

安卓 IPC AIDL Service 用法及原理详解

deepwaterooo

August 14, 2022

Contents

1 基本过程概述：一般实现步骤讲解	1
1.1 在客户端 (Activity 中) 要完成:	1
1.2 在服务端一般要实现:	1
2 abstract class IntentService extends Service 源码通讯简单注释	2
3 透彻解析安卓 Handler: Android--多线程之 Handler	3
3.1 前言	3
3.2 Handler	4
3.3 Post	4
3.4 Message	4
4 安卓 Service 知识细节二 - 绑定服务和 AIDL	6
4.1 一：生命周期	6
4.2 IBinder 相关	8
4.3 Messenger	9
4.4 二：IPC	9
4.5 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。	10
5 深入理解 Android 之 Service 绑定流程：通过一个例子以及源码里必要的细节来理解这个过程	16
5.1 服务绑定，使用 AIDL 进程间通信：主要是相关的 APIs 接口设计	16
5.2 编写 AIDL 文件，定义获取远程服务数据接口：主要是相关的 APIs 接口设计	16
5.2.1 远程服务接口描述	16
5.3 远程服务：申明服务；提供跨进程远程 IBinder 实例的索引以及供客户端调用的公用 APIs	17
5.3.1 AndroidManifest 中申明远程服务	17
5.3.2 RemoteService 远程服务	17
5.3.3 Client 端绑定服务，实现 ServiceConnection 接口	18
5.4 Android IPC 之服务端回调：以一个小例子的方式帮助自己理解定义与回调数据给客户端的过程	18
5.4.1 客户端通过 IPC 从服务端获取学生信息，学生信息封装在 IPC Parcelable 的子类 Student 类里：	18
5.4.2 IPC 接口：Parcelable 的子类 IStudentInfo 学生信息，包括姓名、年龄、分数三个字段。	19
5.4.3 跨进程传递接口	19
5.4.4 AIDL 回调的使用：实现定义过的/服务端和客户端都认得识别的公认 API 交互逻辑	19
5.4.5 注意事项	22

5.4.6 回调在四大组件里的应用	22
5.5 Activity 绑定服务到 ActivityManagerService 过程, 所有相关源码位于 frameworks/base/ 目录下	22
5.5.1 ./core/java/android/content/ContextWrapper.java	22
5.5.2 ./core/java/android/app/ContextImpl.java	22
5.5.3 ./core/java/android/app/LoadedApk.java	23
5.5.4 ./core/java/android/app/LoadedApk.java	24
5.6 AMS 到 ActivityThread 启动绑定 Service 过程	25
5.6.1 服务与 Activity 组件建立绑定关系: 理解起来相对复杂, 今天暂时不根了, 改天头脑清醒的时候再回来看这个	25
5.6.2 ./services/core/java/com/android/server/am/ActiveServices.java	25
5.6.3 ./services/core/java/com/android/server/am/ServiceRecord.java	26
5.6.4 ./services/core/java/com/android/server/am/ActiveServices.java	27
5.6.5 ./core/java/android/app/ActivityThread.java	28
5.6.6 ./services/core/java/com/android/server/am/ActivityManagerService.java	28
5.6.7 ./services/core/java/com/android/server/am/ActivityManagerService.java	29
5.6.8 ./services/core/java/com/android/server/am/ActiveServices.java	29
5.6.9 ./services/core/java/com/android/server/am/ActiveServices.java	30
5.6.10 ./services/core/java/com/android/server/am/ActiveServices.java	30
5.6.11 ./core/java/android/app/ActivityThread.java	30
5.6.12 ./core/java/android/app/ActivityThread.java	31
5.6.13 ./core/java/android/app/ActivityThread.java	31
5.6.14 ./services/core/java/com/android/server/am/ActivityManagerService.java	32
5.6.15 ./services/core/java/com/android/server/am/ActiveServices.java	32
5.6.16 ./core/java/android/app/LoadedApk.java	33
5.6.17 ./core/java/android/app/LoadedApk.java	33
5.6.18 ./core/java/android/app/LoadedApk.java	33
5.7 总结:	34

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 onHandlerIntent() 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 透彻解析安卓 Handler: Android--多线程之 Handler

- https://www.cnblogs.com/plokmju/p/android_Handler.html

3.1 前言

- Android 的消息传递机制是另外一种形式的“事件处理”，这种机制主要是为了解决 Android 应用中多线程的问题，在 Android 中不允许 Activity 新启动的线程访问该 Activity 里的 UI 组件，这样会导致新启动的线程无法改变 UI 组件的属性值。但实际开发中，很多地方需要在工作线程中改变 UI 组件的属性值，比如下载网络图片、动画等等。本篇博客主要介绍 Handler 是如何发送与处理线程上传递来的消息，并讲解 Message 的几种传递数据的方式，最后均会以小 Demo 来演示。

3.2 Handler

- Handler，它直接继承自 Object，一个 Handler 允许发送和处理 Message 或者 Runnable 对象，并且会关联到主线程的 MessageQueue 中。每个 Handler 具有一个单独的线程，并且关联到一个消息队列的线程，就是说一个 Handler 有一个固有的消息队列。当实例化一个 Handler 的时候，它就承载在一个线程和消息队列的线程，这个 Handler 可以把 Message 或 Runnable 压入到消息队列，并且从消息队列中取出 Message 或 Runnable，进而操作它们。
- Handler 主要有两个作用：
 - 在工作线程中发送消息。
 - 在 UI 线程中获取、处理消息。
- 上面介绍到 Handler 可以把一个 Message 对象或者 Runnable 对象压入到消息队列中，进而再在 UI 线程中获取 Message 或者执行 Runnable 对象，所以 Handler 把压入消息队列有两大体系，Post 和 sendMessage：
 - Post: Post 允许把一个 Runnable 对象入队到消息队列中。它的方法有：post(Runnable)、postAtTime(Runnable,long)、postDelayed(Runnable,long)。
 - sendMessage: sendMessage 允许把一个包含消息数据的 Message 对象压入到消息队列中。它的方法有：sendEmptyMessage(int)、sendMessage(Message)、sendMessageAtTime(Message,long)、sendMessageDelayed(Message,long)。
- 从上面的各种方法可以看出，不管是 post 还是 sendMessage 都具有多种方法，它们可以设定 Runnable 对象和 Message 对象被入队到消息队列中，是立即执行还是延迟执行。

3.3 Post

- 对于 Handler 的 Post 方式来说，它会传递一个 Runnable 对象到消息队列中，在这个 Runnable 对象中，重写 run() 方法。一般在这个 run() 方法中写入需要在 UI 线程上的操作。

- 在 Handler 中，关于 Post 方式的方法有：

- boolean post(Runnable r): 把一个 Runnable 入队到消息队列中，UI 线程从消息队列中取出这个对象后，立即执行。
- boolean postAtTime(Runnable r,long uptimeMillis): 把一个 Runnable 入队到消息队列中，UI 线程从消息队列中取出这个对象后，在特定的时间执行。
- boolean postDelayed(Runnable r,long delayMillis): 把一个 Runnable 入队到消息队列中，UI 线程从消息队列中取出这个对象后，延迟 delayMillis 秒执行
- void removeCallbacks(Runnable r): 从消息队列中移除一个 Runnable 对象。

3.4 Message

- Handler 如果使用 sendMessage 的方式把消息入队到消息队列中，需要传递一个 Message 对象，而在 **Handler** 中，需要重写 **handleMessage()** 方法，用于获取工作线程传递过来的消息，此方法运行在 UI 线程上。下面先介绍一下 Message。

- Message 是一个 FINAL 类，所以不可被继承。Message 封装了线程中传递的消息，如果对于一般的数据，Message 提供了 getData() 和 setData() 方法来获取与设置数据，其中操作的数据是一个 Bundle 对象。

