转载请注明链接：https://blog.csdn.net/feather_wch/article/details/81136290

> 本文是我一点点归纳总结的干货，但是难免有疏忽和遗漏，希望不吝赐教。
> Android进程间通信详解。涉及到Binder机制、Bundle、文件共享、Messenger、AIDL、ContentProvider、Socket这五种IPC方式。

有帮助的话请点个赞！万分感谢！

Android进程间通信详解(IPC)

版本：2019/3/11-13:30

多进程(17)

1、什么是IPC?

> Inter-Process Communication(进程间通信)

2、进程间通信是什么?

> 两个进程之间进行数据交换的过程

3、进程是什么?

> 一般指一个执行单元,也是系统分配资源的最小单位。

4、Android中的IPC方法(6种)

> 1-Bundle
> 2-文件共享

> 3-Messenger
> 4-AIDL
> 5-ContentProvider
> 6-Socket

5、线程是什么?

> 是CPU调度的最小单元，而且是有限的系统资源。一个进程可以包含多个线程。

6、ANR导致的原因？如何避免?

> ANR-application not respongding是因为UI线程内部的耗时操作导致界面无响应。应该将耗时操作移到非UI线程即可。

7、什么时候需要用到多进程?

> 比如：当前应用需要从其他应用获取数据

8、开启多进程模式的方法

> 1. 给四大组件添加属性 `android:process`
> 2. 特殊方法：通过JNI在native层去fork一个新的进程。

9、activity的 `android：process` 属性 `=":remote"` 和 `"com.example.remote"` 的区别

> 1. `:remote` 是指在当前进程名前面加上当前的包名 `com.example:remote` ， 且该进程是当前应用的私有进程，其他应用的组件不能和该进程跑在同一个进程内
> 2. 后者是属于全局进程，其他应用可以通过ShareUID的方式和它跑在同一个进程中。

10、多进程会造成的问题:

> 1. 静态成员和单例模式完全失效
> 2. 线程同步机制完全失效
> 3. SharedPreferences的可靠性下降(不支持两个进程同时读写)
> 4. Application会多次创建

11、Serializable和Parcelable接口作用

> 1. 可以完成对象的序列化过程
> 2. 使用Intent和Binder传输数据时就需要Serializable或Parcelable
> 3. 需要把对象持久化到存储设备，或者通过网络传给其他客户端。

12、Serializable接口的作用和使用

> 1. Serializable接口为对象提供了标准的序列化和反序列化操作。
> 2. 使用:
>   1. 这个类实现Serializable接口
>   2. 该类声明一个serialVersionUID(`private static final long serialVersionUID=8711368828010083044L`)。 甚至可以不申明ID，但是这个ID会对反序列化产生影响。

13、serialVersionUID的作用

> 1、序列化后的数据的ID只有和当前类的ID相同才能正常被反序列化。
> 2、可以手动设置ID为1L，这样会自动根据当前类结构去生成它的hash值

14、序列化的两个特别注意点:

> 1、静态成员变量不属于对象，不会参与序列化过程
> 2、用 `transient` 关键字标记的成员变量不会参与序列化过程。

15、java.io.ObjectOutputStream和ObjectInputStream用于对象序列化

```
// 1、创建文件字节输出流对象
FileOutputStream outputStream = new FileOutputStream(f);
// 2、构造对象序列化输出流
ObjectOutputStream objectOutputStream = new ObjectOutputStream(outputStream);
// 3、向文件写入对象
objectOutputStream.writeObject(new User("酒香逢","123"));
// 4、关闭资源。objectOutputStream.close()内部已经将FileOutputStream对象资源释放了
objectOutputStream.close();
```

16、系统中实现了Parcelable接口的类

> Intent、Bundle、Bitmap、List、Map
> 里面的每个元素也都要可序列化

17、Parcelable和Serializable

> 1. Serializable是java中的序列化接口，简单，但开销很大(需要大量IO操作)
> 2. Parcelable是Android首推方法，使用麻烦，效率很高
> 3. Parcelable主要用于内存序列化上
> 4. Serializable适用于将对象序列化到存储设备或通过网络传输(Parcelable也可以只是较复杂)

Binder(39)

1、Binder是什么?

> 1. Binder是android的一个类，实现了IBinder接口
> 2. IPC角度：Binder是android的一种跨进程通信方式
> 3. Android Framework角度：Binder是ServiceManager连接各种Manager(ActivityManager,WindowManager等)和相应ManagerService的桥梁
> 4. Android应用层：Binder是客户端和服务端进行通信的媒介，当bindService的时候，服务端会返回一个包含了服务端业务调用的Binder对象，通过该对象，客户端可以获取服务端提供的服务和数据，服务包括普通服务和基于AIDL的服务。
> 5. Linux层面：Binder也可以看做一种虚拟的物理设备，设备驱动是/dev/binder，Linux中没有这种通信方式

ServiceManager

2、ServiceManager是什么？作用呢?

> 1. Android在init进程启动之后启动一个 特殊的系统服务-ServiceManagerService
> 2. 用来管理其他的所有系统服务
> 3. 比如客户端需要使用 ActivityManagerService的功能 ， 首先 AMS已经在SMS中进行过注册
> 4. 客户端通过 SMS查询到AMS后 ， 直接去调用到AMS的服务

设备驱动

3、设备驱动的作用?

```
// 1. 打开
fd = open("/dev/mem",O_RDWR);
// 2. 读取
read(fd,rdbuf,10);
// 3. 写
write(fd,wrbuf,10);
```

> 1. 设备驱动的作用是让操作设备就好像在读写文件一样，方便快捷。

Binder机制和Linux IPC

4、Binder机制是做什么的?

> 1. Android Binder 用于 进程间通信 。
> 2. Android的应用和系统服务 运行在各自的 进程中 ， 进程之间的通信就需要借助 Binder 实现。

5、Android基于的Linux内核，Linux有哪些IPC方法?

IPC	特点	数据复制次数	同步
管道	分配一个page大小的内存，容量有限	2	×
消息队列	任何大小，不适合频繁大量的数据	2	×
共享内存	任何大小	0	×
Socket	任何大小，多用于不同设备间的网络通信，传输效率较低。	2	×
信号量	进程间通信的同步机制，用于给共享资源上锁。	×	√
信号	不适合进行数据通信，多用于进程中断控制，比如杀死一个进程	×	

6、Binder机制的优势体现在哪里?

