

# Android IPC 机制

deepwaterooo

May 5, 2022

## Contents

<b>1 Android IPC 机制</b>	<b>1</b>
1.1 Android IPC 简介	1
1.2 Android 中的多进程模式	2
1.2.1 开启多进程模式	2
1.2.2 多进程模式的运行机制	3
1.3 IPC 基本概念介绍	4
1.3.1 Serializable 接口	4
1.3.2 Parcelable 接口	6
1.3.3 Binder	7
1.4 Android 中的 IPC 方式	14
1.4.1 使用 Bundle	14
1.4.2 使用文件共享	15
1.4.3 使用 Messenger	16
1.4.4 使用 AIDL	19
1.5 使用 ContentProvider	29
1.6 使用 Socket	35
1.7 Binder 连接池	39
1.8 各个 IPC 方式优缺点	43
<b>2 多进程应用的缺点</b>	<b>43</b>
2.1 静态成员和单例模式的完全失效	43
2.2 线程同步机制完全失效	43
2.3 Application 的 onCreate() 多次执行	43
2.4 调试麻烦	43
2.5 不能内存共享	44
2.6 文件共享问题。	44
2.7 sharedPreferences 不能在多进程中通用	44
2.8 多进程的通信问题	44
2.9 多进程暖启动的恢复问题	45
2.10 内存共享:	45
<b>3 Android 进程保活招式大全</b>	<b>45</b>
3.1 提高进程优先级方面	45
3.1.1 进程优先级介绍	45
3.1.2 如何提高进程优先级	46
3.2 进程死后拉活方案	47
3.2.1 利用系统广播拉活	47
3.2.2 利用系统 Service 机制拉活	47
3.2.3 覆写 Service 的 onDestroy 方法	47
3.2.4 利用 Native 守护进程拉活	47

3.2.5 利用 JobScheduler 机制拉活	48
3.2.6 利用帐号同步机制拉活	48
3.2.7 推送拉活	48
3.2.8 终极武器——客户粘性	49
3.3 使用多进程的其他场景补充	49
<b>4 多进程应用的使用场景</b>	<b>49</b>
<b>5 如何运行多进程</b>	<b>50</b>
5.1 挑战	50
5.2 共享数据	50
5.3 单例	50
5.4 解决办法	50
5.4.1 Content Provider	50
5.4.2 Bound Service	51
<b>6 Serializable 和 Parcelable 两种序列化</b>	<b>52</b>
<b>7 Android Service aidl 分析</b>	<b>52</b>
7.1 IRemoteBookBinder.aidl 文件来做分析	52
7.2 内部类 Default	53
7.3 IRemoteBookBinder.aidl 文件中声明的方法	53
7.4 内部类 Stub	54
7.4.1 DESCRIPTOR	55
7.4.2 构造方法	55
7.4.3 asInterface(android.os.IBinder obj)	55
7.4.4 asBinder() 方法	56
7.4.5 方法 id	56
7.4.6 onTransact() 方法	56
7.4.7 关键字 in, out, inout	56
7.5 内部类 Stub 的内部类 Proxy	57
7.5.1 构造方法	58
7.5.2 方法调用	58
7.6 总结图	60

# 1 Android IPC 机制

- 自从准备第一轮 technical 认真实现了 content provider (server and client) 的项目之后，我终于进阶到了安卓四大组件的 content provider，并对后来的 jetpack libraries, app startup ultimate 以及 (IPC, Messenger, aidl, socket) 等进程间通讯，以及启动优化、布局优化等的底层原理，能清楚地感觉到自己被暴露和引及到一个更深的学习阶段。但还需要通过项目、消化理解得更深入一些 (Binder 以及 content provider 底层原理内存映射等需要深入理解)。
- 此文进程间通讯纪录源自于<https://blog.sweetm.top/2019/03/22/android-ipc-%E6%9C%BA%E5%88%B6/>，写得很好，终于自己实现了一个 socket 相关的帮助理解的项目

## 1.1 Android IPC 简介

- IPC 是 Inter-Process Communication 的缩写，含义为进程间通信或者跨进程通信，是指两个进程之间进行数据交换的过程。说起进程间通信，我们首先要理解什么是进程，什么是线程，进程和线程是截然不同的概念。按照操作系统中的描述，线程是 CPU 调度的最小单元，同时线程是一种有限的系统资源。而进程一般指一个执行单元，在 PC 和移动设备上指一个程序或者一个应用。一个进程可以包含多个线程，因此进程和线程是包含与被包含的关系。最简单的

情况下，一个进程中可以只有一个线程，即主线程，在 Android 里面主线程也叫 UI 线程，在 UI 线程里才能操作界面元素。很多时候，一个进程中需要执行大量耗时的任务，如果这些任务放在主线程中去执行就会造成界面无法响应，严重影响用户体验，这种情况在 PC 系统和移动系统中都存在，在 Android 中有一个特殊的名字叫做 ANR (Application Not Responding)，即应用无响应。解决这个问题就需要用到线程，把一些耗时的任务放在线程中即可。

- IPC 不是 Android 中所独有的，任何一个操作系统都需要有相应的 IPC 机制，比如 Windows 上可以通过剪贴板、管道和邮槽等来进行进程间通信；Linux 上可以通过命名管道、共享内容、信号量等来进行进程间通信。可以看到不同的操作系统平台有着不同的进程间通信方式，对于 Android 来说，它是一种基于 Linux 内核的移动操作系统，它的进程间通信方式并不能完全继承自 Linux，相反，它有自己的进程间通信方式。在 Android 中最有特色的进程间通信方式就是 Binder 了，通过 Binder 可以轻松地实现进程间通信。除了 Binder，Android 还支持 Socket，通过 Socket 也可以实现任意两个终端之间的通信，当然同一个设备上的两个进程通过 Socket 通信自然也是可以的。
- 说到 IPC 的使用场景就必须提到多进程，只有面对多进程这种场景下，才需要考虑进程间通信。这个是很好的理解的，如果只有一个进程在运行，又何谈多进程呢？多进程的情况分为两种。第一种情况是一个应用因为某些原因自身需要采用多进程模式来实现，至于原因，可能有很多，比如有些模块由于特殊原因需要运行在单独的进程中，又或者为了加大一个应用可使用的内存所以需要通过多进程来获取多份内存空间。Android 对单个应用所使用的最大内存做了限制，早期的一些版本可能是 16MB，不同设备有不同的大小。另一种情况是当前应用需要向其他应用获取数据，由于是两个应用，所以必须采用跨进程的方式来获取所需的数据，甚至我们通过系统提供的 ContentProvider 去查询数据的时候，其实也是一种进程间通信，只不过通信细节被系统内部屏蔽了，我们无法感知而已。后续章节会详细介绍 ContentProvider 的底层实现，这里就先不做详细介绍了。总之，不管由于何种原因，我们采用了多进程的设计方法，那么应用中就必须妥善地处理进程间通信的各种问题。

## 1.2 Android 中的多进程模式

- 在正式介绍进程间通信之前，我们必须先要理解 Android 中的多进程模式。通过给四大组件指定
- android:process 属性，我们可以轻易地开启多进程模式，这看起来很简单，但是实际使用过程中却暗藏杀机，多进程远远没有我们想的那么简单，有时候我们通过多进程得到的好处甚至都不足以弥补使用多进程所带来的代码层面的负面影响。下面会详细分析这些问题。

### 1.2.1 开启多进程模式

- 正常情况下，在 Android 中多进程是指一个应用中存在多个进程的情况，因此这里不讨论两个应用之间的多进程情况。首先，在 Android 中使用多进程只有一种方法，那就是给四大组件 (Activity、Service、Receiver、ContentProvider) 在 AndroidManifest 中指定 android:process 属性，除此之外没有其他办法，也就是说我们无法给一个线程或者一个实体类指定其运行时所在的进程。其实还有另一种非常规的多进程方法，那就是通过 JNI 在 native 层去 fork 一个新的进程，但是这种方法属于特殊情况，也不是常用的创建多进程的方式，因此我们暂时不考虑这种方式。下面是一个示例，描述了如何在 Android 中创建多进程：

```
<activity
    android:name=".MainActivity"
    android:configChanges="orientation|screenSize"
    android:label="@string/app_name"
    android:launchMode="standard">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity
    android:name=".SecondActivity"
```

```

android:configChanges="screenLayout"
android:label="@string/app_name"
android:process=":remote" />
<activity
    android:name=".ThirdActivity"
    android:configChanges="screenLayout"
    android:label="@string/app_name"
    android:process="com.ryg.chapter_2.remote" />

```

- 上面的示例分别为 **SecondActivity** 和 **ThirdActivity** 指定了 **process** 属性，并且它们的属性值不同，这意味着当前应用又增加了两个新进程。假设当前应用的包名为 “com.ryg.chapter2”，当 **SecondActivity** 启动时，系统会为它创建一个单独的进程，进程名为
- “com.ryg.chapter 2: remote”；当 **ThirdActivity** 启动时，系统也会为它创建一个单独的进程，进程名为 “com.ryg.chapter\_2.remote”。同时入口 Activity 是 **MainActivity**，没有为它指定 **process** 属性，那么它运行在默认进程中，默认进程的进程名是包名。下面我们运行一下看看效果，如下图。进程列表末尾存在 3 个进程，进程 id 分别为 3039、3056、3098，这说明我们的应用成功地使用了多进程技术，是不是很简单呢？这只是开始，实际使用中多进程是有很多问题需要处理的。
- 使用 **adb shell** 来查看进程信息

```

vbox86p:/ # ps | grep com.ryg.chapter_2
u0_a65    3039   249    856408 102248    ep_poll e8b89bb9 S com.ryg.chapter_2
u0_a65    3056   249    837576 85156    ep_poll e8b89bb9 S com.ryg.chapter_2:remote
u0_a65    3098   249    836072 82460    ep_poll e8b89bb9 S com.ryg.chapter_2.remote

```

- 不知道读者朋友有没有注意到，**SecondActivity** 和 **ThirdActivity** 的 **android:process** 属性分别为 “: remote” 和 “com.ryg.chapter\_2.remote”，那么这两种方式有区别吗？
- 其实是有区别的，区别有两方面：
  - 首先，“:” 的含义是指要在当前的进程名前面附加上当前的包名，这是一种简写的方法，对于 **SecondActivity** 来说，它完整的进程名为 **com.ryg.chapter\_2: remote**，这可以从上面图片的进程信息也能看出来，而对于 **ThirdActivity** 中的声明方式，它是一种完整的命名方式，不会附加包名信息；
  - 其次，进程名以 “:” 开头的进程属于当前应用的私有进程，其他应用的组件不可以和它跑在同一个进程中，而进程名不以 “:” 开头的进程属于全局进程，其他应用通过 **ShareUID** 方式可以和它跑在同一个进程中。
- 我们知道 **Android** 系统会为每个应用分配一个唯一的 **UID**，具有相同 **UID** 的应用才能共享数据。这里要说明的是，两个应用通过 **ShareUID** 跑在同一个进程中是有要求的，需要这两个应用有相同的 **ShareUID** 并且签名相同才可以。在这种情况下，它们可以互相访问对方的私有数据，比如 **data** 目录、组件信息等，不管它们是否跑在同一个进程中。当然如果它们跑在同一个进程中，那么除了能共享 **data** 目录、组件信息，还可以共享内存数据，或者说它们看起来就像是一个应用的两个部分。

### 1.2.2 多进程模式的运行机制

- 如果用一句话来形容多进程，那笔者只能这样说：“当应用开启了多进程以后，各种奇怪的现象都出现了”。为什么这么说呢？这是有原因的。大部分人都认为开启多进程是很简单的事情，只需要给四大组件指定 **android:process** 属性即可。比如说在实际的产品开发中，可能会有多进程的需求，需要把某些组件放在单独的进程中去运行，很多人都会觉得这不很简单吗？
- 然后迅速地给那些组件指定了 **android:process** 属性，然后编译运行，发现“正常地运行起来了”。
- 这里笔者想说的是，那是真的正常地运行起来了吗？现在先不置可否，下面先举个例子，然后引入本节的话题。还是本章刚开始说的那个例子，其中 **SecondActivity** 通过指定 **android:process** 属性从而使其运行在一个独立的进程中，这里做了一些改动，我们新建了一个类，叫做 **UserManager**，这个类中有一个 **public** 的静态成员变量，如下所示。

```
public class UserManager {
    public static int sUserId = 1;
}
```

- 然后在 MainActivity 的 onCreate 中我们把这个 sUserId 重新赋值为 2，打印出这个静态变量的值后再启动 SecondActivity，在 SecondActivity 中我们再打印一下 sUserId 的值。按照正常的逻辑，静态变量是可以在所有的地方共享的，并且一处有修改处处都会同步，下面是运行时所打印的日志，我们看一下结果如何。

```
2019-03-21 20:53:47.451 1682-1682/com.ryg.chapter_2 D/MainActivity: UserManager.sUserId=2
2019-03-21 20:56:03.063 1852-1852/com.ryg.chapter_2:remote D/SecondActivity: UserManager.sUserId=1
```

- 上述问题出现的原因是 SecondActivity 运行在一个单独的进程中，我们知道 Android 为每一个应用分配了一个独立的虚拟机，或者说为每个进程都分配一个独立的虚拟机，不同的虚拟机在内存分配上有不同的地址空间，这就导致在不同的虚拟机中访问同一个类的对象会产生多份副本。拿我们这个例子来说，在进程 com.ryg.chapter\_2 和进程 com.ryg.chapter\_2:remote 中都存在一个 UserManager 类，并且这两个类是互不干扰的，在一个进程中修改 sUserId 的值只会影响当前进程，对其他进程不会造成任何影响，这样我们就可以理解为什么在 MainActivity 中修改了 sUserId 的值，但是在 SecondActivity 中 sUserId 的值却没有发生改变这个现象。
- 所有运行在不同进程中的四大组件，只要它们之间需要通过内存来共享数据，都会共享失败，这也是多进程所带来的主要影响。正常情况下，四大组件中间不可能不通过一些中间层来共享数据，那么通过简单地指定进程名来开启多进程都会无法正确运行。当然，特殊情况下，某些组件之间不需要共享数据，这个时候可以直接指定 android:process 属性来开启多进程，但是这种场景是不常见的，几乎所有情况都需要共享数据。
- 一般来说，使用多进程会造成如下几方面的问题：
  - (1) 静态成员和单例模式完全失效。
  - (2) 线程同步机制完全失效。
  - (3) SharedPreferences 的可靠性下降。
  - (4) Application 会多次创建。
- 第 1 个问题在上面已经进行了分析。
- 第 2 个问题本质上和第一个问题是类似的，既然都不是一块内存了，那么不管是锁对象还是锁全局类都无法保证线程同步，因为不同进程锁的不是同一个对象。
- 第 3 个问题是因为 SharedPreferences 不支持两个进程同时去执行写操作，否则会导致一定几率的数据丢失，这是因为 SharedPreferences 底层是通过读/写 XML 文件来实现的，并发写显然是可能出问题的，甚至并发读/写都有可能出问题。
- 第 4 个问题也是显而易见的，当一个组件跑在一个新的进程中的时候，由于系统要在创建新的进程同时分配独立的虚拟机，所以这个过程其实就是启动一个应用的过程。因此，相当于系统又把这个应用重新启动了一遍，既然重新启动了，那么自然会创建新的 Application。这个问题其实可以这么理解，运行在同一个进程中的组件是属于同一个虚拟机和同一个 Application 的，同理，运行在不同进程中的组件是属于两个不同的虚拟机和 Application 的。为了更加清晰地展示这一点，下面我们来做一个测试，首先在 Application 的 onCreate 方法中打印出当前进程的名字，然后连续启动三个同一个应用内但属于不同进程的 Activity，按照期望，Application 的 onCreate 应该执行三次并打印出三次进程名不同的 log，代码如下所示。

```
2019-03-21 21:06:47.581 2024-2024/com.ryg.chapter_2 D/MyApplication: application start, process name:com.ryg.chapter_2
2019-03-21 21:06:47.702 2041-2041/com.ryg.chapter_2:remote D/MyApplication: application start, process name:com.ryg.chapter_2
2019-03-21 21:07:01.798 2079-2079/? D/MyApplication: application start, process name:com.ryg.chapter_2.remote
```



## 1.3 IPC 基本概念介绍

- 主要介绍 IPC 中的一些基础概念，主要包含三方面内容：Serializable 接口、Parcelable 接口以及 Binder，只有熟悉这三方面的内容后，我们才能更好地理解跨进程通信的各种方式。Serializable 和 Parcelable 接口可以完成对象的序列化过程，当我们需要通过 Intent 和 Binder 传输数据时就需要使用 Parcelable 或者 Serializable。还有的时候我们需要把对象持久化到存储设备上或者通过网络传输给其他客户端，这个时候也需要使用 Serializable 来完成对象的持久化，下面先介绍如何使用 Serializable 来完成对象的序列化。

### 1.3.1 Serializable 接口

- Serializable 是 Java 所提供的一个序列化接口，它是一个空接口，为对象提供标准的序列化和反序列化操作。使用 Serializable 来实现序列化相当简单，只需要在类的声明中指定一个类似下面的标识即可自动实现默认的序列化过程。
- `private static final long serialVersionUID = 519067123721295773L;`
- 在 Android 中也提供了新的序列化方式，那就是 Parcelable 接口，使用 Parcelable 来实现对象的序列号，其过程要稍微复杂一些，本节先介绍 Serializable 接口。上面提到，想让一个对象实现序列化，只需要这个类实现 Serializable 接口并声明一个 serialVersionUID 即可，实际上，甚至这个 serialVersionUID 也不是必需的，我们不声明这个 serialVersionUID 同样也可以实现序列化，但是这将会对反序列化过程产生影响，具体什么影响后面再介绍。
- User 类就是一个实现了 Serializable 接口的类，它是可以被序列化和反序列化的，如下所示。

```
public class User implements Serializable {  
    private static final long serialVersionUID = 519067123721295773L;  
    public int userId;  
    public String userName;  
    public boolean isMale;  
    public Book book;  
}
```

- 通过 Serializable 方式来实现对象的序列化，实现起来非常简单，几乎所有工作都被系统自动完成了。如何进行对象的序列化和反序列化也非常简单，只需要采用 ObjectOutputStream 和 ObjectInputStream 即可轻松实现。下面举个简单的例子。

```
//序列化  
User user = new User(0, "jake", true);  
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("cache.txt"));  
out.writeObject(user);  
out.close();  
//反序列化  
ObjectInputStream in = new ObjectInputStream(new FileInputStream("cache.txt"));  
User newUser = (User) in.readObject();  
in.close();
```

- 上述代码演示了采用 Serializable 方式序列化对象的典型过程，很简单，只需要把实现了 Serializable 接口的 User 对象写到文件中就可以快速恢复了，恢复后的对象 newUser 和 user 的内容完全一样，但是两者并不是同一个对象。
- 刚开始提到，即使不指定 serialVersionUID 也可以实现序列化，那到底要不要指定呢？如果指定的话，serialVersionUID 后面那一长串数字又是什么含义呢？我们要明白，系统既然提供了这个 serialVersionUID，那么它必须是有用的。这个 serialVersionUID 是用来辅助序列化和反序列化过程的，原则上序列化后的数据中的 serialVersionUID 只有和当前类的 serialVersionUID 相同才能够正常地被反序列化。。serialVersionUID 的详细工作机制是这样的：序列化的时候系统会把当前类的 serialVersionUID 写入序列化的文件中（也可能是其他中介），当反序列化的时候系统会去检测文件中的 serialVersionUID，看它是否和当前类的 serialVersionUID 一致，如果一致就说明序列化的类的版本和当前类的版本是相同的，这个时候可以成功反序列化；否则就说明当前类和序列化的类相比发生了某些变换，比如成员变量的数量、类型可能发生了改变，这个时候是无法正常反序列化的，因此会报如下错误：

```
java.io.InvalidClassException: Main; local class incompatible: streamclassdesc serialVersionUID=8711368828010083044, local
```

- 一般来说，我们应该手动指定 `serialVersionUID` 的值，比如 1L，也可以让 Eclipse 根据当前类的结构自动去生成它的 hash 值，这样序列化和反序列化时两者的 `serialVersionUID` 是相同的，因此可以正常进行反序列化。如果不手动指定 `serialVersionUID` 的值，反序列化时当前类有所改变，比如增加或者删除了某些成员变量，那么系统就会重新计算当前类的 hash 值并把它赋值给 `serialVersionUID`，这个时候当前类的 `serialVersionUID` 就和序列化的数据中的 `serialVersionUID` 不一致，于是反序列化失败，程序就会出现 crash。所以，我们可以明显感觉到 `serialVersionUID` 的作用，当我们手动指定了它以后，就可以在很大程度上避免反序列化过程的失败。比如当版本升级后，我们可能删除了某个成员变量也可能增加了一些新的成员变量，这个时候我们的反向序列化过程仍然能够成功，程序仍然能够最大限度地恢复数据，相反，如果不指定 `serialVersionUID` 的话，程序则会挂掉。当然我们还要考虑另外一种情况，如果类结构发生了非常规性改变，比如修改了类名，修改了成员变量的类型，这个时候尽管 `serialVersionUID` 验证通过了，但是反序列化过程还是会失败，因为类结构有了毁灭性的改变，根本无法从老版本的数据中还原出一个新的类结构的对象。
- 根据上面的分析，我们可以知道，给 `serialVersionUID` 指定为 1L 或者采用 Eclipse 根据当前类结构去生成的 hash 值，这两者并没有本质区别，效果完全一样。以下两点需要特别提一下，首先静态成员变量属于类不属于对象，所以不会参与序列化过程；其次用 `transient` 关键字标记的成员变量不参与序列化过程。

### 1.3.2 Parcelable 接口

- 通过 `Serializable` 方式来实现序列化的方法，本节接着介绍另一种序列化方式：`Parcelable`。`Parcelable` 也是一个接口，只要实现这个接口，一个类的对象就可以实现序列化并可以通过 `Intent` 和 `Binder` 传递。下面的示例是一个典型的用法。

```
package com.ryg.chapter_2.model;
import java.io.Serializable;
import com.ryg.chapter_2.aidl.Book;
import android.os.Parcel;
import android.os.Parcelable;
public class User implements Parcelable, Serializable {
    private static final long serialVersionUID = 519067123721295773L;
    public int userId;
    public String userName;
    public boolean isMale;
    public Book book;
    public User() { }
    public User(int userId, String userName, boolean isMale) {
        this.userId = userId;
        this.userName = userName;
        this.isMale = isMale;
    }
    @Override
    public int describeContents() {
        return 0;
    }
    @Override
    public void writeToParcel(Parcel out, int flags) {
        out.writeInt(userId);
        out.writeString(userName);
        out.writeInt(isMale ? 1 : 0);
        out.writeParcelable(book, 0);
    }
    public static final Parcelable.Creator<User> CREATOR = new Parcelable.Creator<User>() {
        @Override
        public User createFromParcel(Parcel in) {
            return new User(in);
        }
        @Override
        public User[] newArray(int size) {
            return new User[size];
        }
    };
    private User(Parcel in) {
        userId = in.readInt();
```

```
        userName = in.readString();
        isMale = in.readInt() == 1;
        book = in.readParcelable(Thread.currentThread().getContextClassLoader());
    }
    @Override
    public String toString() {
        return String.format(
            "User:{userId:%s, userName:%s, isMale:%s}, with child:{%s}",
            userId, userName, isMale, book);
    }
}
```

- 这里先说一下 Parcel，Parcel 内部包装了可序列化的数据，可以在 Binder 中自由传输。从上述代码中可以看出，在序列化过程中需要实现的功能有序列化、反序列化和内容描述。序列化功能由 writeToParcel 方法来完成，最终是通过 Parcel 中的一系列 write 方法来完成的；反序列化功能由 CREATOR 来完成，其内部标明了如何创建序列化对象和数组，并通过 Parcel 的一系列 read 方法来完成反序列化过程；内容描述功能由 describeContents 方法来完成，几乎在所有情况下这个方法都应该返回 0，仅当当前对象中存在文件描述符时，此方法返回 1。需要注意的是，在 User (Parcelin) 方法中，由于 book 是另一个可序列化对象，所以它的反序列化过程需要传递当前线程的上下文类加载器，否则会报无法找到类的错误。
- 详细的方法说明请参看表 2-1。

