转载请注明链接: https://blog.csdn.net/feather_wch/article/details/79415202

> 本文是Binder知识点的总结
> 主要分为两个部分：
> 1、讲解Binder机制的流程图、步骤和实现代码
> 2、从Android整体架构层面讲解Binder IPC相关原理和知识点

Anndroid Binder机制详解

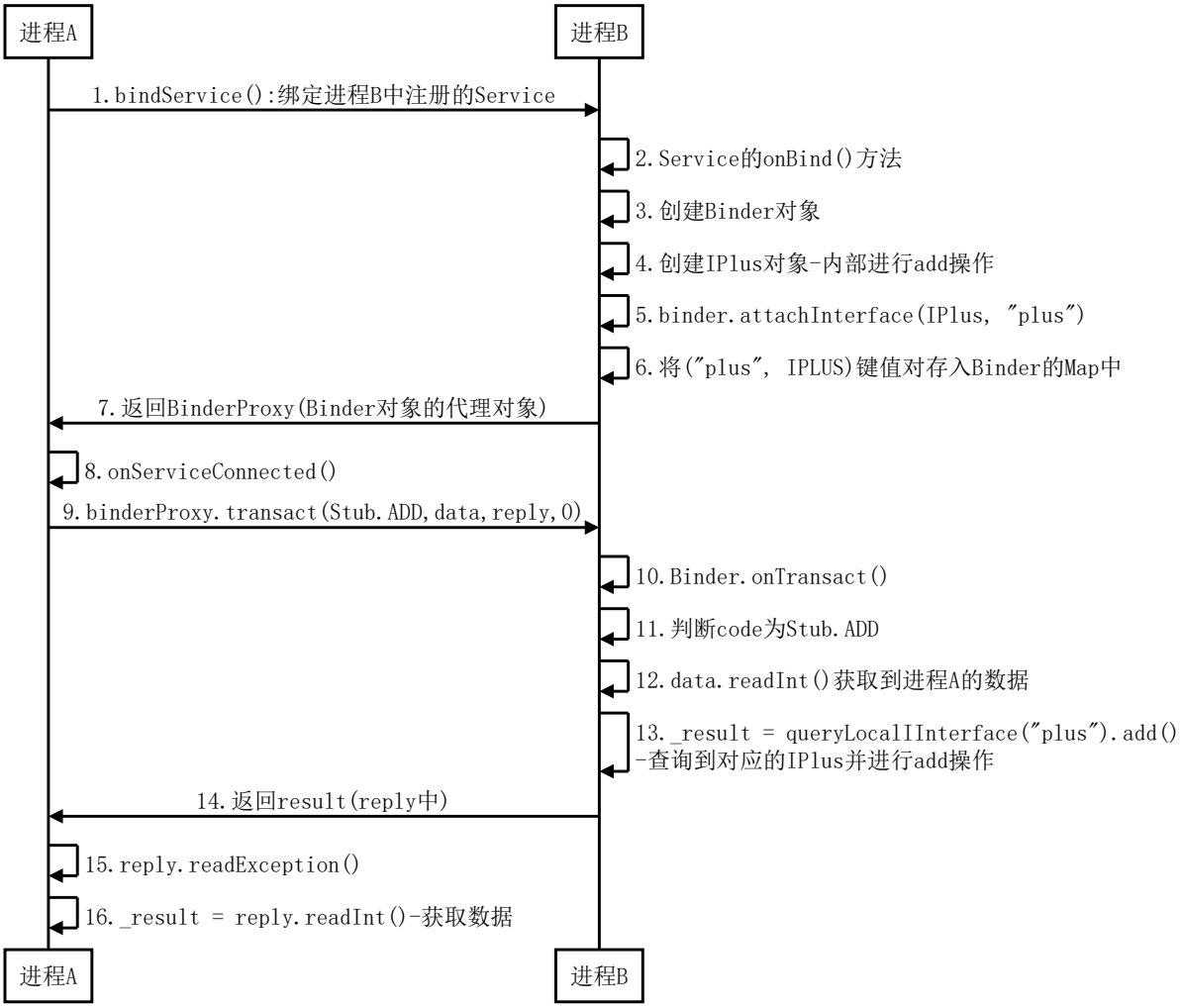
版本：2019/3/14-1

-参考自《Android开发艺术探索》和(https://www.jianshu.com/p/1eff5a13000d)
-参考自：https://www.jianshu.com/p/bdef9e3178c9

Android如何通过Binder机制提供跨进程通信的解决方案?