> 1. 性能 ： 数据只需要复制一次，Linux的大部分IPC需要复制两次，共享内存不需要复制数据。因此 Binder 仅次于 共享内存 。
> 2. 稳定性 ： 共享内存 需要额外进行 数据同步 操作，但是 Binder机制 不需要（本身进行了同步）。
> 3. 安全性 ： Android应用具有UID ， 这是在内核空间的。然而Linux的UID和PID都是在用户空间操作的，因此 Binder安全性更高

7、内核空间是什么？用户空间是什么?

8、UID、PID是什么?

9、为什么UID、PID在用户空间会不安全?

10、Binder主要用在哪?

> 1. Service
> 2. AIDL
> 3. Messenger(底层AIDL)

AIDL

11、AIDL文件的本质作用

> AIDL文件的本质就是系统提供了一种快速实现Binder的工具。

12、通过AIDL快速实现Binder的步骤

1. 新建Book.java(简单的类，没有实际功能，实现Parcelable接口)
2. 新建Book.aidl需要有parcelable Book;
3. 新建IBookManager.aidl,里面需要导入Book类 `import xxx.Book;`
    - 一个interface
    - 声明服务端提供了哪些 服务接口，如:`getBookList`
4. 选择android studio的build中make project

13、AIDL工具快速实现Binder所生成的Java文件中的四个要点

- 例如 `IBookManager.java` -由IBookManager.aidl生成

1. 继承 `IInterface` 接口，本身也为接口
2. 声明了两个IBookManager.aidl中定义的getBookList和addBook方法(并且用两个id标识这两个方法，用于标识在transact中客户端请求的是哪个方法)
3. 声明一个内部类Stub，该Stub就是Binder类
4. Stub的内部代理类Proxy，用于处理逻辑----客户端和服务端都位于一个进程时，方法调用不会走跨进程的transact过程，当位于不同进程时，方法调用走transact过程。

14、Binder的工作流程：

1. Client向Binder发起远程请求，Client同时挂起
2. Binder向data(输入端对象)写入参数，并且通过Transact方法向服务端发起远程调用请求(RPC)
3. Service端调用onTransact方法(运行在服务端线程池中)向reply(输出端对象)写入结果
4. Binder获取reply数据，返回数据并且唤起Client

15、通过AIDL自动生成Binder的java文件

1-新建Book.java(简单的类，没有实际功能，实现Parcelable接口)

```
public class Book implements Parcelable{
    public int bookId;

    public Book(int bookId){
        this.bookId = bookId;
    }
    private Book(Parcel in){
        bookId = in.readInt();
    }

    public static final Creator<Book> CREATOR = new Creator<Book>() {
        @Override
        public Book createFromParcel(Parcel in) {
            return new Book(in);
        }

        @Override
        public Book[] newArray(int size) {
            return new Book[size];
        }
    };

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel parcel, int i) {
        parcel.writeInt(bookId);
    }
}
```

> 2-新建Book.aidl

```
package com.example.administrator.featherdemos.aidl;

parcelable Book;
```

> 3-新建IBookManager.aidl

```
package com.example.administrator.featherdemos.aidl;

//关键：导入Book.java
import com.example.administrator.featherdemos.aidl.Book;

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

> 4-选择android studio的build中make project

- 系统就会自动生成对应java文件 `IBookManager` ， 位于目录 `app\build\generated\source\aidl\debug\`包下

16、Binder所在java文件要点如下:(IBookManager.java)

> 1. 本身是一个接口，并且继承 `IInterface` 接口，
> 2. 声明了两个IBookManager.aidl中定义的getBookList和addBook方法
>   1. 声明了aidl文件中定义的服务端的接口

    2. 并且用一个个id标识这些方法，这些id的作用是：用于服务端知道在transact中客户端请求的是哪个方法
3. 声明一个内部类Stub，该Stub就是Binder类
4. Stub的内部代理类Proxy，内部隐藏了"客户端请求服务端，服务端返回结果"的逻辑。并且用于处理逻辑:
    1. 客户端和服务端都位于一个进程时，方法调用不会走跨进程的transact过程
    2. 当位于不同进程时，方法调用走transact过程。

```
public interface IBookManager extends android.os.IInterface {
    // 1、声明aidl中的服务方法
    public java.util.List<Book> getBookList() throws android.os.RemoteException;
    // 2、实际的Binder类，
    public static abstract class Stub extends android.os.Binder implements IBookManager {
        // Binder的唯一标志
        private static final String DESCRIPTOR = "com.feather.imageview.IBookManager";
        static final int TRANSACTION_getBookList = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);

        public Stub() {
            this.attachInterface(this, DESCRIPTOR);
        }

        // 2、将服务端Binder对象转换为客户端所需要的AIDL接口类型对象
        public static IBookManager asInterface(android.os.IBinder obj) {
            if ((obj == null)) {
                return null;
            }
            android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
            if (((iin != null) && (iin instanceof IBookManager))) {
                return ((IBookManager) iin);
            }
            return new Proxy(obj);
        }

        // 3、返回当前Binder对象
        @Override
        public android.os.IBinder asBinder() {
            return this;
        }

        // 4、运行在服务端，确定客户需要的是哪个服务，将服务结果返回给客户端
        @Override
        public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags) throws an
            String descriptor = DESCRIPTOR;
            switch (code) {
                case INTERFACE_TRANSACTION: {
                    reply.writeString(descriptor);
                    return true;
                }
                case TRANSACTION_getBookList: {
                    data.enforceInterface(descriptor);
                    java.util.List<Book> _result = this.getBookList();
                    reply.writeNoException();
                    reply.writeTypedList(_result);
                    return true;
                }
                default: {
                    return super.onTransact(code, data, reply, flags);
                }
            }
        }

        /**=======================================
         * 5、客户端获取的Binder对象的代理对象
         *   1. 封装客户端请求和接收服务端结果的内部逻辑
         *   2. 对外提供服务端的服务接口
         *=======================================*/
        private static class Proxy implements IBookManager {
            private android.os.IBinder mRemote;

            Proxy(android.os.IBinder remote) {
                mRemote = remote;
            }
```

```
            public String getInterfaceDescriptor() {
                return DESCRIPTOR;
            }           @Override
            public android.os.IBinder asBinder() {
                return mRemote;
            }

