# Android：学习AIDL，这一篇文章就够了(上)（下在下面。。。）

# 正文

## 1，概述

AIDL是一个缩写，全称是Android Interface Definition Language，也就是Android接口定义语言。是的，首先我们知道的第一点就是：AIDL是一种语言。既然是一种语言，那么相应的就很自然的衍生出了一些问题：

* 为什么要设计出这么一门语言？
* 它有哪些语法？
* 我们应该如何使用它？
* 再深入一点，我们可以思考，我们是如何通过它来达到我们的目的的？
* 更深入一点，为什么要这么设计这门语言？会不会有更好的方式来实现我们的目的？

接下来，我们就一步步的来解答上面的这些问题。

ps：**1，**在研究AIDL相关的东西之前，一些必要的知识储备是要有的。一方面是关于Android中service相关的知识，要了解的比较通透才行，关于这方面的东西可以参考 [Android中的Service：默默的奉献者 (1)](https://link.jianshu.com/?t=http://blog.csdn.net/luoyanglizi/article/details/51586437)，[Android中的Service：Binder，Messenger，AIDL（2）](https://link.jianshu.com/?t=http://blog.csdn.net/luoyanglizi/article/details/51594016) 这两篇博文。另一方面是关于Android中序列化的相关知识，这方面的东西文中会简单提及，但是如果想要深入的研究一下的话最好还是去找一些这方面的资料看一下。 **2，**我的编译环境为Android Studio2.1.2，SDK Version 23，JDK 1.7。

## 2，为什么要设计这门语言？

设计这门语言的目的是为了实现**进程间通信**。

每一个进程都有自己的Dalvik VM实例，都有自己的一块独立的内存，都在自己的内存上存储自己的数据，执行着自己的操作，都在自己的那片狭小的空间里过完自己的一生。每个进程之间都你不知我，我不知你，就像是隔江相望的两座小岛一样，都在同一个世界里，但又各自有着自己的世界。而AIDL，就是两座小岛之间沟通的桥梁。相对于它们而言，我们就好像造物主一样，我们可以通过AIDL来制定一些规则，规定它们能进行哪些交流——比如，它们可以在我们制定的规则下传输一些特定规格的数据。

总之，通过这门语言，我们可以愉快的在一个进程访问另一个进程的数据，甚至调用它的一些方法，当然，只能是特定的方法。

## 3，它有哪些语法？

其实AIDL这门语言非常的简单，基本上它的语法和 Java 是一样的，只是在一些细微处有些许差别——毕竟它只是被创造出来简化Android程序员工作的，太复杂不好——所以在这里我就着重的说一下它和 Java 不一样的地方。主要有下面这些点：

* 文件类型：用AIDL书写的文件的后缀是 .aidl，而不是 .java。
* 数据类型：AIDL默认支持一些数据类型，在使用这些数据类型的时候是不需要导包的，但是除了这些类型之外的数据类型，在使用之前必须导包，**就算目标文件与当前正在编写的 .aidl 文件在同一个包下**——在 Java 中，这种情况是不需要导包的。比如，现在我们编写了两个文件，一个叫做 **Book.java** ，另一个叫做 **BookManager.aidl**，它们都在 **com.lypeer.aidldemo** 包下 ，现在我们需要在 .aidl 文件里使用 Book 对象，那么我们就必须在 .aidl 文件里面写上 **import com.lypeer.aidldemo.Book;** 哪怕 .java 文件和 .aidl 文件就在一个包下。  
  默认支持的数据类型包括：
* Java中的八种基本数据类型，包括 byte，short，int，long，float，double，boolean，char。
* String 类型。
* CharSequence类型。
* List类型：List中的所有元素必须是AIDL支持的类型之一，或者是一个其他AIDL生成的接口，或者是定义的parcelable（下文关于这个会有详解）。List可以使用泛型。
* Map类型：Map中的所有元素必须是AIDL支持的类型之一，或者是一个其他AIDL生成的接口，或者是定义的parcelable。Map是不支持泛型的。
* 定向tag：这是一个极易被忽略的点——这里的“被忽略”指的不是大家都不知道，而是很少人会正确的使用它。在我的理解里，定向 tag 是这样的：**AIDL中的定向 tag 表示了在跨进程通信中数据的流向，其中 in 表示数据只能由客户端流向服务端， out 表示数据只能由服务端流向客户端，而 inout 则表示数据可在服务端与客户端之间双向流通。其中，数据流向是针对在客户端中的那个传入方法的对象而言的。in 为定向 tag 的话表现为服务端将会接收到一个那个对象的完整数据，但是客户端的那个对象不会因为服务端对传参的修改而发生变动；out 的话表现为服务端将会接收到那个对象的的空对象，但是在服务端对接收到的空对象有任何修改之后客户端将会同步变动；inout 为定向 tag 的情况下，服务端将会接收到客户端传来对象的完整信息，并且客户端将会同步服务端对该对象的任何变动。**具体的分析大家可以移步我的另一篇博文：[**你真的理解AIDL中的in，out，inout么？**](https://www.jianshu.com/p/ddbb40c7a251)  
  另外，Java 中的基本类型和 String ，CharSequence 的定向 tag **默认且只能是 in** 。还有，请注意，**请不要滥用定向 tag** ，而是要根据需要选取合适的——要是不管三七二十一，全都一上来就用 inout ，等工程大了系统的开销就会大很多——因为排列整理参数的开销是很昂贵的。
* 两种AIDL文件：在我的理解里，所有的AIDL文件大致可以分为两类。一类是用来定义parcelable对象，以供其他AIDL文件使用AIDL中非默认支持的数据类型的。一类是用来定义方法接口，以供系统使用来完成跨进程通信的。可以看到，两类文件都是在“定义”些什么，而不涉及具体的实现，这就是为什么它叫做“Android接口定义语言”。  
  注：**所有的非默认支持数据类型必须通过第一类AIDL文件定义才能被使用。**

下面是两个例子，对于常见的AIDL文件都有所涉及：

/**/ Book.aidl**

**//第一类AIDL文件的例子**

**//这个文件的作用是引入了一个序列化对象 Book 供其他的AIDL文件使用**

**//注意：Book.aidl与Book.java的包名应当是一样的**

**package com.lypeer.ipcclient;**

**//注意parcelable是小写**

**parcelable Book;**

**// BookManager.aidl**

**//第二类AIDL文件的例子**

**package com.lypeer.ipcclient;**

**//导入所需要使用的非默认支持数据类型的包**

**import com.lypeer.ipcclient.Book;**

**interface BookManager {**

**//所有的返回值前都不需要加任何东西，不管是什么数据类型**

**List<Book> getBooks();**

**Book getBook();**

**int getBookCount();**

**//传参时除了Java基本类型以及String，CharSequence之外的类型**

**//都需要在前面加上定向tag，具体加什么量需而定**

**void setBookPrice(in Book book , int price)**

**void setBookName(in Book book , String name)**

**void addBookIn(in Book book);**

**void addBookOut(out Book book);**

**void addBookInout(inout Book book);**

**}**

## 4，如何使用AIDL文件来完成跨进程通信？

在进行跨进程通信的时候，在AIDL中定义的方法里包含非默认支持的数据类型与否，我们要进行的操作是不一样的。如果不包含，那么我们只需要编写一个AIDL文件，如果包含，那么我们通常需要写 n+1 个AIDL文件（ n 为非默认支持的数据类型的种类数）——显然，包含的情况要复杂一些。所以我接下来将只介绍AIDL文件中包含非默认支持的数据类型的情况，至于另一种简单些的情况相信大家是很容易从中触类旁通的。

