

# 安卓 IPC AIDL Service 用法及原理详解

deepwaterooo

July 16, 2022

## Contents

<b>1 基本过程概述：一般实现步骤讲解</b>	<b>1</b>
1.1 在客户端 (Activity 中) 要完成: . . . . .	1
1.2 在服务端一般要实现: . . . . .	1
<b>2 abstract class IntentService extends Service 源码通讯简单注释</b>	<b>2</b>
<b>3 安卓 Service 知识细节二 - 绑定服务和 AIDL</b>	<b>3</b>
3.1 一：生命周期 . . . . .	4
3.2 二：IPC . . . . .	6
3.3 明白了 AIDL 的作用，和 IPC 的基本概念之后，我们来看一下 AIDL 的使用。 . . . . .	6

## 1 基本过程概述：一般实现步骤讲解

- 学习新知识的科学方法是怎样的呢?
  - 我觉得只读源码，然后自己去动脑筋去想，我应该是多花点儿时间，应该也是自己能够最终想得明白的；
  - 但是与其用读很多源码的经验来一再巩固自己的识知：想要 bind 到远程服务，就必须得实现 ServiceConnection 接口，不如自己磨刀不误砍柴工，先化一两个小时的时间，从网络上先把原理弄明白，清楚了，先有一个大概的认知，再去读这些 IPC AIDL IBinder 的远程绑定，是不是能够轻松愉快很多呢？嗯，这是我今天想要尝试的科学学习方法。。。。。

### 1.1 在客户端 (Activity 中) 要完成：

- 1. 客户端通过 BindService() 方法来绑定一个服务对象（业务对象，应用是 IBinder 对象的具体实现类吧，但是 IBinder 对象）
  - 如绑定成功会回调 ServiceConnection 接口方法 onServiceConnected()
- 2. onServiceConnected() 方法的其中一个参数是在 Service 中 OnBind() 返回的 Binder 的实例（IBinder 对象）。
- 3. 通过在 onServiceConnected() 方法中接受 Binder 的实例（IBinder 对象）来调用 Binder 中返回 Service 实例（也就是 IBinder 实例中实现过）的方法，获得 Service 的实现。
- 4. 通过 Service 的实例就可以调用 Service 的公有方法。

## 1.2 在服务端一般要实现:

- 1. 服务端通过创建一个 \*.aidl 文件来定义一个可以被客户端调用的业务接口
  - 一个 AIDL 文件的规范:
    - \* 1> 不能有修饰符, 类似接口的写法
    - \* 2> 支持数据类型, String(存放字符串)\ 自定义类型
  - 自定义类型:
    - \* (要实现 Parcelable 接口, 定义一个 AIDL 文件声明该类型, 在其他 AIDL 中使用该类型需要 import 包)
- 2. 服务端需要提供一个业务接口的实现类, 通常继承 Stub 类
- 3. 通过 Service 的 onBind() 方法返回被绑定的业务对象