            @Override
            public java.util.List<Book> getBookList() throws android.os.RemoteException {
                android.os.Parcel _data = android.os.Parcel.obtain();
                android.os.Parcel _reply = android.os.Parcel.obtain();
                java.util.List<Book> _result;
                try {
                    _data.writeInterfaceToken(DESCRIPTOR);
                    mRemote.transact(Stub.TRANSACTION_getBookList, _data, _reply, 0);
                    _reply.readException();
                    _result = _reply.createTypedArrayList(Book.CREATOR);
                } finally {
                    _reply.recycle();
                    _data.recycle();
                }
                return _result;
            }


        }
    }
}
```

17、Binder类Stub的解析(3个重要方法+多个需求相关方法)

1. DESCRIPTOR
   Binder的唯一标识,一般用类名表示
2. asInterface(android.os.IBinder obj)
   将服务端Binder对象转换成客户端所需AIDL接口类型对象(xxx.aidl.IBookManager)。如果客户端和服务端位于同一进程，此方法返回就是服务端的Stub对象本身，否则返回系统封装后的Stub.proxy
3. asBinder
   返回当前Binder对象
4. onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags)
   (1)运行在服务端的Binder线程池。
   (2)通过code确定Client请求的目标方法，从data中取得方法所需参数，执行目标方法。执行完毕后就向reply中写入返回值。
   (3)如果此方法返回false客户端就请求失败，我们可以用此来进行权限验证。
5. 内部代理类Proxy
   1. Proxy的getBookList
      (1)运行在客户端。
      (2)创建方法所需的data(输入)、reply(输出)和list(返回)对象。把该方法的参数信息写入data，调用transact方法发起RPC远程过程调用请求，同时当前线程挂起。服务端的onTransact会被调用，到RPC过程返回后，当前线程继续执行，并从reply中取出RPC过程的返回结果，最后返回reply中的数据。
   2. Proxy的addBook
      运行在客户端。过程和getBookList类似，但是没有返回值。

18、AIDL如何进行权限验证?

1. 利用Binder类也就是Stub的onTransact方法

2. 该方法如果返回false就代表请求失败，可以用于权限验证

19、Binder注意点

1. 客户端发起远程请求后，当前线程会被挂起直到服务器返回结果，因此不要在UI线程发起远程请求。
2. 服务端的Binder方法运行在Binder的线程池中，所以Binder方法是否耗时都要采用同步方法实现。

20、Binder的两个重要方法/Binder如何检测服务端异常终止?

1. linkToDeath和unlinkToDeath。用于解决: 如果服务端异常终止，而会导致客户端调用失败，甚至可能客户端都不知道binder已经死亡，就会产生问题。
2. linkToDeath作用给Binder设置一个死亡代理，Binder会收到通知，还可以重新发起连接请求从而恢复连接。
3. binder的isBinderAlive也可以判断Binder是否死亡。

21、如何根据当前进程是否在服务端，来直接获取Binder对象，还是代理对象?

1. Stub的asInterface()中进行处理
2. queryLocalInterface()如果能查询到Binder对应的IInterface对象，则表示位于同一个进程，直接返回 `Binder对象`
3. 如果查询不到，则表明进程不同，利用代理类 `Stub.proxy` 进行 远程调用

```
public static IMyAidlInterface asInterface(android.os.IBinder obj) {
    if ((obj == null)) {
        return null;
    }
    android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
    if (((iin != null) && (iin instanceof IMyAidlInterface))) {
        return ((IMyAidlInterface) iin);
    }
    return new Proxy(obj);
}
```

原理深度解析

22、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, 并发送给进程A
3. 进程A在ServiceConnection回调的onServiceConnected方法中接收该特殊对象。
4. 进程A接收到特殊对象后，生成一个代理对象。依靠这个代理对象的帮助，就可以解决进程间通信问题

23、Binder机制的核心步骤(6)

1. 进程A去绑定到进程B注册的Service
2. 进程B创建Binder对象---Servcice的onBind()
3. 进程A接收进程B的Binder对象
4. 进程A利用进程B传过来的对象发起请求
5. 进程B收到并处理进程A的请求
6. 进程A获取进程B返回的处理结果

24、step 2: 进程B创建的Binder对象需要有两个特性:

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

25、Binder对象如何具有能被跨进程传输的能力?

1. Binder对象实现了IBinder接口。
2. 系统会为每个实现了该接口的对象提供跨进程传输。

26、Binder对象如何具有完成特定任务的能力?

1. 核心是 `IInterface`接口 ，服务端实现了该接口，定义了提供的服务的内部处理逻辑。例如: 1 + 1 = 2， 获取图书馆书籍列表，等等。
2. Binder中实现了两个方法：attachInterface()、queryLocalInterface() --- 继承IBinder接口都必须实现。
3. attachIInterface(): 会将IInterface对象根据key存入到Binder的内部。
4. queryLocalInterface(): 根据key值（即参数 descriptor）可以获取到对应的IInterface对象。
5. 借助 `Binder`所关联的`IInterface`接口的实现类，执行特定的功能