### 4.1，使数据类实现 Parcelable 接口

由于不同的进程有着不同的内存区域，并且它们只能访问自己的那一块内存区域，所以我们不能像平时那样，传一个句柄过去就完事了——句柄指向的是一个内存区域，现在目标进程根本不能访问源进程的内存，那把它传过去又有什么用呢？**所以我们必须将要传输的数据转化为能够在内存之间流通的形式。**这个转化的过程就叫做序列化与反序列化。简单来说是这样的：比如现在我们要将一个对象的数据从客户端传到服务端去，我们就可以在客户端对这个对象进行序列化的操作，将其中包含的数据转化为序列化流，然后将这个序列化流传输到服务端的内存中去，再在服务端对这个数据流进行反序列化的操作，从而还原其中包含的数据——通过这种方式，我们就达到了在一个进程中访问另一个进程的数据的目的。

而通常，在我们通过AIDL进行跨进程通信的时候，选择的序列化方式是实现 Parcelable 接口。关于实现 Parcelable 接口之后里面具体有那些方法啦，每个方法是干嘛的啦，这些我就不展开来讲了，那並非这篇文章的重点，我下面主要讲一下如何快速的生成一个合格的可序列化的类（以Book.java为例）。

注：**若AIDL文件中涉及到的所有数据类型均为默认支持的数据类型，则无此步骤。因为默认支持的那些数据类型都是可序列化的。**

#### ****4.1.1，编译器自动生成****

我当前用的编译器是Android Studio 2.1.2，它是自带了 Parcelable 接口的模板的，只需要我们敲几下键盘就可以轻松的生成一个可序列化的 Parcelable 实现类。

首先，创建一个类，正常的书写其成员变量，建立getter和setter并添加一个无参构造，比如：

**public class Book{**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**this.name = name;**

**}**

**public int getPrice() {**

**return price;**

**}**

**public void setPrice(int price) {**

**this.price = price;**

**}**

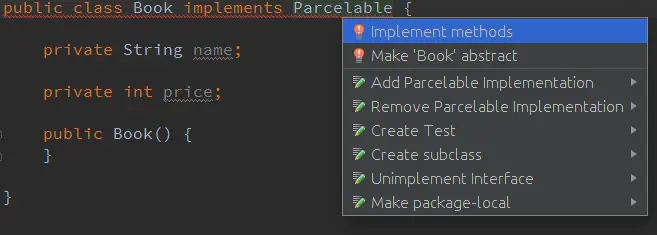
**private String name;**

**private int price;**

**public Book() {}**

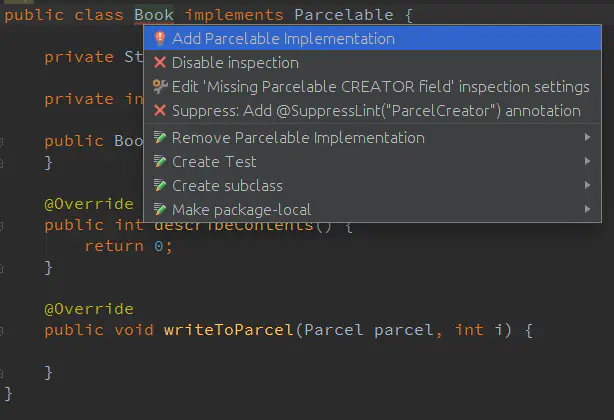
**}**

然后 implements Parcelable ，接着 as 就会报错，将鼠标移到那里，按下 alt+enter（as默认的自动解决错误的快捷键，如果你们的as有修改过快捷键的话以修改后的为准） 让它自动解决错误，这个时候它会帮你完成一部分的工作：



Parcelable实现第一步

在弹出来的框里选择所有的成员变量，然后确定。你会发现类里多了一些代码，但是现在还是会报错，Book下面仍然有一条小横线，再次将鼠标移到那里，按下 alt+enter 让它自动解决错误：



Parcelable实现第二步

这次解决完错误之后就不会报错了，这个 Book 类也基本上实现了 Parcelable 接口，可以执行序列化操作了。

但是请注意，这里有一个坑：**默认生成的模板类的对象只支持为 in 的定向 tag 。**为什么呢？因为默认生成的类里面只有 writeToParcel() 方法，而如果要支持为 out 或者 inout 的定向 tag 的话，还需要实现 readFromParcel() 方法——而这个方法其实并没有在 Parcelable 接口里面，所以需要我们从头写。具体为什么大家可以去看看：[**你真的理解AIDL中的in，out，inout么？**](https://www.jianshu.com/p/ddbb40c7a251)

那么这个 readFromParcel() 方法应当怎么写呢？这样写：

**@Override**

**public void writeToParcel(Parcel dest, int flags) {**

**dest.writeString(name);**

**dest.writeInt(price);**

**}**

**/\*\***

**\* 参数是一个Parcel,用它来存储与传输数据**

**\* @param dest**

**\*/**

**public void readFromParcel(Parcel dest) {**

**//注意，此处的读值顺序应当是和writeToParcel()方法中一致的**

**name = dest.readString();**

**price = dest.readInt();**

**}**

像上面这样添加了 readFromParcel() 方法之后，我们的 Book 类的对象在AIDL文件里就可以用 out 或者 inout 来作为它的定向 tag 了。