方法	功能
createFromParcel(Parcel in)	从序列化后的对象中创建原始对象
newArray(int size)	创建指定长度的原始对象数组
User(Parcel in)	从序列化后的对象中创建原始对象
writeToParcel(Parcel out,int flags)	将当前对象写入序列化结构中，其中 flags 标识有两种值：0 或者 1，为 1 时标识当前对象需要作为返回值返回，不能立即释放资源，几乎
describeContents	返回当前对象的内容描述。如果含有文件描述符，返回 1（参见右侧标记位），否则返回 0，几乎所有情况都返回 0

- 系统已经为我们提供了许多实现了 Parcelable 接口的类，它们都是可以直接序列化的，比如 Intent、Bundle、Bitmap 等，同时 List 和 Map 也可以序列化，前提是它们里面的每个元素都是可序列化的。
- 既然 Parcelable 和 Serializable 都能实现序列化并且都可用于 Intent 间的数据传递，那么二者该如何选取呢？Serializable 是 Java 中的序列化接口，其使用起来简单但是开销很大，序列化和反序列化过程需要大量 I/O 操作。而 Parcelable 是 Android 中的序列化方式，因此更适合用在 Android 平台上，它的缺点就是使用起来稍微麻烦点，但是它的效率很高，这是 Android 推荐的序列化方式，因此我们要首选 Parcelable。Parcelable 主要用在内存序列化上，通过 Parcelable 将对象序列化到存储设备中或者将对象序列化后通过网络传输也都是可以的，但是这个过程会稍显复杂，因此在这两种情况下建议大家使用 Serializable。以上就是 Parcelable 和 Serializable 的区别。
- <https://blog.csdn.net/javazejian/article/details/52665164>

### 1.3.3 Binder

- Binder 是一个很深入的话题，笔者也看过一些别人写的 Binder 相关的文章，发现很少有人能把它介绍清楚，不是深入代码细节不能自拔，就是长篇大论不知所云，看完后都是晕晕的感觉。所以，本节笔者不打算深入探讨 Binder 的底层细节，因为 Binder 太复杂了。
- 直观来说，Binder 是 Android 中的一个类，它实现了 IBinder 接口。从 IPC 角度来说，Binder 是 Android 中的一种跨进程通信方式，Binder 还可以理解作为一种虚拟的物理设备，它的设备驱动是/dev/binder，该通信方式在 Linux 中没有；从 Android Framework 角度来说，Binder 是 ServiceManager 连接各种 Manager（ActivityManager、WindowManager，等等）和相



应 `ManagerService` 的桥梁；从 Android 应用层来说，`Binder` 是客户端和服务端进行通信的媒介，当 `bindService` 的时候，服务端会返回一个包含了服务端业务调用的 `Binder` 对象，通过这个 `Binder` 对象，客户端就可以获取服务端提供的服务或者数据，这里的服务包括普通服务和基于 AIDL 的服务。

- Android 开发中，`Binder` 主要用在 `Service` 中，包括 AIDL 和 `Messenger`，其中普通 `Service` 中的 `Binder` 不涉及进程间通信，所以较为简单，无法触及 `Binder` 的核心，而 `Messenger` 的底层其实是 AIDL，所以这里选择用 AIDL 来分析 `Binder` 的工作机制。为了分析 `Binder` 的工作机制，我们需要新建一个 AIDL 示例，SDK 会自动为我们生产 AIDL 所对应的 `Binder` 类，然后我们就可以分析 `Binder` 的工作过程。还是采用本章开始时用的例子，新建 Java 包 `com.ryg.chapter_2.aidl`，然后新建三个文件 `Book.java`、`Book.aidl` 和 `IBookManager.aidl`，代码如下所示。

```
public class Book implements Parcelable {
    public int bookId;
    public String bookName;
    public Book() {}
    public Book(int bookId, String bookName) {
        this.bookId = bookId;
        this.bookName = bookName;
    }
    public int describeContents() {
        return 0;
    }
    public void writeToParcel(Parcel out, int flags) {
        out.writeInt(bookId);
        out.writeString(bookName);
    }
    public static final Parcelable.Creator<Book> CREATOR = new Parcelable.Creator<Book>() {
        public Book createFromParcel(Parcel in) {
            return new Book(in);
        }
        public Book[] newArray(int size) {
            return new Book[size];
        }
    };
    private Book(Parcel in) {
        bookId = in.readInt();
        bookName = in.readString();
    }
    @Override
    public String toString() {
        return String.format("[bookId:%s, bookName:%s]", bookId, bookName);
    }
}
```

## 1. AIDL 文件

```
//Book.aidl
package com.ryg.chapter_2.aidl;
parcelable Book;

//IBookManager.aidl
package com.ryg.chapter_2.aidl;
import com.ryg.chapter_2.aidl.Book;
interface IBookManager {
    List<Book> getBookList();
    void addBook(in Book book);
}
```

- 上面三个文件中，`Book.java` 是一个表示图书信息的类，它实现了 `Parcelable` 接口。`Book.aidl` 是 `Book` 类在 AIDL 中的声明。`IBookManager.aidl` 是我们定义的一个接口，里面有两个方法：`getBookList` 和 `addBook`，其 `getBookList` 用于从远程服务端获取图书列表，而 `addBook` 用于往图书列表中添加一本书，当然这两个方法主要是示例用，不一定要有实际意义。我们可以看到，尽管 `Book` 类已经和 `IBookManager` 位于相同的包中，但是在 `IBookManager` 中仍然要导入 `Book` 类，这就是 AIDL 的特殊之处。下面我们先看一下系统为 `IBookManager.aidl` 生产的 `Binder` 类，在 `gen` 目录下的 `com.ryg.chapter_2.aidl` 包中有一个 `IBookManager.java` 的类，这就是我们要找的类。接下来我们需要根据这个系统生成的 `Binder` 类来分析 `Binder` 的工作原理，代码如下：

- aidl 编译生成的代码：

```
package com.me.ipc;
// Declare any non-default types here with import statements
public interface IGameManager extends android.os.IInterface {

    /** Default implementation for IGameManager. */
    public static class Default implements com.me.ipc.IGameManager {
        // 首先，它声明了两个方法 getBookList 和 addBook，显然这就是我们在 IBookManageraidl 中所声明的方法
        @Override public java.util.List<com.me.ipc.Game> getGameList() throws android.os.RemoteException {
            return null;
        }
        @Override public void addGame(com.me.ipc.Game game) throws android.os.RemoteException {}
        @Override public android.os.IBinder asBinder() {
            return null;
        }
    }

    /** Local-side IPC implementation stub class. */
    // 声明了一个内部类 Stub，这个 Stub 就是一个 Binder 类
    // 当客户端和服务端都位于同一个进程时，方法调用不会走跨进程的 transact 过程
    public static abstract class Stub extends android.os.Binder implements com.me.ipc.IGameManager {

        private static final java.lang.String DESCRIPTOR = "com.me.ipc.IGameManager"; // Binder 的唯一标识，一般用当前

        /** Construct the stub at attach it to the interface. */
        public Stub() {
            this.attachInterface(this, DESCRIPTOR);
        }

        /**
         * Cast an IBinder object into an com.me.ipc.IGameManager interface,
         * generating a proxy if needed.
         */
        // 用于将服务端的 Binder 对象转换成客户端所需的 AIDL 接口类型的对象，这种转换过程是区分进程的：
        // 如果客户端和服务端位于同一进程，那么此方法返回的就是服务端的 Stub 对象本身，否则返回的是系统封装后的 Stub.proxy 对象
        public static com.me.ipc.IGameManager asInterface(android.os.IBinder obj) {
            if ((obj==null)) return null;
            android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
            if (((iin!=null)&&(iin instanceof com.me.ipc.IGameManager)))
                return ((com.me.ipc.IGameManager)iin);
            return new com.me.ipc.IGameManager.Stub.Proxy(obj);
        }

        @Override public android.os.IBinder asBinder() { // 返回当前 Binder 对象
            return this;
        }

        // 而当两者位于不同进程时，方法调用需要走 transact 过程，这个逻辑由 Stub 的内部代理类 Proxy 来完成
        @Override public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags) throws android.os.RemoteException {
            java.lang.String descriptor = DESCRIPTOR;
            switch (code) {
                case INTERFACE_TRANSACTION: {
                    reply.writeString(descriptor);
                    return true;
                }
                case TRANSACTION_getGameList: {
                    data.enforceInterface(descriptor);
                    java.util.List<com.me.ipc.Game> _result = this.getGameList();
                    reply.writeNoException();
                    reply.writeTypedList(_result);
                    return true;
                }
                case TRANSACTION_addGame: {
                    data.enforceInterface(descriptor);
                    com.me.ipc.Game _arg0;
                    if ((0!=data.readInt())) {
                        _arg0 = com.me.ipc.Game.CREATOR.createFromParcel(data);
                    } else {
                        _arg0 = null;
                    }
                    this.addGame(_arg0);
                    reply.writeNoException();
                    return true;
                }
                default: {
                    return super.onTransact(code, data, reply, flags);
                }
            }
        }

        private static class Proxy implements com.me.ipc.IGameManager {
```

```

private android.os.IBinder mRemote;
Proxy(android.os.IBinder remote) {
    mRemote = remote;
}
@Override public android.os.IBinder asBinder() {
    return mRemote;
}
public java.lang.String getInterfaceDescriptor() {
    return DESCRIPTOR;
}

@Override public java.util.List<com.me.ipc.Game> getGameList() throws android.os.RemoteException { // 这个方法运行在
// 首先创建该方法所需要的输入型 Parcel 对象 _data、输出型 Parcel 对象 _reply 和返回值对象 List
android.os.Parcel _data = android.os.Parcel.obtain();
android.os.Parcel _reply = android.os.Parcel.obtain();
java.util.List<com.me.ipc.Game> _result;
try {
    _data.writeInterfaceToken(DESCRIPTOR); // 把该方法的参数信息写入 _data 中 (如果有参数的话)
    // 接着调用 transact 方法来发起 RPC (远程过程调用) 请求, 同时当前线程挂起
    boolean _status = mRemote.transact(Stub.TRANSACTION_getGameList, _data, _reply, 0);
    // 然后服务端的 onTransact 方法会被调用, 直到 RPC 过程返回后, 当前线程继续执行, 并从 reply 中取出 RPC
    if (!_status && getDefaultImpl() != null) {
        return getDefaultImpl().getGameList();
    }
    _reply.readException();
    _result = _reply.createTypedArrayList(com.me.ipc.Game.CREATOR); // 最后返回 _reply 中的数据
}
finally {
    _reply.recycle();
    _data.recycle();
}
return _result; // 最后返回 _reply 中的数据
}

@Override public void addGame(com.me.ipc.Game game) throws android.os.RemoteException { // 这个方法运行在
android.os.Parcel _data = android.os.Parcel.obtain();
android.os.Parcel _reply = android.os.Parcel.obtain();
try {
    _data.writeInterfaceToken(DESCRIPTOR);
    if ((game!=null)) {
        _data.writeInt(1);
        game.writeToParcel(_data, 0);
    } else {
        _data.writeInt(0);
    }
    boolean _status = mRemote.transact(Stub.TRANSACTION_addGame, _data, _reply, 0);
    if (!_status && getDefaultImpl() != null) {
        getDefaultImpl().addGame(game);
        return;
    }
    _reply.readException();
}
finally {
    _reply.recycle();
    _data.recycle();
}
}

public static com.me.ipc.IGameManager sDefaultImpl;
}
// 标识方法的 整型变量
// 声明了两个整型的 id 分别用于标识这两个方法, 这两个 id 用于标识在 transact 过程中客户端所请求的到底是哪个方法
static final int TRANSACTION_getGameList = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
static final int TRANSACTION_addGame = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);

public static boolean setDefaultImpl(com.me.ipc.IGameManager impl) {
    // Only one user of this interface can use this function
    // at a time. This is a heuristic to detect if two different
    // users in the same process use this function.
    if (Stub.Proxy.sDefaultImpl != null) {
        throw new IllegalStateException("setDefaultImpl() called twice");
    }
    if (impl != null) {
        Stub.Proxy.sDefaultImpl = impl;
        return true;
    }
    return false;
}

public static com.me.ipc.IGameManager getDefaultImpl() {
    return Stub.Proxy.sDefaultImpl;
}
}

```

```

    }

    public java.util.List<com.me.ipc.Game> getGameList() throws android.os.RemoteException;
    public void addGame(com.me.ipc.Game game) throws android.os.RemoteException;
}

```

- 上述代码是系统生成的，为了方便查看笔者稍微做了一下格式上的调整。在 `gen` 目录下，可以看到根据 `IBookManager.aidl` 系统为我们生成了 `IBookManager.java` 这个类，它继承了 `IInterface` 这个接口，同时它自己也还是个接口，所有可以在 `Binder` 中传输的接口都需要继承 `Interface` 接口。
- 这个类刚开始看起来逻辑混乱，但是实际上还是很清晰的，通过它我们可以清楚地了解到 `Binder` 的工作机制。这个类的结构其实很简单：
  - 首先，它声明了两个方法 `getBookList` 和 `addBook`，显然这就是我们在 `IBookManager.aidl` 中所声明的方法，
  - 同时它还声明了两个整型的 `id` 分别用于标识这两个方法，这两个 `id` 用于标识在 `transact` 过程中客户端所请求的到底是哪个方法。
  - 接着，它声明了一个内部类 `Stub`，这个 `Stub` 就是一个 `Binder` 类，当客户端和服务端都位于同一个进程时，方法调用不会走进进程的 `transact` 过程，而当两者位于不同进程时，方法调用需要走 `transact` 过程，这个逻辑由 `Stub` 的内部代理类 `Proxy` 来完成。
- 这么来看，`IBookManager` 这个接口的确很简单，但是我们也应该认识到，这个接口的核心实现就是它的内部类 `Stub` 和 `Stub` 的内部代理类 `Proxy`，下面详细介绍针对这两个类的每个方法的含义。
- **DESCRIPTOR**
  - `Binder` 的唯一标识，一般用当前 `Binder` 的类名表示，比如本例中的“`com.ryg.chapter_2.aidl`”。
- `asInterface(android.os.IBinder obj)`
  - 用于将服务端的 `Binder` 对象转换成客户端所需的 `AIDL` 接口类型的对象，这种转换过程是区分进程的，如果客户端和服务端位于同一进程，那么此方法返回的就是服务端的 `Stub` 对象本身，否则返回的是系统封装后的 `Stub.proxy` 对象。
- `asBinder`
  - 此方法用于返回当前 `Binder` 对象。
- `onTransact`
  - 这个方法运行在服务端中的 `Binder` 线程池中，当客户端发起跨进程请求时，远程请求会通过系统底层封装后交由此方法来处理。该方法的原型为

```

public Boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags)

```

- 服务端通过 `code` 可以确定客户端所请求的目标方法是什么，接着从 `data` 中取出目标方法所需的参数（如果目标方法有参数的话），然后执行目标方法。当目标方法执行完毕后，就向 `reply` 中写入返回值（如果目标方法有返回值的话），`onTransact` 方法的执行过程就是这样的。需要注意的是，如果此方法返回 `false`，那么客户端的请求会失败，因此我们可以利用这个特性来做权限验证，毕竟我们也不希望随便一个进程都能远程调用我们的服务。
- `Proxy#getBookList`
  - 这个方法运行在客户端，当客户端远程调用此方法时，它的内部实现是这样的：首先创建该方法所需要的输入型 `Parcel` 对象 `_data`、输出型 `Parcel` 对象 `reply` 和返回值对象 `List`；然后把该方法的参数信息写入 `_data` 中（如果有参数的话）；接着调用 `transact` 方法来发起 `RPC`（远程过程调用）请求，同时当前线程挂起；然后服务端的 `onTransact` 方法会被调用，直到 `RPC` 过程返回后，当前线程继续执行，并从 `reply` 中取出 `RPC` 过程的返回结果；最后返回 `_reply` 中的数据。

- Proxy#addBook
  - 这个方法运行在客户端，它的执行过程和 `getBookList` 是一样的，`addBook` 没有返回值，所以它不需要从 `_reply` 中取出返回值。
- 通过上面的分析，读者应该已经了解了 `Binder` 的工作机制，但是有两点还是需要额外说明一下：
  - 首先，当客户端发起远程请求时，由于当前线程会被挂起直至服务端进程返回数据，所以如果一个远程方法是很耗时的，那么不能在 `UI` 线程中发起此远程请求；
  - 其次，由于服务端的 `Binder` 方法运行在 `Binder` 的线程池中，所以 `Binder` 方法不管是否耗时都应该采用同步的方式去实现，因为它已经运行在一个线程中了。为了更好地说明 `Binder`，下面给出一个 `Binder` 的工作机制。

## 2. 手写 Binder

- 从上述分析过程来看，我们完全可以不提供 `AIDL` 文件即可实现 `Binder`，之所以提供 `AIDL` 文件，是为了方便系统为我们生成代码。系统根据 `AIDL` 文件生成 `Java` 文件的格式是固定的，我们可以抛开 `AIDL` 文件直接写一个 `Binder` 出来，接下来我们就介绍如何手动写一个 `Binder`。还是上面的例子，但是这次我们不提供 `AIDL` 文件。参考上面系统自动生成的 `IBookManager.java` 这个类的代码，可以发现这个类是相当有规律的，根据它的特点，我们完全可以自己写一个和它一模一样的类出来，然后这个不借助 `AIDL` 文件的 `Binder` 就完成了。但是我们发现系统生成的类看起来结构不清晰，我们想试着对它进行结构上的调整，可以发现这个类主要由两部分组成，首先它本身是一个 `Binder` 的接口（继承了 `IInterface`），其次它的内部由个 `Stub` 类，这个类就是个 `Binder`。还记得我们怎么写一个 `Binder` 的服务端吗？代码如下所示。

```
private Binder mBinder = new IBookManager.Stub() {
    @Override
    public List<Book> getBookList() throws RemoteException {
        return mBookList;
    }
    @Override
    public void addBook(Book book) throws RemoteException {
        mBookList.add(book);
    }
};
```

- 首先我们会实现一个创建了一个 `Stub` 对象并在内部实现 `IBookManager` 的接口方法，然后在 `Service` 的 `onBind` 中返回这个 `Stub` 对象。因此，从这一点来看，我们完全可以把 `Stub` 类提取出来直接作为一个独立的 `Binder` 类来实现，这样 `IBookManager` 中就只剩接口本身了，通过这种分离的方式可以让它的结构变得清晰点。
- 根据上面的思想，手动实现一个 `Binder` 可以通过如下步骤来完成：
- (1) 声明一个 `AIDL` 性质的接口，只需要继承 `IInterface` 接口即可，`IInterface` 接口中只有一个 `asBinder` 方法。这个接口的实现如下：

```
public interface IBookManager extends IInterface {
    static final String DESCRIPTOR = "com.ryg.chapter_2.manualbinder.IBookManager";
    static final int TRANSACTION_getBookList = (IBinder.FIRST_CALL_TRANSACTION + 0);
    static final int TRANSACTION_addBook = (IBinder.FIRST_CALL_TRANSACTION + 1);
    public List<Book> getBookList() throws RemoteException;
    public void addBook(Book book) throws RemoteException;
}
```

- 可以看到，在接口中声明了一个 `Binder` 描述符和另外两个 `id`，这两个 `id` 分别表示的是 `getBookList` 和 `addBook` 方法，这段代码原本也是系统生成的，我们仿照系统生成的规则去手动书写这部分代码。如果我们有三个方法，应该怎么做呢？很显然，我们要再声明一个 `id`，然后按照固定模式声明这个新方法即可，这个比较好理解，不再多说。
- (2) 实现 `Stub` 类和 `Stub` 类中的 `Proxy` 代理类，这段代码我们可以自己写，但是写出来后会发现和系统自动生成的代码是一样的，因此这个 `Stub` 类我们只需要参考系统生成的代码即可，只是结构上需要做一下调整，调整后的代码如下所示。



```

public class BookManagerImpl extends Binder implements IBookManager {
    /** Construct the stub at attach it to the interface. */
    public BookManagerImpl() {
        this.attachInterface(this, DESCRIPTOR);
    }
    /**
     * Cast an IBinder object into an IBookManager interface, generating a proxy
     * if needed.
     */
    public static IBookManager asInterface(IBinder obj) {
        if ((obj == null)) {
            return null;
        }
        android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
        if (((iin != null) && (iin instanceof IBookManager))) {
            return ((IBookManager) iin);
        }
        return new BookManagerImpl.Proxy(obj);
    }
    @Override
    public IBinder asBinder() {
        return this;
    }
    @Override
    public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
        throws RemoteException {
        switch (code) {
            case INTERFACE_TRANSACTION: {
                reply.writeString(DESCRIPTOR);
                return true;
            }
            case TRANSACTION_getBookList: {
                data.enforceInterface(DESCRIPTOR);
                List<Book> result = this.getBookList();
                reply.writeNoException();
                reply.writeTypedList(result);
                return true;
            }
            case TRANSACTION_addBook: {
                data.enforceInterface(DESCRIPTOR);
                Book arg0;
                if ((0 != data.readInt())) {
                    arg0 = Book.CREATOR.createFromParcel(data);
                } else {
                    arg0 = null;
                }
                this.addBook(arg0);
                reply.writeNoException();
                return true;
            }
        }
        return super.onTransact(code, data, reply, flags);
    }
    @Override
    public List<Book> getBookList() throws RemoteException {
        // TODO 待实现
        return null;
    }
    @Override
    public void addBook(Book book) throws RemoteException {
        // TODO 待实现
    }
    private static class Proxy implements IBookManager {
        private IBinder mRemote;
        Proxy(IBinder remote) {
            mRemote = remote;
        }
        @Override
        public IBinder asBinder() {
            return mRemote;
        }
        public java.lang.String getInterfaceDescriptor() {
            return DESCRIPTOR;
        }
        @Override
        public List<Book> getBookList() throws RemoteException {
            Parcel data = Parcel.obtain();
            Parcel reply = Parcel.obtain();
            List<Book> result;

```

```

        try {
            data.writeInterfaceToken(DESCRIPTOR);
            mRemote.transact(TRANSACTION_getBookList, data, reply, 0);
            reply.readException();
            result = reply.createTypedArrayList(Book.CREATOR);
        } finally {
            reply.recycle();
            data.recycle();
        }
        return result;
    }
}
@Override
public void addBook(Book book) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    try {
        data.writeInterfaceToken(DESCRIPTOR);
        if ((book != null)) {
            data.writeInt(1);
            book.writeToParcel(data, 0);
        } else {
            data.writeInt(0);
        }
        mRemote.transact(TRANSACTION_addBook, data, reply, 0);
        reply.readException();
    } finally {
        reply.recycle();
        data.recycle();
    }
}
}
}
}

```

- 通过将上述代码和系统生成的代码对比，可以发现简直是一模一样的。也许有人会问：既然和系统生成的一模一样，那我们为什么要手动去写呢？我们在实际开发中完全可以通过 AIDL 文件让系统去自动生成，手动去写的意义在于可以让我们更加理解 Binder 的工作原理，同时也提供了一种不通过 AIDL 文件来实现 Binder 的新方式。也就是说，AIDL 文件并不是实现 Binder 的必需品。如果是我们手写的 Binder，那么在服务端只需要创建一个 BookManagerImpl 的对象并在 Service 的 onBind 方法中返回即可。最后，是否手动实现 Binder 没有本质区别，二者的工作原理完全一样，AIDL 文件的本质是系统为我们提供了一种快速实现 Binder 的工具，仅此而已。