- 这个 Bundle 对象提供一系列的 getXxx() 和 setXxx() 方法用于传递基本数据类型的键值对，
 - * 对于基本数据类型，使用起来很简单，这里不再详细讲解。
 - * 而对于复杂的数据类型，如一个对象的传递就要相对复杂一些。在 Bundle 中提供了两个方法，专门用来传递对象的，但是这两个方法也有相应的限制，需要实现特定的接口，
- 当然，一些 Android 自带的类，其实已经实现了这两个接口中的某一个，可以直接使用。方法如下：
 - * putParcelable(String key, Parcelable value): 需要传递的对象类实现 Parcelable 接口。
 - * pubSerializable(String key, Serializable value): 需要传递的对象类实现 Serializable 接口。

- 还有另外一种方式在 Message 中传递对象，那就是使用 Message 自带的 obj 属性传值，它是一个 Object 类型，所以可以传递任意类型的对象，Message 自带的有以下几个属性：

- int arg1: 参数一，用于传递不复杂的数据，复杂数据使用 setData() 传递。
- int arg2: 参数二，用于传递不复杂的数据，复杂数据使用 setData() 传递。
- Object obj: 传递一个任意的对象。
- int what: 定义的消息码，一般用于设定消息的标志。

- 对于 Message 对象，一般并不推荐直接使用它的构造方法得到，而是建议通过使用 **Message.obtain()** 这个静态的方法或者 **Handler.obtainMessage()** 获取。

- Message.obtain() 会从消息池中获取一个 Message 对象，如果消息池中是空的，才会使用构造方法实例化一个新 Message，这样有利于消息资源的利用。并不需要担心消息池中的消息过多，它是有上限的，上限为 10 个。
- Handler.obtainMessage() 具有多个重载方法，如果查看源码，会发现其实 Handler.obtainMessage 在内部也是调用的 Message.obtain()。

- Message.obtain() 方法具有多个重载方法，大致可以分为两类：
 - 一类是无需传递 Handler 对象，对于这类的方法，当填充好消息后，需要调用 Handler.sendMessage() 方法来发送消息到消息队列中。
 - 第二类需要传递一个 Handler 对象，这类方法可以直接使用 Message.sendToTarget() 方法发送消息到消息队列中，这是因为在 Message 对象中有一个私有的 Handler 类型的属性 Target，当时 obtain 方法传递进一个 Handler 对象的时候，会给 Target 属性赋值，当调用 sendToTarget() 方法的时候，实际在它内部还是调用的 Target.sendMessage() 方法。
- 在 Handler 中，也定义了一些发送空消息的方法，如：sendEmptyMessage(int what)、sendEmptyMessageDelayed(int what, long delayMillis)，看似这些方法没有使用 Message 就可以发送一个消息，但是如果查看源码就会发现，其实内部也是从 Message.obtain() 方法中获取一个 Message 对象，然后给属性赋值，最后使用 sendMessage() 发送消息到消息队列中。
- Handler 中，与 Message 发送消息相关的方法有：
 - Message obtainMessage(): 获取一个 Message 对象。
 - boolean sendMessage(): 发送一个 Message 对象到消息队列中，并在 UI 线程取到消息后，立即执行。
 - boolean sendMessageDelayed(): 发送一个 Message 对象到消息队列中，在 UI 线程取到消息后，延迟执行。
 - boolean sendEmptyMessage(int what): 发送一个空的 Message 对象到队列中，并在 UI 线程取到消息后，立即执行。
 - boolean sendEmptyMessageDelayed(int what, long delayMillis): 发送一个空 Message 对象到消息队列中，在 UI 线程取到消息后，延迟执行。
 - void removeMessage(): 从消息队列中移除一个未响应的消息。

4 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。

4.1 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 是一个回调接口类，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，即便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，就是如果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()，再调用 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。

4.2 IBinder 相关

- IBinder 是远程对象的基本接口，是为高性能而设计的轻量级远程调用机制的核心部分。但它不仅用于远程调用，也可以用于进程内调用。这个接口定义了与远程对象交互的协议，一般不直接实现这个接口，而是从它的实现类 Binder 中继承。
- 通过 IBinder 进行服务的交互一般有两种方式：
 - 一种方式是使用 **IBinder.transact()** 方法向远端的 IBinder 对象发送一个发出调用，会回调远端的 **Binder.onTransact()** 方法，这个方法传递的数据是 **Parcel**。
 - * Parcel 是一种缓冲区，除了数据外还有有一些描述它内容的元素，如果查看源码的话会发现，Parcel 本质上是一个 Serialize，只是它在内存中完成了序列化和反序列化，利用的是连续的内存空间，因此效率会更高，并且 AIDL 的数据也是通过 Parcel 来交互的。
 - 另外一种方法就是抛弃 **IBinder** 中原生的方法，使用自定义的接口方法进行数据交互，这也是 Android 官方推荐绑定服务的一种数据交互方式。
- 不管是使用 **transact()** 给远程服务交互，还是使用自定义的接口交互，都是同步执行的，直到远程服务执行完并返回结果才会继续向下执行。

4.3 Messenger

- Messenger 引用了一个 Handler 独享？，可以使用 **Messenger.send(Message msg)** 方法跨进程向服务发送消息，只需要在服务中使用 Handler 创建一个 Messenger，宿主（客户端）持有这个 Messenger 就可以与服务进行通信。之前介绍的 handler+Message 的通信方式不同，那都是在同一个进程中的，从工作线程持有一个主线程的 Handler 对象，从而向主线程发送消息，这里不了解的可以看看之前的博客：Android--多线程之 Handler (https://www.cnblogs.com/plokmu/p/android_Handler.html)。
- 上面介绍过了，Android 可以使用 **IBinder** 实现跨进程通信，并且也将 **Handler** 与 **IBinder** 结合起来实现跨进程发送消息。
- 当然这里提一下，Messenger 管理的是一个消息队列，它会依据消息进入的先后次序予以执行（以串口的方式执行，不适用于大量大规模交互），所以也不需要把服务设计为线程安全是。
- 实现 Messenger 实现进程通信，主要有以下几点注意：
 - 在服务中实现一个 Handler 类，并实例化它，在 **handleMessage()** 方法中接收客户端的请求（这是远程服务端）。
 - 在服务中使用这个 Handler 对象创建一个 Messenger 对象（这是远程服务端）。
 - 使用 Messenger 对象的 **getBinder()** 方法返回一个 IBinder 对象作为 **onBind()** 的返回值返回给客户端（这是远程服务端）。
 - 在客户端使用 IBinder 实例化一个 Messenger 对象，并使用它向服务端发送信息（这是客户端）。

4.4 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 不在同一个进程里，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 不在同一个进程里，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。
 - 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。

4.5 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，这意味着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。
- 这里补充一下 IPC AIDL 中数据流向的参数 tag 详解：

- **定向 tag**: AIDL 中，除了基本数据类型，其他类型的方法参数都必须标上数据在跨进程通信中的流向：in、out 或 inout：
 - 1、in 表示输入型参数：只能由客户端流向服务端，服务端收到该参数对象的完整数据，但服务端对该对象的后续修改不会影响到客户端传入的参数对象；
 - 2、out 表示输出型参数：只能由服务端流向客户端，服务端收到该参数的空对象，服务端对该对象的后续修改将同步改动到客户端的相应参数对象；
 - 3、inout 表示输入输出型参数：可在客户端与服务端双向流动，服务端接收到该参数对象的完整数据，且服务端对该对象的后续修改将同步改动到客户端的相应参数对象；
- 定向 tag 需要一定的开销，根据实际需要去确定选择什么 tag，不能滥用。
- 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(null, 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(null, 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** 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>

5 深入理解 Android 之 Service 绑定流程：通过一个例子以及源码里必要的细节来理解这个过程

5.1 服务绑定，使用 AIDL 进程间通信：主要是相关的 APIs 接口设计

这里不会过多的深究 AIDL 进程通信底层原理，而是通过简单 Demo 开启一个新的进程远程服务与 Activity 绑定来简单回顾 bindService 方式 Service 绑定。