此时，完整的 Book 类的代码是这样的：

**package com.lypeer.ipcclient;**

**import android.os.Parcel;**

**import android.os.Parcelable;**

**/\*\***

**\* Book.java**

**\***

**\* Created by lypeer on 2016/7/16.**

**\*/**

**public class Book implements Parcelable{**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**this.name = name;**

**}**

**public int getPrice() {**

**return price;**

**}**

**public void setPrice(int price) {**

**this.price = price;**

**}**

**private String name;**

**private int price;**

**public Book(){}**

**public Book(Parcel in) {**

**name = in.readString();**

**price = in.readInt();**

**}**

**public static final Creator<Book> CREATOR = new Creator<Book>() {**

**@Override**

**public Book createFromParcel(Parcel in) {**

**return new Book(in);**

**}**

**@Override**

**public Book[] newArray(int size) {**

**return new Book[size];**

**}**

**};**

**@Override**

**public int describeContents() {**

**return 0;**

**}**

**@Override**

**public void writeToParcel(Parcel dest, int flags) {**

**dest.writeString(name);**

**dest.writeInt(price);**

**}**

**/\*\***

**\* 参数是一个Parcel,用它来存储与传输数据**

**\* @param dest**

**\*/**

**public void readFromParcel(Parcel dest) {**

**//注意，此处的读值顺序应当是和writeToParcel()方法中一致的**

**name = dest.readString();**

**price = dest.readInt();**

**}**

**//方便打印数据**

**@Override**

**public String toString() {**

**return "name : " + name + " , price : " + price;**

**}**

**}**

至此，关于AIDL中非默认支持数据类型的序列化操作就完成了。

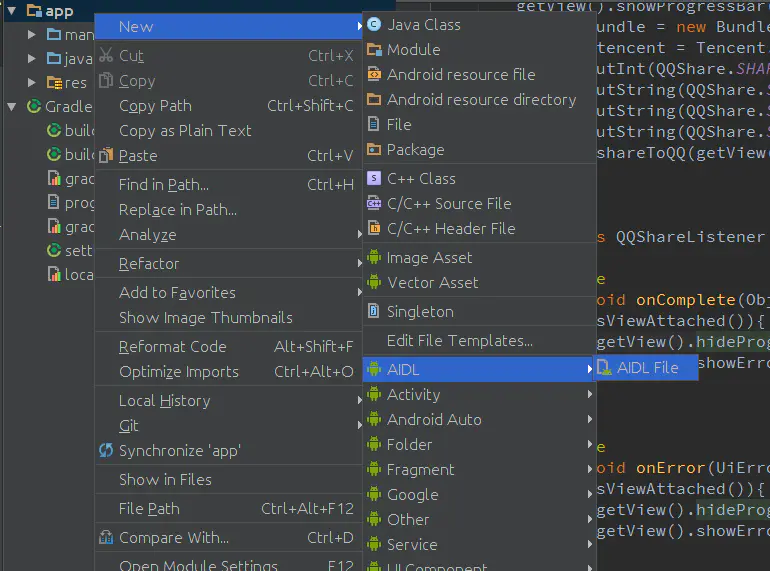
#### ****4.1.2，插件生成****

我不是很清楚 Eclipse 或者较低版本的 as 上会不会像 as 2.1.2 这样帮我们在实现 Parcelable 接口的过程中做如此多的操作，但是就算不会，我们还有其他的招数——通过插件来帮我们实现 Parcelable 接口。

具体的实现方式和实现过程大家可以参见这篇文章：[**告别手写parcelable**](https://link.jianshu.com/?t=http://blog.csdn.net/cheyiliu/article/details/46553129)

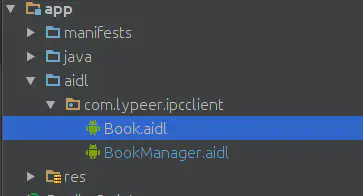
### 4.2，书写AIDL文件

首先我们需要一个 Book.aidl 文件来将 Book 类引入使得其他的 AIDL 文件其中可以使用 Book 对象。那么第一步，如何新建一个 AIDL 文件呢？Android Studio已经帮我们把这个集成进去了：



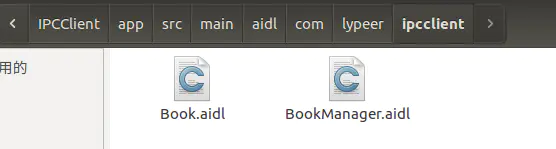
新建AIDL文件

鼠标移到app上面去，点击右键，然后 new->AIDL->AIDL File，按下鼠标左键就会弹出一个框提示生成AIDL文件了。生成AIDL文件之后，项目的目录会变成这样的：



建立AIDL文件后的项目目录

比起以前多了一个叫做 aidl 的包，而且他的层级是和 java 包相同的，并且 aidl 包里默认有着和 java 包里默认的包结构。那么如果你用的是 Eclipse 或者较低版本的 as ，编译器没有这个选项怎么办呢？没关系，我们也可以自己写。打开项目文件夹，依次进入 app->src->main，在 main 包下新建一个和 java 文件夹平级的 aidl 文件夹，然后我们手动在这个文件夹里面新建和 java 文件夹里面的默认结构一样的文件夹结构，再在最里层新建 .aidl 文件就可以了：



自己新建AIDL文件的目录

注意看图中的文件目录。

Ok，如何新建AIDL文件说的差不多了，接下来就该写AIDL文件的内容了。内容的话如果上一节有认真看的话基本上是没什么问题的。在这里，我们需要两个AIDL文件，我是这样写的：

**// Book.aidl**

**//第一类AIDL文件**

**//这个文件的作用是引入了一个序列化对象 Book 供其他的AIDL文件使用**

**//注意：Book.aidl与Book.java的包名应当是一样的**

**package com.lypeer.ipcclient;**

**//注意parcelable是小写**

**parcelable Book;**

**// BookManager.aidl**

**//第二类AIDL文件**

**//作用是定义方法接口**

**package com.lypeer.ipcclient;**

**//导入所需要使用的非默认支持数据类型的包**

**import com.lypeer.ipcclient.Book;**

**interface BookManager {**

**//所有的返回值前都不需要加任何东西，不管是什么数据类型**

**List<Book> getBooks();**

**//传参时除了Java基本类型以及String，CharSequence之外的类型**

**//都需要在前面加上定向tag，具体加什么量需而定**

**void addBook(in Book book);**

**}**

注意：**这里又有一个坑！**大家可能注意到了，在 Book.aidl 文件中，我一直在强调：**Book.aidl与Book.java的包名应当是一样的。**这似乎理所当然的意味着这两个文件应当是在同一个包里面的——事实上，很多比较老的文章里就是这样说的，他们说最好都在 aidl 包里同一个包下，方便移植——然而在 Android Studio 里并不是这样。如果这样做的话，系统根本就找不到 Book.java 文件，从而在其他的AIDL文件里面使用 Book 对象的时候会报 Symbol not found 的错误。为什么会这样呢？因为 Gradle 。大家都知道，Android Studio 是默认使用 Gradle 来构建 Android 项目的，而 Gradle 在构建项目的时候会通过 sourceSets 来配置不同文件的访问路径，从而加快查找速度——问题就出在这里。Gradle 默认是将 java 代码的访问路径设置在 java 包下的，这样一来，如果 java 文件是放在 aidl 包下的话那么理所当然系统是找不到这个 java 文件的。那应该怎么办呢？

又要 java文件和 aidl 文件的包名是一样的，又要能找到这个 java 文件——那么仔细想一下的话，其实解决方法是很显而易见的。首先我们可以把问题转化成：如何在保证两个文件包名一样的情况下，让系统能够找到我们的 java 文件？这样一来思路就很明确了：要么让系统来 aidl 包里面来找 java 文件，要么把 java 文件放到系统能找到的地方去，也即放到 java 包里面去。接下来我详细的讲一下这两种方式具体应该怎么做：

* 修改 build.gradle 文件：在 android{} 中间加上下面的内容：

**sourceSets {**

**main {**

**java.srcDirs = ['src/main/java', 'src/main/aidl']**

**}**

**}**

也就是把 java 代码的访问路径设置成了 java 包和 aidl 包，这样一来系统就会到 aidl 包里面去查找 java 文件，也就达到了我们的目的。只是有一点，这样设置后 Android Studio 中的项目目录会有一些改变，我感觉改得挺难看的。

* 把 java 文件放到 java 包下去：把 Book.java 放到 java 包里任意一个包下，保持其包名不变，与 Book.aidl 一致。只要它的包名不变，Book.aidl 就能找到 Book.java ，而只要 Book.java 在 java 包下，那么系统也是能找到它的。但是这样做的话也有一个问题，就是在移植相关 .aidl 文件和 .java 文件的时候没那么方便，不能直接把整个 aidl 文件夹拿过去完事儿了，还要单独将 .java 文件放到 java 文件夹里去。