### 3. 设置死亡代理

- 接下来，我们介绍 Binder 的两个很重要的方法 linkToDeath 和 unlinkToDeath。我们知道，Binder 运行在服务端进程，如果服务端进程由于某种原因异常终止，这个时候我们到服务端的 Binder 连接断裂（称之为 Binder 死亡），会导致我们的远程调用失败。更为关键的是，如果我们不知道 Binder 连接已经断裂，那么客户端的功能就会受到影响。为了解决这个问题，Binder 中提供了两个配对的方法 linkToDeath 和 unlinkToDeath，通过 linkToDeath 我们可以给 Binder 设置一个死亡代理，当 Binder 死亡时，我们就会收到通知，这个时候我们就可以重新发起连接请求从而恢复连接。那么到底如何给 Binder 设置死亡代理呢？也很简单。
- 首先，声明一个 DeathRecipient 对象。DeathRecipient 是一个接口，其内部只有一个方法 binderDied，我们需要实现这个方法，当 Binder 死亡的时候，系统就会回调 binderDied 方法，然后我们就可以移出之前绑定的 binder 代理并重新绑定远程服务：

```

private IBinder.DeathRecipient mDeathRecipient = new IBinder.DeathRecipient() {
    @Override
    public void binderDied() {
        if (mBookManager == null)
            return;
        mBookManager.asBinder().unlinkToDeath(mDeathRecipient, 0);
        mBookManager = null;
        // TODO: 这里重新绑定远程 Service
    }
};

```

- 其次，在客户端绑定远程服务成功后，给 binder 设置死亡代理：

```

IBookManager bookManager = IBookManager.Stub.asInterface(service);
//绑定链接断开的对象
bookManager.asBinder().linkToDeath(mDeathRecipient, 0);

```

- 其中 `linkToDeath` 的第二个参数是个标记位，我们直接设为 0 即可。经过上面两个步骤，就给我们的 `Binder` 设置了死亡代理，当 `Binder` 死亡的时候我们就可以收到通知了。另外，通过 `Binder` 的方法 `isBinderAlive` 也可以判断 `Binder` 是否死亡。
- 到这里，IPC 的基础知识就介绍完毕了，下面开始进入正题，直面形形色色的进程间通信方式。

## 1.4 Android 中的 IPC 方式

### 1.4.1 使用 Bundle

- 我们知道，四大组件中的三大组件（`Activity`、`Service`、`Receiver`）都是支持在 `Intent` 中传递 `Bundle` 数据的，由于 `Bundle` 实现了 `Parcelable` 接口，所以它可以方便地在不同的进程间传输。基于这一点，当我们在一个进程中启动了另一个进程的 `Activity`、`Service` 和 `Receiver`，我们就可以在 `Bundle` 中附加我们需要传输给远程进程的信息并通过 `Intent` 发送出去。当然，我们传输的数据必须能够被序列化，比如基本类型、实现了 `Parcelable` 接口的对象、实现了 `Serializable` 接口的对象以及一些 Android 支持的特殊对象，具体内容可以看 `Bundle` 这个类，就可以看到所有它支持的类型。`Bundle` 不支持的类型我们无法通过它在进程间传递数据，这个很简单，就不再详细介绍了。这是一种最简单的进程间通信方式。
- 除了直接传递数据这种典型的使用场景，它还有一种特殊的使用场景。比如 A 进程正在进行一个计算，计算完成后它要启动 B 进程的一个组件并把计算结果传递给 B 进程，可是遗憾的是这个计算结果不支持放入 `Bundle` 中，因此无法通过 `Intent` 来传输，这个时候如果我们用其他 IPC 方式就会略显复杂。可以考虑如下方式：我们通过 `Intent` 启动进程 B 的一个 `Service` 组件（比如 `IntentService`），让 `Service` 在后台进行计算，计算完后再启动 B 进程中真正要启动的目标组件，由于 `Service` 也运行在 B 进程中，所以目标组件就可以直接获取计算结果，这样一来就轻松解决了跨进程的问题。这种方式的核心思想在于将原本需要在 A 进程的计算任务转移到 B 进程的后台 `Service` 中去执行，这样就成功地避免了进程间通信问题，而且只用了很小的代价。

### 1.4.2 使用文件共享

- 共享文件也是一种不错的进程间通信方式，两个进程通过读/写同一个文件来交换数据，比如 A 进程把数据写入文件，B 进程通过读取这个文件来获取数据。我们知道，在 Windows 上，一个文件如果被加了排斥锁将会导致其他线程无法对其进行访问，包括读和写，而由于 Android 系统基于 Linux，使得其并发读/写文件可以没有限制地进行，甚至两个线程同时对同一个文件进行写操作都是允许的，尽管这可能出问题。通过文件交换数据很好使用，除了可以交换一些文本信息外，我们还可以序列化一个对象到文件系统中的同时从另一个进程中恢复这个对象，下面就展示这种使用方法。
- 这次我们在 `MainActivity` 的 `onResume` 中序列化一个 `User` 对象到 sd 卡上的一个文件里，然后在 `SecondActivity` 的 `onResume` 中去反序列化，我们期望在 `SecondActivity` 中能够正确地恢复 `User` 对象的值。关键代码如下：

```

// MainActivity 代码
private void persistToFile() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            User user = new User(1, "hello world", false);
            File dir = new File(MyConstants.CHAPTER_2_PATH);
            if (!dir.exists()) {
                dir.mkdirs();
            }
            File cachedFile = new File(MyConstants.CACHE_FILE_PATH);
            ObjectOutputStream outputStream = null;

```

```

        try {
            objectOutputStream = new ObjectOutputStream(
                new FileOutputStream(cachedFile));
            objectOutputStream.writeObject(user);
            Log.d(TAG, "persist user:" + user);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            MyUtils.close(objectOutputStream);
        }
    }
    }).start();
}
//SecondActivity 代码
private void recoverFromFile() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            User user = null;
            File cachedFile = new File(MyConstants.CACHE_FILE_PATH);
            if (cachedFile.exists()) {
                ObjectInputStream objectInputStream = null;
                try {
                    objectInputStream = new ObjectInputStream(
                        new FileInputStream(cachedFile));
                    user = (User) objectInputStream.readObject();
                    Log.d(TAG, "recover user:" + user);
                } catch (IOException e) {
                    e.printStackTrace();
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                } finally {
                    MyUtils.close(objectInputStream);
                }
            }
        }
    }).start();
}
}

```

- 通过文件共享这种方式来共享数据对文件格式是没有具体要求的，比如可以是文本文件，也可以是 XML 文件，只要读/写双方约定数据格式即可。通过文件共享的方式也是有局限性的，比如并发读/写的问题，像上面的那个例子，如果并发读/写，那么我们读出的内容就有可能不是最新的，如果是并发写的话那就更严重了。因此我们要尽量避免并发写这种情况的发生或者考虑使用线程同步来限制多个线程的写操作。通过上面的分析，我们可以知道，文件共享方式适合在对数据同步要求不高的进程之间进行通信，并且要妥善处理并发读/写的问题。
- 当然，SharedPreferences 是个特例，众所周知，SharedPreferences 是 Android 中提供的轻量级存储方案，它通过键值对的方式来存储数据，在底层实现上它采用 XML 文件来存储键值对，每个应用的 SharedPreferences 文件都可以在当前包所在的 data 目录下查看到。一般来说，它的目录位于/data/data/package name/sharedprefs 目录下，其中 package name 表示的是当前应用的包名。从本质上来说，SharedPreferences 也属于文件的一种，但是由于系统对它的读/写有一定的缓存策略，即在内存中会有一份 SharedPreferences 文件的缓存，因此在多进程模式下，系统对它的读/写就变得不可靠，当面对高并发的读/写访问，SharedPreferences 有很大几率会丢失数据，因此，不建议在进程间通信中使用 SharedPreferences。

### 1.4.3 使用 Messenger

- Messenger 可以翻译为信使，顾名思义，通过它可以在不同进程中传递 Message 对象，在 Message 中放入我们需要传递的数据，就可以轻松地实现数据的进程间传递了。Messenger 是一种轻量级的 IPC 方案，它的底层实现是 AIDL，为什么这么说呢，我们大致看一下 Messenger 这个类的构造方法就明白了。下面是 Messenger 的两个构造方法，从构造方法的实现上我们可以明显看出 AIDL 的痕迹，不管是 IMessenger 还是 Stub.asInterface，这种使用方法都表明它的底层是 AIDL。

```

public Messenger(Handler target) {
    mTarget = target.getIMessenger();
}

```

```
public Messenger(IBinder target) {
    mTarget = IMessenger.Stub.asInterface(target);
}
```

- Messenger 的使用方法很简单，它对 AIDL 做了封装，使得我们可以更简便地进行进程间通信。同时，由于它一次处理一个请求，因此在服务端我们不用考虑线程同步的问题，这是因为服务端中不存在并发执行的情形。实现一个 Messenger 有如下几个步骤，分为服务端和客户端。

## • 1. 服务端进程

- 首先，我们需要在服务端创建一个 Service 来处理客户端的连接请求，同时创建一个 Handler 并通过它来创建一个 Messenger 对象，然后在 Service 的 onBind 中返回这个 Messenger 对象底层的 Binder 即可。

## • 2. 客户端进程

- 客户端进程中，首先要绑定服务端的 Service，绑定成功后用服务端返回的 IBinder 对象创建一个 Messenger，通过这个 Messenger 就可以向服务端发送消息了，发消息类型为 Message 对象。如果需要服务端能够回应客户端，就和服务端一样，我们还需要创建一个 Handler 并创建一个新的 Messenger，并把这个 Messenger 对象通过 Message 的 replyTo 参数传递给服务端，服务端通过这个 replyTo 参数就可以回应客户端。这听起来可能还是有点抽象，不过看了下面的两个例子，读者肯定就都明白了。首先，我们来看一个简单点的例子，在这个例子中服务端无法回应客户端。
- 首先看服务端的代码，这是服务端的典型代码，可以看到 MessengerHandler 用来处理客户端发送的消息，并从消息中取出客户端发来的文本信息。而 mMessenger 是一个 Messenger 对象，它和 MessengerHandler 相关联，并在 onBind 方法中返回它里面的 Binder 对象，可以看出，这里 Messenger 的作用是将客户端发送的消息传递给 MessengerHandler 处理。

```
public class MessengerService extends Service {
    private static final String TAG = "MessengerService";
    private static class MessengerHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MyConstants.MSG_FROM_CLIENT:
                    Log.i(TAG, "receive msg from Client:" + msg.getData().getString("msg"));
                default:
                    super.handleMessage(msg);
            }
        }
    }
    private final Messenger mMessenger = new Messenger(new MessengerHandler());
    @Override
    public IBinder onBind(Intent intent) {
        return mMessenger.getBinder();
    }
}
```

- 然后，注册 service，让其运行在单独的进程中：

```
<service
    android:name=".messenger.MessengerService"
    android:process=":remote">
</service>
```

- 接下来再看看客户端的实现，客户端的实现也比较简单，首先需要绑定远程进程的 MessengerService，绑定成功后，根据服务端返回的 binder 对象创建 Messenger 对象并使用此对象向服务端发送消息。下面的代码在 Bundle 中向服务端发送了一句话，在上面的服务端代码中会打印出这句话。

```
public class MessengerActivity extends Activity {
    private static final String TAG = "MessengerActivity";
    private Messenger mService;
```



```

private ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className, IBinder service) {
        mService = new Messenger(service);
        Log.d(TAG, "bind service");
        Message msg = Message.obtain(null, MyConstants.MSG_FROM_CLIENT);
        Bundle data = new Bundle();
        data.putString("msg", "hello, this is client.");
        msg.setData(data);
        try {
            mService.send(msg);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
    public void onServiceDisconnected(ComponentName className) {
    }
};

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_messenger);
    Intent intent = new Intent("com.ryg.MessengerService.launch");
    bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
}

@Override
protected void onDestroy() {
    unbindService(mConnection);
    super.onDestroy();
}
}

```

- 可以从 log 中可以收到 client 发送的消息

2019-03-24 23:35:52.489 3856-3856/com.ryg.chapter\_2:remote I/MessengerService: receive msg from Client:hello, **this is client**

- 通过上面的例子可以看出，在 Messenger 中进行数据传递必须将数据放入 Message 中，而 Messenger 和 Message 都实现了 Parcelable 接口，因此可以跨进程传输。简单来说，Message 中所支持的数据类型就是 Messenger 所支持的传输类型。实际上，通过 Messenger 来传输 Message，Message 中能使用的载体只有 what、arg1、arg2、Bundle 以及 replyTo。Message 中的另一个字段 object 在同一个进程中是很实用的，但是在进程间通信的时候，在 Android2.2 以前 object 字段不支持跨进程传输，即便是 2.2 以后，也仅仅是系统提供的实现了 Parcelable 接口的对象才能通过它来传输。这就意味着我们自定义的 Parcelable 对象是无法通过 object 字段来传输的，读者可以试一下。非系统的 Parcelable 对象的确无法通过 object 字段来传输，这也导致了 object 字段的实用性大大降低，所幸我们还有 Bundle，Bundle 中可以支持大量的数据类型。
- 上面的例子演示了如何在服务端接收客户端中发送的消息，但是有时候我们还需要能回应客户端，下面就介绍如何实现这种效果。还是采用上面的例子，但是稍微做一下修改，每当客户端发来一条消息，服务端就会自动回复一条“嗯，你的消息我已经收到，稍后会回复你。”，这很类似邮箱的自动回复功能。
- 首先看服务端的修改，服务端只需要修改 MessengerHandler，当收到消息后，会立即回复一条消息给客户端。

```

private static class MessengerHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case MyConstants.MSG_FROM_CLIENT:
                Log.i(TAG, "receive msg from Client:" + msg.getData().getString("msg"));
                Messenger client = msg.replyTo;
                Message replyMessage = Message.obtain(null, MyConstants.MSG_FROM_SERVICE);
                Bundle bundle = new Bundle();
                bundle.putString("reply", " 嗯，你的消息我已经收到，稍后会回复你。");
                replyMessage.setData(bundle);
                try {
                    client.send(replyMessage);
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

```

        break;
    default:
        super.handleMessage(msg);
    }
}
}

```

- 接着再看客户端的修改，为了接收服务端的回复，客户端也需要准备一个接收消息的 Messenger 和 Handler，. 如下所示。

```

private static class MessengerHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case MyConstants.MSG_FROM_SERVICE:
                Log.i(TAG, "receive msg from Service:" + msg.getData().getString("reply"));
                break;
            default:
                super.handleMessage(msg);
        }
    }
}
}

```

- 修改后的程序后可以看到 log 变成了

```

2019-03-24 23:35:52.489 3856-3856/com.ryg.chapter_2:remote I/MessengerService: receive msg from Client:hello, this is client
2019-03-24 23:35:52.618 3824-3824/com.ryg.chapter_2 I/MessengerActivity: receive msg from Service:嗯，你的消息我已经收到，稍后

```

- 到这里，我们已经把采用 Messenger 进行进程间通信的方法都介绍完了，读者可以试着通过 Messenger 来实现更复杂的跨进程通信功能。下面给出一张 Messenger 的工作原理图以方便读者更好地理解 Messenger。

#### 1.4.4 使用 AIDL

- 上面我们介绍了使用 Messenger 来进行进程间通信的方法，可以发现，Messenger 是以串行的方式处理客户端发来的消息，如果大量的消息同时发送到服务端，服务端仍然只能一个个处理，如果有大量的并发请求，那么用 Messenger 就不太合适了。同时，Messenger 的作用主要是为了传递消息，很多时候我们可能需要跨进程调用服务端的方法，这种情形用 Messenger 就无法做到了，但是我们可以使用 AIDL 来实现跨进程的方法调用。AIDL 也是 Messenger 的底层实现，因此 Messenger 本质上也是 AIDL，只不过系统为我们做了封装从而方便上层的调用而已。在上一节中，我们介绍了 Binder 的概念，大家对 Binder 也有了一定的了解，在 Binder 的基础上我们可以更加容易地理解 AIDL。这里先介绍使用 AIDL 来进行进程间通信的流程，分为服务端和客户端两个方面。

##### 1. 1. 服务端

- 服务端首先要创建一个 Service 用来监听客户端的连接请求，然后创建一个 AIDL 文件，将暴露给客户端的接口在这个 AIDL 文件中声明，最后在 Service 中实现这个 AIDL 接口即可。

##### 2. 2. 客户端

- 客户端所要做事情就稍微简单一些，首先需要绑定服务端的 Service，绑定成功后，将服务端返回的 Binder 对象转成 AIDL 接口所属的类型，接着就可以调用 AIDL 中的方法了。
- 上面描述的只是一个感性的过程，AIDL 的实现过程远不止这么简单，接下来会对其中的细节和难点进行详细介绍，并完善我们在 Binder 那一节所提供的的实例。

##### 3. 3.AIDL 接口的创建

- 首先看 AIDL 接口的创建，如下所示，我们创建了一个后缀为 AIDL 的文件，在里面声明了一个接口和两个接口方法。

```
interface IBookManager {
    public List<Book> getBookList() throws RemoteException;
    public void addBook(Book book) throws RemoteException;
}
```

- 在 AIDL 文件中，并不是所有的数据类型都是可以使用的，那么到底 AIDL 文件支持哪些数据类型呢？如下所示。
  - 基本数据类型（int、long、char、boolean、double 等）；
  - String 和 CharSequence；
  - List：只支持 ArrayList，里面每个元素都必须能够被 AIDL 支持；
  - Map：只支持 HashMap，里面的每个元素都必须被 AIDL 支持，包括 key 和 value；
  - Parcelable；所有实现了 Parcelable 接口的对象；
  - AIDL：所有的 AIDL 接口本身也可以用在 AIDL 文件中使用。
- 以上 6 种数据类型就是 AIDL 所支持的所有类型，其中自定义的 Parcelable 对象和 AIDL 对象必须要显式 import 进来，不管它们是否和当前的 AIDL 文件位于同一个包内。比如 IBookManager.aidl 这个文件，里面用到了 Book 这个类，这个类实现了 Parcelable 接口并且和 IBookManager.aidl 位于同一个包中，但是遵守 AIDL 的规范，我们仍然需要显式地 import 进来：import com.ryg.chapter 2.aidl.Book。AIDL 中会大量使用到 Parcelable，至于如何使用 Parcelable 接口来序列化对象，在本章的前面已经介绍过，这里就不再赘述。
- 另外一个需要注意的地方是，如果 AIDL 文件中用到了自定义的 Parcelable 对象，那么必须新建一个和它同名的 AIDL 文件，并在其中声明它为 Parcelable 类型。在上面的 IBookManager.aidl 中，我们用到了 Book 这个类，所以，我们必须创建 Book.aidl，然后在里面添加如下内容：

```
parcelable Book;
```

- 我们需要注意，AIDL 中每个实现了 Parcelable 接口的类都需要按照上面那种方式去创建相应的 AIDL 文件并声明那个类为 parcelable。除此之外，AIDL 中除了基本数据类型，其他类型的参数必须标上方向：in、out 或者 inout，in 表示输入型参数，out 表示输出型参数，inout 表示输入输出型参数，至于它们具体的区别，这个就不说了。我们要根据实际需要去指定参数类型，不能一概使用 out 或者 inout，因为这在底层实现是有开销的。最后，AIDL 接口中只支持方法，不支持声明静态常量，这一点区别于传统的接口。
- 为了方便 AIDL 的开发，建议把所有和 AIDL 相关的类和文件全部放入同一个包中，这样做的好处是，当客户端是另外一个应用时，我们可以直接把整个包复制到客户端工程中，对于本例来说，就是要把 com.ryg.chapter 2.aidl 这个包和包中的文件原封不动地复制到客户端中。如果 AIDL 相关的文件位于不同的包中时，那么就需要把这些包一一复制到客户端工程中，这样操作起来比较麻烦而且也容易出错。需要注意的是，AIDL 的包结构在服务端和客户端要保持一致，否则运行会出错，这是因为客户端需要反序列化服务端中和 AIDL 接口相关的所有类，如果类的完整路径不一样的话，就无法成功反序列化，程序也就无法正常运行。为了方便演示，本章的所有示例都是在同一个工程中进行的，但是读者要理解，一个工程和两个工程的多进程本质是一样的，两个工程的情况，除了需要复制 AIDL 接口所相关的包到客户端，其他完全一样，读者可以自行试验。

#### 4. 服务端的实现

```
public class BookManagerService extends Service {
    private static final String TAG = "BMS";
    private AtomicBoolean mIsServiceDestroyed = new AtomicBoolean(false);
    private CopyOnWriteArrayList<Book> mBookList = new CopyOnWriteArrayList<Book>();
    private Binder mBinder = new IBookManager.Stub() {
        @Override
        public List<Book> getBookList() throws RemoteException {
            return mBookList;
        }
        @Override
        public void addBook(Book book) throws RemoteException {
```

```

        mBookList.add(book);
    }
};
@Override
public void onCreate() {
    super.onCreate();
    mBookList.add(new Book(1, "Android"));
    mBookList.add(new Book(2, "Ios"));
}
@Override
public IBinder onBind(Intent intent) {
    return mBinder;
}
}
}

```

- 上面是一个服务端 Service 的典型实现，首先在 onCreate 中初始化添加了两本图书的信息，然后创建了一个 Binder 对象并在 onBind 中返回它，这个对象继承自 IBookManager.Stub 并实现了它内部的 AIDL 方法，这个过程在 Binder 那一节已经介绍过了，这里就不多说了。
- 这里主要看 getBookList 和 addBook 这两个 AIDL 方法的实现，实现过程也比较简单，注意这里采用了 CopyOnWriteArrayList，这个 CopyOnWriteArrayList 支持并发读/写。在前面我们提到，AIDL 方法是在服务端的 Binder 线程池中执行的，因此当多个客户端同时连接的时候，会存在多个线程同时访问的情形，所以我们要在 AIDL 方法中处理线程同步，而我们这里直接使用 CopyOnWriteArrayList 来进行自动的线程同步。

前面我们提到，AIDL 中能够使用的 List 只有 ArrayList，但是我们这里却使用了 CopyOnWriteArrayList（注意它不是继承自 ArrayList），为什么能够正常工作呢？这是因为 AIDL 中所支持的是抽象的 List，而 List 只是一个接口，因此虽然服务端返回的是 CopyOnWriteArrayList，但是在 Binder 中会按照 List 的规范去访问数据并最终形成一个新的 ArrayList 传递给客户端。所以，我们在服务端采用 CopyOnWriteArrayList 是完全可以的。和此类似的还有 ConcurrentHashMap，读者可以体会一下这种转换情形。然后我们需要在 XML 中注册这个 Service，如下所示。注意 BookManagerService 是运行在独立进程中的，它和客户端的 Activity 不在同一个进程中，这样就构成了进程间通信的场景。

```

<service
    android:name=".aidl.BookManagerService"
    android:process=":remote"></service>

```

## 5. 5. 客户端的实现

- 查询书籍
- 客户端的实现就比较简单了，首先要绑定远程服务，绑定成功后将服务端返回的 Binder 对象转换成 AIDL 接口，然后就可以通过这个接口去调用服务端的远程方法了，代码如下所示。

```

private ServiceConnection mConnection = new ServiceConnection() {
    //bind Service 后获取到对象
    public void onServiceConnected(ComponentName className, IBinder service) {
        //转化对象
        IBookManager bookManager = IBookManager.Stub.asInterface(service);
        mRemoteBookManager = bookManager;
        try {
            mRemoteBookManager.asBinder().linkToDeath(mDeathRecipient, 0);
            List<Book> list = bookManager.getBookList();
            Log.i(TAG, "query book list, list type:"
                + list.getClass().getCanonicalName());
            Log.i(TAG, "query book list:" + list.toString());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}

public void onServiceDisconnected(ComponentName className) {
    mRemoteBookManager = null;
    Log.d(TAG, "onServiceDisconnected. tname:" + Thread.currentThread().getName());
}
};

```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_book_manager);
    Intent intent = new Intent(this, BookManagerService.class);
    bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
}

```

- 绑定成功以后，会通过 `bookManager` 去调用 `getBookList` 方法，然后打印出所获取的图书信息。需要注意的是，服务端的方法有可能需要很久才能执行完毕，这个时候下面的代码就会导致 ANR，这一点是需要注意的，后面会再介绍这种情况，之所以先这么写是为了让读者更好地了解 AIDL 的实现步骤。
- 接着在 XML 中注册此 Activity，运行程序，log 如下所示。

```

03-25 16:44:35.622 15908-15908/com.ryg.chapter_2 I/BookManagerActivity: query book list, list type:java.util.ArrayLi
03-25 16:44:35.622 15908-15908/com.ryg.chapter_2 I/BookManagerActivity: query book list:[[bookId:1, bookName:Android