1. 进程A通过bindService方法去绑定在进程B中注册的一个service
2. 系统收到进程A的bindService请求后，会调用进程B中相应service的onBind方法，该方法返回一个特殊对象，系统会接收到这个特殊对象
3. 系统为这个特殊对象生成一个代理对象，再将这个代理对象返回给进程A
4. 进程A在ServiceConnection回调的onServiceConnected方法中接收该代理对象，依靠这个代理对象的帮助，就可以解决进程间通信问题

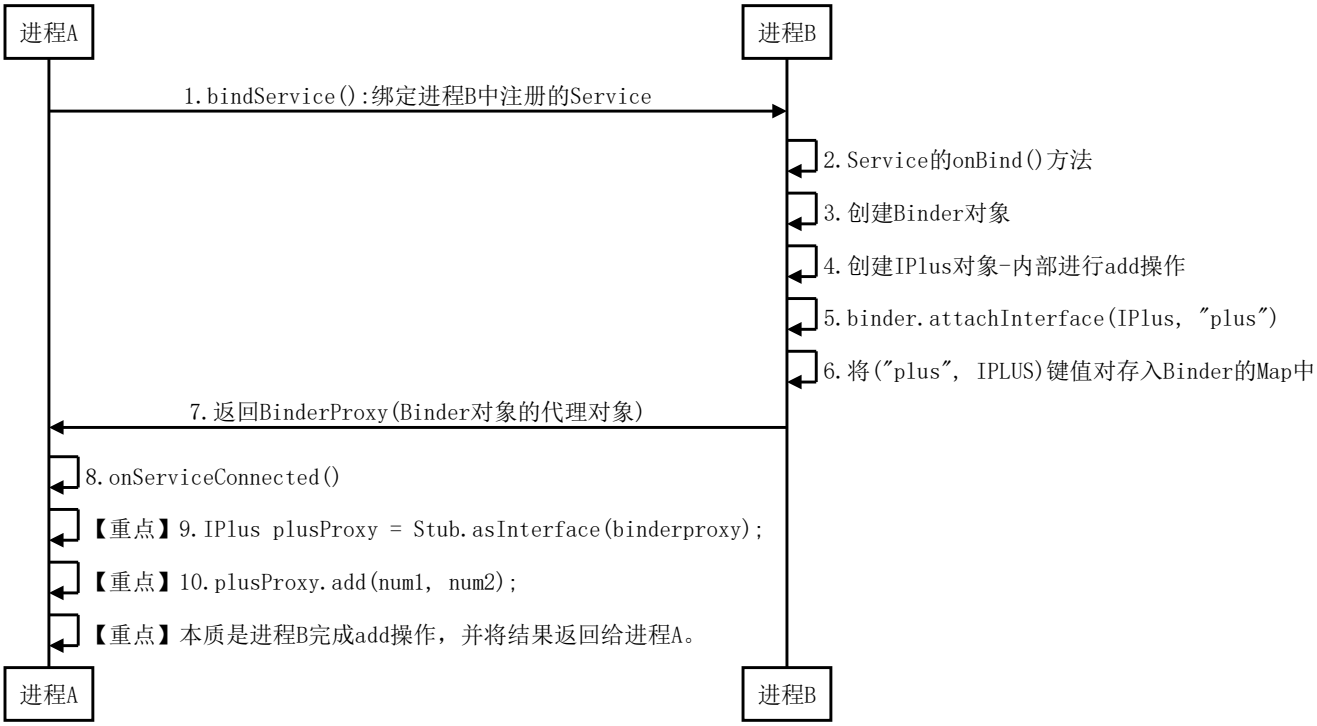
1-总结的流程图



代理的优化：将 步骤9~16 都封装到 PlusProxy 中:

```
public class PlusProxy implements IPlus {
          private IBinder  binderproxy ;
          public PlusProxy(IBinder binderproxy){
                this.binderproxy = binderproxy ;
          }
          public int add (int a ,int b ){
              android.os.Parcel data = android.os.Parcel.obtain();
              android.os.Parcel reply = android.os.Parcel.obtain();
              data.writeInterfaceToken("PLUS TWO INT");
              data.writeInt(a);
              data.writeInt(b);
              binderproxy.transact(1, data, reply, 0);
              int _result;
              reply.readException();
              _result = reply.readInt();
              return _result ;
          }
}
```

给进程A一种假象-进程A获取到了进程B中的IPlus对象，并直接进行了 add() 操作。
最终的流程图如下:



- `Stub.asInterface()` 会先查找本地接口是否存在，判断是否是本地调用，如果是则直接返回 IReporter 的对象，否则返回 PlusProxy 对象。

2-Binder机制步骤分析:

step 1：进程B创建Binder对象

进程B需要实现的 特殊对象(service的onBind方法返回的对象) 需要有两个特性：

1. 具有完成特定任务的能力(例如a+b=c)
2. 具有能被跨进程传输的能力
   这就是**Binder类对象**

Binder类对象如何具有这两个关键能力

- Binder具有被跨进程传输的能力是因为它实现了 `IBinder` 接口。系统会为每个实现了该接口的对象提供跨进程传输。
- Binder具有的完成特定任务的能力是通过调用 `attachInterface()` 而持有的 `IInterface` 对象来实现的

> `attachInterface()` 会将 （ `descriptor`，`plus` ）作为 （ `key,value` ）键值对存入 `Binder` 对象中的一个 `Map<String,IInterface>` 对象中， `Binder` 对象可通过 `attachInterface` 方法持有一个 `IInterface` 对象（即 `plus` ）的引用，并依靠它获得完成特定任务的能力。 `queryLocalInterface` 方法可以认为是根据 `key` 值（即参数 `descriptor` ）查找相应的 `IInterface` 对象。 `onTransact` 方法暂时不用管，后面会讲到。

```
public class Binder implement IBinder{
        void attachInterface(IInterface plus, String descriptor)
        IInterface queryLocalInterface(Stringdescriptor) //从IBinder中继承而来
        boolean onTransact(int code, Parcel data, Parcel reply, int flags)//暂时不用管，后面会讲。
        ......
        final class BinderProxy implements IBinder {
        ......//Binder的一个内部类，暂时不用管，后面会讲。
        }
}
```

我们来实现 `IInterface` 和 `Binder` 对象，概略代码如下：

```
 public interface IPlus extends IInterface {
         public int add(int a,int b);
}

public class Stub extends Binder {
          @Override
          boolean onTransact(int code, Parcel data, Parcel reply, int flags){
          ......//这里我们覆写了onTransact方法，暂时不用管，后面会讲解。
          }
          ......
}
```

实现IInterface对象plus，并将plus根据键值对存入到 Binder 内部的Map中，此时该 Binder 对象持有了该特殊对象 plus

```
IInterface plus = new IPlus(){//匿名内部类
         public int add(int a,int b){//定制我们自己的相加方法
             return a+b;
         }
         public IBinder asBinder(){ //实现IInterface中唯一的方法，
             return null ;
        }
};
Binder binder = new Stub();
binder.attachIInterface(plus,"PLUS TWO INT");
```

step 2: 进程A接收进程B的Binder对象

现在有了这个特殊的对象 binder ，可以在进程B的 service 中的 onBind 方法将它返回，即 return binder ;

1. 系统会首先收到这个 binder 对象，然后，它会生成一个 BinderProxy （就是前面提到的Binder 的内部类）类的对象，姑且称之为 binderproxy
2. 然后将该 BinderProxy 代理对象返回给进程A
3. 最终进程A在 onServiceConnected 方法中接收到了 BinderProxy 对象。

```
public class Binder implement IBinder{
        void attachInterface(IInterface plus, String descriptor)
        IInterface queryLocalInterface(Stringdescriptor) //从IBinder中继承而来
        boolean onTransact(int code, Parcel data, Parcel reply, int flags)//暂时不用管，后面会讲。
        final class BinderProxy implements IBinder {
                    IInterface queryLocalInterface(Stringdescriptor) {
                          return null ;//注意这行代码！！
                                                  //下面会讲到。这行代码只是示例，不是源代码。
                      }
                ......
        }
}
```

此时的进程A接收到 Binder 的代理对象，要通过 queryLocalInterface 方法获取这个 binder 的 plus 对象，利用该对象的加法功能进行加法计算------这样是不行的