```
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。
        }
}
 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。
          }
          ......
}
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");
```

27、step 3: 进程A接收进程B的Binder对象

1. Service的onBind中返回Binder对象。
2. 系统会收到这个binder对象，然后会生成一个BinderProxy。并且返回给进程A。
3. 进程A会在onServiceConnected中接收到了BinderProxy对象。
4. 该对象仅仅是代理类，本身并不能去实现需要实现的功能。
5. 只提供了transact()用于发起请求。

28、step 4: 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将数据和操作通过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

29、step 5: 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。

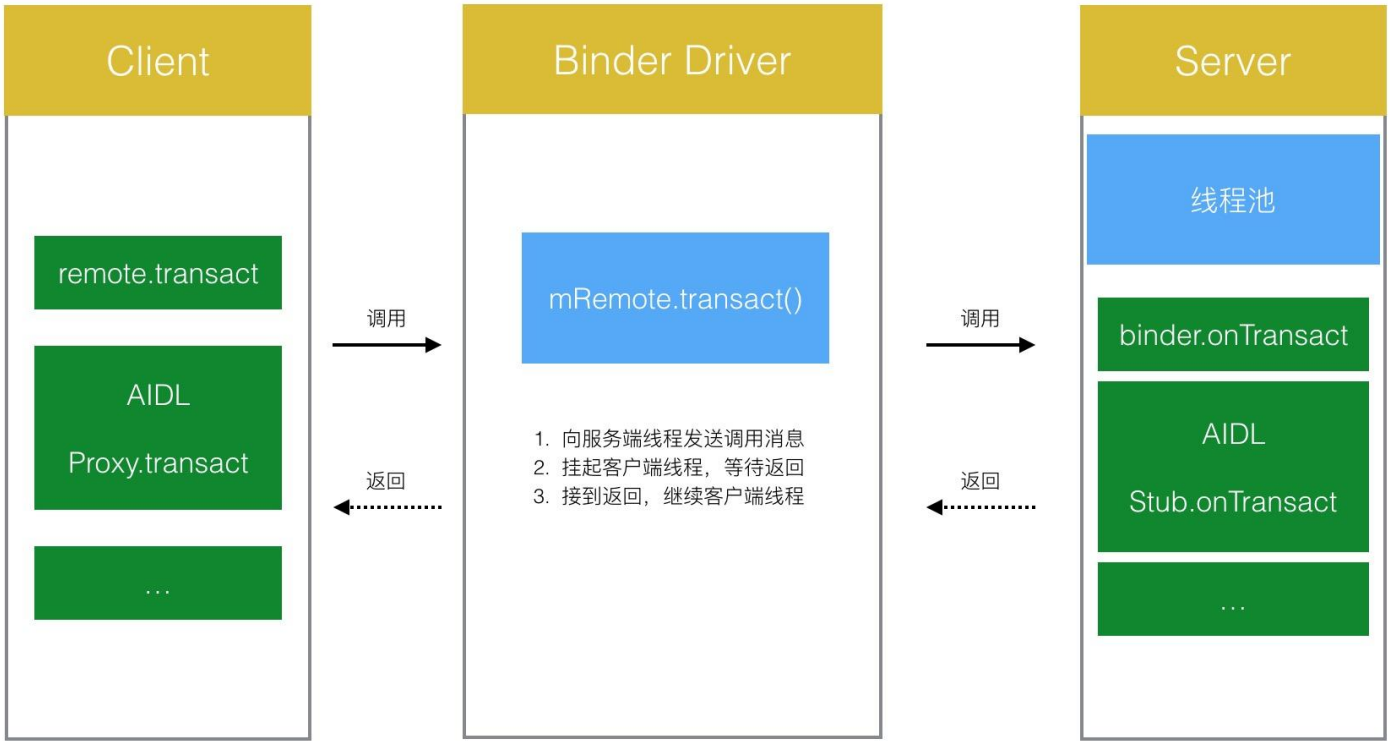
30、step 6: 进程A获取进程B返回的处理结果

1. 进程B把结果写入reply后，进程A就可以从reply读取结果。

31、step 4、5、6的优化

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。

32、Binder的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)

33、Binder的Server为什么不是线程安全的?

Server端使用了线程池决定了Server的代码 不是线程安全的

34、Binder服务端的线程池是如何实现的?

1. 系统实现
2. 线程池实现在Binder内部的native方法中

35、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

36、AIDL中在Service中实现的接口是否是线程安全的?

> 不是, 理由同上

37、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。

38、Binder解析

```
//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。

39、Binder的Driver层作用

> 对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。

Android中的IPC方法(32)

1-Bundle

1、Bundle的作用

> 1. Bundle能携带数据-实现了Parcelable接口，常用于传递数据
> 2. 如Acitivity、Service和Receiver都支持在Intent中通过Bundle传递数据。
> 3. 比如打开另一个App的Activity，可以传递数据进去

2、Bundle传递数据在特殊使用场景无法使用的解决办法?

> 1. 场景：A进程在完成计算后需要启动B进程的一个组件并且将结果传递给B进程，但是这个结果不支持放入Bundle，因此无法通过Intent传输。
> 2. 方案：A进程通过Intent启动进程B的service组件(如IntentService)进行计算，因为Service也在B进程中，目标组件就可以直接使用计算结果。

2-文件共享

3、文件共享的作用

> 两个进程通过读/写同一个文件进行数据交换。也可以通过序列化在进程间传递对象。

4、文件共享的特点:

> 1. 通过序列化在进程间传递对象
> 2. 只适合同步要求不高的进程间通信
> 3. 要妥善处理并发读写问题，高并发情况下很容易出现数据丢失。

3-Messenger

5、Messenger是什么?

> 1. 轻量级的IPC方案
> 2. 底层实现是AIDL
> 3. 一次处理一个请求，因此在服务端不考虑线程同步问题。

6、 Messenger的使用

> 通过messenger在两个进程之间互相发送消息。
> 客户端:
>
> 1. 绑定并启动位于新进程的服务，通过msg发送消息
> 2. 设置接受新进程服务发送来的消息

```
public class MessengerActivity extends Activity {

    private static final String TAG = "MessengerActivity";

    private Messenger mMessenger;

    private ServiceConnection mServiceConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder iBinder) {
            mMessenger = new Messenger(iBinder); //
            Message msg = Message.obtain(null, Constant.MSG_FROM_CLIENT);
            //bundle携带消息
            Bundle data = new Bundle();
            data.putString("msg", "This is Client!");
            //给msg绑定bundle
            msg.setData(data);

            //将用于服务端回复的msger发送给服务端
            msg.replyTo = mGetReplyMessenger;
            try {
                //发送消息
                mMessenger.send(msg);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }

        @Override
        public void onServiceDisconnected(ComponentName componentName) {

        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_messenger);
        //绑定并启动服务
        Intent intent = new Intent(this, MessengerService.class);
        bindService(intent, mServiceConnection, Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onDestroy() {
        //解除服务
        unbindService(mServiceConnection);
        super.onDestroy();
    }

    /**------------------
     *  接受Service回复消息
     * ------------------*/
    private final Messenger mGetReplyMessenger = new Messenger(new MessengerHandler());
    private static class MessengerHandler extends Handler{
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what){
                case Constant.MSG_FROM_SERVICE:
                    Log.i(TAG, "recv msg from service:" + msg.getData().getString("reply"));
                    break;
                default:
                    super.handleMessage(msg);
                    break;
            }
```

```
        }
    }
}
```