## 2 abstract class IntentService extends Service 源码通讯简单注释

```
/*
 * IntentService 是一种特殊的 Service, 它继承于 Service 并且还是个抽象类
 * 所以我们必须创建它的子类才能使用 IntentService
 * IntentService 可以用来执行后台任务, 当任务执行完后就会“自杀”
 * 因为它自己也是个服务, 所以优先级高于普通的线程, 不容易被系统所干掉
 * 所以 IntentService 比较适合执行优先级较高的后台任务
 */
public abstract class IntentService extends Service {
    // HandlerThread 的 looper
    private volatile Looper mServiceLooper;
    // 通过 looper 创建的一个 Handler 对象, 用于处理消息
    private volatile ServiceHandler mServiceHandler;
    private String mName;
    private boolean mRedelivery;
    /**
     * 通过 HandlerThread 线程中的 looper 对象构造的一个 Handler 对象
     * 可以看到这个 handler 中主要就是处理消息
     * 并且将我们的 onHandlerIntent 方法回调出去
     * 然后停止任务, 销毁自己
     */
    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }
        @Override
        public void handleMessage(Message msg) {
            /**
             * 收到消息后, 会将 Intent 对象传递给 onHandlerIntent 方法去处理
             * 注意这个 Intent 对象的内容和外界 Activity 中 startService(intent)
             * 中的 intent 的内容是完全一致的
             * 通过这个 intent 对象可以得到外界启动 IntentService 时所传递的参数
             * 通过这些参数我们就可以区分不同的业务逻辑
             * 这样 onHandlerIntent 就可以对不同的逻辑做出不同的操作了
             * 当 onHandlerIntent 方法执行结束后, IntentService 会通过
             * stopSelf(int startId) 方法尝试停止服务
             *之所以使用 stopSelf(int startId) 而不是 stopSelf() 来停止
             *是因为 stopSelf() 会马上停止服务, 但是有可能还有消息未处理
             * stopSelf(int startId) 则会等所有的消息都处理完后才销毁自己
             *一般来说的话, stopSelf(int startId) 在停止之前会判断
             *最近启动服务的次数和 startId 是不是相等的, 如果相等就立刻停止
             *如果不相等说明还有别的消息没处理, 就不停止服务
             *具体的要看 AMS 中的 stopServiceToken 方法的实现
             */
            onHandleIntent((Intent)msg.obj);
            stopSelf(msg.arg1);
        }
    }
}
```

```

* 构造函数，可以传递一个 name 参数
*/
public IntentService(String name) {
    super();
    mName = name;
}
public void setIntentRedelivery(boolean enabled) {
    mRedelivery = enabled;
}
/**
 * 当我们的 IntentService 第一次启动的时候，onCreate 方法会执行一次
 * 可以看到方法里创建了一个 HandlerThread
 * HandlerThread 继承 Thread，它是一种可以使用 Handler 的 Thread
 * 它的 run 方法里通过 Looper.prepare() 来创建消息队列
 * 并通过 Looper.loop() 来开启消息循环，所以就可以在其中创建 Handler 了
 * 这里我们通过 HandlerThread 得到一个 looper 对象
 * 并且使用它的 looper 对象来构造一个 Handler 对象，就是我们上面看到的那个
 * 这样做的好处就是通过 mServiceHandler 发出的消息都是在 HandlerThread 中执行
 * 所以从这个角度来看，IntentService 是可以执行后台任务的
*/
@Override
    public void onCreate() {
        super.onCreate();
        HandlerThread thread = new HandlerThread("IntentService[" + mName + "]");
        thread.start();
        mServiceLooper = thread.getLooper();
        mServiceHandler = new ServiceHandler(mServiceLooper);
    }
/**
 * 这个方法里的实现其实很简单了，就是通过 handler 发送了一个消息
 * 把我们的 intent 对象和 startId 发送出去
 * 在我们上面的 handleMessage() 会接收到消息
 * 并且通过 onHandleIntent() 方法将对象回调给子类
*/
@Override
    public void onStart(@Nullable Intent intent, int startId) {
        Message msg = mServiceHandler.obtainMessage();
        msg.arg1 = startId;
        msg.obj = intent;
        mServiceHandler.sendMessage(msg);
    }
/**
 * 每次启动 IntentService，onStartCommand() 就会被调用一次
 * 在这个方法里处理每个后台任务的 intent
 * 可以看到在这个方法里调用的是上方的 onStart() 方法
*/
@Override
    public int onStartCommand(@Nullable Intent intent, int flags, int startId) {
        onStart(intent, startId);
        return mRedelivery ? START_REDELIVER_INTENT : START_NOT_STICKY;
    }
/**
 * 因为 looper 是无限循环轮询消息的一个机制，所以当我们明确不需要继续使用的话
 * 那么我们就应该通过它的 quit() 方法来终止它的执行
 * 这是个编程的好习惯，要记住哦！
*/
@Override
    public void onDestroy() {
        mServiceLooper.quit();
    }
/**
 * 这个方法的注释写了：除非你为这个 service 提供了绑定，否则不需要实现这个方法
 * 因为这个方法默认是返回 null 的
 * 所以咱们不用太关注这个方法
*/
@Override
    @Nullable
    public IBinder onBind(Intent intent) {
        return null;
    }
/**
 * 这就是 IntentService 中定义的抽象方法
 * 具体交由它自己的子类来实现
*/
@WorkerThread

```

```
    protected abstract void onHandleIntent(@Nullable Intent intent);  
}
```

## 3 安卓 Service 知识细节二 - 绑定服务和 AIDL

- <https://www.jianshu.com/p/4e6eedff1bf6>

在上一章节了解了后台服务之后，接下来我们来分析一下绑定服务，也是细节最多的一个地方。绑定服务，就是通过 bindService 启动的服务。有了绑定服务，既可以让同一 App 的其他组件与当前的 Service 进行交互，也可以暴露当前 App 的某些功能，从而给其他的 App 来提供服务。绑定服务遵循的是服务器-客户端的模式，与 Service 绑定的组件就是客户端，然后 Service 向这些客户端提供接口对象，接口就代表了两者交互的协议。客户端拿到了之后，根据双方的通信协议，调用相应的函数，来实现自己的功能。所以所谓的服务也就是接口，接口就承担了不同的 App，不同的模块之间的通信。

### 3.1 一：生命周期

- 首先，我们来看一下绑定服务的生命周期。相关的函数除了基本的 onCreate, onDestroy 之外，还有的就是：onBind, onUnBind, onRebind。我们从 bindService 开始一点点分析。
- 当当前的 App 或者其他的 App 的某个组件，想绑定 Service 的时候，就需要调用 bindService 函数，签名如下：

```
public abstract boolean bindService(Intent service, @NonNull ServiceConnection conn, @BindServiceFlags int flags);
```