我们可以用上面两个方法之一来解决找不到 .java 文件的坑，具体用哪个就看大家怎么选了，反正都挺简单的。

到这里我们就已经将AIDL文件新建并且书写完毕了，clean 一下项目，如果没有报错，这一块就算是大功告成了。

### 4.3，移植相关文件

我们需要保证，在客户端和服务端中都有我们需要用到的 .aidl 文件和其中涉及到的 .java 文件，因此不管在哪一端写的这些东西，写完之后我们都要把这些文件复制到另一端去。如果是用的上面两个方法中的第一个解决的找不到 .java 文件的问题，那么直接将 aidl 包复制到另一端的 main 目录下就可以了；如果是使用第二个方法的话，就除了把把整个 aidl 文件夹拿过去，还要单独将 .java 文件放到 java 文件夹里去。

### 4.4，编写服务端代码

通过上面几步，我们已经完成了AIDL及其相关文件的全部内容，那么我们究竟应该如何利用这些东西来进行跨进程通信呢？其实，在我们写完AIDL文件并 clean 或者 rebuild 项目之后，编译器会根据AIDL文件为我们生成一个与AIDL文件同名的 .java 文件，这个 .java 文件才是与我们的跨进程通信密切相关的东西。事实上，基本的操作流程就是：在服务端实现AIDL中定义的方法接口的具体逻辑，然后在客户端调用这些方法接口，从而达到跨进程通信的目的。

接下来我直接贴上我写的服务端代码：

**/\*\***

**\* 服务端的AIDLService.java**

**\* <p/>**

**\* Created by lypeer on 2016/7/17.**

**\*/**

**public class AIDLService extends Service {**

**public final String TAG = this.getClass().getSimpleName();**

**//包含Book对象的list**

**private List<Book> mBooks = new ArrayList<>();**

**//由AIDL文件生成的BookManager**

**private final BookManager.Stub mBookManager = new BookManager.Stub() {**

**@Override**

**public List<Book> getBooks() throws RemoteException {**

**synchronized (this) {**

**Log.e(TAG, "invoking getBooks() method , now the list is : " + mBooks.toString());**

**if (mBooks != null) {**

**return mBooks;**

**}**

**return new ArrayList<>();**

**}**

**}**

**@Override**

**public void addBook(Book book) throws RemoteException {**

**synchronized (this) {**

**if (mBooks == null) {**

**mBooks = new ArrayList<>();**

**}**

**if (book == null) {**

**Log.e(TAG, "Book is null in In");**

**book = new Book();**

**}**

**//尝试修改book的参数，主要是为了观察其到客户端的反馈**

**book.setPrice(2333);**

**if (!mBooks.contains(book)) {**

**mBooks.add(book);**

**}**

**//打印mBooks列表，观察客户端传过来的值**

**Log.e(TAG, "invoking addBooks() method , now the list is : " + mBooks.toString());**

**}**

**}**

**};**

**@Override**

**public void onCreate() {**

**super.onCreate();**

**Book book = new Book();**

**book.setName("Android开发艺术探索");**

**book.setPrice(28);**

**mBooks.add(book);**

**}**

**@Nullable**

**@Override**

**public IBinder onBind(Intent intent) {**

**Log.e(getClass().getSimpleName(), String.format("on bind,intent = %s", intent.toString()));**

**return mBookManager;**

**}**

**}**

整体的代码结构很清晰，大致可以分为三块：第一块是**初始化**。在 onCreate() 方法里面我进行了一些数据的初始化操作。第二块是**重写 BookManager.Stub 中的方法**。在这里面提供AIDL里面定义的方法接口的具体实现逻辑。第三块是**重写 onBind() 方法**。在里面返回写好的 BookManager.Stub 。

接下来在 Manefest 文件里面注册这个我们写好的 Service ，这个不写的话我们前面做的工作都是无用功：

**<service**

**android:name=".service.AIDLService"**

**android:exported="true">**

**<intent-filter>**

**<action android:name="com.lypeer.aidl"/>**

**<category android:name="android.intent.category.DEFAULT"/>**

**</intent-filter>**

**</service>**

到这里我们的服务端代码就编写完毕了，如果你对里面的一些地方感觉有些陌生或者根本不知所云的话，说明你对 Service 相关的知识已经有些遗忘了，建议再去看看这两篇博文：[Android中的Service：默默的奉献者 (1)](https://link.jianshu.com/?t=http://blog.csdn.net/luoyanglizi/article/details/51586437)，[Android中的Service：Binder，Messenger，AIDL（2）](https://link.jianshu.com/?t=http://blog.csdn.net/luoyanglizi/article/details/51594016) 。

