            _data.recycle();
        }
    }
}
```

```
    static final int TRANSACTION_sendMessage = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
}
public void sendMessage(com.example.aidl.data.MessageModel messageModel)
throws android.os.RemoteException;
```

只看代码的话，可能会有点懵逼，相信结合代码再看下方的流程图会更好理解：


asInterface
同一进程?
Yes
No
return Stub
return Stub.Proxy
Client sendMessage
Client sendMessage
Stub.sendMessage
Stub.Proxy.sendMessage
Parcel读写
Binder.transact
Stub.onTransact
End

若处于不同进程，整个数据传递的过程则需要通过Binder底层去进行编组（序列化，传输，接收和反序列化），得到最终的数据后再进行常规的方法调用。

敲黑板：对象跨进程传输的本质就是 序列化，传输，接收和反序列化 这样一个过程，这也是为什么跨进程传输的对象必须实现Parcelable接口

9

跨进程的回调接口

在上面我们已经实现了从客户端发送消息到跨进程服务端的功能，接下来我们还需要将服务端接收到的远程服务器消息，传递到客户端。有同学估计会说："这不就是一个回调接口的事情嘛"，设置回调接口思路是对的，但是在这里使用的回调接口有点不一样，在AIDL中传递的接口，不能是普通的接口，只能是AIDL接口，所以我们需要新建一个AIDL接口传到服务端，作为回调接口。

新建消息收取的AIDL接口MessageReceiver.aidl：

```
package com.example.aidl;
import com.example.aidl.data.MessageModel;

interface MessageReceiver {
    void onMessageReceived(in MessageModel receivedMessage);
}
```

```
package com.example.aidl;
import com.example.aidl.data.MessageModel;
import com.example.aidl.MessageReceiver;

interface MessageSender {
    void sendMessage(in MessageModel messageModel);

    void registerReceiveListener(MessageReceiver messageReceiver);

    void unregisterReceiveListener(MessageReceiver messageReceiver);
}
```

以上就是我们最终修改好的aidl接口，接下来我们需要做出对应的变更：

1. 在服务端中增加MessageSender的注册和反注册接口的方法；
2. 在客户端中实现MessageReceiver接口，并通过MessageSender注册到服务端。

客户端变更，修改MainActivity：

```
protected void onCreate(Bundle savedInstanceState) {
    //...
}
/**
 * 1.unregisterListener
 * 2.unbindService
 */
@Override
protected void onDestroy() {
    //解除消息监听接口
    if (messageSender != null && messageSender.asBinder().isBinderAlive()) {
        try {
            messageSender.unregisterReceiveListener(messageReceiver);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
    unbindService(serviceConnection);
    super.onDestroy();
}
//消息监听回调接口
private MessageReceiver messageReceiver = new MessageReceiver.Stub() {
    @Override
    public void onMessageReceived(MessageModel receivedMessage) throws RemoteException {
        Log.d(TAG, "onMessageReceived: " + receivedMessage.toString());
    }
};
ServiceConnection serviceConnection = new ServiceConnection() {
```

```
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        //使用asInterface方法取得AIDL对应的操作接口
        messageSender = MessageSender.Stub.asInterface(service);
        //生成消息实体对象
        MessageModel messageModel = new MessageModel();
        //...
        try {
```

```
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
        @Override
        public void onServiceDisconnected(ComponentName name) {
        }
    };
}
```

客户端主要有3个变更：

1. 增加了messageReceiver对象，用于监听服务端的消息通知；
2. onServiceConnected方法中，把messageReceiver注册到Service中去；
3. onDestroy时候解除messageReceiver的注册。

下面对服务端MessageServie进行变更：

```
//RemoteCallbackList专门用来管理多进程回调接口
private RemoteCallbackList<MessageReceiver> listenerList = new RemoteCallbackList<>();
public MessageService() {
}
IBinder messageSender = new MessageSender.Stub() {
    @Override
    public void sendMessage(MessageModel messageModel) throws RemoteException {
        Log.e(TAG, "messageModel: " + messageModel.toString());
    }
    @Override
    public void registerReceiveListener(MessageReceiver messageReceiver)
    throws RemoteException {
        listenerList.register(messageReceiver);
    }
    @Override
    public void unregisterReceiveListener(MessageReceiver messageReceiver)
    throws RemoteException {
        listenerList.unregister(messageReceiver);
    }
};
@Override
public IBinder onBind(Intent intent) {
    return messageSender;
}
@Override
public void onCreate() {
    super.onCreate();
    new Thread(new FakeTCPTask()).start();
}
```