- 首先来看一下参数，
- 第一个参数 Intent，代表被绑定的 Service。这里只使用显示 Intent，不要使用隐式 Intent。并且 Android 也在 5.0 之后的系统做了限制，隐式 Intent 是会出错的。
- 第二个参数是 ServiceConnection：

```
private interface ServiceConnection() {  
    public void onServiceConnected(ComponentName name, IBinder servcie) {  
    }  
    public void onServiceDisconnected(ComponentName name) {  
    }  
}
```

- ServiceConnection 是一个回调接口，因为调用 bindService 的时候，绑定成不成功可以通过返回值判断出来。但是 Service 返回的接口对象却是异步的，所以需要提交这个回调来接收。
- ServiceConnection 的两个接口都是在当前应用的主线程里执行，onServiceConnected 是用来接收接口对象的，当连接建立的时候该函数会被调用。
- onServiceDisconnected() 是在客户端与 Service 的连接意外中断的情况下，才会被调用。
- 如果我们是通过 **unBindService** 主动的断开连接，那么这个函数不会被调用。而且即便这个函数被调用了，但在系统眼里，这个连接还存在，所以在内存允许的情况下，系统会帮我们重连。
- 第三个参数是个 int 类型的 Flag，大多数情况下可以是 0 或者 BIND\_AUTO\_CREATE。除此之外，这个参数也可以决定客户端所在进程的优先级对 Service 所在进程的优先级的影响。相关的 Flag 有 BIND\_IMPORTANT, BIND\_ABOVE\_CLIENT, BIND\_WAIVE\_PRIORITY 等等，这个细节就不再这里仔细分析了，有兴趣的朋友可以查看源码。

- 当有客户端通过 bindService 与我们的 Service 建立连接后，Service 的 onBind 函数就会被调用：

```
// return an IBinder through which clients can call on to the service
public abstract IBinder onBind(Intent intent);
```

- 然后 Service 在这里将接口对象返回给客户端，客户端就可以使用 Service 提供的服务了。同时，这个地方有很多细节需要分析一下。

- 首先，对于 Service 来说，一个 Service 可以同时对外提供多个业务接口，这是可以的。要是提供多个接口，就要求客户端传递不同类型的 Intent，从而让 Service 在 onBind 里区分，把相应的接口返回去。这里说的是不同类型的 Intent，有点类似于 Intent 和 IntentFilter 的匹配，规则也类似，只有 Action, Category 这些关键信息会影响匹配，extra 不影响。所以如果客户端将只有 extra 不同的 Intent 传递给 Service，Service 的 onBind 是不会被多次调用的。也就是说，在 Service 眼里，一种类型的 Intent 对应一种接口。针对同一接口，Framework 会缓存返回的 IBinder 对象，所以当多个客户端传递的都是同一类型的 Intent，那么缓存的接口对象直接返回给客户端，onBind 不会调用。换句话说，系统会将 Service 提供的某一接口对象缓存，只有 bindService 传递的是不同类型的 Intent 的时候，onBind 才会被调用。
- 对于客户端来说，如果使用同样类型的 Intent 和同一个 ServiceConnection，多次调用 bindService，没有任何影响，后续的 bindService 都不会调用 onBind 和 ServiceConnection。
- 对于客户端来说，如果是同类型的 Intent，但是多个 ServiceConnection，相当于多个客户端对同一个接口感兴趣，每个 ServiceConnection 都会被调用，onBind 则不会。
- 如果传递的 Intent 不是同一类型的，那么 Service 的 onBind 就会被调用。而当 Service 接收到不同类型的 Intent 之后，Service 可以自己做选择，要么根据 Intent 的类型返回不同的接口，要么不做判断直接返回同一个接口。当返回同一个接口的时候，和前面分析的相同类型的 Intent 就一致了。只不过，当 Framework 发现 Intent 的类型不一样时，一定会调用 Service 的 onBind，有点类似于 Touch 事件的 onIntercept，就是给 Service 一个判断的机会，即使你返回的是同一个接口。
- 当 Service 会根据 Intent 的类型来判断，返回不同的接口时。正常来说，这个时候客户端的 ServiceConnection 也不是同一个，互不干扰。但有一个例外的情况，就是这个时候客户端传递的还是同一个 ServiceConnection。那么这个时候的调用顺序是，先调用这个 ServiceConnection 的 onServiceDisconnected，在调用 onServiceConnected。这种情况是一个错误用法，没有实际意义，只是简单的验证一下。
- 对于 Service 来说，当可以提供多个接口的时候，每一种类型的 Intent 对应一种接口，每一种接口会有多个连接。当其中一种接口的连接全部断开之后，**onUnbind** 会调用。

```
public boolean onUnbind(Intent intent) {
    return false;
}
```