1. binderproxy.queryLocalInterface("PLUS TWO INT") 调用是合法的，因为 queryLocalInterface 方法是 IBinder 中的方法，而 BinderProxy 和 Binder 都实现了 IBinder 接口。
2. 但是， binderproxy 对象显然没有 plus 对象，因为它根本就没有 attachInterface 方法（这是 Binder 具有的）。
3. 因此进程A的 binderproxy.queryLocalInterface("PLUS TWO INT") 调用返回的将是一个 null （参见上面的示例代码）。

step 3: 进程A利用进程B传过来的对象发起请求

1. BinderProxy 对象不能让 进程A 去直接完成需要的操作，但是提供了 transact 方法(在 IBinder 接口中定义的方法).
2. 进程A将数据（两个 int ）和操作（ plus.add ）通过 transact 方法交给 Binder 对象执行，最终结果会通过 BinderProxy 代理对象返回给进程A
   代码如下：

```
android.os.Parcel data = android.os.Parcel.obtain();
android.os.Parcel reply = android.os.Parcel.obtain();
int _result;
data.writeInterfaceToken("PLUS TWO INT");
data.writeInt(a);
data.writeInt(b);
binderproxy.transact(1,  //约定好的数据，用于让B对A传入的数据进行加法运算(含义是自定义的)，可以定义在`Stub`类中，如`public static fina
                     data, //进程A传入的数据
                     reply, //进程B返回的结果数据
                     0);//为简单起见，最后一个0暂时不管它
```

step 4: 进程B收到并处理进程A的请求

1. 系统保存了 Binder 对象和 BinderProxy 对象的对应关系
2. binderproxy.transact 调用发生后，系统会将这个请求中的数据转发给 Binder 对象， Binder 对象将会在 onTransact 中收到 binderproxy 传来的数据（ Stub.ADD,data,reply,0 ）
3. 根据约定好的操作 Stub.ADD 进行运算后，会把结果写回 reply 。

代码概略如下:

```
 public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags) throws android.os.RemoteExceptior
{
          switch (code) {
                case INTERFACE_TRANSACTION: {
                  reply.writeString(DESCRIPTOR);
                     return true;
                }

                //1. 根据code判断需要的操作是`加法`
                case Stub.ADD: {
                   //2. 获取Token
                      data.enforceInterface("PLUS TWO INT");
                        int _arg0;
                        _arg0 = data.readInt();
                       int _arg1;
                        _arg1 = data.readInt();
                   //3. 获取数据并调用IInterface对象的方法
                        int  _result = this.queryLocalIInterface("PLUS TWO INT") .add(_arg0, _arg1);
                        reply.writeNoException();
                         reply.writeInt(_result);
                         return true;
                  }
           }
      return super.onTransact(code, data, reply, flags);
}
```

1. 根据 code 判断当前是加法操作 Stub.ADD
2. 进程A写数据时写入了一个 InterfaceToken (data.writeInterfaceToken("PLUS TWO INT");)
3. 进程B在自己的 Binder 对象中会获取到该 Token 并调用 queryLocalIInterface 方法查找相应的 IInterface 对象
4. 进程A要执行的操作就在该 IInterface 对象中，并执行add操作

step 5: 进程A获取进程B返回的处理结果

进程B把结果写入 reply 后，进程A就可以从 reply 读取结果。
代码概略如下:

```
binderproxy.transact(Stub.ADD, data, reply, 0);
reply.readException();
_result = reply.readInt();
```

3-Binder优化(step 3-4-5 用代理PlusProxy进行封装)

1. 我们可以将传入数据，并将处理结果返回的代码进行优化，原代码如下：

```
android.os.Parcel data = android.os.Parcel.obtain();
android.os.Parcel reply = android.os.Parcel.obtain();
int _result;
data.writeInterfaceToken("PLUS TWO INT");
data.writeInt(a);
data.writeInt(b);
binderproxy.transact(1, data, reply, 0);//为简单起见，最后一个0暂时不管它
reply.readException();
_result = reply.readInt();
```

2. 将其封装为PlusProxy类

```
public class PlusProxy implements IPlus {
          private IBinder  binderproxy ;
          public PlusProxy(IBinder binderproxy){
                this.binderproxy = binderproxy ;
          }
          public int add (int a ,int b ){
              android.os.Parcel data = android.os.Parcel.obtain();
              android.os.Parcel reply = android.os.Parcel.obtain();
              data.writeInterfaceToken("PLUS TWO INT");
              data.writeInt(a);
              data.writeInt(b);
              binderproxy.transact(1, data, reply, 0);
              int _result;
              reply.readException();
              _result = reply.readInt();
              return _result ;
          }
}
```

可以再Stub类中增加一个静态辅助方法 `public static IPlus asInterface(Ibinder)`：

1. 进程A收到 `BinderProxy` 对象时，调用 `Stub.asInterface(binderproxy)`
2. 该方法负责利用 `BinderProxy` 生成一个 `PlusProxy` 代理对象返回给我们
3. 给进程A一种假象：获取到了Plus对象(进程B中的加法操作)，并进行了 `add()` 操作。本质是内部进行了 传入数据给B进行处理，最终从B中获取到结果 等一系列操作。