### 4.5，编写客户端代码

前面说过，在客户端我们要完成的工作主要是调用服务端的方法，但是在那之前，我们首先要连接上服务端，完整的客户端代码是这样的：

**/\*\***

**\* 客户端的AIDLActivity.java**

**\* 由于测试机的无用debug信息太多，故log都是用的e**

**\* <p/>**

**\* Created by lypeer on 2016/7/17.**

**\*/**

**public class AIDLActivity extends AppCompatActivity {**

**//由AIDL文件生成的Java类**

**private BookManager mBookManager = null;**

**//标志当前与服务端连接状况的布尔值，false为未连接，true为连接中**

**private boolean mBound = false;**

**//包含Book对象的list**

**private List<Book> mBooks;**

**@Override**

**protected void onCreate(Bundle savedInstanceState) {**

**super.onCreate(savedInstanceState);**

**setContentView(R.layout.activity\_aidl);**

**}**

**/\*\***

**\* 按钮的点击事件，点击之后调用服务端的addBookIn方法**

**\***

**\* @param view**

**\*/**

**public void addBook(View view) {**

**//如果与服务端的连接处于未连接状态，则尝试连接**

**if (!mBound) {**

**attemptToBindService();**

**Toast.makeText(this, "当前与服务端处于未连接状态，正在尝试重连，请稍后再试", Toast.LENGTH\_SHORT).show();**

**return;**

**}**

**if (mBookManager == null) return;**

**Book book = new Book();**

**book.setName("APP研发录In");**

**book.setPrice(30);**

**try {**

**mBookManager.addBook(book);**

**Log.e(getLocalClassName(), book.toString());**

**} catch (RemoteException e) {**

**e.printStackTrace();**

**}**

**}**

**/\*\***

**\* 尝试与服务端建立连接**

**\*/**

**private void attemptToBindService() {**

**Intent intent = new Intent();**

**intent.setAction("com.lypeer.aidl");**

**intent.setPackage("com.lypeer.ipcserver");**

**bindService(intent, mServiceConnection, Context.BIND\_AUTO\_CREATE);**

**}**

**@Override**

**protected void onStart() {**

**super.onStart();**

**if (!mBound) {**

**attemptToBindService();**

**}**

**}**

**@Override**

**protected void onStop() {**

**super.onStop();**

**if (mBound) {**

**unbindService(mServiceConnection);**

**mBound = false;**

**}**

**}**

**private ServiceConnection mServiceConnection = new ServiceConnection() {**

**@Override**

**public void onServiceConnected(ComponentName name, IBinder service) {**

**Log.e(getLocalClassName(), "service connected");**

**mBookManager = BookManager.Stub.asInterface(service);**

**mBound = true;**

**if (mBookManager != null) {**

**try {**

**mBooks = mBookManager.getBooks();**

**Log.e(getLocalClassName(), mBooks.toString());**

**} catch (RemoteException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

**@Override**

**public void onServiceDisconnected(ComponentName name) {**

**Log.e(getLocalClassName(), "service disconnected");**

**mBound = false;**

**}**

**};**

**}**

同样很清晰，首先建立连接，然后在 ServiceConnection 里面获取 BookManager 对象，接着通过它来调用服务端的方法。

### 4.6，开始通信吧！

通过上面的步骤，我们已经完成了所有的前期工作，接下来就可以通过AIDL来进行跨进程通信了！将两个app同时运行在同一台手机上，然后调用客户端的 addBook() 方法，我们会看到服务端的 logcat 信息是这样的：

//服务端的 log 信息，我把无用的信息头去掉了，然后给它编了个号

1，on bind,intent = Intent { act=com.lypeer.aidl pkg=com.lypeer.ipcserver }

2，invoking getBooks() method , now the list is : [name : Android开发艺术探索 , price : 28]

3，invoking addBooks() method , now the list is : [name : Android开发艺术探索 , price : 28, name : APP研发录In , price : 2333]

客户端的信息是这样的：

//客户端的 log 信息

1，service connected

2，[name : Android开发艺术探索 , price : 28]

3，name : APP研发录In , price : 2333

所有的 log 信息都很正常并且符合预期——这说明我们到这里为止的步骤都是正确的，按照上面说的来做是能够正确的使用AIDL来进行跨进程通信的。

# 结语

这一篇文章主要介绍了我们在概述里提到的前三个问题，即：

* 为什么要设计AIDL语言？
* AIDL的语法是什么？
* 如何使用AIDL语言完成跨进程通信？

本来我是准备在这篇文章里把我那五个问题都讲完的，结果写到这里发现篇幅已经有些长了，再写的话可能就少有人有这个耐性读下去了——那么写在后面的这些又有什么意义呢？于是就干脆从这里截断，将AIDL的工作原理和它的设计思想以及我对于它的这种设计的一些看法放在下一篇博文里来讲述——刚好，有那么点基础篇和提高篇的意思，哈哈。

# Android：学习AIDL，这一篇文章就够了(下)

**简介：** 前言 上一篇博文介绍了关于AIDL是什么，为什么我们需要AIDL，AIDL的语法以及如何使用AIDL等方面的知识，这一篇博文将顺着上一篇的思路往下走，接着介绍关于AIDL的一些更加深入的知识。

## 前言

上一篇博文介绍了关于AIDL是什么，为什么我们需要AIDL，AIDL的语法以及如何使用AIDL等方面的知识，这一篇博文将顺着上一篇的思路往下走，接着介绍关于AIDL的一些更加深入的知识。强烈建议大家在看这篇博文之前先看一下上一篇博文：[**Android：学习AIDL，这一篇文章就够了(上)**](http://blog.csdn.net/chaoyu168/article/details/56673540?spm=a2c6h.12873639.0.0.16a9370exED1ae)

注：文中所有代码均源自上一篇博文中的例子。  
另：在看这篇博文之前，建议先将上一篇博文中的代码下载下来或者敲一遍，然后确定可以正常运行后再接着看。因为文中有大量对于具体代码的分析以及相关代码片段之间的跳转，如果你手头没有一份完整代码的话很容易看得一头雾水，最后浪费了你的时间也浪费了这篇博文。

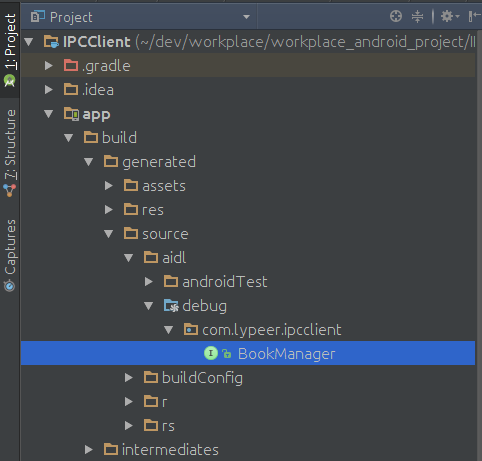
## 1、源码分析：AIDL文件是怎么工作的？

进行到上一篇文章的最后一步，我们已经学会了AIDL的全部用法，接下来让我们透过现象看本质，研究一下究竟AIDL是如何帮助我们进行跨进程通信的。

我们在上一篇提到过，在写完AIDL文件后，编译器会帮我们自动生成一个同名的 .java 文件——也许大家已经发现了，在我们实际编写客户端和服务端代码的过程中，真正协助我们工作的其实是这个文件，而 .aidl 文件从头到尾都没有出现过。这样一来我们就很容易产生一个疑问：**难道我们写AIDL文件的目的其实就是为了生成这个文件么？答案是肯定的。**事实上，就算我们不写AIDL文件，直接按照它生成的 .java 文件那样写一个 .java 文件出来，在服务端和客户端中也可以照常使用这个 .java 类来进行跨进程通信。**所以说AIDL语言只是在简化我们写这个 .java 文件的工作而已，而要研究AIDL是如何帮助我们进行跨进程通信的，其实就是研究这个生成的 .java 文件是如何工作的。**

### 1.1、这个文件在哪儿？

要研究它，首先我们就需要找到它，那么它在哪儿呢？在这里：



它在这儿

它的完整路径是：app->build->generated->source->aidl->debug->com->lypeer->ipcclient->BookManager.java（其中 com.lypeer.ipcclient 是包名，相对应的AIDL文件为 BookManager.aidl ）。在Android Studio里面目录组织方式由默认的 Android 改为 Project 就可以直接按照文件夹结构访问到它。

### 1.2、从应用看原理

和我一贯的分析方式一样，我们先不去看那些冗杂的源码，先从它在实际中的应用着手，辅以思考分析，试图寻找突破点。首先从服务端开始，刨去其他与此无关的东西，从宏观上我们看看它干了些啥：

**private** **final** BookManager.Stub mBookManager = **new** BookManager.Stub() {

@Override

**public** List<Book> getBooks() **throws** RemoteException {

*// getBooks()方法的具体实现*

}

@Override

**public** **void** addBook(Book book) **throws** RemoteException {

*// addBook()方法的具体实现*

}

};

**public** IBinder onBind(Intent intent) {

**return** mBookManager;

}

可以看到首先我们是对 BookManager.Stub 里面的抽象方法进行了重写——实际上，这些抽象方法正是我们在 AIDL 文件里面定义的那些。**也就是说，我们在这里为我们之前定义的方法提供了具体实现。**接着，在 onBind() 方法里我们将这个 BookManager.Stub 作为返回值传了过去。