> 服务器端:
> 接收消息，并通过client传送来的messenger回复消息。

```
public class MessengerService extends Service {

    private static final String TAG = "MessengerService";
    private final Messenger mMessenger = new Messenger(new MessengerHandler());

    private static class MessengerHandler extends Handler{
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what){
                case Constant.MSG_FROM_CLIENT:
                    //接收到Client消息
                    Log.i(TAG, "receive msg from client: " + msg.getData().getString("msg"));
                    /*-----------------------------------------
                    * 回复数据给Client
                    * -----------------------------------------*/
                    Messenger clientMessenger = msg.replyTo; //获取到服务器传来的msger
                    Message message = Message.obtain(null, Constant.MSG_FROM_SERVICE); //设置msg
                    Bundle bundle = new Bundle();
                    bundle.putString("reply", "This is Service!");
                    message.setData(bundle);
                    //发送消息
                    try {
                        clientMessenger.send(message);
                    } catch (RemoteException e) {
                        e.printStackTrace();
                    }

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

    public MessengerService() {
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mMessenger.getBinder();
    }
}
```

> AndroidManifest中注册Service:
> `android:process=":remote"` 代表另开一个Service进程。

```
<service
            android:name=".MessengerService"
            android:enabled="true"
            android:exported="true"
            android:process=":remote">
        </service>
```

7、Messenger缺点：

1. 以串行的方式处理客户端发送的消息，如果大量的消息同时发送到服务端，服务端仍然只能一个个处理，如果有大量的并发请求，Messenger就无法胜任。
2. 如果需要跨进程调用服务端的方法，这种情形Messenger就无法做到。

4-AIDL

8、 AIDL进程间通信流程

1-服务端

1. 创建一个Service来监听客户端的连接请求。
2. 创建一个AIDL文件。
3. 将暴露给客户端的接口在该AIDL文件中声明。
4. 最后在Service中实现这个AIDL接口即可。

2-客户端

1. 绑定服务端的Service
2. 将服务端返回的Binder对象转成AIDL接口所属的类型
3. 最后就可以调用AIDL中的方法。

9、AIDL实例

1. 创建你需要的接口文件：ITuringManager.aidl(这里功能就是获取图灵机列表，以及增加一个图灵机)

```
package com.example.administrator.featherdemos;

import com.example.administrator.featherdemos.TuringMachine;

interface ITuringManager {
    List<TuringMachine> getTuringMachineList();
    void addTuringMachine(in TuringMachine machine);
}
```

2. 实现TuringMachine.java(也就是接口文件导入的类，需要实现Parcelable接口)：

```java
package com.example.administrator.featherdemos;
//import ....需要的包
public class TuringMachine implements Parcelable{
    int machineId;
    String description;

    protected TuringMachine(Parcel in) {
        machineId = in.readInt();
        description = in.readString();
    }

        public TuringMachine(int id, String description){
        this.machineId = id;
        this.description = description;
    }

    public static final Creator<TuringMachine> CREATOR = new Creator<TuringMachine>() {
        @Override
        public TuringMachine createFromParcel(Parcel in) {
            return new TuringMachine(in);
        }

        @Override
        public TuringMachine[] newArray(int size) {
            return new TuringMachine[size];
        }
    };

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel parcel, int i) {
        parcel.writeInt(machineId);
        parcel.writeString(description);
    }
}
```

> 3. 使用到的类(TuringMachine.java)需要一个对应aidl文件-TuringMachine.aidl:

```
package com.example.administrator.featherdemos;

parcelable TuringMachine;
//ITuringMachine.aidl和TuringMachine.aidl需要在aidl文件夹下的包内
//TuringMachine.java要在java文件夹下的包内。
```

> 4. 远程服务端-ITuringMachineManagerService.java

```
public class ITuringMachineManagerService extends Service {

    /**
     * CopyOnWriteArrayList支持并发读/写:
     * 1. AIDL在服务端的Binder线程池中执行，因此当多个客户端同时连接的时候，会存在多个线程同时访问的情况。
     * 2. CopyOnWriteArrayList能进行自动的线程同步。
     */
    private CopyOnWriteArrayList<TuringMachine> mTuringMachineList = new CopyOnWriteArrayList<>();

    private Binder mBinder = new ITuringManager.Stub(){

        @Override
        public List<TuringMachine> getTuringMachineList() throws RemoteException {
            return mTuringMachineList;
        }

        @Override
        public void addTuringMachine(TuringMachine machine) throws RemoteException {
            mTuringMachineList.add(machine);
        }
    };

    public ITuringMachineManagerService() {
        mTuringMachineList.add(new TuringMachine(1, "Machine 1"));
        mTuringMachineList.add(new TuringMachine(2, "Machine 2"));
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }
}
```

5. AndroidManifest中注册Service

```
<service
            android:name=".ITuringMachineManagerService"
            android:enabled="true"
            android:exported="true"
            android:process=":remote">
```

6. 本地客户端

```
public class TuringActivity extends AppCompatActivity {

    private static final String TAG = TuringActivity.class.getName();

    //Service连接：从服务端获取本地AIDL接口对象，并调用远程服务端的方法。
    private ServiceConnection mServiceConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder iBinder) {
            //Binder的asInterface()将binder对象转换为客户端需要的AIDL接口对象
            ITuringManager iTuringManager = ITuringManager.Stub.asInterface(iBinder);
            try {
                //获取服务端的List
                ArrayList<TuringMachine> turingMachineArrayList
                        = (ArrayList<TuringMachine>) iTuringManager.getTuringMachineList();

                for(TuringMachine machine : turingMachineArrayList){
                    Log.i(TAG, "onServiceConnected: "+machine.getMachineId() + "-" + machine.getDescription());
                }
            } catch (RemoteException e){
                e.printStackTrace();
            }
        }

        @Override
        public void onServiceDisconnected(ComponentName componentName) {

        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_turing);

        //绑定远程服务端的 Service 并启动
        Intent intent = new Intent(this, ITuringMachineManagerService.class);
        bindService(intent, mServiceConnection, Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onDestroy() {
        unbindService(mServiceConnection); //解绑
        super.onDestroy();
    }
}
```

10、AIDL支持的数据类型

> - 基本数据类型(int、long、char、boolean、double等)
> - String和CharSequence
> - List：只支持ArrayList，且里面所有元素必须是AIDL支持的数据。
> - Map：只支持HashMap，且里面所有元素必须是AIDL支持的数据，包括key和value
> - Parcelable：所有实现Parcelable接口的对象
> - AIDL：所有AIDL接口都可以在AIDL中使用

11、AIDL中List只能用ArrayList，远程服务端为何使用了CopyOnWriteArrayList(并非继承自ArrayList)：

> Binder会根据List规范去访问数据，并且生成一个新的ArrayList传给客户端，因此没有违反数据类型的规定。
> ConcurrentHashMap也是类似功能