5.2 编写 AIDL 文件，定义获取远程服务数据接口：主要是相关的 APIs 接口设计

5.2.1 远程服务接口描述

```
// IRemoteService.aidl
package mao.com.testaidl;
//注意引用的包名需要与 对应类路径一致
import mao.com.testaidl.Data;
// Declare any non-default types here with import statements
interface IRemoteService {
    /**
     * Demonstrates some basic types that you can use as parameters
     * and return values in AIDL.
     */
    int getPid();
    Data getData();
}
```

- 传输的自定义数据 AIDL 文件

```
// Data.aidl 注意包名路径
package mao.com.testaidl;
//定义 数据声明
Parcelable Data;
```

5.3 远程服务：申明服务；提供跨进程远程 IBinder 实例的索引以及供客户端调用的公用 APIs

5.3.1 AndroidManifest 中申明远程服务

- 要开启一个进程运行，首先要在清单文件添加 process 属性

```
<!--服务在新的进程中启动-->
<service android:name=".RemoteService"
/>
```

5.3.2 RemoteService 远程服务

```
/*
 * @Description: 远程服务
 */
public class RemoteService extends Service {
    private static final String TAG = "RemoteService";
    Data mData;

    // ===== 这些是作为远程服务端，必须提供给绑定后的各个客户端使用的，必须实现
    @Nullable @Override
    public IBinder onBind(Intent intent) {
        Log.i(TAG, "[RemoteService] onBind");
    }
    // ===== 这个 IBinder 实例的引用，会返回给客户端，以便客户端能够与其（这个服务端 IBinder 实例的 reference）为桥梁，调用其公用 APIs，实现调用与交互
    return mBinder;
}

/** 实现 IRemoteService.aidl 中定义的方法 */
// ===== 这些是作为远程服务端，必须提供给绑定后的各个客户端使用的，必须实现
IRemoteService.Stub mBinder = new IRemoteService.Stub() {
    @Override
    public int getPid() throws RemoteException {
        return android.os.Process.myPid();
    }
    // 返回客户端需要获取的数据：提供给客户端调用的公用 APIs
    @Override
    public Data getData() throws RemoteException {
        return mData;
    }
}
// ===== 这些是作为远程服务端，必须提供给绑定后的各个客户端使用的，必须实现 onTransact()
// 该实现可以设置权限
@Override
public boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {
    return super.onTransact(code, data, reply, flags);
}

@Override public boolean onUnbind(Intent intent) {
    return super.onUnbind(intent);
}
@Override void onCreate() {
    super.onCreate();
    Log.i(TAG, "[RemoteService] onCreate");
    initData();
}
@Override public void onDestroy() {
    super.onDestroy();
    Log.i(TAG, "[RemoteService] onDestroy");
}
/** * 初始化 Data 数据 */
private void initData() {
    mData = new Data();
    mData.setData1(10);
    mData.setData2("远程服务返回数据");
}
}
```

5.3.3 Client 端绑定服务，实现 ServiceConnection 接口

- 实现 ServiceConnection 接口

```
// 监听服务连接状态
private ServiceConnection serviceConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        // ===== 这里拿到了可以调用操控远程公用 API 的句柄或是引用或说是实例
        mRemoteService = IRemoteService.Stub.asInterface(service);
        String pidInfo = null;
        try {
            Data data = mRemoteService.getData(); // =====
            pidInfo = "pid=" + mRemoteService.getPid() +
                ", data1 = " + data.getData1() +
                ", data2=" + data.getData2();
        }
```

```

        } catch (RemoteException e) {
            e.printStackTrace();
        }
        Log.i(TAG, "[ClientActivity] ServiceConnection");
        Log.i(TAG, "绑定服务组件获取的远程服务数据: "+pidInfo);
        mCallBackTv.setText(pidInfo); // <<<<<< 操作客户端本地 UI 相关
    }
    @Override
    public void onServiceDisconnected(ComponentName name) {
        Log.i(TAG, "[ClientActivity] onServiceDisconnected");
        mCallBackTv.setText("onServiceDisconnected"); // <<<<<< 操作客户端本地 UI 相关
        mRemoteService = null; // 置空远程服务端的引用, 以免资源泄露
    }
};

• 绑定服务

```

```

Log.i(TAG, "[ClientActivity] bindRemoteService");
Intent intent = new Intent(MainActivity.this, RemoteService.class);
bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);

```

- Demo 最终运行日志

```

mao.com.testaidl I/ClientActivity: [ClientActivity] bindRemoteService
? I/RemoteService: [RemoteService] onCreate
? I/RemoteService: [RemoteService] onBind
mao.com.testaidl I/ClientActivity: [ClientActivity] ServiceConnection
mao.com.testaidl I/ClientActivity: 绑定服务组件获取的远程服务数据: pid=26575, data1 = 10, data2= 远程服务返回数据

```

5.4 Android IPC 之服务端回调: 以一个小例子的方式帮助自己理解定义与回调数据给客户端的过程

- 之前接触的都是最基础的, 只需要绑定服务端就可以了; 可是如果客户端想要拿服务端的数据要如何回调呢? 这一节是这个最基础的实现原理逻辑
- 以一个小例子的方式帮助自己理解定义与回调数据给客户端的过程

5.4.1 客户端通过 IPC 从服务端获取学生信息, 学生信息封装在 IPC Parcelable 的子类 Student 类里:

```

public class Student implements Parcelable {
    private String name;
    private int age;
    private float score;
}

```

5.4.2 IPC 接口: Parcelable 的子类 IStudentInfo 学生信息, 包括姓名、年龄、分数三个字段。

- 我们定义 AIDL 接口如下:

```

interface IStudentInfo {
    // 主动获取
    Student getStudentInfo();
}

```

- 客户端通过调用 getStudentInfo() 方法即可获取从服务端返回的学生信息。

5.4.3 跨进程传递接口

- 客户端想要获取学生信息, 需要主动调用 getStudentInfo() 方法。考虑一种场景:
 - 1、学生每一门考试, 分数都在变化, 客户端需要一直轮询去调用 getStudentInfo() 方法才能获取最新的成绩。我们知道轮询是效率比较低的做法, 要尽量避免。

- 2、我们就会想到学生成绩发生了变化了，服务端就主动通知我们就好啦。
- 现在的问题重点是：**服务端如何主动通知客户端**。
- 依据以往的经验，有两种方式可以实现：
 - 1、客户端通过绑定服务端的 Service，进而与服务端通信，那么可以换种思路，客户端也可以定义 Service，而后服务端通过绑定客户端，进而调用客户端的接口，主动给客户端传递消息。
 - 2、客户端绑定了服务端的 Service，两者之间就能够通信。实际上服务端传递了 Binder 给客户端，客户端拿到 Binder 之后就可以进行通信了，这就说明了 Binder 对象本身能够跨进程传输。
- 于是改造之前的接口：
 - 客户端调用服务端接口的时候将自己生成的 Binder 传递给服务端，那么服务端发生变化的时候就可以通过这个 Binder 来通知客户端了。(具体是这样的吗？还是说还可以有更好的实现呢？)
- 通过比对 1、2 两种方式：
 - 第一种方式过于复杂，对于客户端、服务端的角色容易搞混。
 - 第二种方式符合我们认知的“回调”，也就是说跨进程的回调和同一个进程里的回调理解上是一致的。

5.4.4 AIDL 回调的使用：实现定义过的/服务端和客户端都认得识别的公认 API 交互逻辑

1. 服务端声明回调接口

- 定义 AIDL 回调接口：

```
import com.fish.ipcserver.Student;
interface RemoteCallback {
    // 回调
    // oneway：表示调用 onCallback(xx) 方法的线程立即返回，不阻塞等待方法调用结果
    oneway void onCallback(in Student student);
}
```

- Student 为学生信息类，该对象支持跨进程传输。
- in 表示数据流方向，表示该 Student 对象传递给客户端（注意：这里仍然是之前曾经提醒过自己的，以客户端的眼光来看问题；对客户端来说，是数据进来）。
- **oneway** 表示调用 **onCallback(xx)** 方法的线程立即返回，不阻塞等待方法调用结果。