```

- 可以发现，虽然我们在服务端返回的是 `CopyOnWriteArrayList` 类型，但是客户端收到的仍然是 `ArrayList` 类型，这也证实了我们在前面所做的分析。第二行 log 表明客户端成功地得到了服务端的图书列表信息。
- 这就是一次完完整整的使用 AIDL 进行 IPC 的过程，到这里相信读者对 AIDL 应该有了一个整体的认识了，但是还没完，AIDL 的复杂性远不止这些，下面继续介绍 AIDL 中常见的一些难点。
- 添加书籍
- 我们接着再调用一下另外一个接口 `addBook`，我们在客户端给服务端添加一本书，然后再获取一次，看程序是否能够正常工作。还是上面的代码，客户端在服务连接后，在 `onServiceConnected` 中做如下改动：

```

public void onServiceConnected(ComponentName className, IBinder service) {
    IBookManager bookManager = IBookManager.Stub.asInterface(service);
    mRemoteBookManager = bookManager;
    try {
        mRemoteBookManager.asBinder().linkToDeath(mDeathRecipient, 0);
        List<Book> list = bookManager.getBookList();
        Log.i(TAG, "query book list, list type:"
            + list.getClass().getCanonicalName());
        Log.i(TAG, "query book list:" + list.toString());
        Book newBook = new Book(3, "Android 进阶");
        bookManager.addBook(newBook);
        Log.i(TAG, "add book:" + newBook);
        List<Book> newList = bookManager.getBookList();
        Log.i(TAG, "query book list:" + newList.toString());
        bookManager.registerListener(mOnNewBookArrivedListener);
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}

```

- 可以看到 log 日志

```

03-25 16:48:08.522 15908-15908/com.ryg.chapter_2 I/BookManagerActivity: query book list, list type:java.util.ArrayLi
03-25 16:48:08.522 15908-15908/com.ryg.chapter_2 I/BookManagerActivity: query book list:[[bookId:1, bookName:Android
03-25 16:48:08.522 15908-15908/com.ryg.chapter_2 I/BookManagerActivity: add book:[bookId:3, bookName:Android进阶]
03-25 16:48:13.532 15908-15908/com.ryg.chapter_2 I/BookManagerActivity: query book list:[[bookId:1, bookName:Android

```

- 动态监听添加的书籍
  - 现在我们考虑一种情况，假设有一种需求：用户不想时不时地去查询图书列表了，太累了，于是，他去问图书馆，“当有新书时能不能把书的信息告诉我呢？”。大家应该明白了，这就是一种典型的观察者模式，每个感兴趣的用户都观察新书，当新书到的时候，图书馆就通知每一个对这本书感兴趣的用户，这种模式在实际开发中用得很多，下面我们就来模拟这种情形。首先，我们需要提供一个 AIDL 接口，每个用户都需要实现这个接口并且向图书馆申请新书的提醒功能，当然用户也可以随时取消这种提醒。之所以选择 AIDL 接口而不是普通接口，是因为 AIDL 中无法使用普通接



口。这里我们创建一个 `IOOnNewBookArrivedListener.aidl` 文件，我们所期望的情况是：当服务端有新书到来时，就会通知每一个已经申请提醒功能的用户。从程序上来说就是调用所有 `IOOnNewBookArrivedListener` 对象中的 `onNewBookArrived` 方法，并把新书的对象通过参数传递给客户端，内容如下所示。

```
interface IOOnNewBookArrivedListener {
    void onNewBookArrived(in Book newBook);
}
```

- 除了要新加一个 AIDL 接口，还需要在原有的接口中添加两个新方法，代码如下所示。

```
interface IBookManager {
    List<Book> getBookList();
    void addBook(in Book book);
    void registerListener(IOOnNewBookArrivedListener listener);
    void unregisterListener(IOOnNewBookArrivedListener listener);
}
```

- 接着，服务端中 `Service` 的实现也要稍微修改一下，主要是 `Service` 中 `IBookManager.Stub` 的实现，因为我们在 `IBookManager` 新加了两个方法，所以在 `IBookManager.Stub` 中也要实现这两个方法。同时，在 `BookManagerService` 中还开启了一个线程，每隔 5s 就向书库中增加一本新书并通知所有感兴趣的用户，整个代码如下所示。

```
public class BookManagerService extends Service {
    private static final String TAG = "BMS";
    private AtomicBoolean mIsServiceDestroyed = new AtomicBoolean(false);
    private CopyOnWriteArrayList<Book> mBookList = new CopyOnWriteArrayList<Book>();
    private CopyOnWriteArrayList<IOOnNewBookArrivedListener> mListenerList = new CopyOnWriteArrayList<IOOnNewBookArrivedListener>();
    private Binder mBinder = new IBookManager.Stub() {
        @Override
        public List<Book> getBookList() throws RemoteException {
            SystemClock.sleep(5000);
            return mBookList;
        }
        @Override
        public void addBook(Book book) throws RemoteException {
            mBookList.add(book);
        }
        @Override
        public void registerListener(IOOnNewBookArrivedListener listener)
            throws RemoteException {
            if (!mListenerList.contains(listener)) {
                mListenerList.add(listener);
            } else {
                Log.d(TAG, "already exists");
            }
            Log.d(TAG, "registerListener, size:" + mListenerList.size());
        }
        @Override
        public void unregisterListener(IOOnNewBookArrivedListener listener)
            throws RemoteException {
            if (mListenerList.contains(listener)) {
                mListenerList.remove(listener);
                Log.d(TAG, "unregister listener succeed");
            } else {
                Log.d(TAG, "not found,can not unregister");
            }
            Log.d(TAG, "unregisterListener,current size:" + mListenerList.size());
        }
    };
    @Override
    public void onCreate() {
        super.onCreate();
        mBookList.add(new Book(1, "Android"));
        mBookList.add(new Book(2, "Ios"));
        new Thread(new ServiceWorker()).start();
    }
    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }
    @Override
    public void onDestroy() {
        mIsServiceDestroyed.set(true);
    }
}
```

```

        super.onDestroy();
    }
    private void onNewBookArrived(Book book) throws RemoteException {
        mBookList.add(book);
        Log.d(TAG, "onNewBookArrived,notify listeners:" + mListenerList.size());
        //遍历集合通知书籍到达
        for (int i = 0; i < mListenerList.size(); i++) {
            IOnNewBookArrivedListener listener = mListenerList.get(i);
            Log.d(TAG, "onNewBookArrived,notify listener:" + listener);
            listener.onNewBookArrived(book);
        }
    }
    private class ServiceWorker implements Runnable {
        @Override
        public void run() {
            // do background processing here.....
            while (!mIsServiceDestroyed.get()) {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                int bookId = mBookList.size() + 1;
                Book newBook = new Book(bookId, "new book#" + bookId);
                try {
                    onNewBookArrived(newBook);
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

- 最后，我们还需要修改一下客户端的代码，主要有两方面：首先客户端要注册 IOnNewBookArrivedListener 到远程服务端，这样当有新书时服务端才能通知当前客户端，同时我们要在 Activity 退出时解除这个注册；另一方面，当有新书时，服务端会回调客户端的 IOnNewBookArrivedListener 对象中的 onNewBookArrived 方法，但是这个方法是在客户端的 Binder 线程池中执行的，因此，为了便于进行 UI 操作，我们需要有一个 Handler 可以将其切换到客户端的主线程中去执行，这个原理在 Binder 中已经做了分析，这里就不多说了。客户端的代码修改如下：

```

public class BookManagerActivity extends Activity {
    private static final String TAG = "BookManagerActivity";
    private static final int MESSAGE_NEW_BOOK_ARRIVED = 1;
    private IBookManager mRemoteBookManager;
    @SuppressWarnings("HandlerLeak")
    private Handler mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MESSAGE_NEW_BOOK_ARRIVED:
                    Log.d(TAG, "receive new book :" + msg.obj);
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    };
    private ServiceConnection mConnection = new ServiceConnection() {
        //bind Service 后获取到对象
        public void onServiceConnected(ComponentName className, IBinder service) {
            IBookManager bookManager = IBookManager.Stub.asInterface(service);
            mRemoteBookManager = bookManager;
            try {
                mRemoteBookManager.asBinder().linkToDeath(mDeathRecipient, 0);
                List<Book> list = bookManager.getBookList();
                Log.i(TAG, "query book list, list type:"
                    + list.getClass().getCanonicalName());
                Log.i(TAG, "query book list:" + list.toString());
                Book newBook = new Book(3, "Android 进阶");
                bookManager.addBook(newBook);
                Log.i(TAG, "add book:" + newBook);
                List<Book> newList = bookManager.getBookList();
                Log.i(TAG, "query book list:" + newList.toString());
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
    };
}

```

```

        bookManager.registerListener(mOnNewBookArrivedListener);
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}

public void onServiceDisconnected(ComponentName className) {
    mRemoteBookManager = null;
    Log.d(TAG, "onServiceDisconnected. tname:" + Thread.currentThread().getName());
}

};
private IOnNewBookArrivedListener mOnNewBookArrivedListener = new IOnNewBookArrivedListener.Stub() {
    @Override
    public void onNewBookArrived(Book newBook) throws RemoteException {
        mHandler.obtainMessage(MESSAGE_NEW_BOOK_ARRIVED, newBook)
            .sendToTarget();
    }
};
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_book_manager);
    Intent intent = new Intent(this, BookManagerService.class);
    bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
}
@Override
protected void onDestroy() {
    if (mRemoteBookManager != null
        && mRemoteBookManager.asBinder().isBinderAlive()) {
        try {
            Log.i(TAG, "unregister listener:" + mOnNewBookArrivedListener);
            mRemoteBookManager
                .unregisterListener(mOnNewBookArrivedListener);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
    unbindService(mConnection);
    super.onDestroy();
}
}
}

```

## • 打印的 log 日志

```

2019-03-25 22:44:58.927 5594-5815/com.ryg.chapter_2:remote D/BMS: onNewBookArrived,notify listeners:1
2019-03-25 22:44:58.927 5594-5815/com.ryg.chapter_2:remote D/BMS: onNewBookArrived,notify listener:com.ryg.chapter_2
2019-03-25 22:44:58.928 5560-5560/com.ryg.chapter_2 D/BookManagerActivity: receive new book :[bookId:290, bookName:n
2019-03-25 22:45:00.020 1425-1445/? D/hwcomposer: hw_composer sent 66 syncs in 60s

```

- 回调事件的销毁错误的方式
- 如果你以为到这里 AIDL 的介绍就结束了，那你就错了，之前就说过，AIDL 远不止这么简单，目前还有一些难点是我们还没有涉及的，接下来将继续为读者介绍。
- 从上面的代码可以看出，当 BookManagerActivity 关闭时，我们会在 onDestroy 中去解除已经注册到服务端的 listener，这就相当于我们不想再接收图书馆的新书提醒了，所以我们可以随时取消这个提醒服务。按 back 键退出 BookManagerActivity，下面是打印出的 log。

```

2019-03-25 22:52:43.777 5594-5834/com.ryg.chapter_2:remote D/BMS: onTransact: com.ryg.chapter_2
2019-03-25 22:52:43.777 5594-5834/com.ryg.chapter_2:remote D/BMS: not found,can not unregister
2019-03-25 22:52:43.778 5594-5834/com.ryg.chapter_2:remote D/BMS: unregisterListener,current size:1

```

- 从上面的 log 可以看出，程序没有像我们所预期的那样执行。在解注册的过程中，服务端竟然无法找到我们之前注册的那个 listener，在客户端我们注册和解注册时明明传递的是同一个 listener 啊！最终，服务端由于无法找到要解除的 listener 而宣告解注册失败！这当然不是我们想要的结果，但是仔细想想，好像这种方式的确无法完成解注册。其实，这是必然的，这种解注册的处理方式在日常开发过程中时常使用到，但是放到多进程中却无法奏效，因为 Binder 会把客户端传递过来的对象重新转化并生成一个新的对象。虽然我们在注册和解注册过程中使用的是同一个客户端对象，但是通过 Binder 传递到服务端后，却会产生两个全新的对象。别忘了对象是不能跨进程直接传输的，对象的

跨进程传输本质上都是反序列化的过程，这就是为什么 AIDL 中的自定义对象都必须要实现 Parcelable 接口的原因。那么到底我们该怎么做才能实现解注册功能呢？答案是使用 RemoteCallbackList，这看起来很抽象，不过没关系，请看接下来的详细分析。

- 回调事件销毁正确的方式
- RemoteCallbackList 是系统专门提供的用于删除跨进程 listener 的接口。RemoteCallbackList 是一个泛型，支持管理任意的 AIDL 接口，这点从它的声明就可以看出，因为所有的 AIDL 接口都继承自 IInterface 接口，读者还有印象吗？
- public class RemoteCallbackList<E extends IInterface>
- 它的工作原理很简单，在它的内部有一个 Map 结构专门用来保存所有的 AIDL 回调，这个 Map 的 key 是 IBinder 类型，value 是 Callback 类型，如下所示。
- ArrayMap<IBinder, Callback> mCallbacks= new ArrayMap<IBinder, Callback>();
- 其中 Callback 中封装了真正的远程 listener。当客户端注册 listener 的时候，它会把这个 listener 的信息存入 mCallbacks 中，其中 key 和 value 分别通过下面的方式获得：

```
IBinder key=listener.asBinder()
Callback value=new Callback(listener, cookie)
```

- 到这里，读者应该都明白了，虽然说多次跨进程传输客户端的同一个对象会在服务端生成不同的对象，但是这些新生成的对象有一个共同点，那就是它们底层的 Binder 对象是同一个，利用这个特性，就可以实现上面我们无法实现的功能。当客户端解注册的时候，我们只要遍历服务端所有的 listener，找出那个和解注册 listener 具有相同 Binder 对象的服务端 listener 并把它删掉即可，这就是 RemoteCallbackList 为我们做的事情。同时 RemoteCallbackList 还有一个很有用的功能，那就是当客户端进程终止后，它能够自动移除客户端所注册的 listener。另外，RemoteCallbackList 内部自动实现了线程同步的功能，所以我们使用它来注册和解注册时，不需要做额外的线程同步工作。由此可见，RemoteCallbackList 的确是个很有价值的类，下面就演示如何使用它来完成解注册。
- RemoteCallbackList 使用起来很简单，我们要对 BookManagerService 做一些修改，首先要创建一个 RemoteCallbackList 对象来替代之前的 CopyOnWriteArrayList，如下所示。

```
private RemoteCallbackList<IOnNewBookArrivedListener> mListenerList = new RemoteCallbackList<IOnNewBookArrivedListener>
```

- 然后修改 registerListener 和 unregisterListener 这两个接口的实现，如下所示。

```
@Override
public void registerListener(IOnNewBookArrivedListener listener)
    throws RemoteException {
    mListenerList.register(listener);
}
@Override
public void unregisterListener(IOnNewBookArrivedListener listener)
    throws RemoteException {
    boolean success = mListenerList.unregister(listener);
}
```

- 怎么样？是不是用起来很简单，接着要修改 onNewBookArrived 方法，当有新书时，我们就要通知所有已注册的 listener，如下所示。

```
private void onNewBookArrived(Book book) throws RemoteException {
    mBookList.add(book);
    final int N = mListenerList.beginBroadcast();
    for (int i = 0; i < N; i++) {
        IOnNewBookArrivedListener l = mListenerList.getBroadcastItem(i);
        if (l != null) {
            try {
                l.onNewBookArrived(book);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
    }
    mListenerList.finishBroadcast();
}
```

- `BookManagerService` 的修改已经完毕了，为了方便我们验证程序的功能，我们还需要添加一些 `log`，在注册和解注册后我们分别打印出所有 `listener` 的数量。如果程序正常工作的话，那么注册之后 `listener` 总数量是 1 解注册之后总数量应该是 0，我们再次运行一下程序，看是否如此。从下面的 `log` 来看，很显然，使用 `RemoteCallbackList` 的确可以完成跨进程的解注册功能。

```
2019-03-26 00:04:26.056 11421-11616/com.ryg.chapter_2:remote D/BMS: unregister success.
2019-03-26 00:04:26.056 11421-11616/com.ryg.chapter_2:remote D/BMS: unregisterListener, current size:0
```

- 使用 `RemoteCallbackList`，有一点需要注意，我们无法像操作 `List` 一样去操作它，尽管它的名字中也带个 `List`，但是它并不是一个 `List`。遍历 `RemoteCallbackList`，必须要按照下面的方式进行，其中 `beginBroadcast` 和 `finishBroadcast` 必须要配对使用，哪怕我们仅仅是想要获取 `RemoteCallbackList` 中的元素个数，这是必须要注意的地方。

```
final int N = mListenerList.beginBroadcast();
for (int i = 0; i < N; i++) {
    IOnNewBookArrivedListener l = mListenerList.getBroadcastItem(i);
    if (l != null) {
        //todo
    }
}
mListenerList.finishBroadcast();
```

- 到这里，`AIDL` 的基本使用方法已经介绍完了，但是有几点还需要再次说明一下。我们知道，客户端调用远程服务的方法，被调用的方法运行在服务端的 `Binder` 线程池中，同时客户端线程会被挂起，这个时候如果服务端方法执行比较耗时，就会导致客户端线程长时间地阻塞在这里，而如果这个客户端线程是 `UI` 线程的话，就会导致客户端 `ANR`，这当然不是我们想要看到的。因此，如果我们明确知道某个远程方法是耗时的，那么就要避免在客户端的 `UI` 线程中去访问远程方法。由于客户端的 `onServiceConnected` 和 `onServiceDisconnected` 方法都运行在 `UI` 线程中，所以也不可以在它们里面直接调用服务端的耗时方法，这点要尤其注意。另外，由于服务端的方法本身就运行在服务端的 `Binder` 线程池中，所以服务端方法本身就可以执行大量耗时操作，这个时候切记不要在服务端方法中开线程去进行异步任务，除非你明确知道自己在干什么，否则不建议这么做。下面我们稍微改造一下服务端的 `getBookList` 方法，我们假定这个方法是耗时的，那么服务端可以这么实现：

```
@Override
public List<Book> getBookList() throws RemoteException {
    SystemClock.sleep(5000);
    return mBookList;
}
```

- 然后在客户端中放一个按钮，单击它的时候就会调用服务端的 `getBookList` 方法，可以预知，连续单击几次，客户端就 `ANR` 了，如下图，感兴趣读者可以自行试一下。
- 避免出现上述这种 `ANR` 其实很简单，我们只需要把调用放在非 `UI` 线程即可，如下所示。

```
public void onButton1Click(View view) {
    Toast.makeText(this, "click button1", Toast.LENGTH_SHORT).show();
    new Thread(new Runnable() {
        @Override
        public void run() {
            if (mRemoteBookManager != null) {
                try {
                    List<Book> newList = mRemoteBookManager.getBookList();
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
            }
        }
    }).start();
}
```

- 同理，当远程服务端需要调用客户端的 `listener` 中的方法时，被调用的方法也运行在 `Binder` 线程池中，只不过是客户端的线程池。所以，我们同样不可以在服务端中调用客



户端的耗时方法。比如针对 BookManagerService 的 onNewBookArrived 方法，如下所示。在它内部调用了客户端的 IOnNewBookArrivedListener 中的 onNewBookArrived 方法，如果客户端的这个 onNewBookArrived 方法比较耗时的话，那么请确保 BookManagerService 中的 onNewBookArrived 运行在非 UI 线程中，否则将导致服务端无法响应。

```
private void onNewBookArrived(Book book) throws RemoteException {
    mBookList.add(book);
    Log.d(TAG, "onNewBookArrived,notify listeners:" + mListenerList.size());
    //遍历集合通知书籍到达
    for (int i = 0; i < mListenerList.size(); i++) {
        IOnNewBookArrivedListener listener = mListenerList.get(i);
        Log.d(TAG, "onNewBookArrived,notify listener:" + listener);
        listener.onNewBookArrived(book);
    }
}
```

- 另外，由于客户端的 IOnNewBookArrivedListener 中的 onNewBookArrived 方法运行在客户端的 Binder 线程池中，所以不能在它里面去访问 UI 相关的内容，如果要访问 UI，请使用 Handler 切换到 UI 线程，这一点在前面的代码实例中已经有所体现，这里就不再详细描述了。
- 为了程序的健壮性，我们还需要做一件事。Binder 是可能意外死亡的，这往往是由于服务端进程意外停止了，这时我们需要重新连接服务。有两种方法，第一种方法是给 Binder 设置 DeathRecipient 监听，当 Binder 死亡时，我们会收到 binderDied 方法的回调，在 binderDied 方法中我们可以重连远程服务，具体方法在 Binder 那一节已经介绍过了，这里就不再详细描述了。另一种方法是在 onServiceDisconnected 中重连远程服务。这两种方法我们可以随便选择一种来使用，它们的区别在于：onServiceDisconnected 在客户端的 UI 线程中被回调，而 binderDied 在客户端的 Binder 线程池中被回调。也就是说，在 binderDied 方法中我们不能访问 UI，这就是它们的区别。

## 6. 6.AIDL 权限验证

- 到此为止，我们已经对 AIDL 有了一个系统性的认识，但是还差最后一步：如何在 AIDL 中使用权限验证功能。默认情况下，我们的远程服务任何人都可以连接，但这应该不是我们愿意看到的，所以我们必须给服务加入权限验证功能，权限验证失败则无法调用服务中的方法。在 AIDL 中进行权限验证，这里介绍两种常用的方法。
- 第一种方法
  - 我们可以在 onBind 中进行验证，验证不通过就直接返回 null，这样验证失败的客户端直接无法绑定服务，至于验证方式可以有多种，比如使用 permission 验证。
  - 使用这种验证方式，我们要先在 AndroidManifest 中声明所需的权限，比如：

```
<permission
    android:name="com.ryg.chapter_2.permission.ACCESS_BOOK_SERVICE"
    android:protectionLevel="normal" />
```

- 关于 permission 的定义方式请读者查看相关资料，这里就不详细展开了，毕竟本节的主要内容是介绍 AIDL。定义了权限以后，就可以在 BookManagerService 的 onBind 方法中做权限验证了，如下所示。

```
@Override
public IBinder onBind(Intent intent) {
    int check = checkCallingOrSelfPermission("com.ryg.chapter_2.permission.ACCESS_BOOK_SERVICE");
    Log.d(TAG, "onbind check=" + check);
    if (check == PackageManager.PERMISSION_DENIED) {
        return null;
    }
    return mBinder;
}
```

- 一个应用来绑定我们的服务时，会验证这个应用的权限，如果它没有使用这个权限，onBind 方法就会直接返回 null，最终结果是这个应用无法绑定到我们的服务，这样就达到了权限验证的效果，这种方法同样适用于 Messenger 中，读者可以自行扩展。

如果我们自己内部的应用想绑定到我们的服务中，只需要在它的 `AndroidManifest` 文件中采用如下方式使用 `permission` 即可。

```
<uses-permission android:name="com.ryg.chapter_2.permission.ACCESS_BOOK_SERVICE" />
```

- 第二种方法
- 我们可以在服务端的 `onTransact` 方法中进行权限验证，如果验证失败就直接返回 `false`，这样服务端就不会终止执行 AIDL 中的方法从而达到保护服务端的效果。
- 至于具体的验证方式有很多，可以采用 `permission` 验证，具体实现方式和第一种方法一样。还可以采用 `Uid` 和 `Pid` 来做验证，通过 `getCallingUid` 和 `getCallingPid` 可以拿到客户端所属应用的 `Uid` 和 `Pid`，通过这两个参数我们可以做一些验证工作，比如验证包名。在下面的代码中，既验证了 `permission`，又验证了包名。一个应用如果想远程调用服务中的方法，首先要使用我们的自定义权限“`com.ryg.chapter_2.permission.ACCESS_BOOK_SERVICE`”，其次包名必须以“`com.ryg`”开始，否则调用服务端的方法会失败。