- 然后如果还有客户端要建立这种接口的连接，那 Service 可能调用 onBind，也可能调用 onRebind，取决于 onUnbind 的返回值。返回 true，调用 onRebind，返回 false，就是 onBind。
- 当所有接口的所有连接全断开之后，这个 Service 也就没存在的必要了，就会被系统销毁，onDestroy 会被调用。
- 对于刚才提到的错误情况，不同的接口使用同一个 ServiceConnection。这个时候，系统认为 Service 只有一个连接，但 Service 确实对外提供了多个接口。比如你调用了三次 bindService，当调用 unBindService 断开连接的时候，由于这是唯一的连接，系统会销毁 Service；但同时由于 Service 提供了三个接口，所以 onUnbind 会被调用三次。

- 可见，绑定服务的生命周期不会像后台服务那样长期的在后台运行，直到被明确的关闭为止。在没有连接的情况下，系统会自动杀死。
  - 这里有两个例外，一个是有**可能 bindService** 返回的是 `false`，也就是说我们的连接没有成功。
  - 另一种是，**onBind** 可能在某些情况下，返回的是 `null`，那么 **ServiceConnection** 的回调函数也就不会被调用了。
- 上面的这两种情况下，我们还是要调用 **unBindService** 来解绑，否则就会阻止 **Framework** 销毁 **Service**，造成资源的浪费。
  - 还有一点就是，**unBindService** 不要调用多次，否则会报错。
  - 如果 **unBindService** 里面是一个没有通过 **bindService** 的 **ServiceConnection**，也会报错。
- 上面的这两种情况，**系统都会抛出这个 ServiceConnection 没有被注册的异常**。
- 对于生命周期的分析，我们现在还只局限在单纯的绑定服务的角度来分析，但开发中，更多的可能是混合的，也就是既被 **startService**，又被 **bindService** 了。这个时候，即便所有的连接全断开了，**Service** 也不会被系统回收，因为它还会做为后台服务继续运行，直到关闭为止。

## 3.2 二：IPC

- 对于绑定服务，前面分析到，采用的是服务-客户端的交互模式，由服务向客户端提供接口来访问，而提供接口的方式还要根据具体情况来确定，具体有：
  - **创建 Binder 的子类：**要求客户端和 **Service** 在同一个进程中
  - **Messenger：**客户端和 **Service** 不在同一个进程中，但不允许 **Service** 并发，必须串行的处理客户端的请求，这在某些情况下就会有串行压力
  - **AIDL：**客户端和 **Service** 不在同一个进程中，但对 **Service** 串行与并行有很大的自由，都可以。只不过一般串行处理的话，**Messenger** 就可以了，所以一般使用 **AIDL**，都是为了并行处理。
- 这三种方式，我们首先来看 AIDL。AIDL,Android Interface Definition Language，也就是 Android 接口定义语言。概念上算是一种语言，有自己的规范，主要目的是为了定义接口，这个接口比较特殊，是为了实现进程间通信的，也就是 IPC。
- 当两个不同的进程通信的时候，会有很多问题。
- 每个进程都有自己的内存空间，并且不共享，所以在进行数据传递的时候，尤其是自定义的复杂的类型，就需要把这些对象先分解为操作系统可以识别的原始的类型，到了另一个进程在组装成对象，这个过程也就是序列化与反序列化的过程。
- 对于 IPC 来说，要解决两个核心问题：数据序列化和执行线程。
  - 对于线程这块，我们不需要操心，系统已经帮我们做了。系统会给每个进程维护一个专门用于进程间调用的线程池，进程 A 调用进程 B 的一个对象的函数的时候，进程 A 就在自己的执行线程里执行，而进程 B 是在这个线程池里执行，而且默认是同步阻塞的。所以如果 **Service** 某个接口是耗时的，那么客户端就要避免在主线程里直接调用，避免 **ANR**。客户端调用 **Service** 的函数是这样，**Service** 回调客户端的接口同样是这样的。
  - 线程没问题了之后，下一个就是序列化，包括接口的序列化和请求参数和返回结果的序列化，因为我们也需要把接口的对象，跨越进程传递，而这部分就是 AIDL 帮我们完成的。

### 3.3 明白了 AIDL 的作用，和 IPC 的基本概念之后，我们来看一下 AIDL 的使用。

- 首先是定义给客户端的接口，文件格式为 aidl，里面的语法和 java 一样，例子如下：

```
package com.me.prac;
interface IServer {
    String getName();
    int getPID();
    void error();
}
```