2. 服务端：需要负责提供和暴露给客户端用来注册监听回调的公用 API 接口方法

- 服务端定义了回调接口，客户端需要给服务端传递接口的实现（实现了接口实例的 reference）。因此服务端还需要将注册回调的接口暴露给客户端。
- 定义 AIDL 文件如下：

```
import com.fish.ipcserver.Student;
import com.fish.ipcserver.RemoteCallback;
interface IStudentInfo {
    // 主动获取
    Student getStudentInfo();
    // 客户端注册监听回调：观察者模式，只在第一次注册监听和数据发生变化的时候回调通知
    oneway void register(in RemoteCallback callback);
}
```

- 至此，服务端提供了两个方法：
 - 1、getStudentInfo() 客户端调用此方法主动获取学生信息。
 - 2、register(xx) 客户端调用此方法注册回调实例。

3. 服务端编写回调逻辑

```

public class StudentService extends Service {
    private Student student;
    private RemoteCallback remoteCallback;

    // 声明了 IStudentInfo 实例 (子类实例)
    private MyStudent myStudent;

    class MyStudent extends IStudentInfo.Stub {
        // BinderProxy, 这是一种写法; 还有直接返回服务端 IBinder references 的 ?
        @Override public Student getStudentInfo() throws RemoteException {
            return student;
        }
        @Override
        public void register(RemoteCallback callback) throws RemoteException {
            // 客户端注册监听回调: 客户端注册的回调实例保存到成员变量 remoteCallback
            remoteCallback = callback;
        }
        public void changeScore() { // 公用 APIs
            // 学生成绩发生改变
            student.setScore((float)(Math.random() * 100));
            try {
                if (remoteCallback != null)
                    // 调用回调实例方法, 将变化后的学生信息传递给客户端
                    remoteCallback.onCallback(student);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
    }
    @Nullable @Override public IBinder onBind(Intent intent) {
        // 将 Stub 返回给客户端; 还有直接返回服务端 IBinder references 的 ?
        return myStudent.asBinder();
    }

    @Override public void onCreate() {
        super.onCreate();
        // <<<<<<===== 这些都只是初始化的占位符
        student = new Student();
        student.setAge(19);
        student.setName(" 小明");
        myStudent = new MyStudent();
    }
}

```

- 可以看出，声明了 IStudentInfo 实例。
- 小结上面的逻辑：
 - 1、服务端声明了 Stub(桩，实际上是 **IBinder 实例 (的引用)**)，并将 **Stub (IBinder 实例的引用)** 返回给客户端。
 - 2、客户端收到 Stub(实际上是 BinderProxy)，然后转换为 IStudentInfo 接口。而该接口里声明了两个方法，分别是 getStudentInfo() 和 register(客户端监听回调函数)。
 - 3、客户端调用 register(RemoteCallback) 将监听回调注册 (传递) 给服务端。
 - 4、服务端发生变化的时候通过 RemoteCallback 通知客户端数据已经发生改变。

4. 客户端编写调用逻辑

- 分三步：
 - (1)、客户端绑定服务端 Service。

- (2)、建立连接后客户端将 IBinder 转化为 IStudentInfo 接口，并注册回调。
- (3)、客户端处理回调内容。
- 来看看代码实现：

5. (1) 绑定服务

```
// 参数 1：运行远程服务的包名
// 参数 2：远程服务全限定类名
ComponentName componentName = new ComponentName("com.fish.ipcserver", "com.fish.ipcserver.StudentService");
Intent intent = new Intent();
intent.setComponent(componentName);
// 绑定远程服务
v.getContext().bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);
```

6. (2)IBinder 转换为 IStudentInfo 接口

```
ServiceConnection serviceConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        isConnected = true;
        // 转为对应接口
        iStudentInfo = IStudentInfo.Stub.asInterface(service);
        try {
            // 注册回调
            iStudentInfo.register(remoteCallback);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
    @Override public void onServiceDisconnected(ComponentName name) {
        isConnected = false;
    }
};
```

7. (3) 客户端处理回调

```
// 声明回调：RemoteCallback 类必须是跨进程 IPC 可以公认的类或是数据结构，以确保远程与客户端可以沟通（互相认得）
RemoteCallback remoteCallback = new RemoteCallback.Stub() {
    @Override public void onCallback(Student student) throws RemoteException {
        Log.d("fish", "call back student:" + student);
        runOnUiThread(new Runnable() {
            @Override public void run() {
                // <<<<<<===== 客户端接收到远程数据后，更新其 UI 相关的内容
                Toast.makeText(IPCActivity.this, "client receive change:" + student.toString(), Toast.LENGTH_SHORT);
            }
        });
    }
};
```

- 此处收到服务端的回调后，仅仅 Toast 学生信息。

5.4.5 注意事项

- 1、自定义类型 Student.java 与 Student.aidl 需要在同一个包名下。这样学生类（学生实例）才能被跨进程多个不同的进程认得识别。
- 2、客户端与服务端定义的 aidl 文件需要在同一个包名下。通常来说，一般先定义服务端 aidl 接口，最后将这些 aidl 文件拷贝到客户端相同包名下。
- 3、bindService Intent 需要指定 ComponentName。

5.4.6 回调在四大组件里的应用

- 以 ContentProvider 为例：

- 想要获取相册数据，可以通过 ContentProvider 获取，而相册是公共的存储图片区域，其它 App 都可以往里面插入数据或者删除数据。
- 而系统也提供了监听相册变化的回调：

```

Handler handler = new Handler(Looper.getMainLooper());
ContentObserver contentObserver = new ContentObserver(handler) {
    @Override
    public void onChange(boolean selfChange) {
        // 数据变化回调
        super.onChange(selfChange);
    }
};
getContentResolver().registerContentObserver(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, true, contentObserver);

```

- 如上，通过 registerContentObserver(xx) 向系统（服务端）注册了回调接口，当有数据变化的时候服务端会调用 onChange(xx) 通知客户端。
- 不仅 ContentProvider 运用到了回调，Service、Activity、Broadcast 也用到了。

5.5 Activity 绑定服务到 ActivityManagerService 过程，所有相关源码位于 frameworks/base/ 目录下

- 经过上一小节的例子，根据打印结果，通过源码探究服务绑定过程。还是从 Activity 的 bindService 方法开始

5.5.1 ./core/java/android/content/ContextWrapper.java

```

@Override
public boolean bindService(Intent service, ServiceConnection conn, int flags) {
    return mBase.bindService(service, conn, flags); //1
}

```

- 由以上源码注释 1，通过上一篇文章深入理解 Android 之 Service 启动流程第一小节分析，mBase 指向的就是 ContextImpl 对象，接着看到它的 bindService 方法

5.5.2 ./core/java/android/app/ContextImpl.java

```

@Override
public boolean bindService(Intent service, ServiceConnection conn, int flags) {
    warnIfCallingFromSystemProcess();
    return bindServiceCommon(service, conn, flags, null, mMainThread.getHandler(), null,
        getUser()); //1
}
private boolean bindServiceCommon(Intent service, ServiceConnection conn, int flags,
    String instanceName, Handler handler, Executor executor, UserHandle user) {
    // Keep this in sync with DevicePolicyManager.bindDeviceAdminServiceAsUser.
    IServiceConnection sd; //2
    if (mPackageInfo != null) {
        if (executor != null) {
            sd = mPackageInfo.getServiceDispatcher(conn, getOuterContext(), executor, flags);
        } else {
            sd = mPackageInfo.getServiceDispatcher(conn, getOuterContext(), handler, flags); //3
        }
    } else { }
    validateServiceIntent(service);
    try {
        service.prepareToLeaveProcess(this);
        int res = ActivityManager.getService().bindIsolatedService(
            mMainThread.getApplicationThread(), getActivityToken(), service,
            service.resolveTypeIfNeeded(getApplicationContext()),
            sd, flags, instanceName, getOpPackageName(), user.getIdentityManager()); //4
        return res != 0;
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}