```
public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
    throws RemoteException {
    int check = checkCallingOrSelfPermission("com.ryg.chapter_2.permission.ACCESS_BOOK_SERVICE");
    Log.d(TAG, "check=" + check);
    if (check == PackageManager.PERMISSION_DENIED) {
        return false;
    }
    String packageName = null;
    String[] packages = getPackageManager().getPackagesForUid(
        getCallingUid());
    if (packages != null && packages.length > 0) {
        packageName = packages[0];
    }
    Log.d(TAG, "onTransact: " + packageName);
    if (!packageName.startsWith("com.ryg")) {
        return false;
    }
    return super.onTransact(code, data, reply, flags);
}
```

- 上面介绍了两种 AIDL 中常用的权限验证方法，但是肯定还有其他方法可以做权限验证，比如为 `Service` 指定 `android:permission` 属性等，这里就不一一进行介绍了。

## 1.5 使用 ContentProvider

- `ContentProvider` 是 Android 中提供的专门用于不同应用间进行数据共享的方式，从这一点来看，它天生就适合进程间通信。和 `Messenger` 一样，`ContentProvider` 的底层实现同样也是 `Binder`，由此可见，`Binder` 在 Android 系统中是何等的重要。虽然 `ContentProvider` 的底层实现是 `Binder`，但是它的使用过程要比 AIDL 简单许多，这是因为系统已经为我们做了封装，使得我们无须关心底层细节即可轻松实现 IPC。`ContentProvider` 虽然使用起来很简单，包括自己创建一个 `ContentProvider` 也不是什么难事，尽管如此，它的细节还是相当多，比如 `CRUD` 操作、防止 `SQL` 注入和权限控制等。由于章节主题限制，在本节中，笔者暂时不对 `ContentProvider` 的使用细节以及工作机制进行详细分析，而是为读者介绍采用 `ContentProvider` 进行跨进程通信的主要流程，至于使用细节和内部工作机制会在后续章节进行详细分析。
- 系统预置了许多 `ContentProvider`，比如通讯录信息、日程表信息等，要跨进程访问这些信息，只需要通过 `ContentResolver` 的 `query`、`update`、`insert` 和 `delete` 方法即可。在本节中，我们来实现一个自定义的 `ContentProvider`，并演示如何在其他应用中获取 `ContentProvider` 中的数据从而实现进程间通信这一目的。首先，我们创建一个 `ContentProvider`，名字就叫 `BookProvider`。创建一个自定义的 `ContentProvider` 很简单，只需要继承 `ContentProvider` 类并实现六个抽象方法即可：`onCreate`、`query`、`update`、`insert`、`delete` 和 `getType`。这六个抽象方法都很好理解，`onCreate` 代表 `ContentProvider` 的创建，一般来说我们需要做一些初始化工作；`getType` 用来返回一个 `Uri` 请求所对应的 `MIME` 类型（媒体类型），比如图片、视频等，这个媒体类型还是有点复杂的，如果我们的应用不关注这个选项，可以直接在这个方法

中返回 `null` 或者 “\*\*”; 剩下的四个方法对应于 CRUD 操作, 即实现对数据表的增删改查功能。根据 Binder 的工作原理, 我们知道这六个方法均运行在 `ContentProvider` 的进程中, 除了 `onCreate` 由系统回调并运行在主线程里, 其他五个方法均由外界回调并运行在 Binder 线程池中, 这一点在接下来的例子中可以再次证明。

- `ContentProvider` 主要以表格的形式来组织数据, 并且可以包含多个表, 对于每个表格来说, 它们都具有行和列的层次性, 行往往对应一条记录, 而列对应一条记录中的一个字段, 这点和数据库很类似。除了表格的形式, `ContentProvider` 还支持文件数据, 比如图片、视频等。文件数据和表格数据的结构不同, 因此处理这类数据时可以在 `ContentProvider` 中返回文件的句柄给外界从而让文件来访问 `ContentProvider` 中的文件信息。Android 系统所提供的 `MediaStore` 功能就是文件类型的 `ContentProvider`, 详细实现可以参考 `MediaStore`。另外, 虽然 `ContentProvider` 的底层数据看起来很像一个 `SQLite` 数据库, 但是 `ContentProvider` 对底层的数据存储方式没有任何要求, 我们既可以使用 `SQLite` 数据库, 也可以使用普通的文件, 甚至可以采用内存中的一个对象来进行数据的存储, 这一点在后续的章节中会再次介绍, 所以这里不再深入了。
- 下面看一个最简单的示例, 它演示了 `ContentProvider` 的工作工程。首先创建一个 `BookProvider` 类, 它继承自 `ContentProvider` 并实现了 `ContentProvider` 的六个必须需要实现的抽象方法。在下面的代码中, 我们什么都没干, 尽管如此, 这个 `BookProvider` 也是可以工作的, 只是它无法向外界提供有效的数据而已。

```
public class BookProvider extends ContentProvider {
    private static final String TAG = "BookProvider";
    @Override
    public boolean onCreate() {
        Log.d(TAG, "onCreate, current thread:"
            + Thread.currentThread().getName());
        return false;
    }
    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
        String[] selectionArgs, String sortOrder) {
        Log.d(TAG, "query, current thread:" + Thread.currentThread().getName());
        return null;
    }
    @Override
    public String getType(Uri uri) {
        Log.d(TAG, "getType");
        return null;
    }
    @Override
    public Uri insert(Uri uri, ContentValues values) {
        Log.d(TAG, "insert");
        return null;
    }
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        Log.d(TAG, "delete");
        return 0;
    }
    @Override
    public int update(Uri uri, ContentValues values, String selection,
        String[] selectionArgs) {
        Log.d(TAG, "update");
        return 0;
    }
}
```

- 接着我们需要注册这个 `BookProvider`, 如下所示。其中 `android:authorities` 是 `ContentProvider` 的唯一标识, 通过这个属性外部应用就可以访问我们的 `BookProvider`, 因此, `android:authorities` 必须是唯一的, 这里建议读者在命名的时候加上包名前缀。为了演示进程间通信, 我们让 `BookProvider` 运行在独立的进程中并给它添加了权限, 这样外界应用如果想访问 `BookProvider`, 就必须声明 “`com.ry.PROVIDER`” 这个权限。`ContentProvider` 的权限还可以细分为读权限和写权限, 分别对应 `android:readPermission` 和 `android:writePermission` 属性, 如果分别声明了读权限和写权限, 那么外界应用也必须依样声明相应的权限才可以进行读/写操作, 否则外界应用会异常终止。关于权限这一块, 请读者自行查阅相关资料, 本章不进行详细介绍。

```
<provider
    android:name=".provider.BookProvider"
    android:authorities="com.ryg.chapter_2.book.provider"
    android:permission="com.ryg.PROVIDER"
    android:process=":provider"></provider>
```

- 注册了 `ContentProvider` 以后，我们就可以在外部应用中访问它了。为了方便演示，这里仍然选择在同一个应用的其他进程中去访问这个 `BookProvider`，至于在单独的应用中去访问这个 `BookProvider`，和同一个应用中访问的效果是一样的，读者可以自行试一下（注意要声明对应权限）。

```
public class ProviderActivity extends Activity {
    private static final String TAG = "ProviderActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_provider);
        Uri uri = Uri.parse("content://com.ryg.chapter_2.book.provider");
        getContentResolver().query(uri, null, null, null, null);
        getContentResolver().query(uri, null, null, null, null);
        getContentResolver().query(uri, null, null, null, null);
    }
}
```

- 在上面的代码中，我们通过 `ContentResolver` 对象的 `query` 方法去查询 `BookProvider` 中的数据，其中“`content://com.ryg.chapter_2.boobook.provider`”唯一标识了 `BookProvider`，而这个标识正是我们前面为 `BookProvider` 的 `android:authorities` 属性所指定的值。我们运行后看一下 log。从下面 log 可以看出，`BookProvider` 中的 `query` 方法被调用了三次，并且这三次调用不在同一个线程中。可以看出，它们运行在一个 `Binder` 线程中，前面提 `update`、`insert` 和 `delete` 方法同样也运行在 `Binder` 线程中。另外，`onCreate` 运行在 `main` 线程中，也就是 `UI` 线程，所以我们不能在 `onCreate` 中做耗时操作。

```
2019-03-26 11:38:01.381 6215-6215/com.ryg.chapter_2:provider D/BookProvider: onCreate, current thread:main
2019-03-26 11:38:01.395 6215-6228/com.ryg.chapter_2:provider D/BookProvider: query, current thread:Binder:6215_2
2019-03-26 11:38:01.397 6215-6215/com.ryg.chapter_2:provider D/MyApplication: application start, process name:com.ryg.chapter_2
2019-03-26 11:38:01.399 6215-6227/com.ryg.chapter_2:provider D/BookProvider: query, current thread:Binder:6215_1
2019-03-26 11:38:01.401 6215-6228/com.ryg.chapter_2:provider D/BookProvider: query, current thread:Binder:6215_2
```

- 到这里，整个 `ContentProvider` 的流程我们已经跑通了，虽然 `ContentProvider` 中没有返回任何数据。接下来，在上面的基础上，我们继续完善 `BookProvider`，从而使其能够对外部应用提供数据。继续本章提到的那个例子，现在我们要提供一个 `BookProvider`，外部应用可以通过 `BookProvider` 来访问图书信息，为了更好地演示 `ContentProvider` 的使用，用户还可以通过 `BookProvider` 访问到用户信息。为了完成上述功能，我们需要一个数据库来管理图书和用户信息，这个数据库不难实现，代码如下：

```
public class DbOpenHelper extends SQLiteOpenHelper {
    private static final String DB_NAME = "book_provider.db";
    public static final String BOOK_TABLE_NAME = "book";
    public static final String USER_TABLE_NAME = "user";
    private static final int DB_VERSION = 3;
    private String CREATE_BOOK_TABLE = "CREATE TABLE IF NOT EXISTS "
        + BOOK_TABLE_NAME + "(_id INTEGER PRIMARY KEY," + "name TEXT)";
    private String CREATE_USER_TABLE = "CREATE TABLE IF NOT EXISTS "
        + USER_TABLE_NAME + "(_id INTEGER PRIMARY KEY," + "name TEXT,"
        + "sex INT)";
    public DbOpenHelper(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_BOOK_TABLE);
        db.execSQL(CREATE_USER_TABLE);
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // TODO ignored
    }
}
```

- 上述代码是一个最简单的数据库的实现，我们借助 SQLiteOpenHelper 来管理数据库的创建、升级和降级。下面我们就要通过 BookProvider 向外界提供上述数据库中的信息了。我们知道，ContentProvider 通过 Uri 来区分外界要访问的数据集合，在本例中支持外界对 BookProvider 中的 book 表和 user 表进行访问，为了知道外界要访问的是哪个表，我们需要为它们定义单独的 Uri 和 Uri Code，并将 Uri 和对应的 Uri\_Code 相关联，我们可以使用 UriMatcher 的 addURI 方法将 Uri 和 Uri\_Code 关联到一起。这样，当外界请求访问 BookProvider 时，我们就可以根据请求的 Uri 来得到 Uri\_Code，有了 Uri\_Code 我们就可以知道外界想要访问哪个表，然后就可以进行相应的数据操作了，具体代码如下所示。

```
public class BookProvider extends ContentProvider {
    private static final String TAG = "BookProvider";
    public static final String AUTHORITY = "com.ryg.chapter_2.book.provider";
    /* 创建匹配路径的 Uri */
    public static final Uri BOOK_CONTENT_URI = Uri.parse("content://"
        + AUTHORITY + "/book");
    public static final Uri USER_CONTENT_URI = Uri.parse("content://"
        + AUTHORITY + "/user");

    public static final int BOOK_URI_CODE = 0;
    public static final int USER_URI_CODE = 1;
    private static final UriMatcher sUriMatcher = new UriMatcher(
        UriMatcher.NO_MATCH);
    static {
        sUriMatcher.addURI(AUTHORITY, "book", BOOK_URI_CODE);
        sUriMatcher.addURI(AUTHORITY, "user", USER_URI_CODE);
    }
}
```

- 从上面代码可以看出，我们分别为 book 表和 user 表指定了 Uri，分别为“content://com.ryg.chapter\_2”和“content://com.ryg.chapter\_2.book.provider/user”，这两个 Uri 所关联的 Uri\_Code 分别为 0 和 1。这个关联过程是通过下面的语句来完成的：

```
private String getTableName(Uri uri) {
    String tableName = null;
    switch (sUriMatcher.match(uri)) {
        case BOOK_URI_CODE:
            tableName = DbOpenHelper.BOOK_TABLE_NAME;
            break;
        case USER_URI_CODE:
            tableName = DbOpenHelper.USER_TALBE_NAME;
            break;
        default:break;
    }
    return tableName;
}
```

- 接着，我们就可以实现 query、update、insert、delete 方法了。如下是 query 方法的实现，首先我们要从 Uri 中取出外界要访问的表的名称，然后根据外界传递的查询参数就可以进行数据库的查询操作了，这个过程比较简单。

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
    String[] selectionArgs, String sortOrder) {
    Log.d(TAG, "query, current thread:" + Thread.currentThread().getName());
    String table = getTableName(uri);
    if (table == null) {
        throw new IllegalArgumentException("Unsupported URI: " + uri);
    }
    return mDb.query(table, projection, selection, selectionArgs, null, null, sortOrder, null);
}
```

- 另外三个方法的实现思想和 query 是类似的，只有一点不同，那就是 update、insert 和 delete 方法会引起数据源的改变，这个时候我们需要通过 ContentResolver 的 notifyChange 方法来通知外界当前 ContentProvider 中的数据已经发生改变。要观察一个 ContentProvider 中的数据改变情况，可以通过 ContentResolver 的 registerContentObserver 方法来注册观察者，通过 unregisterContentObserver 方法来解除观察者。对于这三个方法，这里不再详细解释了，BookProvider 的完整代码如下：



```

public class BookProvider extends ContentProvider {
    private static final String TAG = "BookProvider";
    public static final String AUTHORITY = "com.ryg.chapter_2.book.provider";
    public static final Uri BOOK_CONTENT_URI = Uri.parse("content://"
        + AUTHORITY + "/book");
    public static final Uri USER_CONTENT_URI = Uri.parse("content://"
        + AUTHORITY + "/user");
    public static final int BOOK_URI_CODE = 0;
    public static final int USER_URI_CODE = 1;
    private static final UriMatcher sUriMatcher = new UriMatcher(
        UriMatcher.NO_MATCH);
    static {
        sUriMatcher.addURI(AUTHORITY, "book", BOOK_URI_CODE);
        sUriMatcher.addURI(AUTHORITY, "user", USER_URI_CODE);
    }
    private Context mContext;
    private SQLiteDatabase mDb;
    @Override
    public boolean onCreate() {
        Log.d(TAG, "onCreate, current thread:" + Thread.currentThread().getName());
        mContext = getContext();
        initProviderData();
        return true;
    }
    private void initProviderData() {
        mDb = new DbOpenHelper(mContext).getWritableDatabase();
        mDb.execSQL("delete from " + DbOpenHelper.BOOK_TABLE_NAME);
        mDb.execSQL("delete from " + DbOpenHelper.USER_TABLE_NAME);
        mDb.execSQL("insert into book values(3,'Android');");
        mDb.execSQL("insert into book values(4,'Ios');");
        mDb.execSQL("insert into book values(5,'Html5');");
        mDb.execSQL("insert into user values(1,'jake',1);");
        mDb.execSQL("insert into user values(2,'jasmine',0);");
    }
    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
        String[] selectionArgs, String sortOrder) {
        Log.d(TAG, "query, current thread:" + Thread.currentThread().getName());
        String table = getTableName(uri);
        if (table == null) {
            throw new IllegalArgumentException("Unsupported URI: " + uri);
        }
        return mDb.query(table, projection, selection, selectionArgs, null, null, sortOrder, null);
    }
    @Override
    public String getType(Uri uri) {
        Log.d(TAG, "getType");
        return null;
    }
    @Override
    public Uri insert(Uri uri, ContentValues values) {
        Log.d(TAG, "insert");
        String table = getTableName(uri);
        if (table == null) {
            throw new IllegalArgumentException("Unsupported URI: " + uri);
        }
        mDb.insert(table, null, values);
        mContext.getContentResolver().notifyChange(uri, null);
        return uri;
    }
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        Log.d(TAG, "delete");
        String table = getTableName(uri);
        if (table == null) {
            throw new IllegalArgumentException("Unsupported URI: " + uri);
        }
        int count = mDb.delete(table, selection, selectionArgs);
        if (count > 0) {
            getContext().getContentResolver().notifyChange(uri, null);
        }
        return count;
    }
    @Override
    public int update(Uri uri, ContentValues values, String selection,
        String[] selectionArgs) {
        Log.d(TAG, "update");
        String table = getTableName(uri);
        if (table == null) {

```

```

        throw new IllegalArgumentException("Unsupported URI: " + uri);
    }
    int row = mDb.update(table, values, selection, selectionArgs);
    if (row > 0) {
        getContext().getContentResolver().notifyChange(uri, null);
    }
    return row;
}
private String getTableName(Uri uri) {
    String tableName = null;
    switch (sUriMatcher.match(uri)) {
        case BOOK_URI_CODE:
            tableName = DbOpenHelper.BOOK_TABLE_NAME;
            break;
        case USER_URI_CODE:
            tableName = DbOpenHelper.USER_TALBE_NAME;
            break;
        default:break;
    }
    return tableName;
}
}
}

```

- 需要注意的是，query、update、insert、delete 四大方法是存在多线程并发访问的，因此方法内部要做好线程同步。在本例中，由于采用的是 SQLite 并且只有一个 SQLiteDatabase 的连接，所以可以正确应对多线程的情况。具体原因是 SQLiteDatabase 内部对数据库的操作是有同步处理的，但是如果通过多个 SQLiteDatabase 对象来操作数据库就无法保证线程同步，因为 SQLiteDatabase 对象之间无法进行线程同步。如果 ContentProvider 的底层数据集是一块内存的话，比如是 List，在这种情况下同 List 的遍历、插入、删除操作就需要进行线程同步，否则就会引起并发错误，这点是尤其需要注意的。到这里 BookProvider 已经实现完成了，接着我们在外部访问一下它，看看是否能够正常工作。

```

public class ProviderActivity extends Activity {
    private static final String TAG = "ProviderActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_provider);
        // Uri uri = Uri.parse("content://com.ryg.chapter_2.book.provider");
        // getContentResolver().query(uri, null, null, null, null);
        // getContentResolver().query(uri, null, null, null, null);
        // getContentResolver().query(uri, null, null, null, null);
        //
        Uri bookUri = Uri.parse("content://com.ryg.chapter_2.book.provider/book");
        ContentValues values = new ContentValues();
        values.put("_id", 6);
        values.put("name", " 程序设计的艺术");
        getContentResolver().insert(bookUri, values);
        Cursor bookCursor = getContentResolver().query(bookUri, new String[]{"_id", "name"}, null, null, null);
        while (bookCursor.moveToNext()) {
            Book book = new Book();
            book.bookId = bookCursor.getInt(0);
            book.bookName = bookCursor.getString(1);
            Log.d(TAG, "query book:" + book.toString());
        }
        bookCursor.close();
        Uri userUri = Uri.parse("content://com.ryg.chapter_2.book.provider/user");
        Cursor userCursor = getContentResolver().query(userUri, new String[]{"_id", "name", "sex"}, null, null, null);
        while (userCursor.moveToNext()) {
            User user = new User();
            user.userId = userCursor.getInt(0);
            user.userName = userCursor.getString(1);
            user.isMale = userCursor.getInt(2) == 1;
            Log.d(TAG, "query user:" + user.toString());
        }
        userCursor.close();
    }
}
}

```

- 默认情况下，BookProvider 的数据库中有三本书和两个用户，在上面的代码中，我们首先添加一本书：“程序设计的艺术”。接着查询所有的图书，这个时候应该查询出四本书，因为我们刚刚添加了一本。然后查询所有的用户，这个时候应该查询出两个用户。是不是这样呢？我们运行一下程序，看一下 log。

```

2019-03-26 15:43:18.451 18553-18553/com.ryg.chapter_2 D/ProviderActivity: query book:[bookId:3, bookName:Android]
2019-03-26 15:43:18.451 18553-18553/com.ryg.chapter_2 D/ProviderActivity: query book:[bookId:4, bookName:Ios]
2019-03-26 15:43:18.452 18553-18553/com.ryg.chapter_2 D/ProviderActivity: query book:[bookId:5, bookName:Html5]
2019-03-26 15:43:18.453 18553-18553/com.ryg.chapter_2 D/ProviderActivity: query book:[bookId:6, bookName:程序设计的艺术]
2019-03-26 15:43:18.460 18553-18553/com.ryg.chapter_2 D/ProviderActivity: query user:User:{userId:1, userName:jake, isMale:
2019-03-26 15:43:18.461 18553-18553/com.ryg.chapter_2 D/ProviderActivity: query user:User:{userId:2, userName:jasmine, isMa

```

- 从上述 log 可以看到，我们的确查询到了 4 本书和 2 个用户，这说明 BookProvider 已经能够正确地处理外部的请求了，读者可以自行验证一下 update 和 delete 操作，这里就不再验证了。同时，由于 ProviderActivity 和 BookProvider 运行在两个不同的进程中，因此，这也构成了进程间的通信。ContentProvider 除了支持对数据源的增删改查这四个操作，还支持自定义调用，这个过程是通过 ContentResolver 的 Call 方法和 ContentProvider 的 Call 方法来完成的。关于使用 ContentProvider 来进行 IPC 就介绍到这里。

## 1.6 使用 Socket

- 我们通过 Socket 来实现进程间的通信。Socket 也称为“套接字”，是网络通信中的概念，它分为流式套接字和用户数据报套接字两种，分别对应于网络的传输控制层中的 TCP 和 UDP 协议。TCP 协议是面向连接的协议，提供稳定的双向通信功能，TCP 连接的建立需要经过“三次握手”才能完成，为了提供稳定的数据传输功能，其本身提供了超时重传机制，因此具有很高的稳定性；而 UDP 是无连接的，提供不稳定的单向通信功能，当然 UDP 也可以实现双向通信功能。在性能上，UDP 具有更好的效率，其缺点是不保证数据一定能够正确传输，尤其是在网络拥塞的情况下。关于 TCP 和 UDP 的介绍就这么多，更详细的资料请查看相关网络资料。接下来我们演示一个跨进程的聊天程序，两个进程可以通过 Socket 来实现信息的传输，Socket 本身可以支持传输任意字节流，这里为了简单起见，仅仅传输文本信息，很显然，这是一种 IPC 方式。
- 使用 Socket 来进行通信，有两点需要注意，首先需要声明权限：

```

<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

```

- 其次要注意不能在主线程中访问网络，因为这会导致我们的程序无法在 Android4.0 及其以上的设备中运行，会抛出如下异常：android.os.NetworkOnMainThreadException。而且进行网络操作很可能是耗时的，如果放在主线程中会影响程序的响应效率，从这方面来说，也不应该在主线程中访问网络。下面就开始设计我们的聊天室程序了，比较简单，首先在远程 Service 建立一个 TCP 服务，然后在主界面中连接 TCP 服务，连接上了以后，就可以给服务端发消息。对于我们发送的每一条文本消息，服务端都会随机地回应我们一句话。为了更好地展示 Socket 的工作机制，在服务端我们做了处理，使其能够和多个客户端同时建立连接并响应。
- 先看一下服务端的设计，当 Service 启动时，会在线程中建立 TCP 服务，这里监听的是 8688 端口，然后就可以等待客户端的连接请求。当有客户端连接时，就会生成一个新的 Socket，通过每次新创建的 Socket 就可以分别和不同的客户端通信了。服务端每收到一次客户端的消息就会随机回复一句话给客户端。当客户端断开连接时，服务端这边也会相应的关闭对应 Socket 并结束通话线程，这点是如何做到的呢？方法有很多，这里是通过判断服务端输入流的返回值来确定的，当客户端断开连接后，服务端这边的输入流会返回 null，这个时候我们就知道客户端退出了。服务端的代码如下所示。

```

public class TCPServerService extends Service {
    private boolean mIsServiceDestoryed = false;
    private String[] mDefinedMessages = new String[] {
        " 你好啊，哈哈",
        " 请问你叫什么名字呀？",
        " 今天北京天气不错啊，shy",
        " 你知道吗？我可是可以和多个人同时聊天的哦",
        " 给你讲个笑话吧：据说爱笑的人运气不会太差，不知道真假。"
    };
}
@Override
public void onCreate() {
    new Thread(new TcpServer()).start();
}