接着看看客户端：

**private** BookManager mBookManager = **null**;

**private** ServiceConnection mServiceConnection = **new** ServiceConnection() {

@Override

**public** **void** onServiceConnected(ComponentName name, IBinder service)

mBookManager = BookManager.Stub.asInterface(service);

*//省略*

}

@Override

**public** **void** onServiceDisconnected(ComponentName name) {

*//省略*

}

};

**public** **void** addBook(View view) {

*//省略*

mBookManager.addBook(book);

}

简单的来说，客户端就做了这些事：获取 BookManager 对象，然后调用它里面的方法。

现在结合服务端与客户端做的事情，好好思考一下，我们会发现这样一个怪事情：**它们配合的如此紧密，以至于它们之间的交互竟像是同一个进程中的两个类那么自然！**大家可以回想下平时项目里的接口回调，基本流程与此一般无二。明明是在两个线程里面，数据不能直接互通，何以他们能交流的如此愉快呢？答案在 BookManager.java 里。

### 1.3、从客户端开始

一点开 BookManager.java ，我发现的第一件事是：BookManager 是一个接口类！**一看到它是个接口，我就知道，突破口有了。**为什么呢？接口意味着什么？方法都没有具体实现。但是明明在客户端里面我们调用了 mBookManager.addBook() ！那么就说明我们在客户端里面用到的 BookManager 绝不仅仅是 BookManager，而是它的一个实现类！那么我们就可以从这个实现类入手，看看在我们的客户端调用 addBook() 方法的时候，究竟 BookManager 在背后帮我们完成了哪些操作。首先看下客户端的 BookManager 对象是怎么来的：

**public** **void** onServiceConnected(ComponentName name, IBinder service)

mBookManager = BookManager.Stub.asInterface(service);

}

在这里我首先注意到的是方法的传参：IBinder service 。这是个什么东西呢？通过调试，我们可以发现，**这是个 BinderProxy 对象。**但随后我们会惊讶的发现：Java中并没有这个类！似乎研究就此陷入了僵局——其实不然。在这里我们没办法进一步的探究下去，那我们就先把这个问题存疑，从后面它的一些应用来推测关于它的更多的东西。

接下来顺藤摸瓜去看下这个 BookManager.Stub.asInterface() 是怎么回事：

**public** **static** com.lypeer.ipcclient.BookManager asInterface(android.os.IBinder obj) {

*//验空*

**if** ((obj == **null**)) {

**return** **null**;

}

*//DESCRIPTOR = "com.lypeer.ipcclient.BookManager"，搜索本地是否已經*

*//有可用的对象了，如果有就将其返回*

android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);

**if** (((iin != **null**) && (iin **instanceof** com.lypeer.ipcclient.BookManager))) {

**return** ((com.lypeer.ipcclient.BookManager) iin);

}

*//如果本地没有的话就新建一个返回*

**return** **new** com.lypeer.ipcclient.BookManager.Stub.Proxy(obj);

}

方法里首先进行了验空，这个很正常。第二步操作是调用了 **queryLocalInterface()** 方法，这个方法是 IBinder 接口里面的一个方法，而这里传进来的 IBinder 对象就是上文我们提到过的那个 service 对象。由于对 service 对象我们还没有一个很清晰的认识，这里也没法深究这个**queryLocalInterface()** 方法：它是 IBinder 接口里面的一个方法，那么显然，具体实现是在 service 的里面的，我们无从窥探。但是望文生义我们也能体会到它的作用，这里就姑且这么理解吧。第三步是创建了一个对象返回——很显然，这就是我们的目标，那个实现了 BookManager 接口的实现类。果断去看这个 BookManager.Stub.Proxy 类：

**private** **static** **class** Proxy **implements** com.lypeer.ipcclient.BookManager {

**private** android.os.IBinder mRemote;

Proxy(android.os.IBinder remote) {

*//此处的 remote 正是前面我们提到的 IBinder service*

mRemote = remote;

}

@Override

**public** java.util.List<com.lypeer.ipcclient.Book> getBooks() **throws** android.os.RemoteException {

*//省略*

}

@Override

**public** **void** addBook(com.lypeer.ipcclient.Book book) **throws** android.os.RemoteException {

*//省略*

}

*//省略部分方法*

}

看到这里，我们几乎可以确定：Proxy 类确实是我们的目标，客户端最终通过这个类与服务端进行通信。

那么接下来看看 getBooks() 方法里面具体做了什么：

@Override

**public** java.util.List<com.lypeer.ipcclient.Book> getBooks() **throws** android.os.RemoteException {

*//很容易可以分析出来，\_data用来存储流向服务端的数据流，*

*//\_reply用来存储服务端流回客户端的数据流*

android.os.Parcel \_data = android.os.Parcel.obtain();

android.os.Parcel \_reply = android.os.Parcel.obtain();

java.util.List<com.lypeer.ipcclient.Book> \_result;

**try** {

\_data.writeInterfaceToken(DESCRIPTOR);

*//调用 transact() 方法将方法id和两个 Parcel 容器传过去*

mRemote.transact(Stub.TRANSACTION\_getBooks, \_data, \_reply, 0);

\_reply.readException();

*//从\_reply中取出服务端执行方法的结果*

\_result = \_reply.createTypedArrayList(com.lypeer.ipcclient.Book.CREATOR);

} **finally** {

\_reply.recycle();

\_data.recycle();

}

*//将结果返回*

**return** \_result;

}

在这段代码里有几个需要说明的地方，不然容易看得云里雾里的：

* 关于 \_data 与 \_reply 对象：一般来说，我们会将方法的传参的数据存入\_data 中，而将方法的返回值的数据存入 \_reply 中——在没涉及定向 tag 的情况下。如果涉及了定向 tag ，情况将会变得稍微复杂些，具体是怎么回事请参见这篇博文：[你真的理解AIDL中的in，out，inout么？](http://www.open-open.com/lib/view/open1469494342021.html)
* 关于 Parcel ：简单的来说，Parcel 是一个用来存放和读取数据的容器。我们可以用它来进行客户端和服务端之间的数据传输，当然，它能传输的只能是可序列化的数据。具体 Parcel 的使用方法和相关原理可以参见这篇文章：[Android中Parcel的分析以及使用](http://blog.csdn.net/qinjuning/article/details/6785517)
* 关于 transact() 方法：这是客户端和服务端通信的核心方法。调用这个方法之后，客户端将会挂起当前线程，等候服务端执行完相关任务后通知并接收返回的 \_reply 数据流。关于这个方法的传参，这里有两点需要说明的地方：
  + 方法 ID ：transact() 方法的第一个参数是一个方法 ID ，这个是客户端与服务端约定好的给方法的编码，彼此一一对应。在AIDL文件转化为 .java 文件的时候，系统将会自动给AIDL文件里面的每一个方法自动分配一个方法 ID。
  + 第四个参数：transact() 方法的第四个参数是一个 int 值，它的作用是设置进行 IPC 的模式，为 0 表示数据可以双向流通，即 \_reply 流可以正常的携带数据回来，如果为 1 的话那么数据将只能单向流通，从服务端回来的 \_reply 流将不携带任何数据。  
    注：AIDL生成的 .java 文件的这个参数均为 0。