```

- 由以上源码注释 2 和注释 3，分为以下几点

- 首先声明了 IServiceConnection 对象，通过 ContextImpl 的 LoadApk 类型的 mPackageInfo 对象来封装获取 IServiceConnection 对象，由前面注释 1 处得知 executor 对象传入为 null，所以调用的是注释 2；
- 在构造 IServiceConnection 时，首先传入了开始绑定服务的 ServiceConnection 接口引用，其次是 ContextImpl 的 mOuterContext，它是 Context 类型，指向的就是当前需要绑定服务的 Activity 组件；
- 其次第三个参数为 Handler 对象，它指向就是 ActivityThread 中代表应用程序主线程的 H 对象，方便后续将服务绑定完成回调到主线程中的 ServiceConnection 接口回调方法
- 第四个参数则是绑定服务的 flag 参数 BIND_AUTO_CREATE

- 接着看到 LoadApk 的 getServiceDispatcher 方法

5.5.3 ./core/java/android/app/LoadedApk.java

```

@UnsupportedAppUsage
public final IServiceConnection getServiceDispatcher(
    ServiceConnection c, Context context, Handler handler, int flags) {
    return getServiceDispatcherCommon(c, context, handler, null, flags);
}

private IServiceConnection getServiceDispatcherCommon(
    ServiceConnection c, Context context, Handler handler, Executor executor, int flags) {
    synchronized (mServices) {
        LoadedApk.ServiceDispatcher sd = null;
        // <<<<<<< 这里借助图表，来保存所有不同 ServiceConnection，以及他们所对应的 loadedApk.ServiceDispatcher
        // 有点儿类似（不是同一个）：自己先前有所接触的同一个远程服务，对应多个不同的客户端连接监听的情况
        ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher> map = mServices.get(context); // 1: 每种绑定连接，和建立连接
        if (map != null) {
            sd = map.get(c);
        }
        if (sd == null) { // map 中还没有这样的数据
            if (executor != null) {
                sd = new ServiceDispatcher(c, context, executor, flags);
            } else {
                sd = new ServiceDispatcher(c, context, handler, flags); // 2 不存在则注释 2 处新建 ServiceDispatcher 对象
            }
            if (map == null) {
                map = new ArrayMap<>();
                mServices.put(context, map);
            }
            map.put(c, sd); // 3:
        } else {
            sd.validate(context, handler, executor);
        }
        return sd.getServiceConnection(); // 最终通过 ServiceDispatcher 对象来获取 IServiceConnection
    }
}

```

- 由以上源码，

- 注释 1 处通过 ArrayMap（类似 HashMap）类型的 mServices 获取与对应服务绑定 ServiceConnectoin c 的 Activity 组件的 ServiceDispatcher 对象，
- 不存在则注释 2 处新建 ServiceDispatcher 对象，并将刚刚传入的数据作为构造方法参数，
- 最终通过 ServiceDispatcher 对象来获取 IServiceConnection，

- 继续看到 LoadApk 的内部静态类 ServiceDispatcher

5.5.4 ./core/java/android/app/LoadedApk.java

```
static final class ServiceDispatcher {
    private final ServiceDispatcher.InnerConnection mServiceConnection;

    @UnsupportedAppUsage
    private final ServiceConnection mConnection;
    // @UnsupportedAppUsage(maxTargetSdk = Build.VERSION_CODES.P, trackingBug = 115609023)
    // private final Context mContext;
    // private final Handler mActivityThread;
    // private final Executor mActivityExecutor;
    // private final ServiceConnectionLeaked mLocation;
    // private final int mFlags;
    private static class InnerConnection extends IServiceConnection.Stub {
        @UnsupportedAppUsage
        final WeakReference<LoadedApk.ServiceDispatcher> mDispatcher;

        InnerConnection(LoadedApk.ServiceDispatcher sd) {
            mDispatcher = new WeakReference<LoadedApk.ServiceDispatcher>(sd);
        }
    }

    @UnsupportedAppUsage
    ServiceDispatcher(ServiceConnection conn,
    // 最终将 ServiceConnection, 指向 Activity 组件主线程 Handlerd 对象和 Context 包装到 ServiceDispatcher 对象中
    Context context, Handler activityThread,
    int flags) {
        // 在内部实现了 IServiceConnection 接口的 InnerConnection
        mServiceConnection = new InnerConnection(this);

        // mConnection = conn;
        // mContext = context;
        // mActivityThread = activityThread;
        // mActivityExecutor = null;
        // mLocation = new ServiceConnectionLeaked(null);
        // mLocation.fillInStackTrace();
        // mFlags = flags;
    }
    // 通过 getIServiceConnection 方法返回将其赋值给最开始创建的 sd 对象
    // 也就是可以和远程服务端进行 IBinder 通信的本地引用
    @UnsupportedAppUsage
    IServiceConnection getIServiceConnection() {
        return mServiceConnection;
    }
}
```

- 由以上源码，最终将 ServiceConnection，指向 Activity 组件主线程 Handler 对象和 Context 包装到 ServiceDispatcher 对象中，并在内部实现了 IServiceConnection 接口的 InnerConnection，最终通过 getIServiceConnection 方法返回将其赋值给最开始创建的 sd 对象，也就是可以和远程服务进行 Binder 通信的本地引用
- 接着我们回到 bindServiceCommon 方法的注释 4，获取当前进程的 IApplicationThread 对象等，和刚刚封装获取的 IServiceConnection 一并作为参数请求到 AMS 的 bindIsolatedService 方法来绑定 Service。
- Activity 绑定服务调用到 AMS 过程时序图

5.6 AMS 到 ActivityThread 启动绑定 Service 过程

5.6.1 服务与 Activity 组件建立绑定关系：理解起来相对复杂，今天暂时不根了，改天头脑清醒的时候再回来看这个

- 原文链接：<https://juejin.cn/post/6844904153043451917>
- 接着上一小节，分析到 ActivityManagerService 方法，接着跟进
- ./services/core/java/com/android/server/am/ActivityManagerService.java

```

public int bindIsolatedService(IApplicationThread caller, IBinder token, Intent service,
        String resolvedType, IServiceConnection connection, int flags, String instanceName,
        String callingPackage, int userId) throws TransactionTooLargeException {
    enforceNotIsolatedCaller("bindService");
    .....
    synchronized(this) {
        return mServices.bindServiceLocked(caller, token, service,
                resolvedType, connection, flags, instanceName, callingPackage, userId); //1
    }
}

```

- 由以上源码注释 1，同样是调用了用来管理 Service 的 ActiveServices 类的 bindServiceLocked 方法，

5.6.2 ./services/core/java/com/android/server/am/ActiveServices.java

```

int bindServiceLocked(IApplicationThread caller, IBinder token, Intent service,
        String resolvedType, final IServiceConnection connection, int flags,
        String instanceName, String callingPackage, final int userId)
        throws TransactionTooLargeException {
    .....
    final ProcessRecord callerApp = mAm.getRecordForAppLocked(caller); //1

    ActivityServiceConnectionsHolder<ConnectionRecord> activity = null;
    if (token != null) {
        activity = mAm.mAtmInternal.getServiceConnectionsHolder(token); //2
        .....
    }
    .....
    ServiceLookupResult res =
        retrieveServiceLocked(service, instanceName, resolvedType, callingPackage,
            Binder.getCallingPid(), Binder.getCallingUid(), userId, true,
            callerFg, isBindExternal, allowInstant); //3

    ServiceRecord s = res.record; //4
    .....
    AppBindRecord b = s.retrieveAppBindingLocked(service, callerApp); //5
    ConnectionRecord c = new ConnectionRecord(b, activity,
        connection, flags, clientLabel, clientIntent,
        callerApp.uid, callerApp.processName, callingPackage); //6
    IBinder binder = connection.asBinder();
    s.addConnection(binder, c);
    b.connections.add(c);
    if (activity != null) {
        activity.addConnection(c);
    }
    b.client.connections.add(c);
    c.startAssociationIfNeeded();
    .....
    ArrayList<ConnectionRecord> clist = mServiceConnections.get(binder);
    if (clist == null) {
        clist = new ArrayList<>();
        mServiceConnections.put(binder, clist);
    }
    clist.add(c); //7

    /**
     * 8
     */
    if ((flags & Context.BIND_AUTO_CREATE) != 0) {
        s.lastActivity = SystemClock.uptimeMillis();
        if (bringUpServiceLocked(s, service.getFlags(), callerFg, false,
            permissionsReviewRequired) != null) {
            return 0;
        }
    }
    .....
    if (s.app != null && b.intent.received) {
        // Service is already running, so we can immediately
        // publish the connection.
        try {
            c.conn.connected(s.name, b.intent.binder, false); //9
        } catch (Exception e) {
    }
}