```

```

        super.onCreate();
    }
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
    @Override
    public void onDestroy() {
        mIsServiceDestroyed = true;
        super.onDestroy();
    }
    private class TcpServer implements Runnable {
        @SuppressWarnings("resource")
        @Override
        public void run() {
            ServerSocket serverSocket = null;
            try {
                serverSocket = new ServerSocket(8688);
            } catch (IOException e) {
                System.err.println("establish tcp server failed, port:8688");
                e.printStackTrace();
                return;
            }
            while (!mIsServiceDestroyed) {
                try {
                    // 接受客户端请求
                    final Socket client = serverSocket.accept();
                    System.out.println("accept");
                    new Thread() {
                        @Override
                        public void run() {
                            try {
                                responseClient(client);
                            } catch (IOException e) {
                                e.printStackTrace();
                            }
                        }
                    }.start();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    private void responseClient(Socket client) throws IOException {
        // 用于接收客户端消息
        BufferedReader in = new BufferedReader(new InputStreamReader(
            client.getInputStream()));
        // 用于向客户端发送消息
        PrintWriter out = new PrintWriter(new BufferedWriter(
            new OutputStreamWriter(client.getOutputStream())), true);
        out.println(" 欢迎来到聊天室! ");
        while (!mIsServiceDestroyed) {
            String str = in.readLine();
            System.out.println("msg from client:" + str);
            if (str == null) {
                break;
            }
            int i = new Random().nextInt(mDefinedMessages.length);
            String msg = mDefinedMessages[i];
            out.println(msg);
            System.out.println("send :" + msg);
        }
        System.out.println("client quit.");
        // 关闭流
        MyUtils.close(out);
        MyUtils.close(in);
        client.close();
    }
}

```

- 接着看一下客户端，客户端 Activity 启动时，会在 onCreate 中开启一个线程去连接服务端 Socket，至于为什么用线程在前面已经做了介绍。为了确定能够连接成功，这里采用了超时重连的策略，每次连接失败后都会重新建立尝试建立连接。当然为了降低重试机制的开销，我们加入了休眠机制，即每次重试的时间间隔为 1000 毫秒。

```
Socket socket = null;
```

```

while (socket == null) {
    try {
        socket = new Socket("localhost", 8688);
        mClientSocket = socket;
        mPrintWriter = new PrintWriter(new BufferedWriter(
            new OutputStreamWriter(socket.getOutputStream()), true);
        mHandler.sendMessage(MESSAGE_SOCKET_CONNECTED);
        System.out.println("connect server success");
    } catch (IOException e) {
        SystemClock.sleep(1000);
        System.out.println("connect tcp server failed, retry...");
    }
}
}

```

- 服务端连接成功以后，就可以和服务端进行通信了。下面的代码在线程中通过 while 循环不断地去读取服务端发送过来的消息，同时当 Activity 退出时，就退出循环并终止线程。

```

try {
    // 接收服务器端的消息
    BufferedReader br = new BufferedReader(new InputStreamReader(
        socket.getInputStream()));
    while (!TCPClientActivity.this.isFinishing()) {
        String msg = br.readLine();
        System.out.println("receive :" + msg);
        if (msg != null) {
            String time = formatDateTime(System.currentTimeMillis());
            final String showedMsg = "server " + time + ":" + msg + "\n";
            mHandler.obtainMessage(MESSAGE_RECEIVE_NEW_MSG, showedMsg).sendToTarget();
        }
    }
    System.out.println("quit...");
    MyUtils.close(mPrintWriter);
    MyUtils.close(br);
    socket.close();
} catch (IOException e) {
    e.printStackTrace();
}
}

```

- 同时，当 Activity 退出时，还要关闭当前的 Socket，如下所示。

```

@Override
protected void onDestroy() {
    if (mClientSocket != null) {
        try {
            mClientSocket.shutdownInput();
            mClientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    super.onDestroy();
}
}

```

- 接着是发送消息的过程，这个就很简单了，这里不再详细说明。客户端的完整代码如下：

```

public class TCPClientActivity extends Activity implements OnClickListener {
    private static final int MESSAGE_RECEIVE_NEW_MSG = 1;
    private static final int MESSAGE_SOCKET_CONNECTED = 2;
    private Button mSendButton;
    private TextView mMessageTextView;
    private EditText mMessageEditText;
    private PrintWriter mPrintWriter;
    private Socket mClientSocket;
    @SuppressWarnings("HandlerLeak")
    private Handler mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MESSAGE_RECEIVE_NEW_MSG: {
                    mMessageTextView.setText(mMessageTextView.getText() + (String) msg.obj);
                    break;
                }
                case MESSAGE_SOCKET_CONNECTED: {
                    mSendButton.setEnabled(true);
                    break;
                }
            }
        }
    }
}

```



```

        default:
            break;
    }
}
};
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_tcpclient);
    mMessageTextView = (TextView) findViewById(R.id.msg_container);
    mSendButton = (Button) findViewById(R.id.send);
    mSendButton.setOnClickListener(this);
    mMessageEditText = (EditText) findViewById(R.id.msg);
    Intent service = new Intent(this, TCPServerService.class);
    startService(service);
    new Thread() {
        @Override
        public void run() {
            connectTCPServer();
        }
    }.start();
}
@Override
protected void onDestroy() {
    if (mClientSocket != null) {
        try {
            mClientSocket.shutdownInput();
            mClientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    super.onDestroy();
}
@Override
public void onClick(View v) {
    if (v == mSendButton) {
        final String msg = mMessageEditText.getText().toString();
        if (!TextUtils.isEmpty(msg) && mPrintWriter != null) {
            new Thread() {
                @Override
                public void run() {
                    mPrintWriter.println(msg);
                }
            }.start();
            mMessageEditText.setText("");
            String time = formatDate(System.currentTimeMillis());
            final String showedMsg = "self " + time + ":" + msg + "\n";
            mMessageTextView.setText(mMessageTextView.getText() + showedMsg);
        }
    }
}
@SuppressLint("SimpleDateFormat")
private String formatDate(long time) {
    return new SimpleDateFormat("HH:mm:ss").format(new Date(time));
}
private void connectTCPServer() {
    Socket socket = null;
    while (socket == null) {
        try {
            socket = new Socket("localhost", 8688);
            mClientSocket = socket;
            mPrintWriter = new PrintWriter(new BufferedWriter(
                new OutputStreamWriter(socket.getOutputStream()), true);
            mHandler.sendMessage(MESSAGE_SOCKET_CONNECTED);
            System.out.println("connect server success");
        } catch (IOException e) {
            SystemClock.sleep(1000);
            System.out.println("connect tcp server failed, retry...");
        }
    }
    try {
        // 接收服务器端的消息
        BufferedReader br = new BufferedReader(new InputStreamReader(
            socket.getInputStream()));
        while (!TCPClientActivity.this.isFinishing()) {
            String msg = br.readLine();
            System.out.println("receive : " + msg);
            if (msg != null) {

```

```

        String time = formatDateTime(System.currentTimeMillis());
        final String showedMsg = "server " + time + ":" + msg + "\n";
        mHandler.obtainMessage(MESSAGE_RECEIVE_NEW_MSG, showedMsg).sendToTarget();
    }
    System.out.println("quit...");
    MyUtils.close(mPrintWriter);
    MyUtils.close(br);
    socket.close();
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

- 上述就是通过 Socket 来进行进程间通信的实例，除了采用 TCP 套接字，还可以采用 UDP 套接字。另外，上面的例子仅仅是一个示例，实际上通过 Socket 不仅仅能实现进程间的通信，还可以实现设备间的通信，当然前提是这些设备之间的 IP 地址互相可见，这其中又涉及许多复杂的概念，这里就不一一介绍了。下面看一下上述例子的运行效果，如图。

## 1.7 Binder 连接池

- 如何使用 AIDL 在上面已经进行了介绍，这里在回顾一下大致流程：首先创建一个 Service 和一个 AIDL 接口，接着创建一个类继承自 AIDL 接口中的 Stub 类并实现 Stub 中的抽象方法，在 Service 的 onBind 方法中返回这个类的对象，然后客户端就可以绑定服务端 Service，建立连接后就可以访问远程服务端的方法了。
- 上述过程就是典型的 AIDL 的使用流程。这本来也没什么问题，但是现在考虑一种情况：公司的项目越来越庞大了，现在有 10 个不同的业务模块都需要使用 AIDL 来进行进程间通信，那我们该怎么处理呢？也许你会说：“就按照 AIDL 的实现方式一个个来吧”，这是可以的，如果用这种方法，首先我们需要创建 10 个 Service，这好像有点多啊！如果有 100 个地方需要用到 AIDL 呢，先创建 100 个 Service？到这里，读者应该明白问题所在了。随着 AIDL 数量的增加，我们不能无限制地增加 Service，Service 是四大组件之一，本身就是一种系统资源。而且太多的 Service 会使得我们的应用看起来很重量级，因为正在运行的 Service 可以在应用详情页看到，当我们的应用详情显示有 10 个服务正在运行时，这看起来并不是什么好事。针对上述问题，我们需要减少 Service 的数量，将所有的 AIDL 放在同一个 Service 中去管理。
- 在这种模式下，整个工作机制是这样的：每个业务模块创建自己的 AIDL 接口并实现此接口，这个时候不同业务模块之间是不能有耦合的，所有实现细节我们要单独开来，然后向服务端提供自己的唯一标识和其对应的 Binder 对象；对于服务端来说，只需要一个 Service 就可以了，服务端提供一个 queryBinder 接口，这个接口能够根据业务模块的特征来返回相应的 Binder 对象给它们，不同的业务模块拿到所需的 Binder 对象后就可以进行远程方法调用了。由此可见，Binder 连接池的主要作用就是将每个业务模块的 Binder 请求统一转发到远程 Service 中去执行，从而避免了重复创建 Service 的过程，它的工作原理如图 2-10 所示。
- 通过上面的理论介绍，也许还有点不好理解，下面对 Binder 连接池的代码实现做一下说明。首先，为了说明问题，我们提供了两个 AIDL 接口 (ISecurityCenter 和 ICompute) 来模拟上面提到的多个业务模块都要使用 AIDL 的情况，其中 ISecurityCenter 接口提供加解密功能，声明如下：

```

interface ISecurityCenter {
    String encrypt(String content);
    String decrypt(String password);
}

```

- 而 ICompute 接口提供计算加法的功能，声明如下：

```

interface ICompute {
    int add(int a, int b);
}

```

- 虽然上面两个接口的功能都比较简单，但是用于分析 Binder 连接池的工作原理已经足够了，读者可以写出更复杂的例子。接着看一下上面两个 AIDL 接口的实现，也比较简单，代码如下：

```
public class SecurityCenterImpl extends ISecurityCenter.Stub {
    private static final char SECRET_CODE = '^';
    @Override
    public String encrypt(String content) throws RemoteException {
        char[] chars = content.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            chars[i] ^= SECRET_CODE;
        }
        return new String(chars);
    }
    @Override
    public String decrypt(String password) throws RemoteException {
        return encrypt(password);
    }
}
```

- 现在业务模块的 AIDL 接口定义和实现都已经完成了，注意这里并没有为每个模块的 AIDL 单独创建 Service，接下来就是服务端和 Binder 连接池的工作了。首先，为 Binder 连接池创建 AIDL 接口 IBinderPool.aidl，代码如下所示。

```
interface IBinderPool {
    /**
     * @param binderCode, the unique token of specific Binder<br/>
     * @return specific Binder who's token is binderCode.
     */
    IBinder queryBinder(int binderCode);
}
```

- 接着，为 Binder 连接池创建远程 Service 并实现 IBinderPool，下面是 queryBinder 的具体实现，可以看到请求转发的实现方法，当 Binder 连接池连接上远程服务时，会根据不同模块的标识即 binderCode 返回不同的 Binder 对象，通过这个 Binder 对象所执行的操作全部发生在远程服务端。

```
public static class BinderPoolImpl extends IBinderPool.Stub {
    public BinderPoolImpl() {
        super();
    }
    @Override
    public IBinder queryBinder(int binderCode) throws RemoteException {
        IBinder binder = null;
        switch (binderCode) {
            case BINDER_SECURITY_CENTER: {
                binder = new SecurityCenterImpl();
                break;
            }
            case BINDER_COMPUTE: {
                binder = new ComputeImpl();
                break;
            }
            default:
                break;
        }
        return binder;
    }
}
```

- 远程 Service 的实现就比较简单了，代码如下所示。

```
public class BinderPoolService extends Service {
    private static final String TAG = "BinderPoolService";
    private Binder mBinderPool = new BinderPool.BinderPoolImpl();
    @Override
    public void onCreate() {
        super.onCreate();
    }
    @Override
    public IBinder onBind(Intent intent) {
        Log.d(TAG, "onBind");
        return mBinderPool;
    }
}
```

```

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

```

- 下面还剩下 Binder 连接池的具体实现，在它的内部首先它要去绑定远程服务，绑定成功后，客户端就可以通过它的 queryBinder 方法去获取各自对应的 Binder，拿到所需的 Binder 以后，不同业务模块就可以进行各自的操作了，Binder 连接池的代码如下所示。

```

public class BinderPool {
    private static final String TAG = "BinderPool";
    public static final int BINDER_NONE = -1;
    public static final int BINDER_COMPUTE = 0;
    public static final int BINDER_SECURITY_CENTER = 1;
    private Context mContext;
    private IBinderPool mBinderPool;
    private static volatile BinderPool sInstance;
    private CountDownLatch mConnectBinderPoolCountDownLatch;
    private BinderPool(Context context) {
        mContext = context.getApplicationContext();
        connectBinderPoolService();
    }
    public static BinderPool getInsance(Context context) {
        if (sInstance == null) {
            synchronized (BinderPool.class) {
                if (sInstance == null) {
                    sInstance = new BinderPool(context);
                }
            }
        }
        return sInstance;
    }
    private synchronized void connectBinderPoolService() {
        //等待线程的数量
        mConnectBinderPoolCountDownLatch = new CountDownLatch(1);
        Intent service = new Intent(mContext, BinderPoolService.class);
        mContext.bindService(service, mBinderPoolConnection, Context.BIND_AUTO_CREATE);
        try {
            mConnectBinderPoolCountDownLatch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    /**
     * query binder by binderCode from binder pool
     *
     * @param binderCode
     *         the unique token of binder
     * @return binder who's token is binderCode<br>
     *         return null when not found or BinderPoolService died.
     */
    public IBinder queryBinder(int binderCode) {
        IBinder binder = null;
        try {
            if (mBinderPool != null) {
                binder = mBinderPool.queryBinder(binderCode);
            }
        } catch (RemoteException e) {
            e.printStackTrace();
        }
        return binder;
    }
    private ServiceConnection mBinderPoolConnection = new ServiceConnection() {
        @Override
        public void onServiceDisconnected(ComponentName name) {
            // ignored.
        }
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            mBinderPool = IBinderPool.Stub.asInterface(service);
            try {
                mBinderPool.asBinder().linkToDeath(mBinderPoolDeathRecipient, 0);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
            mConnectBinderPoolCountDownLatch.countDown();
        }
    }
}

```

```

};
private IBinder.DeathRecipient mBinderPoolDeathRecipient = new IBinder.DeathRecipient() {
    @Override
    public void binderDied() {
        Log.w(TAG, "binder died.");
        mBinderPool.asBinder().unlinkToDeath(mBinderPoolDeathRecipient, 0);
        mBinderPool = null;
        connectBinderPoolService();
    }
};
public static class BinderPoolImpl extends IBinderPool.Stub {
    public BinderPoolImpl() {
        super();
    }
    @Override
    public IBinder queryBinder(int binderCode) throws RemoteException {
        IBinder binder = null;
        switch (binderCode) {
            case BINDER_SECURITY_CENTER: {
                binder = new SecurityCenterImpl();
                break;
            }
            case BINDER_COMPUTE: {
                binder = new ComputeImpl();
                break;
            }
            default:
                break;
        }
        return binder;
    }
}
}
}

```

- Binder 连接池的具体实现就分析完了，它的好处是显而易见的，针对上面的例子，我们只需要创建一个 Service 即可完成多个 AIDL 接口的工作，下面我们来验证一下效果。新创建一个 Activity，在线程中执行如下操作：

```

private void doWork() {
    BinderPool binderPool = BinderPool.getInstance(BinderPoolActivity.this);
    IBinder securityBinder = binderPool.queryBinder(BinderPool.BINDER_SECURITY_CENTER);
    mSecurityCenter = (ISecurityCenter) SecurityCenterImpl.asInterface(securityBinder);
    Log.d(TAG, "visit ISecurityCenter");
    String msg = "helloworld-安卓";
    System.out.println("content:" + msg);
    try {
        String password = mSecurityCenter.encrypt(msg);
        System.out.println("encrypt:" + password);
        System.out.println("decrypt:" + mSecurityCenter.decrypt(password));
    } catch (RemoteException e) {
        e.printStackTrace();
    }
    Log.d(TAG, "visit ICompute");
    IBinder computeBinder = binderPool.queryBinder(BinderPool.BINDER_COMPUTE);
    mCompute = ComputeImpl.asInterface(computeBinder);
    try {
        System.out.println("3+5=" + mCompute.add(3, 5));
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}
}

```

- 可以看到日志

```

2019-03-27 14:15:16.006 4043-4236/com.ryg.chapter_2 D/BinderPoolActivity: visit ISecurityCenter
2019-03-27 14:15:16.006 4043-4236/com.ryg.chapter_2 I/System.out: content:helloworld-安卓
2019-03-27 14:15:16.007 4043-4236/com.ryg.chapter_2 I/System.out: encrypt:6;221)1,2:s国匍
2019-03-27 14:15:16.008 4043-4236/com.ryg.chapter_2 I/System.out: decrypt:helloworld-安卓
2019-03-27 14:15:16.009 4043-4236/com.ryg.chapter_2 D/BinderPoolActivity: visit ICompute
2019-03-27 14:15:16.025 4043-4236/com.ryg.chapter_2 I/System.out: 3+5=8
2019-03-27 14:15:16.197 4043-4094/com.ryg.chapter_2 D/EGL_emulation: eglMakeCurrent: 0x96af8180: ver 3 0 (tinfo 0xa61ff580)

```

- 这里需要额外说明一下，为什么要在线程中去执行呢？这是因为在 Binder 连接池的实现中，我们通过 CountDownLatch 将 bindService 这一异步操作转换成了同步操作，这就意味着它有可能是耗时的，然后就是 Binder 方法的调用过程也可能是耗时的，因此不建议放在主线程



去执行。注意到 `BinderPool` 是一个单例实现，因此在同一个进程中只会初始化一次，所以如果我们提前初始化 `BinderPool`，那么可以优化程序的体验，比如我们可以放在 `Application` 中提前对 `BinderPool` 进行初始化，虽然这不能保证当我们调用 `BinderPool` 时它一定是初始化好的，但是在大多数情况下，这种初始化工作（绑定远程服务）的时间开销（如果 `BinderPool` 没有提前初始化完成的话）是可以接受的。另外，`BinderPool` 中有断线重连的机制，当远程服务意外终止时，`BinderPool` 会重新建立连接，这个时候如果业务模块中的 `Binder` 调用出现了异常，也需要手动去重新获取最新的 `Binder` 对象，这个是需要注意的。

- 有了 `BinderPool` 可以大大方便日常的开发工作，比如如果有一个新的业务模块需要添加新的 AIDL，那么在它实现了自己的 AIDL 接口后，只需要修改 `BinderPoolImpl` 中的 `queryBinder` 方法，给自己添加一个新的 `binderCode` 并返回对应的 `Binder` 对象即可，不需要做其他修改，也不需要创建新的 `Service`。由此可见，`BinderPool` 能够极大地提高 AIDL 的开发效率，并且可以避免大量的 `Service` 创建，因此，建议在 AIDL 开发工作中引入 `BinderPool` 机制。

## 1.8 各个 IPC 方式优缺点

名称	优点	缺点
Bundle	简单易用	只能传输 Bundle 支持的数据
文件共享	简单易用	不适合高并发场景，并且无法做到进程内
AIDL	功能强大，支持一对多并发通信，支持实时通信	使用稍微复杂，需要处理多线程同步
Messenger	功能一般，支持一对多串行通信，支持实时通信	不能很好处理高并发情形，不支持 RPC 进行传输，因此只能传输 Bundle 支持的
Content Provider	在数据源访问方面功能强大，支持一对多并发数据共享，可以通过 Call 方法进行扩展其他操作	可以理解为受约束的 AIDL，主要提供数据源的 CURD 操作
Socket	功能强大，可以通过网络传输字节流，支持一对多并发实时通信	实现细节稍微有点繁琐，不支持直接的

## 2 多进程应用的缺点

### 2.1 静态成员和单例模式的完全失效

- Android 为每个应用分配一个独立的虚拟机，或者说为每一个进程分配一个独立的虚拟机，不同的虚拟机在内存分配上有不同的地址空间，这就导致在不同虚拟机中访问同一个类的对象会产生多份副本。

### 2.2 线程同步机制完全失效

- 因为都不是同一块内存空间，所以不管是锁对象还是锁全局类都无法保证线程同步，因为不同进程锁的不是同一个对象。

### 2.3 Application 的 onCreate() 多次执行

多进程被创建好了，应用运行时就会对进程进行初始化，如果一个 application 中有多个进程，在进行全局初始化时，多进程会被初始化多次。解决方法：通过判断当前进程是不是应用包名（默认进程名就是应用包名），来决定需不需要进行初始化相关的操作。

```
private String getCurrentProcessName() { String currentProcName = ""; int pid = android.os.Process.myPid(); ActivityManager manager = (ActivityManager) this.getSystemService(Context.ACTIVITY_SERVICE); for (ActivityManager.RunningAppProcessInfo processInfo : manager.getRunningAppProcesses())
```

```
{ if (processInfo.pid == pid) { currentProcName = processInfo.processName; break; } } return currentProcName; }
```

如果子进程也需要一些初始化，则可根据进程名称来分别做对应的初始化即可，并且只初始化当前进程需要的业务，即按照进程按需初始化。

## 2.4 调试麻烦

- 多进程开发，调试起来很麻烦，比如希望在子进程创建后的首行代码里添加断点，这是做不到。因为 Android Studio 添加调试器的时候，会让你选一个调试的进程，而这时这个子进程还没有启动起来，一旦你启动起来后，首行代码就执行完毕了，就无法再添加调试了
- 解决方法：
  - 1 可以在子进程启动后，添加几秒的延迟再来启动业务逻辑，比如延迟 5 秒，而这 5 秒就是留给你去 Android Studio 中添加断点的
  - 2 调试时关闭子进程，例如某些操作是在一个 Service 时执行的，这时先把 service 放到主进程中，等业务代码写完了，再放到子进程进行功能测试。

## 2.5 不能内存共享

- 同一个进程的多个线程是共享该进程的所有资源，但多个进程间内存是不可见的，也就是说多个进程间内存是不共享的。会引起一些问题，如：静态成员的失效。

## 2.6 文件共享问题。

- 多进程情况下会出现两个进程在同一时刻访问同一个数据库文件的情况。这就可能造成资源的竞争访问，导致诸如数据库损坏数据丢失等。在多线程的情况下我们有锁机制控制资源的共享，但是在多进程中比较难，虽然有文件锁排队等机制，但是在 Android 里很难实现。解决办法就是多进程的时候不并发访问同一个文件，比如子进程涉及到操作数据库，就可以考虑调用主进程进行数据库的操作。

## 2.7 sharePreference 不能在多进程中通用

- 在主进程中往 sharePreference 存的值，在子进程不能立即可见。
- 解决方法
  - 1: 使用 SharedPreferences 时, SharedPreferences 在 MODE\_PRIVATE MODE\_PUBLIC 之外其实还可以设置多进程的 Flag，即 MODE\_MULTI\_PROCESS。