- 其实看代码，和一个 java 普通接口没什么区别。
- 需要注意的地方是 **AIDL 支持的数据格式**：原始类型，String，CharSequence，List，Map，Parcelable 和 Aidl 接口。
  - 其中 **List** 和 **Map** 里面的元素也要是被支持的数据类型，由于它只支持 **Aidl** 接口，这意味这如果客户端将来要向 **Service** 注册一个回调接口，也必须是 **Aidl** 接口。
  - 对于 Map，可以用 Bundle 来替代。Bundle 就可以理解为支持 Parcelable 的 key-value 的数据结构，在这比 Map 更好一些。
  - 只不过使用 Bundle 的时候，读取数据之前，先设置 Classloader。在请求参数中，除了原始类型之外，其他的都应该标明方向，**in**, **out**, **inout**。这里要根据实际需要标明，毕竟是有损耗的。同时，接口里的函数可以用 **oneway** 关键字来标明。因为进程间函数调用默认是同步的，使用 **oneway** 可以更改这一行为，调用方调用完了函数可以即刻返回，不会阻塞。
- 以上就是接口定义需要注意的细节，如果其中引用了自定义的 Parcelable 类型，即便和接口在同一个包下，也要显示的导入。而且对于 AIDL 相关的接口，类，交互的两个进程都需要有一份，并且路径一致。因为数据的传输就是序列化，反序列化的过程，接收的进程需要有这个类才能反序列化成功。所以 AIDL 相关的东西，最好放在同一个包下，到时候直接拷贝到另一个进程的程序里即可。
- 在遵循 AIDL 的语言规范，定义完了 ADIL 接口之后，SDK 会自动帮我们生成一个同名的 java 接口，并且里面有个叫 Stub 的内部抽象类，针对上面那个例子，生成的部分代码为：

```
public interface IServer extends android.os.IInterface
{
    /** Local-side IPC implementation stub class. */
    public static abstract class Stub extends android.os.Binder implements com.timestamp.practice.uipractice.IServer
    {
        private static final java.lang.String DESCRIPTOR = "com.timestamp.practice.uipractice.IServer";
        /** Construct the stub at attach it to the interface. */
        public Stub() { this.attachInterface(owner, DESCRIPTOR); }
        /**
         * Cast an IBinder object into an com.timestamp.practice.uipractice.IServer interface,
         * generating a proxy if needed.
         */
        public static com.timestamp.practice.uipractice.IServer asInterface(android.os.IBinder obj)
        {
            if ((obj==null)) {
                return null;
            }
            android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
            if (((iin!=null)&&(iin instanceof com.timestamp.practice.uipractice.IServer))) {
                return ((com.timestamp.practice.uipractice.IServer)iin);
            }
            return new com.timestamp.practice.uipractice.IServer.Stub.Proxy(obj);
        }
    }
}
```

- 为了方便截图，这里只截取了一部分，格式也进行了调整。这个地方就涉及到了 AIDL 的精髓：Binder。
- AIDL 从定义上看只是一个接口定义语言，但在 IPC 通信的过程中，真正起作用的是生成的同名 Java 接口和内部的 Stub 类。

- 也就是说，AIDL 只是 Android 给我提供的一个工具类，简化了我们的开发，但并不是 IPC 的必需品，真正的核心是里面的 Binder。
- Binder 最核心的两个数据类型是 IInterface 和 IBinder。先来看一下 IInterface:

```
public interface IInterface {
    public IBinder asBinder();
}
```

- **IInterface** 是一个接口，是 IPC 需要使用到的根接口。当调用另一个进程的接口的时候，这个接口就一定要实现 IInterface，而它里面的函数很简单，就一个 asBinder，转化为对应的 IBinder。IBinder 和 IInterface 的对应关系后面会讲。
- 接下来就是 **IBinder**，它也是一个接口，但一般不直接使用，使用的是它的实现类 **Binder**。
  - Binder 是个特殊的数据类型，是 Android 实现 IPC 的 Binder 机制的核心。它可以理解为一个媒介或者传输介质，可以在进程间传递，并且跨越进程边界之后，两个进程的 Binder 还是同一个。也就是说，**Binder** 从一个进程传递到另一个进程之后，也会保持唯一。
  - 这是一个很重要的特性，可以在 IPC 过程中，作为 id 啊或者 token 之类的，Activity 启动的过程中就使用到了这个特性。
  - 同时 Binder 会对应一个接口，也就是 Service 给客户端提供的接口。但是在 Binder 进程间传递的过程中，对应的接口会发生变化。在发送方进程里，这个接口就是 **Binder** 自己。到了接收方进程里，就会变成一个代理。这些东西可以在 SDK 生成的 Stub 里看到，由于不方便截图，这里就不贴代码了。
- 以上就是 Binder 能够实现 IPC 的基础，其他的一些细节都是围绕这个展开。

#### • **IBinder** 里面的函数主要包括这么几个：

- **pingBinder() / isBinderAlive()**: 这两个函数主要用来判断 Binder 所在的进程是否还存活。对于客户端来说，就是判断 Service 的进程是否还存活，从而决定，我们还可不可以正常的调用服务。毕竟 Service 进程是有可能因为内存等原因意外中止的。
- **linkToDeath() / unlinkToDeath()**: 这是一对函数，里面的参数类型都是 DeathRecipient。这是一个接口，里面只有一个函数：
- **DeathRecipient**: 这是一个回调接口，当 Service 所在的进程被意外中止的时候，里面的 binderDied() 函数就会被调用。由于这是一个回调接口，而且还是跨进程调用，所以它会在客户端的线程池里执行。

```
public interface DeathRecipient {
    public void binderDied();
}
```