```

```

    ....
}

// If this is the first app connected back to this binding,
// and the service had previously asked to be told when
// rebound, then do so.
if (b.intent.apps.size() == 1 && b.intent.doRebind) {
    requestServiceBindingLocked(s, b.intent, callerFg, true);
}
else if (!b.intent.requested) {
    requestServiceBindingLocked(s, b.intent, callerFg, false); //10
}
getServiceMapLocked(s.userId).ensureNotStartingBackgroundLocked(s);
finally {
    Binder.restoreCallingIdentity(origId);
}
return 1;
}

```

- 由以上源码，
- 注释 1 处，通过上一小节获取的 IApplicationThread 对象得到正在请求绑定服务的 Activity 组件所在的应用程序进程
- 注释 2 处根据 token 也就是 binder 引用获取对应的 ActivityServiceConnectionsHolder，也就是需要绑定服务的 Activity 组件
- 注释 3 和 4 处通过 Intent 类型的 service 参数最终获取了描述需要绑定的 Service 对象描述 ServiceRecord
- 注释 5 处则调用了 ServiceRecord 的 retrieveAppBindingLocked 方法
- 接着看到 retrieveAppBindingLocked 方法

5.6.3 ./services/core/java/com/android/server/am/ServiceRecord.java

```

final ArrayMap<Intent.FilterComparison, IntentBindRecord> bindings
    = new ArrayMap<Intent.FilterComparison, IntentBindRecord>();
public AppBindRecord retrieveAppBindingLocked(Intent intent,
    ProcessRecord app) {
    Intent.FilterComparison filter = new Intent.FilterComparison(intent);
    IntentBindRecord i = bindings.get(filter);
    if (i == null) { //1
        i = new IntentBindRecord(this, filter);
        bindings.put(filter, i);
    }
    AppBindRecord a = i.apps.get(app); //2
    if (a != null) {
        return a;
    }
    a = new AppBindRecord(this, i, app);
    i.apps.put(app, a);
    return a;
}

```

- 远程服务是一对多的关系，可以和多个应用程序进程绑定的，所以，每个绑定服务的应该程序进程 ProcessRecord 在服务 ServiceRecord 中被描述成了 IntentBindRecord，并保存在 ArrayMap 类型的 bindings 的 map 中，
- 由以上源码注释 1，通过的 Intent.FilterComparison 来查看当前应用程序进程是否绑定了服务，没有绑定则新建 IntentBindRecord 并保存
- 源码注释 2 处获取 IntentBindRecord 中是否存在 AppBindRecord，它服务与其应用程序进程之间的关联对象，如果存在也说明之前就绑定过，否则新建 AppBindRecord 并返回。

- 接着回看前一个步骤 ActiveServices 的 bindServiceLocked 方法注释 6，此时又把 AppBindRecord、描述 Activity 组件的 ActivityServiceConnectionsHolder 和 IServiceConnection 封装成了代表 Activity 组件与远程服务 ServiceRecord 建立了连接，描述对象为 ConnectionRecord，前面也说过服务绑定是一对多关系，所以服务也能被多个 Activity 绑定，也就是会有多个 ConnectionRecord 对象，这里就将它们全部放入 ArrayList 保存，也就是注释 7 处的clist，并且以 IServiceConnection 引用为 key，值为 ArrayList 保存到 AMS 的 ArrayMap 中保存，也就是 mServiceConnections，以待后续服务启动回调绑定
- 远程 Service 进程 fork
- 接着前一个步骤 ActiveServices 的 bindServiceLocked 方法注释 8，根据之前的分析，绑定服务设置的 flag 为 BIND_AUTO_CREATE，所以该判断条件成立，则调用 bringUpServiceLocked 方法。(ActiveServices 的 bindServiceLocked 方法注释 10 将在下一小节继续分析)

5.6.4 ./services/core/java/com/android/server/am/ActiveServices.java

```

private String bringUpServiceLocked(ServiceRecord r, int intentFlags, boolean execInFg,
    boolean whileRestarting, boolean permissionsReviewRequired)
throws TransactionTooLargeException {

    .....
    ProcessRecord app;
    HostingRecord hostingRecord = new HostingRecord("service", r.instanceName);
    if (!isolated) {
        app = mAm.getProcessRecordLocked(procName, r.appInfo.uid, false);
        .....
        if (app != null && app.thread != null) {
            try {
                app.addPackage(r.appInfo.packageName, r.appInfo.longVersionCode, mAm.mProcessStats);
                realStartServiceLocked(r, app, execInFg); //1
                return null;
            }
            .....
        } else {
            // If this service runs in an isolated process, then each time
            // we call startProcessLocked() we will get a new isolated
            // process, starting another process if we are currently waiting
            // for a previous process to come up. To deal with this, we store
            // in the service any current isolated process it is running in or
            // waiting to have come up.
            app = r.isolatedProc;
            if (WebViewZygote.isMultiprocessEnabled()
                && r.serviceInfo.packageName.equals(WebViewZygote.getPackageName())) {
                hostingRecord = HostingRecord.byWebviewZygote(r.instanceName);
            }
            if ((r.serviceInfo.flags & ServiceInfo.FLAG_USE_APP_ZYGOTE) != 0) {
                hostingRecord = HostingRecord.byAppZygote(r.instanceName, r.definingPackageName,
                    r.definingUid); //2
            }
        }
    }

    // Not running -- get it started, and enqueue this service record
    // to be executed when the app comes up.
    if (app == null && !permissionsReviewRequired) {
        if ((app=mAm.startProcessLocked(procName, r.appInfo, true, intentFlags,
            hostingRecord, false, isolated, false)) == null) { //3
            .....
        }
        .....
        if (!mPendingServices.contains(r)) {
            mPendingServices.add(r);
        } //4
    }
}

```

- 如以上源码，文章开头例子使用的远程服务，也就是服务启动进程与 Activity 进程不同，所以在注释 2 处获取启动进程描述对象 HostingRecord，如果没有启动，则通过注释 3

处调用 ActivityManagerService.startProcessLocked 启动 Service 进程，并在注释 4 处保存需要启动的服务描述 ServiceRecord，进程启动 fork Zygote 进程可查看前面文章（深入理解 Android 之应用程序进程启动），应用进程 fork 成功后会调用应用程序进程的 ActivityThread 的 main 方法，最后注释 4 会把待启动的 Service 描述 ServiceRecord 保存到 mPendingServices(ArrayList)，以便后续应用进程启动后使用，接着看到 ActivityThread 的 main 方法。

5.6.5 ./core/java/android/app/ActivityThread.java

```
public static void main(String[] args) {
    .....
    ActivityThread thread = new ActivityThread();
    thread.attach(false, startSeq); //1
    .....
}

private void attach(boolean system, long startSeq) {
    sCurrentActivityThread = this;
    mSystemThread = system;
    if (!system) {
        android.ddm.DdmHandleAppName.setAppName("<pre-initialized>",
            UserHandle.myUserId());
        RuntimeInit.setApplicationObject(mAppThread.asBinder());
        final IActivityManager mgr = ActivityManager.getService(); //2
        try {
            mgr.attachApplication(mAppThread, startSeq); //3
        } catch (RemoteException ex) {
            throw ex.rethrowFromSystemServer();
        }
    }
    .....
}
```