```
SharedPreferences myPrefs = context.getSharedPreferences(MY_FILE_NAME, Context.MODE_MULTI_PROCESS | Context.MODE_PRIVATE);
```

- 一旦我们设置了这个 Flag，每次调用 Context.getSharedPreferences 的时候系统会重新从 SP 文件中读入数据，因此我们在使用的時候每次读取和存入都要使用 Context.getSharedPreferences 重新获取 SP 实例。即使是这样，由于 SP 本质上并不是多进程安全的，所以还是无法保证数据的同步，因此该方法我们并没有使用，我们也不推荐使用。
- 2: tray 安卓多进程安全的 sharePreferences - <https://github.com/GCX-HCI/tray>
  - 如果 SP 不是多进程安全的，那么是否有多进程安全的，又有 SP 功能的第三方项目呢。答案是有的，Tray——一个多进程安全的 SharedPreferences，我们可以在 Github 上找到它，如果是 AndroidStudio，可以直接使用 Gradle 引入，可谓是十分方便，如下是使用的代码，十分简单，没有 apply commit，看起来比 SP 还要简单。
  - 但是最终我们并没有选择使用它，主要的原因是它需要 minSdk 为 15，而我们是支持 sdk14 的，所以只能果断放弃了。

## 2.8 多进程的通信问题

- 同一个进程的多个线程是共享该进程的所有资源，但多个进程间内存是不可见的，也就是说多个进程间内存是不共享的。但通常我们的业务代码都是有关联，很少有能开启多进程同时又不需要进程通信的业务场景，因此这就需要多进程之间进行通信。
- Android 中提供了三种进程通信方法：
  - 系统实现。
  - AIDL (Android Interface Definition Language, Android 接口定义语言)：大部分应用程序不应该使用 AIDL 去创建一个绑定服务，因为它需要多线程能力，并可能导致一个更复杂的实现。
  - Messenger：利用 Handler 实现。(适用于多进程单线程，不需要考虑线程安全)，其底层基于 AIDL。

## 2.9 多进程暖启动的恢复问题

- 在主进程的 MainActivity 退出后，子进程还在运行，这时用户又从手机桌面上点击应用图标启动了主进程，若业务功能中主进程与子进程有通信，那么此时应该让上次断开的主进程与子进程连接再重新连接上。这里就需要额外的判断。

## 2.10 内存共享：

- 不同进程之间内存不能共享，最大的弊端是他们之间通信麻烦，不能将公用数据放在 Application 中，堆栈信息、文件操作也是独立的，如果他们之间传递的数据不大并且是可序列

# 3 Android 进程保活招式大全

- 目前市面上的应用，貌似除了微信和 QQ 都会比较担心被用户或者系统厂商杀死的问题。Android 进程拉活包括两个层面：(1) 提供进程优先级，降低进程被杀死的概率。(2) 在进程被杀死后，进行拉活。本文下面就从这两个方面做一下总结。

## 3.1 提高进程优先级方面

### 3.1.1 进程优先级介绍

- Android 系统将尽量长时间地保持应用进程，但为了新建进程或运行更重要的进程，需要清除旧进程来回收内存。为了确定保留或终止哪些进程，系统会根据进程中正在运行的组件以及这些组件的状态，将每个进程赋予“重要性”。必要时系统会首先消除重要性最低的进程以回收系统资源。进程的重要性从高到低可以划分为以下五级：
- (1) 前台进程 (Foreground process)，即用户当前操作所必需的进程，通常数量不多。举例如下：
  - //拥有用户正在交互的 Activity (已调用 onResume())
  - //拥有某个 Service，后者绑定到用户正在交互的 Activity
  - //拥有正在“前台”运行的 Service (服务已调用 startForeground())
  - //拥有正执行一个生命周期回调的 Service (onCreate()、onStart() 或 onDestroy())
  - //拥有正执行其 onReceive() 方法的 BroadcastReceiver
- (2) 可见进程 (Visible process)，没有任何前台组件、但仍会影响用户在屏幕上所见内容的进程。举例如下：

- //拥有不在前台、但仍对用户可见的 Activity（已调用 onPause()）
- //拥有绑定到可见（或前台）Activity 的 Service
- (3) 服务进程 (Service process)，尽管服务进程与用户所见内容没有直接关联，但是它们通常在执行一些用户关心的操作（例如在后台播放音乐或从网络下载数据）。举例如下：
  - //正在运行通过 startService() 方法启动的服务，且不属于上述两个更高类别进程的进程
- (4) 后台进程 (Background process)，后台进程对用户体验没有直接影响，系统可能随时终止它们。通常会有很多后台进程在运行，因此它们会保存在 LRU 列表中，以确保包含用户最近查看的 Activity 所在进程最后一个被终止。举例如下：
  - //对用户不可见的 Activity 的进程（已调用 Activity 的 onStop() 方法）
- (5) 空进程 (Empty process)，保留这种进程的的唯一目的是用作缓存，以缩短下次在其中运行组件所需的启动时间。为使总体系统资源在进程缓存和底层内核缓存之间保持平衡，系统往往会终止这些进程。举例如下：
  - //不含任何活动应用组件的进程

## 1. Low Memory Killer

- Android 的 Low Memory Killer 基于 Linux 的 OOM 机制，Low Memory Killer 会根据进程的 adj 级别以及所占的内存，来决定是否杀掉该进程，adj 越大，占用内存越多，进程越容易被杀掉。
- 关于 adj 的分级，我们可以参考 ProcessList.java，这里的常量定义了 ADJ 的分级。

```
UNKNOWN_ADJ = 16
//级别最低级的进程，通常是被缓存的进程，但是系统也不清楚缓存的内容

CACHED_APP_MAX_ADJ = 15
//这是一个只托管不可见的活动的进程，因此可以在没有任何中断的情况下被杀死

CACHED_APP_MIN_ADJ = 9
//缓存进程，没有英文解释

SERVICE_B_ADJ = 8
//不活跃的服务，不像 adj=5 的服务那么活跃
//在 root 以后，有的系统优化大师会把所有服务都调成 adj=8 来达到内存优化的目的
//因为当所有人 adj 都比较高时，这样才能保证名正言顺的杀进程

PREVIOUS_APP_ADJ = 7
//被切换的进程，一般是用户前一个使用的进程。两个应用来回切换，那么前一个应用一般 adj 设置为 7

HOME_APP_ADJ = 6
//与主应用程序有交互的进程

SERVICE_ADJ = 5
//活跃的服务进程

HEAVY_WEIGHT_APP_ADJ = 4
//高权重进程

BACKUP_APP_ADJ = 3
//正在备份的进程

PERCEPTIBLE_APP_ADJ = 2
//可感知进程，通常是前台 Service 进程

VISIBLE_APP_ADJ = 1
//可见进程

FOREGROUND_APP_ADJ = 0
//前台进程
```

- 剩下的就是 adj 值为负数的进程，基本上都是系统集成，不在本文的讨论范围内。负数进程是不会被 lmk 杀掉的。
- Android 进程调度之 adj 算法
  - <http://gityuan.com/2016/08/07/android-adj/>

### 3.1.2 如何提高进程优先级

#### 1. 1 利用 Activity 提升权限

- 本方案针对的是某些工具或应用会在检测到锁屏事件后会杀死后台进程，（已达到省电的目的问题）。因此本方案可以监控手机锁屏解锁事件，在屏幕锁屏时启动 1 个像素的 Activity，在用户解锁时将 Activity 销毁掉。通过该方案，可以使进程的优先级在屏幕锁屏时间提升为最高优先级。
- 具体实现方案如下：
  - 首先定义 Activity，并设置 Activity 的大小为 1 像素：
  - 其次，从 AndroidManifest 中通过如下属性，避免 Activity 在 RecentTask 中的显示：
  - 最后，控制 Activity 为透明：
- Activity 启动与销毁时机的控制：

#### 2. 2 后台 Service 设置为前台 Service

- 通过 setForeground 可以将后台 Service 设置为前台 Service，使进程的优先级由 4 提升为 2，使进程被杀死的概率大大降低。
- 但是从 Android3 开始调用 setForeground 将后台 Service 设置为前台 Service 时，必须在系统的通知栏发送一条通知，也就是说前台 Service 与一条可见的通知是绑定在一起的。解决方案的话，可以增加一个内部 Service，在 Service 和它内部的 Service 中，同时发送具有相同 ID 的 Notification，然后再将这个内部 Service 结束掉。随着内部 Service 的结束，通知将会消失，但系统优先级依然保持为

## 3.2 进程死后拉活方案

### 3.2.1 利用系统广播拉活

- 在发生特定系统事件时，系统会发出响应的广播，通过在 Manifest 中静态注册对应的广播监听器，即可在发生响应事件时拉活。常用的用于拉活的广播事件包括：

### 3.2.2 利用系统 Service 机制拉活

- 在 service 的 onStartCommand 方法里返回 STATR\_STICK，系统会重新创建这个服务并且调用 onStartCommand() 方法，当然了它不会重新传递 Intent 对象。
- 但是这种方案有弊端：
  - （1）Service 第一次被异常杀死后会在 5 秒内重启，第二次被杀死会在 10 秒内重启，第三次会在 20 秒内重启，一旦在短时间内 Service 被杀死达到 5 次，则系统不再拉起。
  - （2）如果我们的进程被取得 Root 权限的管理工具强制停止（forestop）后也无法重启。

### 3.2.3 覆写 Service 的 onDestroy 方法

- 这种方案针对的是在设置里面，对正在运行的应用点击关闭，会走 onDestroy 方法，可以在这里可以把自己启动起来。
- 但是如果被取得 Root 权限的管理工具强制停止（forestop）的话，压根不会走到 onDestroy 这个方法。



### 3.2.4 利用 Native 守护进程拉活

- 原理就是，在 Android 中所有进程和系统组件的生命周期受 ActivityManagerService 的统一管理。但是通过 Linux 的 fork 机制创建的进程是 Native 进程，生命周期不受 Android 的管理。在 Native 守护进程中监控主进程的存活，当主进程挂掉后，在 Native 进程中立即对主进程进行拉活。
- 该方案主要适用于 Android5.0 以下版本的手机，并且不受强制停止的影响。
- 对于 Android5.0 以上手机，系统杀死进程的时候，不仅把主进程给杀死了，另外把主进程所属的进程组一并杀死，因此 native 进程可能会被杀死，但是这种情况不一定就拉不活。因为这其实是一个系统依次杀死进程的时间与拉活逻辑执行时间赛跑的问题，如果可以跑的比系统逻辑快，依然可以有效拉起。
- 记得网上有人做过实验，该结论在某些 Android 5.0 以上机型是成立的。该方案中的几个重要问题补充如下：

#### 1. Native 进程中如何感知主进程死亡

- (1) 在 Native 进程中通过死循环或定时器，轮询判断主进程是否存活，当主进程不存活时进行拉活。但是缺点也很明显，就是轮询判断逻辑会非常耗电。
- (2) 具体判断主进程存活的方案是，在主进程中创建一个监控文件，并且在主进程中持有文件锁。在拉活进程启动后就去申请这个文件锁，然后就将会被堵塞，一旦拉活进程可以成功获取到锁，说明主进程挂掉，即可进行拉活。
- 由于 Java 层的文件锁与 Linux 层的文件锁是不同的，需要封装 Linux 层的文件锁供上层调用。

#### 2. Native 进程中如何拉活主进程

- 在 Native 进程进行拉活的时候，本质上是通过 am 命令进行拉活。（通过指定 “--include-stopped-packages” 参数拉活主进程被 forestop 的情况）

### 3.2.5 利用 JobScheduler 机制拉活

- Android5.0 以上提供了一个叫 JobScheduler 的接口，可以把它理解成是一个操作系统层面的全局定时器，而不是像 Alarm 一样是以 APP 为界限的。它会定时唤醒进程，使应用进行一些逻辑操作。该方案在 Android5.0 以上版本中不受 forestop 影响，被强制停止的应用依然可以被拉活，在 Android5.0 以上版本拉活效果非常好。仅在小米手机可能会出现有时无法拉活的问题。

### 3.2.6 利用帐号同步机制拉活

- Android 系统里有一个帐号系统，系统会通过账户同步机制，定期进行帐号的同步，我们可以自己设定同步的事件间隔。当系统发起更新后进行进程的拉活。
- 这个方案适用于除 Android 7.0 以外所有的 Android 版本，包括被 forestop 掉的进程也可以进行拉活。
- 但是这个方案弊端太多了：
  - (1) 在 7.0 上系统对帐号同步做了一些变动。
  - (2) 从用户角度来说，用户会在系统设置里的账户列表上看到一个不认识的账户；而且某些手机比如 note3 需要手动设置账户。
  - (3) 从设备商角度看，同步的事件间隔是有限制的，最短 1 分钟，见源码，而且各种国产机怎么改的源码我们未知，是不是都能用仍然未知；

- (4) 从 Google 的角度来看, Google 提供这个机制是让用于同步账户信息, 必须联网, 不联网则无法同步。
- 添加账号和设置同步周期的代码如下:
- 该方案需要在 `AndroidManifest` 中定义账号授权与同步服务。

### 3.2.7 推送拉活

- 根据终端不同, 在小米手机 (包括 MIUI) 接入小米推送、华为手机接入华为推送, 可以作为拉活方案的有效补充。

### 3.2.8 终极武器——客户粘性

- 为什么 QQ、微信这些应用推送服务不会被杀死呢, 是因为手机厂商不让他死, 如果一个人买了个手机然后收不到 QQ 消息, 估计这款手机就卖不出去了。
- 这就是客户粘性带来的效益, 应用的客户粘性越高, 就越有可能出现在手机厂商进程保活的白名单里。

## 3.3 使用多进程的其他场景补充

- 多进程还有一种非常有用的场景, 就是多模块应用。比如我做的应用大而全, 里面肯定会有很多模块, 假如有地图模块、大图浏览、自定义 `WebView` 等等 (这些都是吃内存大户), 一个成熟的应用一定是多模块化的。首先多进程开发能为应用解决了 OOM 问题, 因为 `Android` 对内存的限制是针对进程的, 所以, 当我们需要加载大图之类的操作, 可以在新的进程中去执行, 避免主进程 OOM。而且假如图片浏览进程打开了一个过大的图片, `java heap` 申请内存失败, 该进程崩溃并不影响我主进程的使用。
- 举个例子, 现在要做一款音乐播放器, 现在有以下几种方案:
  - A. 在 `Activity` 中直接播放音乐。
  - B. 启动后台 `Service`, 播放音乐。
  - C. 启动前台 `Service`, 播放音乐。
  - D. 在新的进程中, 启动后台 `Service`, 播放音乐。
  - E. 在新的进程中, 启动前台 `Service`, 播放音乐。
- E 也是使用了多进程, 并且在新进程中, 使用了前台 `service`. 这个进程的值是 2, 像 C 方案, 非常小, 非常稳定, 而且我们还可以在系统进入后台后, 手动杀掉主进程, 使整个应用的内存消耗降到最低。内存低, 优先级又高, E 获得了今天的“最稳定方案奖”。

## 4 多进程应用的使用场景

- 可以让 `Android` 应用中的不同组件运行在不同的进程。有时这种做法是必要的, 可以改进应用性能。但你必须意识到, 官方网站上并没有太多关于 `Android` 多进程应用的文档, 而且多进程应用也不容易理解。我们看看何时它会很有用, 有何挑战以及如何解决。
- 缺省时所有 `Android` 应用组件都运行在相同的 `Linux` 进程中。但也可以让不同组件运行在不同的进程。许多场景下多进程很有用。我们来看看。
  - `Android` 会不时杀死进程回收内存, 以提供给更重要的进程。`Android` 使用重要性等级来决定该杀死和保留哪个进程。占用内存高的后台进程更可能被 `Android` 杀死。如果你将后台组件 (`service`, `content provider`) 和前台组件 (`activity`) 放在同一个进程, 该进程会使用更多的内存。当应用进入后台, 它更可能被杀死。`Android` 杀死进程时后杀死其中所有的组件。为了将后台组件跟 UI 的生命周期解耦, 最好将其独立到一个独立的进程。

- 比如，如果应用需要在客户端和服务端之间同步数据时，它可以在独立进程中的 `service` 中进行同步。这可以让后台进程更少重启，让其独立于 UI 进程的生命周期。另外，UI 进程中的崩溃和异常对后台进程没有任何影响。反之亦然。
- 使用多进程的另一个原因是多个特性完全不同且独立。比如，如果应用支持 `Email`, `Calendar`, `Contacts`, `Notes` 以及 `Tasks`，其中每个特性可以在独立进程中运行以保持各自的生命周期独立。

## 5 如何运行多进程

- `<activity>`, `<service>`, `<receiver>` 以及 `<provider>` 都支持 `<android:process>` 属性，该属性可以指定组件运行的进程。多个组件也可以共享同一个进程，指定多少进程就能启动多少进程。

### 5.1 挑战

- 当 Android 应用使用多进程时，需要处理进程间的数据一致性。甚至你的代码结构良好，在运行时也不大容易知道哪段代码在哪个进程运行。

### 5.2 共享数据

- 尽管可以在多个独立进程中运行不同独立逻辑，某些时候它们仍然需要共享数据。如果你使用文件，数据库或者 `shared preferences`，你可能遇到跨进程数据不一致问题。如果使用文件，你需要监听其他进程对文件进行的修改。如果从多个进程访问数据库，你可能会遇到非常难以调试的数据库冲突或死锁问题。如果使用 `shared preferences`，必须使用多进程模式打开它，并且使用 `OnSharedPreferenceChangeListener` 监听其他进程导致的变化。

### 5.3 单例

- 单例是在 Android 应用中跨组件共享状态信息和数据的简单办法。可以使用 `synchronized` 关键字保证单例的方法是线程安全的。但如果在多进程应用中使用单例，可能生成的单例对象数量跟进程数量一样多。因为进程并不共享地址空间，所以一个进程中的单例对象对其他进程并不可见。如果使用 `shared preferences`，数据库或文件中的数据初始化单例，要让 (不同进程中的) 每个单例对象保持一致的数据非常困难，而且很可能它们在运行时有不同的数据状态。另外，`synchronized` 对这种情况不起作用。看看代码中单例的通常用法。
- 假设我们有一个单例类，不妨称之为 `SubscriptionManager`，它会跟踪用户订阅。在这个类中更新订阅内容。同时你将数据保存到文件以供离线访问。如果数据文件存在的话，由这些文件初始化单例。有两个进程：主进程和后台进程。在一个工具类中检查用户是否订阅了某个内容，该类会被主进程和后台进程调用。用户调用 `SubscriptionManager.getInstance(context).updateSubscription(feature, subscription)`；从主进程订阅。这个调用更新了订阅内容，发送到后台并且更新了文件。
- 在这种情况下，后台进程中的 `SubscriptionManager` 并不知道主进程中发生的这次订阅，除非你在数据文件中设置了 `FileObserver`，监听数据变化更进行相应更新。或者，你也可以使用广播机制来通知这次订阅。所以说，使用这些方式在进程间共享数据非常复杂，并不是最佳方案。

### 5.4 解决办法

- Android 通过 `Binder` 接口提供进程间通信 (IPC)。Content providers 和 Bound services 使用 `binder` 接口来跨进程通信。所以本方案使用这种方式来维护多进程间数据和状态的一致性。

### 5.4.1 Content Provider

- **ContentProvider** 用于管理对结构化数据的访问。它们封装数据并且提供数据安全机制。**ContentProvider** 是在当前进程的数据跟其他进程的代码之间建立联系的标准接口。(Content providers are the standard interface that connects data in one process with code running in another process)
- 尽管 **ContentProvider** 本是用于应用间共享数据，它也可以用于在多进程之间共享数据。**Android** 保证跨进程时 **ContentProvider** 的单一性。**ContentResolver** 提供易用的接口，所以应用代码不必担心 **IPC** 细节。
- 注意：如果不想在应用外共享数据，不要在 **manifest** 中对外暴露 **ContentProvider**(**android:export**="fa
- 尽管 **ContentProvider** 和 **ContentResolver** 的”**CURD**” **API** 主要被设计为共享 **SQL** 数据库中的数据，也可以扩展更通用的 **call(android.net.Uri, java.lang.String, java.lang.String, android.os.Bundle)** **API** 以实现应用特定的目的。
- 比如：在上述例子的 **SubscriptionManager**，你可以提供如下 **API** 来检查和更新订阅：

```
ContentResolver resolver = getContentResolver();  
Bundle result = resolver.call("SubscriptionCheck", arg, extras);
```

### 5.4.2 Bound Service

- **Bound service** 是客户端-服务器接口中的服务器端。**Bound service** 允许组件 (比如 **activity**) 绑定到 **service**，发送请求，接收响应，甚至执行 **IPC**
- 有两种方式实现 **IPC**：**Messengers** 和 **AIDL**。**Messenger** 方式更简单一些。两种方式中 **Android** 框架都做好了 **IPC** 底层工作 (**marshalling**, **unmarshalling**, **RPC**)，**IPC** 对调用方是透明的。跟 **service** 通信的步骤如下：
- **bindService**
- 连接到 **service** 时收到回调
- 使用 **binder** 接口发送请求，或调用 **service API**
- 从 **service** 接收响应
- **Service**
- 可以使用 **intent** 在另一个进程中启动 **service**，或者从代码中的任何地方发送广播，并且在另外的进程中接收该广播并启动一个 **service** 来处理它。
- 比如你有一个测量系统用于捕获应用中的某些事件，可以将每次测量发送到 **service**，让 **service** 将其写入数据库或发送到后台，而不是让代码中到处都是数据库或网络操作。
- **Application** 类的坑
- 你可以实现 **Application** 的子类，并且在 **AndroidManifest.xml** 的 **<application>** 标签的 **<android:name>** 属性中指定这个类的全名。当应用进程启动时，你实现的 **Application** 子类会最先初始化。
- 要注意的是，应用中启动的每一个进程都会执行上述初始化。不幸的是，并不能为不同的进程指定不同的 **Application** 类。
- 如果你使用 **Application** 类，很可能你会在它的 **onCreate()** 方法中做一些初始化工作。请确保这些初始化工作对进程是恰当且必要的也在 **Application.onCreate()** 中使用下面的代码确定进程上下文：

```

int pid = android.os.Process.myPid();
ActivityManager manager = (ActivityManager) this.getSystemService(Context.ACTIVITY_SERVICE);
for (ActivityManager.RunningAppProcessInfo processInfo : manager.getRunningAppProcesses()) {
    if (processInfo.pid == pid) {
        String currentProcName = processInfo.processName;
        if (!TextUtils.isEmpty(currentProcName) && currentProcName.equals(":background")) {
            //Rest of the initializations are not needed for the background
            //process
            return;
        }
    }
}
}
/* Initializations for the UI process */

```

- 通过这种方式你可以降低内存占用，避免某些大块内存分配，甚至有可能加快应用启动速度。这里提到初始化包括：加载专有字体，初始化 Google Maps，及以初始化第三方 SDK。通常只需要在主进程中做这些初始化。

## 6 Serializable 和 Parcelable 两种序列化

- 什么时候用的到序列化？
  - 1、永久性保存对象，保存对象的字节序列到本地文件中
  - 2、通过序列化对象在网络中传递对象
  - 3、通过序列化在进程之间传递对象
- 在安卓中，可以用作在 Activity 中传递复杂类型，主要是将自己的类转换为基础的字节数组，Android 中 Activity 传递主要靠 Intent 这个类，Intent 类内部传递数据的效率仍然需要拷贝，如果数据太大，其实可以考虑用单例模式，定义一个全局的静态类，活用设计模式。
- Parcelable