12、AIDL实例：如何使用观察者模式

> 在AIDL基础上有如下步骤：
>
> 1. 建立观察者接口(Observer)-ITMachineObserver.aidl
> 2. 在ITuringManager.aidl中增加注册和解注册功能(register\unregister)
> 3. 在服务端ITuringMachineManagerService中的binder对象里实现额外增加的注册和解注册功能。
> 4. 在客户端中的binder对象里实现观察者接口中的更新方法。
>
> 使用：
>
> 1. 客户端中通过从服务端获得的Binder对象，调用register/unregister等方法
> 2. 服务端中通过Client客户端注册的Observer去调用客户端Binder中的更新方法

13、AIDL观察者模式源码：

> 1. ITMachineObserver.aidl

```
package com.example.administrator.featherdemos;

import com.example.administrator.featherdemos.TuringMachine;

interface ITMachineObserver {
    void inform(in TuringMachine machine);
}
```

> 2. ITuringManager.aidl

```
package com.example.administrator.featherdemos;

import com.example.administrator.featherdemos.TuringMachine;
import com.example.administrator.featherdemos.ITMachineObServer;

interface ITuringManager {
    List<TuringMachine> getTuringMachineList();
    void addTuringMachine(in TuringMachine machine);
    void registerListener(in ITMachineObserver observer);
    void unregisterListener(in ITMachineObserver observer);
}
```

> 3. ITuringMachineManagerService:
>
> 只修改了private Binder mBinder = new ITuringManager.Stub()的内容

```
public class ITuringMachineManagerService extends Service {

    /**
     * CopyOnWriteArrayList支持并发读/写：
     * 1. AIDL在服务端的Binder线程池中执行，因此当多个客户端同时连接的时候，会存在多个线程同时访问的情况。
     * 2. CopyOnWriteArrayList能进行自动的线程同步。
     */
    private CopyOnWriteArrayList<TuringMachine> mTuringMachineList = new CopyOnWriteArrayList<>();
    private CopyOnWriteArrayList<ITMachineObserver> mObserverList = new CopyOnWriteArrayList<>();

    private Binder mBinder = new ITuringManager.Stub(){

        @Override
        public List<TuringMachine> getTuringMachineList() throws RemoteException {
            return mTuringMachineList;
        }

        @Override
        public void addTuringMachine(TuringMachine machine) throws RemoteException {
            mTuringMachineList.add(machine);
            for(ITMachineObserver observer : mObserverList){
                observer.inform(machine);
            }
        }

        @Override
        public void registerListener(ITMachineObserver observer) throws RemoteException {
            mObserverList.add(observer);
        }

        @Override
        public void unregisterListener(ITMachineObserver observer) throws RemoteException {
            mObserverList.remove(observer);
        }
    };

    public ITuringMachineManagerService() {
        mTuringMachineList.add(new TuringMachine(1, "Old Machine 1949"));
        mTuringMachineList.add(new TuringMachine(2, "Old Machine 1949-2"));
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }
}
```

4. TuringActivity:

    1. private ITMachineObserver mITMachineObserver获得观察者的Binder对象，并实现接口方法
    2. 调用iTuringManager.registerListener(mITMachineObserver)在服务端进行注册

```
public class TuringActivity extends AppCompatActivity{

    private static final String TAG = TuringActivity.class.getName();

    //Service连接：从服务端获取本地AIDL接口对象，并调用远程服务端的方法。
    private ServiceConnection mServiceConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder iBinder) {
            //Binder的asInterface()将binder对象转换为客户端需要的AIDL接口对象
            ITuringManager iTuringManager = ITuringManager.Stub.asInterface(iBinder);
            try {
                iTuringManager.addTuringMachine(new TuringMachine(10, "Machine 2008"));

                //获取服务端的List
                ArrayList<TuringMachine> turingMachineArrayList
                        = (ArrayList<TuringMachine>) iTuringManager.getTuringMachineList();

                for(TuringMachine machine : turingMachineArrayList){
                    Log.i(TAG, "onServiceConnected: "+machine.getMachineId() + "-" + machine.getDescription());
                }

                iTuringManager.registerListener(mITMachineObserver);

                iTuringManager.addTuringMachine(new TuringMachine(3, "New Machine 2018!"));
            } catch (RemoteException e){
                e.printStackTrace();
            }
        }

        @Override
        public void onServiceDisconnected(ComponentName componentName) {

        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_turing);

        //绑定远程服务端的 Service 并启动
        Intent intent = new Intent(this, ITuringMachineManagerService.class);
        bindService(intent, mServiceConnection, Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onDestroy() {
        unbindService(mServiceConnection); //解绑
        super.onDestroy();
    }

    private ITMachineObserver mITMachineObserver = new ITMachineObserver.Stub() {
        @Override
        public void inform(TuringMachine machine) throws RemoteException {
            Log.i(TAG, "inform: get a new machine-"+machine.getDescription());
        }
    };

}
```

14、AIDL观察中解除注册引发问题(RemoteCallbackList)

1. 在onDestory时，我们可以解除在服务端的注册：

```
@Override
    protected void onDestroy() {
        //确保远程服务的Binder任然存活，就进行unregister
        if(mRemoteITuringManager != null && mRemoteITuringManager.asBinder().isBinderAlive()){
            try {
                mRemoteITuringManager.unregisterListener(mITMachineObserver);
                Log.i(TAG, "onDestroy: unregister!");
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
        unbindService(mServiceConnection); //解绑
        super.onDestroy();
    }
```

2. 但在服务端却无法在列表中找到该 `Observer` ： 因为已经是两个不同的对象了
3. 对象的跨进程传输本身是反序列化的过程，而对象在服务端和客户端早已不是同一个对象。
4. RemoteCallbackList
   系统专门提供用于删除跨进程Listener的接口。该类本身是一个泛型，支持任何AIDL接口。

15、RemoteCallbackList工作原理

1. 内部有一个Map结构，key是Ibinder，value是Callback类型。该结构能保存所有AIDL回调。将Listener的信息存入CallBack：

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

2. 虽然每次跨进程Client的同一个对象，会在服务端生成多个对象。但是这些对象本身的Binder都是同一个。
   以Binder作为Key，这样Listener对应着唯一Binder。

16、使用RemoteCallbackList的流程和特点

1. 客户端解注册，服务端会遍历所有listener，找出那个和解注册的listener具有相同Binder的listener，并删除。
2. 客户端终止后，会自动移除客户端注册的所有listener
3. 自动实现线程同步，无需额外操作。