4-Android整体架构层面的Binder机制

1-提出问题

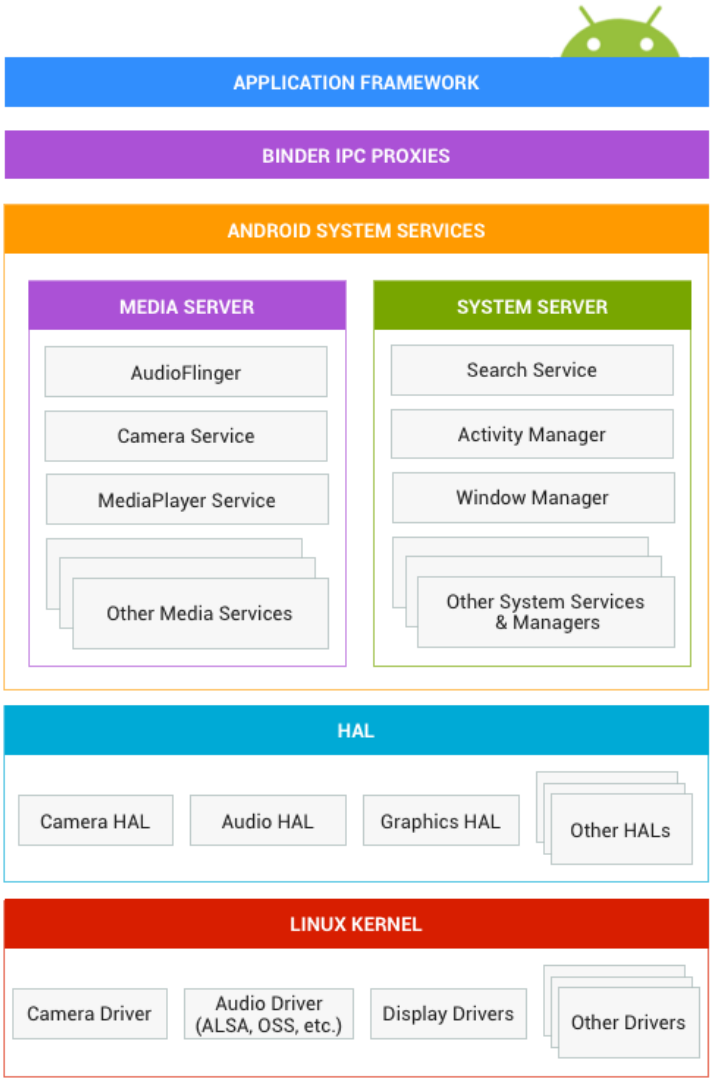
1、ContentProvider中的CRUD是否是线程安全的?

> 不是, 因为底层的Binder机制中的Server端不是线程安全的

2、AIDL中在Service中实现的接口是否是线程安全的?