- 由以上源码 1，首先创建了 ActivityThread 对象，并调用了 ActivityThread 的私有方法 attach，在该方法中注释 2 应该很熟悉，获取了与 AMS 进行 binder 通信的本地引用，然后注释 3 处调用 AMS 的 attachApplication 与 AMS 进行通信，接着往下看到 AMS 的 attachApplication 方法。

5.6.6 ./services/core/java/com/android/server/am/ActivityManagerService.java

```
@Override
public final void attachApplication(IApplicationThread thread, long startSeq) {
    synchronized (this) {
        attachApplicationLocked(thread, callingPid, callingUid, startSeq); //1
        Binder.restoreCallingIdentity(origId);
    }
}
```

- 由以上源码注释 1 处 ActivityManagerService 继续调用了 attachApplicationLocked，接着往下看

5.6.7 ./services/core/java/com/android/server/am/ActivityManagerService.java

```
@GuardedBy("this")
private final boolean attachApplicationLocked(IApplicationThread thread,
    int pid, int callingUid, long startSeq) {

    .....
    // Find any services that should be running in this process...
    if (!badApp) {
        try {
            didSomething |= mServices.attachApplicationLocked(app, processName);
            checkTime(startTime, "attachApplicationLocked: after mServices.attachApplicationLocked");
        } catch (Exception e) {
            .....
        }
    }
}
```

```
.....
```

- 由以上源码，attachApplicationLocked 方法本身逻辑是很多的，这里先忽略大部分不是本文分析的代码，直接看到注释 1 处，从因为注释也可以看出是找到应该在此进程中运行的任何服务，从而调用了 ActiveServices 的 attachApplicationLocked 方法，接着往下看。
- 远程 Service 组件创建

5.6.8 ./services/core/java/com/android/server/am/ActiveServices.java

```
boolean attachApplicationLocked(ProcessRecord proc, String processName)
    throws RemoteException {
    boolean didSomething = false;
    // Collect any services that are waiting for this process to come up.
    if (mPendingServices.size() > 0) {
        ServiceRecord sr = null;
        try {
            for (int i=0; i<mPendingServices.size(); i++) {
                sr = mPendingServices.get(i); //1
                if (proc != sr.isolatedProc && (proc.uid != sr.appInfo.uid
                    || !processName.equals(sr.processName))) {
                    continue;
                }
                mPendingServices.remove(i);
                i--;
                proc.addPackage(sr.appInfo.packageName, sr.appInfo.longVersionCode,
                    mAm.mProcessStats);
                realStartServiceLocked(sr, proc, sr.createdFromFg); //2
                didSomething = true;
                .....
            }
        } catch (RemoteException e) {
            .....
        }
    }
    .....
```

- 由以上源码注释 1，遍历之前保存待启动 Service 组件的 **mPendingServices(ArrayList)** 集合，获取 ServiceRecord，注释 2 处调用 realStartServiceLocked 方法，之后启动 Service 过程和上一篇文章深入理解 Android 之 Service 启动流程基本相同，最终调用到 Service 的 OnCreate 方法完成 Service 组件启动创建，这里不再进行展开。
- AMS 到 ActivityThread 启动绑定 Service 过程时序图
- 绑定 Service 组件的 Activity 组件的 ServiceConnection 回调
- 本小节接着回到上一节 ActiveServices 注释 10 的 bindServiceLocked 方法

5.6.9 ./services/core/java/com/android/server/am/ActiveServices.java

```
int bindServiceLocked(IApplicationThread caller, IBinder token, Intent service,
    String resolvedType, final IServiceConnection connection, int flags,
    String instanceName, String callingPackage, final int userId)
    throws TransactionTooLargeException {

    .....
    if (s.app != null && b.intent.received) {
        // Service is already running, so we can immediately
        // publish the connection.
        try {
            c.conn.connected(s.name, b.intent.binder, false); //1
        } catch (Exception e) {
            Slog.w(TAG, "Failure sending service " + s.shortInstanceName
                + " to connection " + c.conn.asBinder()
                + " (in " + c.binding.client.processName + ")", e);
        }
    }
}
```

```

        }
        // If this is the first app connected back to this binding,
        // and the service had previously asked to be told when
        // rebound, then do so.
        if (b.intent.apps.size() == 1 && b.intent.doRebind) {
            requestServiceBindingLocked(s, b.intent, callerFg, true);
        }
    } else if (!b.intent.requested) {
        requestServiceBindingLocked(s, b.intent, callerFg, false); //2
    }
    getServiceMapLocked(s.userId).ensureNotStartingBackgroundLocked(s);
} finally {
    Binder.restoreCallingIdentity(origId);
}
return 1;
}

```

- 如上代码所示，注释 1 处如果服务已经启动，则可以直接发布链接。否则调用注释 2 处的方法，接着看 ActiveServices 的 requestServiceBindingLocked 方法

5.6.10 ./services/core/java/com/android/server/am/ActiveServices.java

```

private final boolean requestServiceBindingLocked(ServiceRecord r, IntentBindRecord i,
    boolean execInFg, boolean rebind) throws TransactionTooLargeException {
    ...
    if ((!i.requested || rebind) && i.apps.size() > 0) {
        try {
            bumpServiceExecutingLocked(r, execInFg, "bind");
            r.app.forceProcessStateUpTo(ActivityManager.PROCESS_STATE_SERVICE);
            r.app.thread.scheduleBindService(r, i.intent.getIntent(), rebind,
                r.app.getReportedProcState()); //1
            if (!rebind) {
                i.requested = true;
            }
            i.hasBound = true;
            i.doRebind = false;
        } catch (TransactionTooLargeException e) {
            ...
        } catch (RemoteException e) {
            ...
        }
    }
    return true;
}

```

- 由以上代码注释 1，ServiceRecord 中保存的就是 Activity 组件所在应用程序进程的 ActivityThread 内部类 IApplicationThread 的实现类 ApplicationThread 引用，我们接着看到 ApplicationThread 的 scheduleBindService 方法

5.6.11 ./core/java/android/app/ActivityThread.java

```

public final void scheduleBindService(IBinder token, Intent intent,
    boolean rebind, int processState) {
    updateProcessState(processState, false);
    BindServiceData s = new BindServiceData();
    s.token = token;
    s.intent = intent;
    s.rebind = rebind;
    if (DEBUG_SERVICE)
        Slog.v(TAG, "scheduleBindService token=" + token + " intent=" + intent + " uid="
            + Binder.getCallingUid() + " pid=" + Binder.getCallingPid());
    sendMessage(H.BIND_SERVICE, s); //1
}

```

- 由以上代码，还是熟悉的操作，这里调用了 ActivityThread 中代表 Android 主线程处理的内部 Handler 类 H 发送了 BIND_SERVICE 消息，将 ServiceRecord 等数据封装成了 BindServiceData 对象，继续往下看

5.6.12 ./core/java/android/app/ActivityThread.java

```
class H extends Handler {
    .....
    public void handleMessage(Message msg) {
        .....
        switch (msg.what) {
            case BIND_SERVICE:
                Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "serviceBind");
                handleBindService((BindServiceData)msg.obj); //1
                Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
                break;
        }
    }
}
```

- 由以上代码注释 1，很清晰，Handler 在主线程处理消息调用了 ActivityThread 的 handleBindService 方法

5.6.13 ./core/java/android/app/ActivityThread.java

```
private void handleBindService(BindServiceData data) {
    Service s = mServices.get(data.token);//1
    .....
    if (s != null) {
        try {
            data.intent.setExtrasClassLoader(s.getClassLoader());
            data.intent.prepareToEnterProcess();
            try {
                if (!data.rebind) {
                    IBinder binder = s.onBind(data.intent); //2
                    ActivityManager.getService().publishService(
                        data.token, data.intent, binder); //3
                } else {
                    s.onRebind(data.intent);
                    ActivityManager.getService().serviceDoneExecuting(
                        data.token, SERVICE_DONE_EXECUTING_ANON, 0, 0);
                }
            } catch (RemoteException ex) {
                throw ex.rethrowFromSystemServer();
            }
        } catch (Exception e) {
            .....
        }
    }
}
```