```

putExtra(String name, Parcelable value) // 传递一个 parcelable 对象
putExtra(String name, Parcelable[] value) // 传递一个 Parcelable 数组
putExtra(String name, Bundle value) // 传递一个 Bundle 对象
putParcelableArrayListExtra(String name, ArrayList<? extends Parcelable> value) // 传递一个实现 Parcelable 接口的对象数组
putStringArrayListExtra(String name, ArrayList<String> value) // 传递一个字符串数组
putIntegerArrayListExtra(String name, ArrayList<Integer> value) // 传递一个整形数组。

```

- Integer 这些装箱类型，Intent 还实现了 putExtra(String name, int[] value) 这样的基础类型数组，同时 Parcelable 效率比标准的 Java 序列化更高，同时还被用作在 AIDL 中交换数据，所以掌握 Parcelable 这样的显示序列化技术，可以大幅提高数据交换性能。
- Serializable
- 传统的 Java，串行化技术，实现 Serializable 接口，原理和 Parceble 差不多，均将自己的类转换为基本的类型比如说字节数组，在 Android 中不是很实用，比 Parcelable 效率低，在 Intent 中使用 Intent.putExtra(String name, Serializable value) 这个方法即可。

## 7 Android Service aidl 分析

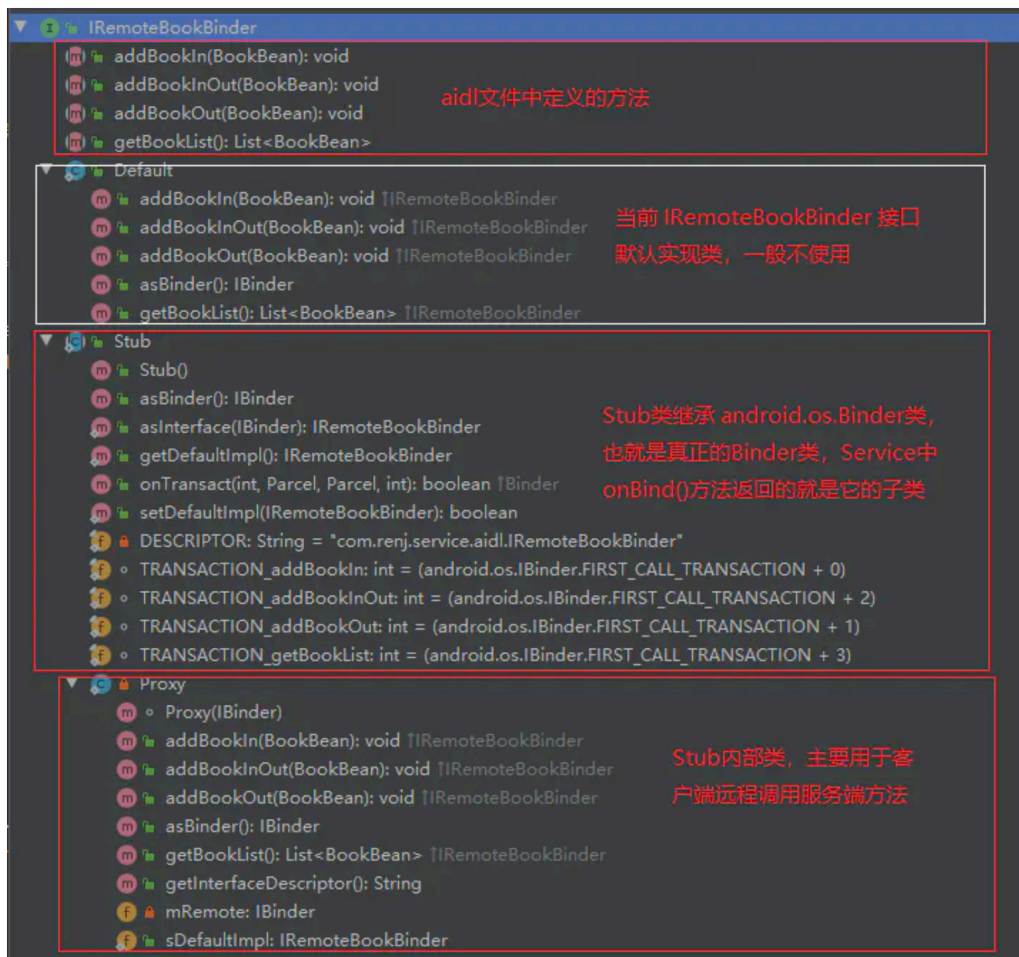
- <https://www.jianshu.com/p/90889446fb4d>

### 7.1 IRemoteBookBinder.aidl 文件来做分析

```

interface IRemoteBookBinder {
    void addBookIn(in BookBean bookBean);
    void addBookOut(out BookBean bookBean);
    void addBookInOut(inout BookBean bookBean);
    List<BookBean> getBookList();
}

```



## 7.2 内部类 Default

```
public static class Default implements com.renj.service.aidl.IRemoteBookBinder {
    @Override
    public void addBookIn(com.renj.service.bean.BookBean bookBean) throws android.os.RemoteException {}
    @Override
    public void addBookOut(com.renj.service.bean.BookBean bookBean) throws android.os.RemoteException {}
    @Override
    public void addBookInOut(com.renj.service.bean.BookBean bookBean) throws android.os.RemoteException {}
    @Override
    public java.util.List<com.renj.service.bean.BookBean> getBookList() throws android.os.RemoteException {
        return null;
    }
    @Override
    public android.os.IBinder asBinder() {
        return null;
    }
}
```

- 该类实现了 IRemoteBookBinder 类，作为 IRemoteBookBinder 接口的默认实现，在实际中并无太多实际意义。

## 7.3 IRemoteBookBinder.aidl 文件中声明的方法

```
public void addBookIn(com.renj.service.bean.BookBean bookBean) throws android.os.RemoteException;
public void addBookOut(com.renj.service.bean.BookBean bookBean) throws android.os.RemoteException;
public void addBookInOut(com.renj.service.bean.BookBean bookBean) throws android.os.RemoteException;
public java.util.List<com.renj.service.bean.BookBean> getBookList() throws android.os.RemoteException;
```



- 这几个方法都是在 `IRemoteBookBinder.aidl` 进行了声明的，直接作为接口的声明方法，具体有子类实现。

## 7.4 内部类 Stub

```
public static abstract class Stub extends android.os.Binder implements com.renj.service.aidl.IRemoteBookBinder {
    private static final java.lang.String DESCRIPTOR = "com.renj.service.aidl.IRemoteBookBinder";

    public Stub() {
        this.attachInterface(this, DESCRIPTOR);
    }
    public static com.renj.service.aidl.IRemoteBookBinder asInterface(android.os.IBinder obj) {
        if ((obj == null))
            return null;
        android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
        if (((iin != null) && (iin instanceof com.renj.service.aidl.IRemoteBookBinder)))
            return ((com.renj.service.aidl.IRemoteBookBinder) iin);
        return new com.renj.service.aidl.IRemoteBookBinder.Stub.Proxy(obj);
    }
    @Override
    public android.os.IBinder asBinder() {
        return this;
    }
    @Override
    public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags) throws android.os.R
        java.lang.String descriptor = DESCRIPTOR;
        switch (code) {
            case TRANSACTION_addBookIn: {
                data.enforceInterface(descriptor);
                com.renj.service.bean.BookBean _arg0;
                if ((0 != data.readInt())) {
                    _arg0 = com.renj.service.bean.BookBean.CREATOR.createFromParcel(data);
                } else {
                    _arg0 = null;
                }
                this.addBookIn(_arg0);
                reply.writeNoException();
                return true;
            }
            case TRANSACTION_addBookOut: {
                data.enforceInterface(descriptor);
                com.renj.service.bean.BookBean _arg0;
                _arg0 = new com.renj.service.bean.BookBean();
                this.addBookOut(_arg0);
                reply.writeNoException();
                if ((_arg0 != null)) {
                    reply.writeInt(1);
                    _arg0.writeToParcel(reply, android.os.Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
                } else {
                    reply.writeInt(0);
                }
                return true;
            }
            case TRANSACTION_addBookInOut: {
                // ...
                return true;
            }
            case TRANSACTION_getBookList: {
                data.enforceInterface(descriptor);
                java.util.List<com.renj.service.bean.BookBean> _result = this.getBookList();
                reply.writeNoException();
                reply.writeTypedList(_result);
                return true;
            }
            default: {
                return super.onTransact(code, data, reply, flags);
            }
        }
    }
    private static class Proxy implements com.renj.service.aidl.IRemoteBookBinder {
        // ...
    }
    static final int TRANSACTION_addBookIn = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
    static final int TRANSACTION_addBookOut = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);
    static final int TRANSACTION_addBookInOut = (android.os.IBinder.FIRST_CALL_TRANSACTION + 2);
    static final int TRANSACTION_getBookList = (android.os.IBinder.FIRST_CALL_TRANSACTION + 3);
    public static boolean setDefaultImpl(com.renj.service.aidl.IRemoteBookBinder impl) {
```

```

        if (Stub.Proxy.sDefaultImpl == null && impl != null) {
            Stub.Proxy.sDefaultImpl = impl;
            return true;
        }
        return false;
    }
    public static com.renj.service.aidl.IRemoteBookBinder getDefaultImpl() {
        return Stub.Proxy.sDefaultImpl;
    }
}

```

- 以上代码,去掉了 Proxy 内部类的实现,在后面再说,然后就是去掉了 TRANSACTION\_addBookInOut 部分的实现,因为这部分就是 TRANSACTION\_addBookIn 和 TRANSACTION\_addBookOut 的结合。
- Stub 类继承了 android.os.Binder 并且实现了 IRemoteBookBinder 接口,她就是真正的 Binder 类,是在 Service 中的 onBind() 方法返回给客户端的 Binder 类(一般我们在 onBind() 方法中返回的都是 Stub 的子类,如 RemoteBookBinderImpl extends IRemoteBookBinder.Stub),也是客户端 ServiceConnection 回调方法 onServiceConnected() 中的参数 service

```

private ServiceConnection bindConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        iRemoteBookBinder = IRemoteBookBinder.Stub.asInterface(service);
    }
    @Override
    public void onServiceDisconnected(ComponentName name) {
    }
};

```

#### 7.4.1 DESCRIPTOR

- Binder 的唯一标识,一般使用当前类的全路径名表示,如上面的"com.renj.service.aidl.IRemoteBookBinder"

#### 7.4.2 构造方法

- 将本身对象 (this) 和唯一标识 (DESCRIPTOR) 通过 attachInterface() 传递给父类 Binder 类,这里需要注意,后面会有地方用到。

```

public void attachInterface(@Nullable IInterface owner, @Nullable String descriptor) {
    mOwner = owner;
    mDescriptor = descriptor;
}

```

#### 7.4.3 asInterface(android.os.IBinder obj)

- 将服务端的 Binder 对象转换成客户端所需要的 AIDL 接口对象, ServiceConnection 回调方法 onServiceConnected() 中就是使用这个方法对 service 参数进行转换。这个方法内部的转换是区分进程的,如果客户端和服务端在同一个进程,那么返回的就是服务端的 Stub 对象本身,否则返回的就是系统封装后的 IRemoteBookBinder.Stub.Proxy 类。注意创建 IRemoteBookBinder.Stub.Proxy(obj) 对象时,参数 obj,就是 onBind() 方法返回的 Binder 对象,也就是回调 onServiceConnected() 方法中的参数 service。
- 在 asInterface() 方法内部是通过 android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR) 方法判断客户端和服务端是否在同一进程的, queryLocalInterface() 方法在 Binder 中的实现:

```

public @Nullable IInterface queryLocalInterface(@NonNull String descriptor) {
    if (mDescriptor != null && mDescriptor.equals(descriptor))
        return mOwner;
    return null;
}

```

- 上述方法中的 mDescriptor 和 mOwner 就是通过 Stub 的构造方法设置的值。

#### 7.4.4 asBinder() 方法

- 该方法返回当前的类对象，也就是 Binder

#### 7.4.5 方法 id

```
static final int TRANSACTION_addBookIn = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
static final int TRANSACTION_addBookOut = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);
static final int TRANSACTION_addBookInOut = (android.os.IBinder.FIRST_CALL_TRANSACTION + 2);
static final int TRANSACTION_getBookList = (android.os.IBinder.FIRST_CALL_TRANSACTION + 3);
```

- 这里就是 aidl 接口中的方法，为他们都声明一个 id，用于在 onTransact() 方法中确定调用的是哪个方法。

#### 7.4.6 onTransact() 方法

- 该方法运行在服务端的 Binder 线程池中，当客户端跨进程调用服务器方法时，远程请求会通过系统的封装 (Proxy) 来调用该方法，然后由该方法根据方法 id 调用对应的方法，该方法的声明为：

```
boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags)
```

- code: 调用的方法 id
- data: 传递进来的数据
- reply: 需要返回的数据
- flags: 额外标记，正常传 0(1: 单向 RPC, 无返回值)
- 返回值: Boolean 类型, true 表示调用成功, false 表示调用失败, 可以在这个方法内做权限控制。

#### 7.4.7 关键字 in, out, inout

- 我们在《Android Service aidl 使用及进阶》说过，非基本数据类型的数据的走向标记 in、out、inout 的作用，这里再次进行说明一下，因为以下内容需要明白他们的作用才能更好的理解。
  - in: 服务端只能接收输入，不能输出给客户端。当客户端传递 BookBean 给服务端时，服务端能获取到 BookBean 对象信息，但是服务端进行修改之后，客户端不会发生改变，还是原来的。
  - out: 服务端不能接收输入，只能输出给客户端。当客户端传递 BookBean 给服务端时，服务端获取到的 BookBean 对象之后，并不能获取 BookBean 里面的字段信息 (都是默认值)，但是服务端修改了这些值后，客户端的信息会发生改变，变成服务端修改之后的值。
  - inout: 服务端既能接收输入，也能输出给客户端。当客户端传递 BookBean 给服务端时，服务端能获取到 BookBean 对象信息；服务端修改了这些值后，客户端的信息也会发生改变，变成服务端修改之后的值。

- 下面对 onTransact() 内对方法 addBookIn() 和 addBookOut() 方法具体说明：

##### 1. TRANSACTION\_addBookIn

- 表示调用的是 addBookIn() 方法，该方法在 IRemoteBookBinder.aidl 中的定义是 void addBookIn(in BookBean bookBean);，注意关键字 in；

```
data.enforceInterface(descriptor); // 服务端标识，这个数据该由谁处理
com.renj.service.bean.BookBean _arg0; // 声明一个新的对象
if ((0 != data.readInt())) { // 判断 data 是否有数据需要读取
    // 根据传递进来的 data 创建一个新的对象并赋值给 _arg0
    _arg0 = com.renj.service.bean.BookBean.CREATOR.createFromParcel(data);
} else {
    _arg0 = null;
}
this.addBookIn(_arg0); // 调用我们定义方法，具体由子类实现，将新的对象 _arg0 作为参数传递给方法
reply.writeNoException(); // 没有发生异常
return true; // 返回成功
```

- 当确定调用的 `addBookIn()` 方法，先将 `Binder` 的唯一标识对数据标记，然后服务器创建一个新的对象，因为使用了 `in` 关键字，所以需要读取老的数据，然后调用抽象 `addBookIn()` 方法（由子类实现），并且将新建的对象作为参数传递，并没有将新的数据写入到 `reply` 中，因为没有 `out` 关键字。

## 2. TRANSACTION\_addBookOut

- 表示调用的是 `addBookOut()` 方法，该方法在 `IRemoteBookBinder.aidl` 中的定义是 `void addBookOut(out BookBean bookBean);`，注意关键字 `out`；

```
data.enforceInterface(descriptor); // 服务端标识，这个数据该由谁处理
com.renj.service.bean.BookBean _arg0; // 声明一个新的对象
_arg0 = new com.renj.service.bean.BookBean(); // 因为只有 out 关键字，所以不会传递老的对象中的数据
this.addBookOut(_arg0); // 调用我们定义的方法，传递新的对象，所有字段都是默认值
reply.writeNoException(); // 没有发生异常
if ((_arg0 != null)) {
    reply.writeInt(1); // 当数据发送变化后，修改标记
    // 把变化后的数据写入 reply 中
    _arg0.writeToParcel(reply, android.os.Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
} else {
    reply.writeInt(0);
}
return true; // 返回成功
```

- 当确定调用的 `addBookOut()` 方法，先将 `Binder` 的唯一标识对数据标记，然后服务器创建一个新的对象，因为使用了 `out` 关键字，所以只会创建一个包含默认值的东西，而不会读取老的数据，然后调用抽象 `addBookOut()` 方法（由子类实现），并且将新建的对象作为参数传递，方法调用完成后，就修改的数据写入 `reply` 中。
- 对于 `IRemoteBookBinder.aidl` 中的定义的 `addBookInOut()` 方法，也差不多。只是有 `in` 和 `out` 两个关键字，所以既能传入原始数据，也会返回修改后的数据。

## 3. TRANSACTION\_getBookList

- 表示调用的是 `getBookList()` 方法，该方法在 `IRemoteBookBinder.aidl` 中的定义是 `List<BookBean> getBookList();`

```
data.enforceInterface(descriptor);
java.util.List<com.renj.service.bean.BookBean> _result = this.getBookList();
reply.writeNoException();
reply.writeTypedList(_result);
```

- 这个方法直接调用抽象 `getBookList()` 方法（由子类实现），然后将结果写入到 `reply` 中返回。
- `setDefaultImpl()` 和 `getDefaultImpl()` 方法
- `setDefaultImpl()` 方法是设置默认 `aidl` 接口实现，当远程调用 `transact()` 方法（实际最终调用的就是内部类 `Stub` 的 `onTransact()` 方法）中的对应方法，返回结果为 `false`，并且通过 `getDefaultImpl()` 方法获取到的默认实现不为 `null` 就会调用默认实现的对应方法。

## 7.5 内部类 Stub 的内部类 Proxy

```
private static class Proxy implements com.renj.service.aidl.IRemoteBookBinder {
    private android.os.IBinder mRemote;
    Proxy(android.os.IBinder remote) {
        mRemote = remote;
    }
    @Override
    public android.os.IBinder asBinder() {
        return mRemote;
    }
    public java.lang.String getInterfaceDescriptor() {
        return DESCRIPTOR;
    }
    @Override
    public void addBookInOut(com.renj.service.bean.BookBean bookBean) throws android.os.RemoteException {
```

```

        android.os.Parcel _data = android.os.Parcel.obtain();
        android.os.Parcel _reply = android.os.Parcel.obtain();
        try {
            _data.writeInterfaceToken(DESCRIPTOR);
            if ((bookBean != null)) {
                _data.writeInt(1);
                bookBean.writeToParcel(_data, 0);
            } else {
                _data.writeInt(0);
            }
            boolean _status = mRemote.transact(Stub.TRANSACTION_addBookInOut, _data, _reply, 0);
            if (!_status && getDefaultImpl() != null) {
                getDefaultImpl().addBookInOut(bookBean);
                return;
            }
            _reply.readException();
            if ((0 != _reply.readInt())) {
                bookBean.readFromParcel(_reply);
            }
        } finally {
            _reply.recycle();
            _data.recycle();
        }
    }
}
@Override
public java.util.List<com.renj.service.bean.BookBean> getBookList() throws android.os.RemoteException {
    android.os.Parcel _data = android.os.Parcel.obtain();
    android.os.Parcel _reply = android.os.Parcel.obtain();
    java.util.List<com.renj.service.bean.BookBean> _result;
    try {
        _data.writeInterfaceToken(DESCRIPTOR);
        boolean _status = mRemote.transact(Stub.TRANSACTION_getBookList, _data, _reply, 0);
        if (!_status && getDefaultImpl() != null) {
            return getDefaultImpl().getBookList();
        }
        _reply.readException();
        _result = _reply.createTypedArrayList(com.renj.service.bean.BookBean.CREATOR);
    } finally {
        _reply.recycle();
        _data.recycle();
    }
    return _result;
}
public static com.renj.service.aidl.IRemoteBookBinder sDefaultImpl;
}

```

首先使用 Proxy 的情况是因为我们在 Stub 的 asInterface() 方法中判断了客户端和服务端是不是在同一个进程，不是在同一个进程时，才返回 Proxy 对象。

### 7.5.1 构造方法

- 在解析 asInterface() 方法时，就说明了 Proxy 的构造方法需要一个参数 obj，就是 onBind() 方法返回的 Binder 对象，也就是回调 onServiceConnected() 方法中的参数 service。

#### 1. asBinder() 和 getInterfaceDescriptor()

asBinder() 方法返回通过构造方法传递进来的 Binder 对象  
getInterfaceDescriptor() 方法返回 Stub 的 DESCRIPTOR 字段，Binder 的唯一标识。

### 7.5.2 方法调用

- 就是当我们在客户端通过 service 调用方法时，实际上调用的就是 Proxy 中的方法，理由通过前面的分析我们已经知道了（对 asInterface(android.os.IBinder obj) 方法分析）。
- 对于 Stub 类，我们分析了 addBookIn() 和 addBookOut() 方法，那么在 Proxy 类中，我们分析一下 addBookInOut() 方法。

#### 1. void addBookInOut(com.renj.service.bean.BookBean bookBean) 方法

```

android.os.Parcel _data = android.os.Parcel.obtain(); // 创建参数对象
android.os.Parcel _reply = android.os.Parcel.obtain(); // 创建返回值对象
try {
    _data.writeInterfaceToken(DESCRIPTOR); // 唯一标识
    if ((bookBean != null)) { // 参数不为 null 时, 将客户端传递的参数数据写到新建对象中 (因为关键字为 inout)
        _data.writeInt(1);
        bookBean.writeToParcel(_data, 0);
    } else {
        _data.writeInt(0);
    }
    // 通过 mRemote 调用 transact() 方法, 传递调用方法 id (Stub 中定义的方法 id), 输入输出对象和标志位,
    // 这些参数就时 Stub#onTransact() 方法所接收的参数
    boolean _status = mRemote.transact(Stub.TRANSACTION_addBookInOut, _data, _reply, 0);
    // Stub#onTransact() 方法返回失败, 并且设置了默认 aidl 接口实现, 就调用默认的实现
    if (!_status && getDefaultImpl() != null) {
        getDefaultImpl().addBookInOut(bookBean);
        return;
    }
    _reply.readException();
    // 因为是有 inout 关键字, 所以需要将修改后的结果重写更新到客户端的对象中
    if ((0 != _reply.readInt())) {
        bookBean.readFromParcel(_reply);
    }
} finally {
    _reply.recycle();
    _data.recycle();
}

```

- 上面的代码中, 已经对代码做了注释了, 我们发现在 Proxy 中的方法内, 其实就是对参数和结果数据的处理, 并没有其他的逻辑, 然后直接调用了 mRemote 的 transact() 方法, mRemote 就是 Binder, 这个我们是知道的, 那看看 Binder#transact() 方法做了什么操作了

```

public final boolean transact(int code, @NonNull Parcel data, @Nullable Parcel reply,
    int flags) throws RemoteException {
    if (false) Log.v("Binder", "Transact: " + code + " to " + this);
    if (data != null) {
        data.setDataPosition(0);
    }
    // 调用了 onTransact 方法
    boolean r = onTransact(code, data, reply, flags);
    if (reply != null) {
        reply.setDataPosition(0);
    }
    return r;
}

```

- 在 Binder#transact() 中调用了 onTransact() 方法, 这个方法也就是我们刚刚在分析 Stub 类时分析的方法。到了这里, 我们就已经知道了客户端在跨进程调用服务端方法时的流程, 以及 in、out 和 inout 关键字作用的原理了。

## 2. java.util.List<com.renj.service.bean.BookBean> getBookList() 方法

```

android.os.Parcel _data = android.os.Parcel.obtain();
android.os.Parcel _reply = android.os.Parcel.obtain();
java.util.List<com.renj.service.bean.BookBean> _result;
try {
    _data.writeInterfaceToken(DESCRIPTOR);
    boolean _status = mRemote.transact(Stub.TRANSACTION_getBookList, _data, _reply, 0);
    if (!_status && getDefaultImpl() != null) {
        return getDefaultImpl().getBookList();
    }
    _reply.readException();
    // 将返回 _reply 对象封装成客户端需要的类型
    _result = _reply.createTypedArrayList(com.renj.service.bean.BookBean.CREATOR);
} finally {
    _reply.recycle();
    _data.recycle();
}
return _result;

```

- 和 addBookInOut() 方法类似, 封装参数, 然后调用 mRemote 的 transact() 方法, 最后将结果 \_reply 对象封装成客户端需要的类型返回。



# 7.6 总结图