> 不是, 理由同上

2-Android整体架构

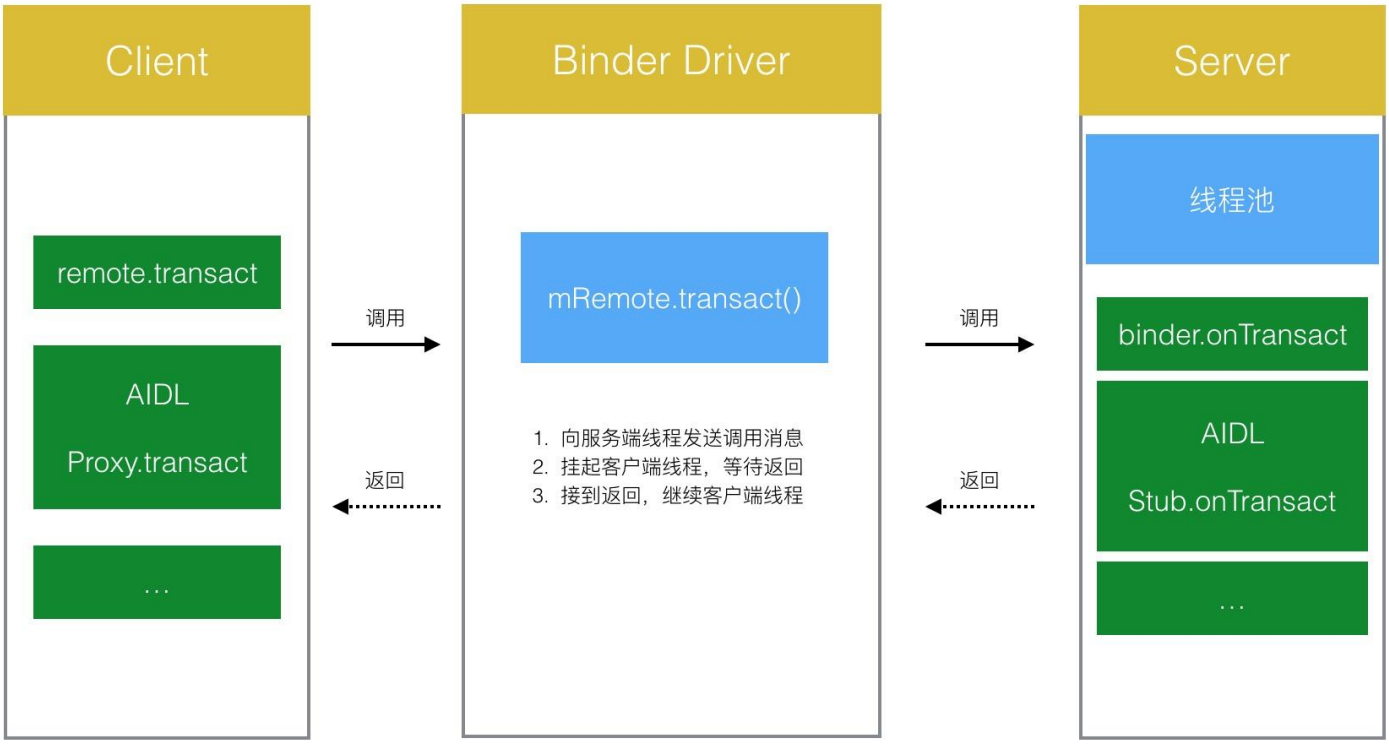


`Android整体架构` 从底层到上层大致分为五层，依次为：

1. Linux内核层: Binder IPC驱动在此层。该层包含各类硬件设备的驱动。
2. HAL(硬件抽象层)：封装Linux内核层的硬件驱动，提供可供 `Android System Services` 层级调用的统一硬件接口。
3. Android System Service(系统服务层): 提供核心服务，包括-Activity Manager、WindowManager、Camera Service(ServiceManagerService)等等
4. 【`Binder IPC层`】：作为 `系统服务层` 和 `FrameWork层` 的IPC桥梁，互相传递接口调用的数据以实现跨进程的通信。
5. Application Framework(应用程序框架层): 提供四大组件，View绘制等内容(ServiceManager)。

- 每个应用框架层的 `ServiceManager` 在系统服务层都有一个对应的 `ServiceManagerService` 。
  例如：WindowManager和WindowManagerService

3-Binder IPC C/S架构:



1. Client: 用户实现的代码，如AIDL自动生成的接口类
2. Binder Driver: 内核层实现的驱动
3. Server: Service中的onBind返回的IBinder对象

绿色区域：用户自行实现部分
蓝色区域：系统实现部分(包括Servcer端的线程池-线程池实现在Binder内部的native方法中)

- 线程池决定了Server的代码 不是线程安全的

4-IBinder接口作用

1. IBinder是用于远程对象的基本接口，是轻量级远程调用机制的核心部分，用于高性能地进行进程内部和进程间调用。
2. IBinder描述了远程对象间交互的抽象协议。
3. 不能直接实现IBinder接口，而是应该继承自Binder类。

5-Binder解析

Binder的初始化代码, Binder.java:

```
public Binder(){
  init();
  ...
}
...
private native final void init();
```

- init()是native方法，是对底层 `Binder Driver` 的封装。
- 客户端发起请求的时候(调用IBinder的 `transact()`接口 也就是调用驱动层的 `mRemote.transact()`)，`Binder Driver` 会调用 `execTransact()` ，并在内部调用服务端 `Binder` 的 `onTransact()` 方法。

6-Driver层

Binder类进行了完美的封装，开发者只需要继承 `Binder` 和实现 `onTransact()` 即可。

小结

1. AIDL自动生成了Stub类型
2. 在 Service 端继承 Stub 类，Stub 类中实现了 onTransact 方法实现了 数据解析 的功能

3. 在 Client 端使用 Stub 类的 Proxy 对象，该对象实现了 数据打包 并且调用 transact 的功能