- 前面已经讨论了远程 Service 组件目前已经启动，并且在 ActivityThread 的创建启动 Service 方法 handleCreateService 中将启动的 Service 实例加入到了 ArrayMap 类型的 mServices 中，
- 注释 1 处则根据 ServiceRecord 获取对应的 Service 组件
- 注释 2 处回调了 Servcie 组件的 onBind 方法，将 Service 远程服务的 IBinder 引用获取返回，也就是文章开始 Demo 中的 IRemoteService.Stub 类型的 mBinder
- 注释 3 处再次调用 AMS 服务的 publishService 方法传递刚刚获取的 mBinder 引用数据
- 接着看 AMS 的 publishService 方法，如下所示

5.6.14 ./services/core/java/com/android/server/am/ActivityManagerService.java

```
public void publishService(IBinder token, Intent intent, IBinder service) {
    // Refuse possible leaked file descriptors
    .....
    synchronized(this) {
        if (!(token instanceof ServiceRecord)) {
            throw new IllegalArgumentException("Invalid service token");
        }
    }
}
```

```

        mServices.publishServiceLocked((ServiceRecord)token, intent, service); //1
    }
}

```

- 由以上源码，还是老样子继续调用了 ActiveServices 的 publishServiceLocked 方法

5.6.15 ./services/core/java/com/android/server/am/ActiveServices.java

```

void publishServiceLocked(ServiceRecord r, Intent intent, IBinder service) {
    final long origId = Binder.clearCallingIdentity();
    try {
        .....
        if (r != null) {
            Intent.FilterComparison filter
                = new Intent.FilterComparison(intent); //1
            IntentBindRecord b = r.bindings.get(filter); //2
            if (b != null && !b.received) {
                b.binder = service; //3
                b.requested = true; //4
                b.received = true; //5
            }
            ArrayMap<IBinder, ArrayList<ConnectionRecord>> connections = r.getConnections(); //6
            for (int conni = connections.size() - 1; conni >= 0; conni--) {
                ArrayList<ConnectionRecord> clist = connections.valueAt(conni);
                for (int i=0; i<clist.size(); i++) {
                    ConnectionRecord c = clist.get(i);
                    .....
                    try {
                        c.conn.connected(r.name, service, false); //7
                    } catch (Exception e) {
                        .....
                    }
                }
            }
        }
        .....
    } finally {
        .....
    }
}

```

- 由以上源码，看完似乎又找到了豁然开朗的感觉，结合前面第二小节服务与 Activity 组件建立绑定关系的分析
- 注释 1 处还是根据 Intentl 来获取 Intent.FilterComparison
- 注释 2 处根据第一步中获取的 Intent.FilterComparison 从 ServiceRecord 的 bindings 获取 IntentBindRecord，这都是在前面第二小节分析中创建的，接着注释 3 将刚刚通过 Service 的 OnBind 方法获取的 IBinder 引用赋值给 IntentBindRecord，并在注释 4 和注释 5 处将值设置为 true，避免后续的 AMS 重复请求
- 前面说过 Service 绑定的一对多属性，所以他们都是使用了同一个 key 也就是 InnerConnection，之前我们分析中是将其封装在了描述 Actiivty 组件与 Service 组件之间联系的描述类 ConnectionRecord 中
- 接着从泛型为 ConnectionRecord 的 ArrayList 结合最后一个开始遍历获取 ConnectionRecord 对象，然后获取与之对应的 InnerConnection，然后注释 7 处调用它的 connected 方法，接着我们再次看回 LoadedApk 类的 InnerConnection

5.6.16 ./core/java/android/app/LoadedApk.java

```

static final class ServiceDispatcher {
private final ServiceDispatcher.InnerConnection mIServiceConnection; //1

private static class InnerConnection extends IServiceConnection.Stub {
    @UnsupportedAppUsage

```

```

        final WeakReference<LoadedApk.ServiceDispatcher> mDispatcher;
        InnerConnection(LoadedApk.ServiceDispatcher sd) {
            mDispatcher = new WeakReference<LoadedApk.ServiceDispatcher>(sd);
        }
        public void connected(ComponentName name, IBinder service, boolean dead)
            throws RemoteException {
            LoadedApk.ServiceDispatcher sd = mDispatcher.get();
            if (sd != null) {
                sd.connected(name, service, dead); //2
            }
        }
    }
}

```

- 在文章第一小节分析 ContextImpl 的 bindServiceCommon 方法就已经创建了 IServiceConnection 的实现类 InnerConnection 并封装到了 LoadedApk 的 ServiceDispatcher，以上源码注释 2 实际调用的是 ServiceDispatcher 的 connected，接着往下看

5.6.17 ./core/java/android/app/LoadedApk.java

```

static final class ServiceDispatcher {
    ...
    public void connected(ComponentName name, IBinder service, boolean dead) {
        if (mActivityExecutor != null) {
            mActivityExecutor.execute(new RunConnection(name, service, 0, dead));
        } else if (mActivityThread != null) {
            mActivityThread.post(new RunConnection(name, service, 0, dead));//2
        } else {
            doConnected(name, service, dead);
        }
        ...
    }
    ...
}

```

- 根据前面的分析，mActivityThread 指向的 ActivityThread 的 handler 成员 H，也就是说这里可以给 Activity 组件所在应用程序进程的主线程发送消息，让其运行 Runnable 类型的 RunConnection，也就是说 run 方法在主线程运行，接着看 run 方法逻辑

5.6.18 ./core/java/android/app/LoadedApk.java

```

private final ServiceConnection mConnection;
private final class RunConnection implements Runnable {
    RunConnection(ComponentName name, IBinder service, int command, boolean dead) {
        mName = name;
        mService = service;
        mCommand = command;
        mDead = dead;
    }
    public void run() {
        if (mCommand == 0) {
            doConnected(mName, mService, mDead); //1
        } ...
    }
    ...
}

public void doConnected(ComponentName name, IBinder service, boolean dead) {
    ServiceDispatcher.ConnectionInfo old;
    ServiceDispatcher.ConnectionInfo info;
    ...
    // If there is a new viable service, it is now connected.
    if (service != null) {
        mConnection.onServiceConnected(name, service); //2
    } else {
        // The binding machinery worked, but the remote returned null from onBind().
        mConnection.onNullBinding(name);
    }
}

```

- 如以上源码，注释 1 处调用了 doConnected 方法，它传入了一个 mService 对象，经前面分析已经很清晰，它就是指向远程 Service 组件中组件的 Binder 对象，也可以理解成 Demo 例子中的 IRemoteService.Stub 类型 mBinder，接着在注释 2 处调用之前在 Activity 组件创建的 ServiceConnection 实现类引用，并调用 onServiceConnected 将远程服务 mBinder 回调给 Activity 组件。
- 绑定 Service 组件的 Activity 组件的 ServiceConnection 回调时序图

5.7 总结：

- 和应用程序进程启动一样，服务绑定过程也涉及了四个进程，其中最核心的就是 AMS 所在的 SystemServer 进程
- 涉及进程间通信方式有两种，一种是 socket 方式 fork Zygote 进程，另一种是 Binder，线程间通信方式为 Handler
- 组件与服务绑定关系被描述成 ConnectionRecord，它包含的 IServiceConnection 指向与服务绑定的组件，InnerConnection 实现了 IServiceConnection，ServiceDispatcher 包含 InnerConnection、ConnectionRecord、ServiceConnection，最终通过绑定组件的主线程 Handler.post 将 onServiceConnected 回调方法运行在绑定服务组件的主线程。
- ContextImpl 和 ContextWrapper 之间联系使用的是静态代理模式，ContextWrapper 和 ContextThemeWrapper 使用了装饰模式
- 最后：经过分析，服务绑定过程还是步骤比较多的，需要一些耐心，在阅读源码的过程中也能清晰看到服务绑定流程方法调用顺序和文章开头 Demo 执行顺序是一样的。