- **transact()**: 这是 Binder 机制中最重要的函数，翻译过来是交易。在 Android 眼里，两个不同的进程的通信就类似于交易。既有请求参数的输入，也有响应结果的输出，这个函数会在 Binder 类里面有实现，后面会有分析。
- 以上就是对 IBinder 这个核心类的简要分析，实际开发中，我们不会使用它，而是使用的实现类 Binder。接下来，我们就以 IServer 这个简单的例子来分析一下 Aidl 的调用流程：

```
public interface IServer extends android.os.IInterface
{
    public java.lang.String getName() throws android.os.RemoteException;
    public int getPID() throws android.os.RemoteException;
    public void error() throws android.os.RemoteException;

    /* Local-side IPC implementation stub class. */
    public static abstract class Stub extends android.os.Binder implements com.timestamp.practice.uipr
    {
```

- 在我们创建了 Aidl 接口之后，这就是 sdk 帮我们自动生成的同名的 Java 接口。继承了 IInterface，这是要求，给其他进程调用的远程对象，都得实现这个接口，并且这个 Java 接口里有和 Aidl 里面定义一样的函数。这个接口里有一个名为 **Stub** 的内部类：

```
/** Local-side IPC implementation stub class. */
public static abstract class Stub extends android.os.Binder implements com.timestamp.practice.uipractice.IServer
{
    private static final java.lang.String DESCRIPTOR = "com.timestamp.practice.uipractice.IServer";
    /** Construct the stub at attach it to the interface. */
    public Stub()
    {
        this.attachInterface((IInterface) this, DESCRIPTOR);
    }
    /**
     * Cast an IBinder object into an com.timestamp.practice.uipractice.IServer interface,
     * generating a proxy if needed.
     */
    public static com.timestamp.practice.uipractice.IServer asInterface(android.os.IBinder obj)
    {
        if ((obj==null)) {
            return null;
        }
        android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
        if (((iin!=null)&&(iin instanceof com.timestamp.practice.uipractice.IServer))) {
            return ((com.timestamp.practice.uipractice.IServer)iin);
        }
        return new com.timestamp.practice.uipractice.IServer.Stub.Proxy(obj);
    }
    @Override public android.os.IBinder asBinder()
    {
        return this;
    }
}
```

- 这个 Stub 继承了 Binder，又实现了刚才的 IServer 接口。这个类我们很熟悉，使用的时候，就是创建它的子类，然后将 IServer 里定义的函数实现，至于需不需要多线程啊，线程安全这些问题看自己的需要，这里不在详细分析了，重点看 **Aidl 的调用流程**。
- 接口里的逻辑定义完了之后，然后创建自定义 **Stub** 子类的对象，将它通过 **onBind()** 函数返回给客户端。
- 我们看一下 Stub 的构造函数，它内部调用了 attachInterface 函数，这是一个定义在 IBinder 的函数，Binder 中的实现为：

```
/**
 * Convenience method for associating a specific interface with the Binder.
 * After calling, queryLocalInterface() will be implemented for you
 * to return the given owner IInterface when the corresponding
 * descriptor is requested.
 */
public void attachInterface(@Nullable IInterface owner, @Nullable String descriptor) {
    mOwner = owner;
    mDescriptor = descriptor;
}
```

- 代码很简单，就是很字段赋值。我们前面提到，可以认为每个 Binder 对象都对应一个 IInterface 类型的业务接口，它内部有个字段来存储。所以这个函数的意思就是赋值，并且这个接口有自己对应的一个描述符：

```
/** Local-side IPC implementation stub class. */
public static abstract class Stub extends android.os.Binder implements com.timestamp.practice.uipractice.IServer
{
    private static final java.lang.String DESCRIPTOR = "com.timestamp.practice.uipractice.IServer"; ←
    /** Construct the stub at attach it to the interface. */
}
```

- 通过 attachInterface 的代码实现，我们可以确定，在 Service 端，Binder 对应的业务接口就是自己。接下来我们来看客户端，由于 Aidl 相关的接口在客户端也有一份，所以 **SDK 同样也会给客户端构建对应的 Java 接口和 Stub**，只不过没有了自定义的 **Stub** 子类。而客户端绑定服务成功之后，就会以在 ServiceConnection 中接收到的 IBinder 对象为参数，调用 IServer.Stub.asInterface 函数，将其转化为对应的业务接口 IServer。
- 接下来我们看 **asInterface()** 的逻辑，它是一个静态函数：

```

/**
 * Cast an IBinder object into an com.timestring.practice.uipractice.IServer interface,
 * generating a proxy if needed.
 */
public static com.timestring.practice.uipractice.IServer asInterface(android.os.IBinder obj)
{
    if ((obj==null)) {
        return null;
    }
    android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
    if (((iin!=null)&&(iin instanceof com.timestring.practice.uipractice.IServer))) {
        return ((com.timestring.practice.uipractice.IServer)iin);
    }
    return new com.timestring.practice.uipractice.IServer.Stub.Proxy(obj);
}