```
@Override
public void onDestroy() {
    serviceStop.set(true);
    super.onDestroy();
}
//模拟长连接，通知客户端有新消息到达
private class FakeTCPTask implements Runnable {
    @Override
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            MessageModel messageModel = new MessageModel();
            messageModel.setFrom("Service");
            messageModel.setTo("Client");
            messageModel.setContent(String.valueOf(System.currentTimeMillis()));
            /**
             * RemoteCallbackList的遍历方式
             * beginBroadcast和finishBroadcast一定要配对使用
             */
            final int listenerCount = listenerList.beginBroadcast();
            Log.d(TAG, "listenerCount == " + listenerCount);
            for (int i = 0; i < listenerCount; i++) {
                MessageReceiver messageReceiver = listenerList.getBroadcastItem(i);
                if (messageReceiver != null) {
                    try {
                        messageReceiver.onMessageReceived(messageModel);
                    } catch (RemoteException e) {
                        e.printStackTrace();
                    }
                }
            }
            listenerList.finishBroadcast();
        }
    }
}
```

服务端主要变更：

1. MessageSender.Stub实现了注册和反注册回调接口的方法；
2. 增加了RemoteCallbackList来管理AIDL远程接口；

这里还有一个需要讲一下的，就是RemoteCallbackList，为什么要用RemoteCallbackList，普通ArrayList不行吗？当然不行，不然干嘛又整一个RemoteCallbackList □，registerReceiveListener 和 unregisterReceiveListener在客户端传输过来的对象，经过Binder处理，在服务端接收到的时候其实是一个新的对象，这样导致在 unregisterReceiveListener 的时候，普通的ArrayList是无法找到在 registerReceiveListener 时候添加到List的那个对象的，但是它们底层使用的Binder对象是同一个，RemoteCallbackList利用这个特性做到了可以找到同一个对象，这样我们就可以顺利反注册客户端传递过来的接口对象了。RemoteCallbackList在客户端进程终止后，它能自动移除客户端所注册的listener，它内部还实现了线程同步，所以我们在注册和反注册都不需要考虑线程同步，的确是个666的类。（至于使用ArrayList的幺蛾子现象，大家可以自己试试，篇幅问题，这里就不演示了）

到此，服务端通知客户端相关的代码也写完了，运行结果无非就是正确打印□就不贴图了，可以自己Run一下，打印的时候注意去选择不同的进程，不然瞪坏屏幕也没有日志。

10

DeathRecipient

你以为这样就完了？too young too simple…

不知道你有没有感觉到，两个进程交互总是觉得缺乏那么一点安全感…比如说服务端进程Crash了，而客户端进程想要调用服务端方法，这样就调用不到了。此时我们可以给Binder设置一个DeathRecipient对象，当Binder意外挂了的时候，我们可以在DeathRecipient接口的回调方法中收到通知，并作出相应的操作，比如重连服务等等。

DeathRecipient的使用如下：

2. 给Binder对象设置DeathRecipient对象。

在客户端MainActivity声明DeathRecipient：

```
 */
IBinder.DeathRecipient deathRecipient = new IBinder.DeathRecipient() {
    @Override
    public void binderDied() {
        Log.d(TAG, "binderDied");
        if (messageSender != null) {
            messageSender.asBinder().unlinkToDeath(this, 0);
            messageSender = null;
        }
        //// TODO: 2017/2/28 重连服务或其他操作
        setupService();
    }
};

ServiceConnection serviceConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        //...
        try {
            //设置Binder死亡监听
            messageSender.asBinder().linkToDeath(deathRecipient, 0);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
    //...
};
```

Binder中两个重要方法：

1. linkToDeath -> 设置死亡代理 DeathRecipient 对象；

2. unlinkToDeath -> Binder死亡的情况下，解除该代理。

权限验证

就算是公交车，上车也得嘀卡对不，如果希望我们的服务进程不想像公交车一样谁想上就上，那么我们可以加入权限验证。

介绍两种常用验证方法：

1. 在服务端的onBind中校验自定义permission，如果通过了我们的校验，正常返回Binder对象，校验不通过返回null，返回null的情况下客户端无法绑定到我们的服务；
2. 在服务端的onTransact方法校验客户端包名，不通过校验直接return false，校验通过执行正常的流程。

自定义permission，在Androidmanifest.xml中增加自定义的权限：