上面的这些如果要去一步步探究出结果的话也不是不可以，但是那将会涉及到 Binder 机制里比较底层的东西，一点点说完势必会将文章的重心带偏，那样就不好了——所以我就直接以上帝视角把结论给出来了。

另外的那个 addBook() 方法我就不去分析了，殊途同归，只是由于它涉及到了定向 tag ，所以有那么一点点的不一样，有兴趣的读者可以自己去试着阅读一下。接下来我总结一下在 Proxy 类的方法里面一般的工作流程：

* 1，生成 \_data 和 \_reply 数据流，并向 \_data 中存入客户端的数据。
* 2，通过 transact() 方法将它们传递给服务端，并请求服务端调用指定方法。
* 3，接收 \_reply 数据流，并从中取出服务端传回来的数据。

纵观客户端的所有行为，我们不难发现，其实一开始我们不能理解的那个 IBinder service 恰恰是客户端与服务端通信的灵魂人物——**正是通过用它调用的 transact() 方法，我们得以将客户端的数据和请求发送到服务端去。**从这个角度来看，这个 service 就像是服务端在客户端的代理一样——你想要找服务端？要传数据过去？行啊！你来找我，我给你把数据送过去——而 BookManager.java 中的那个 Proxy 类，就只能沦为二级代理了，我们在外部通过它来调动 service 对象。

至此，客户端在 IPC 中进行的工作已经分析完了，接下来我们看一下服务端。

### 1.4，接着看服务端

前面说了客户端通过调用 transact() 方法将数据和请求发送过去，那么理所当然的，服务端应当有一个方法来接收这些传过来的东西：在 BookManager.java 里面我们可以很轻易的找到一个叫做 onTransact() 的方法——看这名字就知道，多半和它脱不了关系，再一看它的传参

(int code, android.os.Parcel data, android.os.Parcel reply, int flags)——和 transact() 方法的传参是一样的！如果说他们没有什么 py 交易把我眼珠子挖出来当泡踩！下面来看看它是怎么做的：

@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\_getBooks: {

*//省略*

**return** **true**;

}

**case** TRANSACTION\_addBook: {

*//省略*

**return** **true**;

}

}

**return** **super**.onTransact(code, data, reply, flags);

}

可以看到，它在接收了客户端的 transact() 方法传过来的参数后，什么废话都没说就直接进入了一个 switch 选择：根据传进来的方法 ID 不同执行不同的操作。接下来看一下每个方法里面它具体做了些什么，以 getBooks() 方法为例：

**case** TRANSACTION\_getBooks: {

data.enforceInterface(DESCRIPTOR);

*//调用 this.getBooks() 方法，在这里开始执行具体的事务逻辑*

*//result 列表为调用 getBooks() 方法的返回值*

java.util.List<com.lypeer.ipcclient.Book> \_result = **this**.getBooks();

reply.writeNoException();

*//将方法执行的结果写入 reply ，*

reply.writeTypedList(\_result);

**return** **true**;

}

非常的简单直了，直接调用服务端这边的具体方法实现，然后获取返回值并将其写入 reply 流——当然，这是由于这个方法没有传入参数并且不涉及定向 tag 的关系，不然还会涉及到将传入参数从 data 中读取出来，以及针对定向 tag 的操作，具体的可以参考这篇博文：[你真的理解AIDL中的in，out，inout么？](http://www.open-open.com/lib/view/open1469494342021.html)。

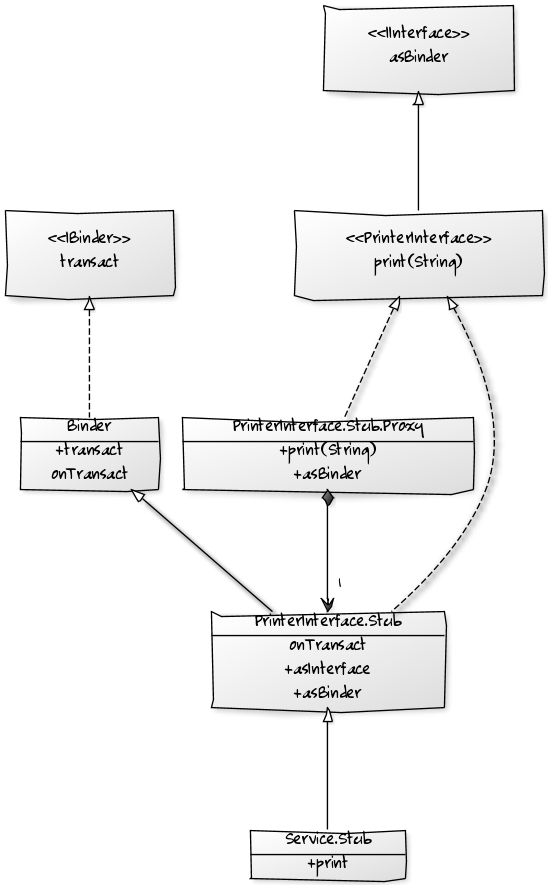
另外，还有一个问题，有些读者可能会疑惑，为什么这里没有看到关于将 reply 回传到客户端的相关代码？事实上，在客户端我们也没有看到它将相关参数传向服务端的相关代码——它只是把这些参数都传入了一个方法，其中过程同样是对我们隐藏的——服务端也同样，在执行完 return true 之后系统将会把 reply 流传回客户端，具体是怎么做的就不足为外人道也了。**不知道大家发现了没有，通过隐藏了这些细节，我们在 transact() 与 onTransact() 之间的调用以及数据传送看起来就像是发生在同一个进程甚至同一个类里面一样。**我们的操作就像是在一条直线上面走，根本感受不出来其中原来有过曲折——也许这套机制在设计之初，就是为了达到这样的目的。

分析到这里，服务端的工作我们也分析的差不多了，下面我们总结一下服务端的一般工作流程：

* 1，获取客户端传过来的数据，根据方法 ID 执行相应操作。
* 2，将传过来的数据取出来，调用本地写好的对应方法。
* 3，将需要回传的数据写入 reply 流，传回客户端。

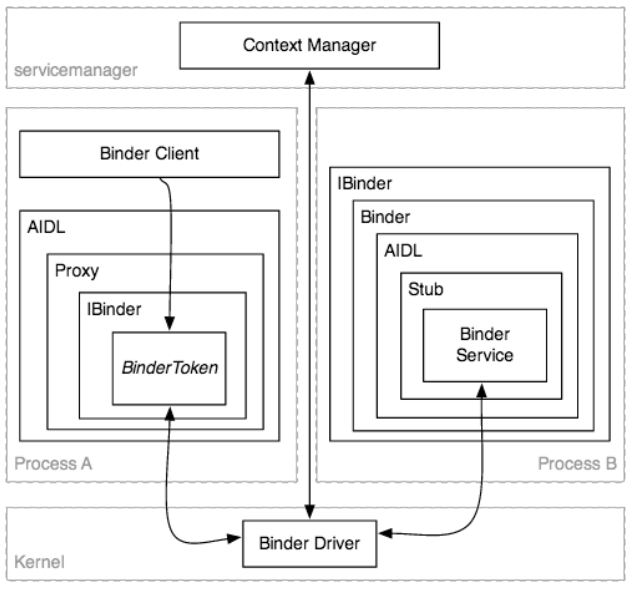
### 1.5，总结

现在我们已经完成了 BookManager.java 几乎所有的分析工作，接下来我想用两张图片来做一个总结。第一张是它的 UML 结构图：



AIDL的结构

第二张是客户端与服务端使用其进行 IPC 的工作流程：



AIDL的工作流程

剩下的就大家自己体味一下吧——如果前面的东西你看懂了，这里有没有我说的几句总结都差不多；如果前面你看的似懂非懂，看看这两张图片也就懂了；如果前面你几乎没有看懂，那么我写几句总结你还是看不懂。。。