17、RemoteCallbackList的使用

1. Service服务端

```
private RemoteCallbackList<ITMachineObserver> mObserverList = new RemoteCallbackList<>();
```

2. 注册/解除注册

```
@Override
        public void registerListener(ITMachineObserver observer) throws RemoteException {
            mObserverList.register(observer);
        }

        @Override
        public void unregisterListener(ITMachineObserver observer) throws RemoteException {
            Log.i("ITuringService", "unregisterListener: size-"+mObserverList.getRegisteredCallbackCount());
            mObserverList.unregister(observer);
            Log.i("ITuringService", "unregisterListener: size-"+mObserverList.getRegisteredCallbackCount());
        }
```

3. 需要按照RemoteCallbackList的方法进行遍历。

```
        @Override
        public void addTuringMachine(TuringMachine machine) throws RemoteException {
            mTuringMachineList.add(machine);

            final int count = mObserverList.beginBroadcast();
            //进行通知
            for (int i = 0; i < count; i++){
                mObserverList.getBroadcastItem(i).inform(machine);
            }
            mObserverList.finishBroadcast();
        }
```

18、AIDL的注意点

1. Client客户端调用远程方法，该方法运行在服务端的Binder线程池中，Client会被挂起。
2. 服务端的方法执行过于耗时，会导致Client长时间阻塞，若该线程是UI线程，会导致ANR。
3. 客户端的onServiceConnected和onServiceDisconnected方法都运行在UI线程中，不可以进行耗时操作
4. 服务端的方法本身就在服务端的Binder线程池中，所以服务端方法本身就可以执行大量耗时操作，不要再开线程进行异步操作。
5. 远程方法因为运行在Binder线程池中，如果要操作UI要通过Handler切换到UI线程(服务端通过远程方法去操作Cilent客户端的控件)

19、Binder意外死亡的两种解决方法：

1. 给Binder设置DeathRecipient监听，Binder死亡时回调binderDied
2. 在onServiceDisconnected中重连远程服务。

20、Binder意外死亡的解决方法的区别

1. onServiceDisconnected在客户端UI线程中被回调
2. binderDied在客户端的Binder线程池中被回调(无法操作UI)

21、DeathRecipient处理Binder意外死亡的实现：

```
//设置Binder死亡代理
    private IBinder.DeathRecipient mDeathRecipient = new IBinder.DeathRecipient(){

        @Override
        public void binderDied() {
            //远程服务端die，就不重新连接
            if(mRemoteITuringManager == null){
                return;
            }
            mRemoteITuringManager.asBinder().unlinkToDeath(mDeathRecipient, 0);
            mRemoteITuringManager = null;
            //重新绑定远程Service
            //绑定远程服务端的 Service 并启动
            Intent intent = new Intent(TuringActivity.this, ITuringMachineManagerService.class);
            bindService(intent, mServiceConnection, Context.BIND_AUTO_CREATE);
        }
    };
```

> 客户端的onServiceConnected中:

```
//Binder的asInterface()将binder对象转换为客户端需要的AIDL接口对象
ITuringManager iTuringManager = ITuringManager.Stub.asInterface(iBinder);

try {
       //设置死亡代理
       iBinder.linkToDeath(mDeathRecipient, 0); //这里!
} catch (RemoteException e) {
       e.printStackTrace();
}
```

22、onServiceDisconnected解决Binder死亡问题：

> onServiceDisconnected是ServiceConnection的内部方法，在服务端Service断开后就会调用。

23、AIDL中进行权限验证的方法

> 1. 直接在onBind中进行权限验证，比如可以使用permission验证等等。如果验证不通过则bind服务就失败。
> 2. 在服务端onTransact中验证
>   验证失败直接返回false，可以通过uid和pid验证，这样就可以验证包名也可以和第一种方法一样，验证permission。

Binder连接池

24、AIDL使用流程

> 1. 创建一个Service和一个AIDL接口
> 2. 创建一个Binder(自定义)继承自AIDL接口中的Stub类，并实现其中抽象方法
> 3. 在Service的onBind方法中返回该Binder类的对象
> 4. 客户端绑定Service
> 5. 建立连接后就可以访问远程服务端的方法

25、大量业务模块都需要使用AIDL进行进程间通信，如何在不创建大量Service的情况下解决。

> 1. 可以将所有的AIDL放在一个Service中管理。
> 2. 服务端Service提供一个queryBinder接口，根据不同业务返回相应的Binder对象。
> 3. 就是使用Binder池

5-ContentProvider

26、ContentProvider是什么

> 1. ContentProvider是Android中提供的专门用于不同应用间数据共享的方式。
> 2. 底层实现是Binder，但是使用比AIDL简单很多，系统进行了封装。

27、ContentProvider的使用
系统预置了许多ContentProvider，比如通讯录信息、日程表信息等。访问这些数据，只需要通过 `ContentResolver` 的query、update、insert和delete方法。

28、自定义ContentProvider的步骤

> 1. 继承ContentProvider
> 2. 实现：onCreate-创建的初始化工作
> 3. getType-返回url请求对应的MIME类型(媒体类型)，比如图片、视频等。不关注该类型，可以返回null或者 “*/*”
> 4. query-查数据
> 5. insert-插入数据
> 6. update-更新数据
> 7. delete-删除数据

29、ContentProvider六种方法原理：

> 1. 六种方法均运行在ContentProvider的进程中
> 2. onCreate由系统回调并运行在主线程里
> 3. 其余五种方法由外界回调，并运行在Binder线程池中

30、ContentProvider存储方式

> 底层很像SQLite数据库，但是存储方式没有限制，可以使用数据，也可以使用普通文件，甚至可以采用内存中的对象存储数据。

31、ContentProvider实例

> 1. 第一个app用于提供Provider功能，主要包含两个文件：DbOpenHelper和PetProvider两个文件（底层使用数据库存储数据）。
> 2. 第二个app使用Provider提供的数据
>   DbOpenHelper:

```
public class DbOpenHelper extends SQLiteOpenHelper{

    private static final String DB_NAME = "pet_provider.db";
    public static final String PET_TABLE_NAME = "pet";
    public static final String MASTER_TABLE_NAME = "master";

    private static final int DB_VERSION = 1;

    private static final String CREATE_PET_TABLE = "create table if not exists "
            + PET_TABLE_NAME + "(_id integer primary key, name text)";
    private static final String CREATE_MASTER_TABLE = "create table if not exists "
            + MASTER_TABLE_NAME + "(_id integer primary key, name text, sex int)";

    public DbOpenHelper(Context context) {
        //创建数据库(name和版本)
        super(context, DB_NAME, null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase) {
        //创建数据库
        sqLiteDatabase.execSQL(CREATE_MASTER_TABLE);
        sqLiteDatabase.execSQL(CREATE_PET_TABLE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase sqLiteDatabase, int oldVersion, int newVersion) {
       //TODO ignored
    }
}
```