```

- 首先根据接口对应的描述符，来调用 queryLocalInterface：

```

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

```

- 这个函数的意思是，在本地查询 descriptor 对应的接口的实现类。而这个实现类在客户端肯定是没有的，所以 queryLocalInterface 就会返回 null。然后 asInterface 就会以接收到的 IBinder 为参数，构建一个 IServer.Stub.Proxy 对象，可以理解为，远程接口在本地的代理对象。

```

private static class Proxy implements com.me.prac.IServer {
    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;
    }
}

```

- Proxy 内部有一个 mRemote 字段代表它指向的远程服务接口，那么客户端转化后的接口对象，实际上就是一个 Proxy，我们调用服务，其实就是调用 Proxy 里面的函数。所以，Binder 机制可以通俗点理解为 Service 给客户端提供接口，由 Binder 负责传递，Binder 保持唯一，但在 Service 端对应的接口是我们自定义的 Stub 的子类，而在客户端对应的就是一个 Proxy，并且这个 Proxy 通过 mRemote 字段指向远程接口。
- 客户端拿到了接口，接下来就会调用某个函数来实现自己的逻辑。这里以 getName 为例，调用的也就是 Proxy 的 getName()：

```

@Override public java.lang.String getName() throws android.os.RemoteException
{
    android.os.Parcel _data = android.os.Parcel.obtain();
    android.os.Parcel _reply = android.os.Parcel.obtain();
    java.lang.String _result;
    try {
        _data.writeInterfaceToken(DESCRIPTOR);
        mRemote.transact(Stub.TRANSACTION_getName, _data, _reply, (flags: 0));
        _reply.readException();
        _result = _reply.readString();
    }
    finally {
        _reply.recycle();
        _data.recycle();
    }
    return _result;
}

```

- 首先声明了两个 Parcel，分别代表这个函数的请求参数和返回结果。Parcel 大家都不陌生，和 Parcelable 一块使用。源码给的解释是不能把 Parcelable 当成普通的序列化机制，它主要应用在 Binder 中，在内存的序列化方面性能极其高效，特别适用于在内存中，跨越进程的传递，但不适合持久化到存储设备和网络中传输。至于为什么 Parcelable 比较高效，笔者暂时还没仔细研究，网上的说法很多比较笼统，类似于一次拷贝之类的，这个等以后在补充吧。使用 Parcelable 的时候，是用 Parcel 作为数据的载体，将对象的状态存储其中，可以将 Parcel 理解为操作系统能识别的字节序列。由于函数的请求参数和返回结果都要跨进程，这个地方使用了 Parcel 来传递。到这里，我们可以说，Parcel 和 Binder 就是 IPC 的两大关键媒介，一个传输请求参数这些数据，一个传输接口，有了它俩，才有了 IPC 通信的可能。
- 对于 getName 来说，不需要参数，所以其中的 \_data 字段是空的。创建了 \_data 和 \_reply 两个 Parcel，并在需要的情况下将参数塞到 Parcel 里，然后调用了 mRemote 的 transact 函数，开始了跨进程的通信，交易。这里 transact 还是在客户端的执行线程里执行，并且会到导致当前线程挂起，等待结果的返回。而 mRemote 位于 Service 进程，所以 Service 那边会在系统维护的线程池里继续进行。transact 的实现为：

```
/**  
 * Default implementation rewinds the parcels and calls onTransact(). On  
 * the remote side, transact calls into the binder to do the IPC.  
 */  
public final boolean transact(int code, @NonNull Parcel data, @Nullable Parcel reply,  
    int flags) throws RemoteException {  
    if (false) Log.v("Binder", msg: "Transact: " + code + " to " + this);  
    if (data != null) {  
        data.setDataPosition(0);  
    }  
    boolean r = onTransact(code, data, reply, flags);  
    if (reply != null) {  
        reply.setDataPosition(0);  
    }  
    return r;  
}
```

- 这个函数是 final 的，不可重写。它的第一个参数是方法对应的 code。在系统生成的 Stub 里面，会给每一个 Aidl 接口定义的函数声明一个 code：

```
static final int TRANSACTION_getName = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);  
static final int TRANSACTION_getPID = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);  
static final int TRANSACTION_error = (android.os.IBinder.FIRST_CALL_TRANSACTION + 2);
```