```
<permission
    android:name="com.example.aidl.permission.REMOTE_SERVICE_PERMISSION"
    android:protectionLevel="normal" />

<uses-permission android:name="com.example.aidl.permission.REMOTE_SERVICE_PERMISSION" />
```

服务端检查权限的方法：

```
    public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
    throws RemoteException {
        /**
         * 包名验证方式
         */
        String packageName = null;
        String[] packages = getPackageManager().getPackagesForUid(getCallingUid());
        if (packages != null && packages.length > 0) {
            packageName = packages[0];
        }
        if (packageName == null || !packageName.startsWith("com.example.aidl")) {
            Log.d("onTransact", "拒绝调用: " + packageName);
            return false;
        }
        return super.onTransact(code, data, reply, flags);
    }
};
@Override
public IBinder onBind(Intent intent) {
    //自定义permission方式检查权限
    if (checkCallingOrSelfPermission("com.example.aidl.permission.REMOTE_SERVICE_PERMISSION")
                == PackageManager.PERMISSION_DENIED) {
        return null;
    }
    return messageSender;
}
```

12

根据不同进程，做不同的初始化工作

相信前一两年很多朋友还在使用Android-Universal-Image-Loader来加载图片，它是需要在Application类进行初始化的。打个比如，我们用它来加载图片，而且还有一个图片选择进

这里提供一个简单粗暴的方法，博主也是这么干的...直接拿到进程名判断，作出相应操作即可：

```
public class MyApp extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        Log.d("process name", getProcessName());
    }
    //取得进程名
    private String getProcessName() {
        ActivityManager am = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
        List<ActivityManager.RunningAppProcessInfo> runningApps = am.getRunningAppProcesses();
        if (runningApps == null) {
            return null;
        }
        for (ActivityManager.RunningAppProcessInfo procInfo : runningApps) {
            if (procInfo.pid == Process.myPid()) {
                return procInfo.processName;
            }
        }
        return null;
    }
}
```

每个进程创建，都会调用Application的onCreate方法，这是一个需要注意的地方，我们也可以根据当前进程的pid，拿到当前进程的名字去做判断，然后做一些我们需要的逻辑，我们这个例子，拿到的两个进程名分别是：

1. 客户端进程：com.example.aidl
2. 服务端进程：com.example.aidl:remote

1. 多进程app可以在系统中申请多份内存，但应合理使用，建议把一些高消耗但不常用的模块放到独立的进程，不使用的进程可及时手动关闭；
2. 实现多进程的方式有多种：四大组件传递Bundle、Messenger、AIDL等，选择适合自己的使用场景；
3. Android中实现多进程通讯，建议使用系统提供的Binder类，该类已经实现了多进程通讯而不需要我们做底层工作；
4. 多进程应用，Application将会被创建多次；

14

结语

这篇文章断断续续写了很久，而且我相信真正使用起来的同学可能不多，选择这样一个话题我是吃力不讨好… 但是我还是希望可以在这里提供一个完整的解决方案给大家。简单的多进程使用，而且效果显著的，比如把图片选择和WebView配置到独立的进程，这个我希望可以大家行动起来。这篇文章的知识点非常多，理解起来可能不是太容易，如果有兴趣，我建议你手动去写一下，然后不理解的地方，打断点看看是什么样的运行步骤。

对于面试的同学，如果在面试过程中说到多进程，跟面试官聊得开，估计也是能加点分的，或者在实际工作中，一些使用多进程可以更好地解决问题的地方，你可以在会议中拍桌猛起，跟主管说：“我有一个大胆的想法…”，这样装逼也不错（当然，被炒了的话就不关我的事了…）

往日文章集合：

花了4个月整理了50篇Android干货文章

如果文章对您有用，请随意打赏。

Reward

people gave a reward

长按二维码向我转账

受苹果公司新规定影响，微信 iOS 版的赞赏功能被关闭，可通过二维码转账支持公众号。

Android

相关热门文章

你最不想错过的 **2017** 早期 **25** 个 **Android** 开源库

亦枫 59 2

一篇文章，全面总结**Android**面试知识点

Ruheng 239 2

Android开发：**RecyclerView**平滑流畅的滑动到指定位置

MichaelX 19