> PetProvider:
> 继承自ContentProvider：将表的uri和uricode相绑定，overide五种方法并底层使用数据库实现。

```java
public class PetProvider extends ContentProvider{
    private static final String TAG = PetProvider.class.getSimpleName();

    private SQLiteDatabase mDb;
    /**
     * privoder的数据访问通过uri来实现，因此自定义Provider也采用此方法：
     *   用UriMatcher将Content_Uri和code相关联,这样通过uri就能知道访问哪个数据库表
     */
    private static final String AUTHORITIES = "customPrivoderName";
    private static final String PET_CONTENT_URI = "content://"+AUTHORITIES+"/pet";
    private static final String MASTER_CONTENT_URI = "content://"+AUTHORITIES+"/master";
    private static final int PET_CONTENT_CODE = 0;
    private static final int MASTER_CONTENT_CODE = 1;
    private static final UriMatcher sUrilMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    static{
        sUrilMatcher.addURI(AUTHORITIES, "pet", PET_CONTENT_CODE); //指明authorites+路径：pet和CODE对应
        sUrilMatcher.addURI(AUTHORITIES, "master", MASTER_CONTENT_CODE);
    }
    /**
     *  根据uri获得相应的table name
     */
    private String getTableName(Uri uri){
        String tableName = null;
        switch (sUrilMatcher.match(uri)){
            case PET_CONTENT_CODE:
                tableName = DbOpenHelper.PET_TABLE_NAME;
                break;
            case MASTER_CONTENT_CODE:
                tableName = DbOpenHelper.MASTER_TABLE_NAME;
                break;
            default:break;
        }
        return tableName;
    }
    @Override
    public boolean onCreate() {
        Log.i(TAG,"onCreate()");
        //创建数据库
        mDb = new DbOpenHelper(getContext()).getReadableDatabase();
        return true;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)
        Log.i(TAG,"query():"+uri);
        String tableName = getTableName(uri);
        if(tableName == null){
            throw new IllegalArgumentException("Unsupported URI:" + uri);
        }
        return mDb.query(tableName, projection, selection, selectionArgs, null, null, sortOrder, null);
    }

    @Override
    public String getType(Uri uri) {
        Log.i(TAG,"getType()");
        return null;
    }

    @Override
    public Uri insert(Uri uri, ContentValues contentValues) {
        Log.i(TAG,"insert()");
        String tableName = getTableName(uri);
        if(tableName == null){
            throw new IllegalArgumentException("Unsupported URI:" + uri);
```

```
        }
        mDb.insert(tableName, null, contentValues);
        getContext().getContentResolver().notifyChange(uri, null);
        return uri;
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        Log.i(TAG,"delete()");
        String tableName = getTableName(uri);
        if(tableName == null){
            throw new IllegalArgumentException("Unsupported URI:" + uri);
        }
        int count = mDb.delete(tableName, selection, selectionArgs);
        if(count > 0){
            getContext().getContentResolver().notifyChange(uri, null);
        }
        return count;
    }

    @Override
    public int update(Uri uri, ContentValues contentValues, String selection, String[] selectionArgs) {
        Log.i(TAG,"update()");
        String tableName = getTableName(uri);
        if(tableName == null){
            throw new IllegalArgumentException("Unsupported URI:" + uri);
        }
        int row = mDb.update(tableName, contentValues, selection, selectionArgs);
        if(row > 0){
            getContext().getContentResolver().notifyChange(uri, null);
        }
        return row;
    }
}
```

> AndroidMainifest权限:

```
        <!--自定义ContentProvider-->
        <provider
            android:authorities="customPrivoderName"
            android:name=".PetProvider"
            android:exported="true"
            >
        </provider>
```

> 在客户端使用ContentProvider(增删改查):
> 增:

```
ContentValues values = new ContentValues();
values.put("_id", 1);
values.put("name", "dog");
getContentResolver().insert(uri, values);
```

> 删:

```
//删除
getContentResolver().delete(uri, "_id=?", new String[]{"2"});
```

> 改:

```
//更新
ContentValues values = new ContentValues();
values.put("_id", 3);
values.put("name", "fox");
getContentResolver().update(uri, values, "_id=?",new String[]{"1"});
```

> 查找:

```
Cursor petCursor = getContentResolver().query(uri, new String[]{"_id", "name"}, null, null, null);
if(petCursor != null){
   while(petCursor.moveToNext()){
       Log.i("MainActivity", ""+petCursor.getInt(0));
       Log.i("MainActivity", ""+petCursor.getString(1));
   }
   petCursor.close();
}
```

6-Socket

32、Sokcet就可以进行进程间通信

选用合适的IPC方式(1)

1、IPC各种方法的优缺点和适用场景:

名称	优点	缺点	适用场景
Bundle	简单易用	只能传输Bundle支持的数据类型	四大组件间的进程间通信
文件共享	简单易用	不适合高并发场景， 无法做到进程间的即时通信	无并发访问情形， 交换简单数据实时性不高的场景
AIDL	功能强大， 支持一对多并发通信， 支持实时通信	适用稍复杂， 需要处理好线程同步	一对多通信且有RPC需求
Messenger	功能一般， 支持一对多串行通信， 支持实时通信	不能很好处理高并发情形， 不支持RPC， 数据通过Message传输因此只能传输Bundle支持的数据类型	低并发的一对多即时通信， 无RPC需求， 或者无须要返回结果的RPC需求
ContentProvider	在数据源访问方面功能强大， 支持一对多并发数据共享， 可通过Call方法扩展其他操作	可以理解为受约束的AIDL， 主要提供数据源的CRUD操作	一对多的进程间的数据共享
Socket	功能强大， 可以通过网络传输字节流， 支持一对多并发实时通信	实现细节稍微繁琐， 不支持直接的RPC	网络数据交换

参考资料

1. 面试题-听说你精通Binder?

2. Android笔记九（ServiceManager浅析）
3. Android-ServiceManager