## 2、为什么要这样设计？

这个问题可以拆分成两个子问题：

* 为什么AIDL的语法要这样设计？
* 为什么它生成的 .java 文件的结构要这样设计？

首先我有一个总的观点：**在程序设计领域，任何的解决方案，无非是基于需求和性能两方面的考虑。**首先是保证把需求完成，在这个大前提下保证性能最佳——这里的性能，就包括了代码的健壮性，可维护性等等林林总总的东西。

关于AIDL的语法为什么要这么设计，其实没有太大的研究的必要——因为他的语法实际上和 Java 没有多大区别，区别的地方也很容易想通，多是因为一些很显然的原因而不得不那样做。接下来我主要分析一下 BookManager.java 的设计之道。首先我们要明确需求：

* 基本需求当然是实现 IPC 。
* 在此基础上要尽可能的对开发者友好，即使用方便，且最好让开发者有那种在同一个进程中调用方法传输数据的爽感。

既然要实现 IPC ，一些核心的要素就不能少，**比如客户端接收到的 IBinder service ，比如 transact() 方法，比如 onTransact() 方法**——但是能让开发者察觉到这些这些东西的存在甚至自己写这些东西么？不能。为什么？因为这些东西做的事情其实非常的单调，无非就是那么几步，但是偏偏又涉及到很多对数据的写入读出的操作——涉及到数据流的东西一般都很繁琐。**把这些东西暴露出去显然是不合适的，还是建立一套模板把它封装起来比较的好。**但是归根结底，我们实现 IPC 是需要用到它们的，所以我们需要有一种途径去访问它们——在这个时候，**代理-桩的设计理念就初步成型了。**为了达到我们的目的，我们可以在客户端建立一个服务端的代理，在服务端建立一个客户端的桩，这样一来，客户端有什么需求可以直接跟代理说，代理跟它说你等等，我马上给你处理，然后它就告诉桩，客户端有这个需求了，桩就马上让服务端开始执行相应的事件，在执行结束后再通过桩把结果告诉代理，代理最后把结果给客户端。这样一来，客户端以为代理就是服务端，并且事实上它也只与代理进行了交互，而客户端与代理是在同一个进程中的，在服务端那边亦然——通过这种方式，我们就可以让客户端与服务端的通信看上去简单无比，像是从头到尾我们都在一个进程中工作一样。

在上面的设计思想指导之下，BookManager.java 为什么是我们看到的这个样子就很清楚明白了。

## 3、有没有更好的方式来完成 IPC ?

首先我要阐述的观点是：**如果你对这篇文章中上面叙述的那些内容有一定的掌握与理解了的话，完全脱离AIDL来手动书写客户端与服务端的相关文件来进行 IPC 是绝对没有问题的。**并且在了解了 IPC 得以进行的根本之后，你甚至完全没有必要照着 BookManager.java 来写，只要那几个点在，你想怎么写就怎么写。

**但是要说明的是，相较于使用AIDL来进行IPC，手动实现基本上是没有什么优势的。**毕竟AIDL是一门用来简化我们的工作的语言，用它确实可以省很多事。

那么现在除了AIDL与自己手动写，有没有其他的方式来进行 IPC 呢？答案是：有的。前段时间饿了么（这不算打广告吧。。。毕竟没有利益相关，只是纯粹的讨论技术）的一个工程师开源了一套 IPC 的框架，地址在这里：[**Hermes**](https://github.com/Xiaofei-it/Hermes)。这套框架的核心还是 IBinder service , transact() ，onTransact() 那些东西（事实上，任何和IPC有关的操作最终都还是要落在这些东西上面），但是他采取了一种巧妙的方式来实现：在服务端开启了一条默认进程，让这条进程来负责所有针对服务端的请求，同时采用注解的方式来注册类和方法，使得客户端能用这种形式和服务端建立约定，并且，这个框架对绑定service的那些细节隐藏的比较好，我们甚至都不需要在服务端写service，在客户端调用 bindService了——**三管齐下，使得我们可以远离以前那些烦人的有关service的操作了。**但是也并不是说这套框架就完全超越了AIDL，在某些方面它也有一些不足。比如，不知道是他的那个 Readme 写的太晦涩了还是怎么回事，我觉得使用它需要付出的学习成本还是比较大的；另外，在这套框架里面是将所有传向服务端的数据都放在一个 Mail 类里面的，而这个类的传输方式相当于AIDL里面定向 tag 为 in 的情况——也就是说，不要再想像AIDL里面那样客户端数据还能在服务端完成操作之后同步变化了。更多的东西我也还没看出来，还没用过这个框架，只是简单的看了下它的源码，不过总的来说能过看出来的是作者写的很用心，作者本身的Android功底也很强大，至少不知道比我强大到哪里去了......另外，想微微的吐槽一下，为什么这个框架用来进行IPC的核心类 IHermesService 里面长得和AIDL生成的 .java 一模一样啊一模一样......

总之，我想说的就是，虽然已经有AIDL了，但是并不意味着就不会出现比它更好的实现了——不止在这里是这样，这个观点可以推广到所有领域。

结语

这篇文章说是学习AIDL的，其实大部分的内容都是在通过AIDL生成的那个.java 文件讲 IPC 相关的知识——其实也就是 Binder 机制的利用的一部分——这也是为什么文中其实有很多地方没有深入下去讲，而是匆匆忙忙的给出了结论，因为再往下就不是应用层的东西了，讲起来比较麻烦，而且容易把人看烦。

讲到这里，基本上关于Android里面 IPC 相关的东西都已经讲得差不多了，如果你是从我写的[Android中的Service：默默的奉献者 (1)](http://www.open-open.com/lib/view/open1469493749528.html) --> [Android中的Service：Binder，Messenger，AIDL（2）](http://www.open-open.com/lib/view/open1469493830770.html) --> [Android：学习AIDL，这一篇文章就够了(上)](http://www.open-open.com/lib/view/open1469493649028.html) --> 现在这篇，这样一路看下来，并且是认真的看下来的话，基本上这一块的问题都难不倒你了。

另外，除了知识，我更希望通过我的博文传递的是一些解决问题分析问题的思路或者说是方法，所以我的很多博文都重在叙述思考过程而不是阐述结果——这样有好处也有坏处，好处是如果看懂了，能够收获更多，坏处是，大部分人都没有那个耐性慢慢的来看懂它，毕竟这需要思考，而当前很多的人都已经没有思考的时间，甚至丧失思考的能力了。

谢谢大家。

另：关于脱离AIDL自己写IPC的代码，我自己写了一份，大家可以聊作参考，[传送门](http://download.csdn.net/detail/luoyanglizi/9585999)。