- 这些 code 都和相应的函数相匹配，一般都是从 IBinder.FIRST\_CALL\_TRANSACTION 逐步累加。然后接下来的两个参数就是装载请求参数和返回结果的 Parcel。最后一个参数是个 flag，如果是默认的调用，也就是同步的调用，就是 0。如果该函数被 oneway 修饰了，那么就是 FLAG\_ONEWAY：

```
/**  
 * Flag to {@link #transact}: this is a one-way call, meaning that the  
 * caller returns immediately, without waiting for a result from the  
 * callee. Applies only if the caller and callee are in different  
 * processes.  
 */  
int FLAG_ONEWAY = 0x00000001;
```

- 然后 transact 就会调用 onTransact 函数，这个函数才是真正发生 IPC 的地方。我们这里只需要关注 Stub 的 onTransact 就好：

```

@Override public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags) throws android.os.RemoteException
{
    switch (code)
    {
        case INTERFACE_TRANSACTION:
        {
            reply.writeString(DESCRIPTOR);
            return true;
        }
        case TRANSACTION_getName:
        {
            data.enforceInterface(DESCRIPTOR);
            java.lang.String _result = this.getName();
            reply.writeNoException();
            reply.writeString(_result);
            return true;
        }
        case TRANSACTION_getID:
        {
            data.enforceInterface(DESCRIPTOR);
            int _result = this.getID();
            reply.writeNoException();
            reply.writeInt(_result);
            return true;
        }
        case TRANSACTION_error:
        {
            data.enforceInterface(DESCRIPTOR);
            this.error();
            reply.writeNoException();
            return true;
        }
    }
    return super.onTransact(code, data, reply, flags);
}

```

- onTransact 的代码也很好理解，根据 code 调用对应的函数。而 getName 就会调用我们自定义 Stub 子类的 getName() 函数，调用我们自定义的业务逻辑。执行完毕后，将返回值塞到 reply 里面。而这个 reply 也就是客户端传过来的，然后 onTransact 和 transact 函数依次返回，再回到 Proxy 的 getName 里面：

```

@Override public java.lang.String getName() throws android.os.RemoteException
{
    android.os.Parcel _data = android.os.Parcel.obtain();
    android.os.Parcel _reply = android.os.Parcel.obtain();
    java.lang.String _result;
    try {
        _data.writeInterfaceToken(DESCRIPTOR);
        mRemote.transact(Stub.TRANSACTION_getName, _data, _reply, (flags: 0));
        _reply.readException();
        _result = _reply.readString();
    }
    finally {
        _reply.recycle();
        _data.recycle();
    }
    return _result;
}

```



- 当 mRemote.transact 返回的时候，其中的 \_reply 也就有数据了。它从 \_reply 里面读取结果，从而返回，这样客户端就可以拿到 getName 的返回值。可见，Parcel 也和 Binder 一样算是一个载体，自由的跨进程传输，里面装载数据，从而实现进程间的数据传递和通信。
- 至此，我们通过一个最简单的例子，把 AIDL 的调用过程分析完了。这里总结一下，对于 Android 系统下的 IPC 而言，真正的核心就是 Binder。AIDL 只是辅助，相当于是 sdk 给我们提供的工具类，帮我们自动生成了代码，但他并不是必须的。即便没有 Aidl 接口，我们也可以自己写出对应的 Java 接口和 Stub。同时，bindService 和 ServiceConnection 严格意义上来说，也不是必须的。比如在 Framework 中，像 ServiceManager，AMS 也大量的使用到了 Binder，但他们就没有用到 bindService。基于 Binder 实现 IPC 的话，真正要做的是 Binder 的传递和 Binder 到接口的转换，bindService 只是 Android 系统为我们普通的 App 提供的传递 Binder 对象的方式而已。Android IPC 真正的核心就是 Binder，在依赖于 Parcel，分别实现了接口和数据跨进程的传递，从而才有了跨进程通信/交易的可能，其他的只是基于此的辅助手段。
- 同时，任玉刚大神的《Android 开发艺术探索》里也提到了 AIDL 的一些常见问题扩展，包括权限验证，死亡通知，回调接口注册和 Binder 连接池等等。回调接口注册用到了 RemoteCallbackList，源码也不复杂，关键用到了 Binder 的唯一性。大家可以自己看一下，

这里就不再多说了。其中 Binder 连接池，在 Android 系统里也有类似的概念。比如 Activity Manager Service，这个 AMS 服务，归根结底也就是系统的服务进程给我们提供的一个接口对象，类型为 IActivityManager，在获得这个接口对象的时候，就是通过 ServiceManager 的 getService 来查询到的，和连接池的概念很相似，具体的在后续的 Activity 的启动过程中会有详细描述。

- 至此，我们把 Service 给客户端提供接口的方式之一 AIDL 的细节分析完了。除此之外，Service 还可以通过 Messenger 和 Binder 子类的方式来提供接口，由于篇幅原因，下一章节会在分析。
- 参考：<https://developer.android.google.cn/guide/components/aidl>