**IPC**(Inter-Process Communication)**机制**

参考https://mp.weixin.qq.com/s/LwQ53yQ6d\_sbDAJ2CguTJQ

**基础概念**

IPC机制：进程间通信或者跨进程通信，是指两个进程间进行数据交换的过程。

线程：CPU最小的调度单元，它是一种有限的资源。

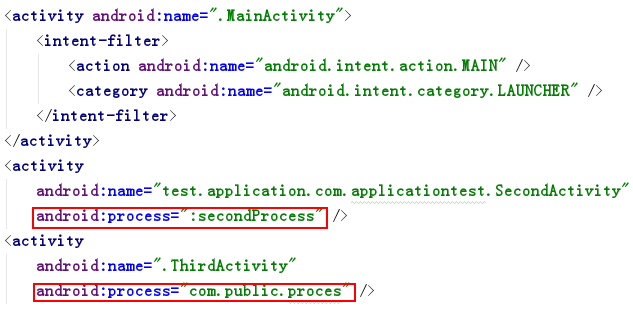
进程：指一个执行单元。执行线程的单元，线程由CPU调度，然后由进程执行。

**Android的多进程模式**

在android中使用多进程一般是指**一个应用中**存在多个进程，所以这里不涉及两个应用之间的多进程。同一个应用中的多进程可以理解为两个不同的应用采用了相同的SharedUID的模式。

* **开启多进程步骤**

在android中开启多进程只有一种方式，那就是在AndroidMenifest中给android的四大组件指定android：process属性。如下：



根据上面的配置在启动三个Activity的时候，会启动三个进程：

MainActivity：在默认的进程中运行，进程名为应用包名。

SecondActivity:单独的进程，进程名为“包名+:secondProcess”

ThirdActivity:单独的进程，进程名为“com.public.proces”

* **Android：process的命名方式：**

1. 以“：”开头，这是一种简写方式，它完整的进程名为“包名 + : + 进程名”。
2. 不以“：”开头，它是一种完整的命名方式，不会附加包名信息。如ThirdActivity。

区别：“包名 + : + 进程名”的进程属于当前应用的私有进程其他应用的组件不可以和它跑在同一个进程中。其他的组属于全局进程，其他应用通过设置相同的ShareUID并且签名相同才可以和全局进程跑在同一个进程。

另外，只要拥有相同UID的进程就能互相访问对方的私有数据，比如data目录、组件信息等。但是如果这两个进程能跑在同一个进程中，还可以共享内存中的数据，或者说它们看起来像一个应用的两个部分。

* **Android多进程模式的运行机制**

**Android系统为每一个进程（或应用）分配了独立的虚拟机。**不同的虚拟机在内存分配上拥有不同的地址空间，所以所有运行在不同进程中的四大组件，**只要通过内存来共享数据，都会共享失败，**因为在一个进程中去访问另一个进程的对象只会访问这个对象的副本，它们是两个不同的对象。

以上是多进程的本质机制，由于这个机制会使不同进程的组件拥有各自独立的虚拟机、Application、内存空间，所以也会出现一些列的问题：

1. 静态成员、单例模式、线程同步机制将完全失效，因为他们访问的不是同一个对象。
2. SharedPreferences的可靠性下降，因为SharedPreferences底层是通过读/写XML文件来实现的，

两个进程并发的读写显然是会出问题的。

1. Application会多次创建。当一个应用启动一个新的进程的时候，这时候系统在创建新的进程的时

候会分配一个新的独立的虚拟机，这个过程就是启动一个应用的过程，那么自然会为新的进程创建一

个新的Application。

**跨进程通信基础**

跨进程通信用到一些基础概念，主要包括：Serializable接口、Parcelable接口、Binder。Serializable和Parcelable接口可以完成对象的序列化过程。在使用Intent和Binder传输数据时就需要使用Serializable和Parcelable接口。

如何对对象进行序列化和反序列只要采用ObjectOoutputStream和ObjectInputStream就能实现，如下：



* **Serializable接口**

Serializable接口是Java提供的一个序列化接口，它是一个空接口，为对象提供标准的序列化和反序列化操作。使用方式很简单，只要在想要序列化的类上实现这个接口就可以。几乎所有的序列化工作都被系统做了。

* **Parcelable接口**



Parcelable是android提供的一种序列化方式，它也是一个接口，上面就是Parcelable序列化典型的使用方式，下面是接口中实现方法：

createFromParcel(Paecel in)：从序列化的对象中创建原始对象。

newArray(int size)：创建指定长度的原始对象数组。

User(Parcel in): 从序列化后的对象中创建原始对象。

writeToParcel(...)：将当前对象写入到序列化结构中。其中flags有两种值：当值为1时标识当前对象需要作为返回值返回，不能立即释放资源，几乎所有情况下值都为0。

describeContents：返回当前对象的内容描述，如果含有文件描述返回1，几乎所有情况下都返回0。

* **Serializable和Parcelable两个接口区别**

可参考：<http://blog.csdn.net/u010794180/article/details/43796045>

它们两个共同特点：首先静态成员不属于对象，所以不会参加序列化过程。其次用transient关键字标识的成员变量不参与序列化过程。

Serializable：java中序列化接口，使用起来简单，但是开销很大，序列化和反序列化过程需要大量的I/O操作。如果将对象序列化到存储设备中或者序列化后通过网络传输建议使用这个接口，因为Parcelable接口使用起来很复杂。

Parcelable：是android中的序列化方式，是android推荐的序列化方式，使用起来麻烦但是效率很高，**它主要用在内存序列化上。**

* **Binder**

Binder有两个概念，一个是内核空间的Binder驱动。另一个是运行在用户空间的Binder对象。这里只介绍用户空间的Binder对象，在AIDL篇对Binder对象的使用完成了跨进程通信。在服务端通过sdk编译，生成由AIDL定义的接口相对应的Java接口，这个Java接口中的方法就是AIDL中定义的方法,同时这个java接口也继承自IInterface接口。在这个接口中还生成了一个内部类，真正用于跨进程通信的的Binder的实现类Stub，它是一个抽象类，同时也继承了由AIDL定义的Java接口，我们只需要在服务端实现抽象类Stub即AIDL中定义的具体接口。了解了Stub类的实现就了解了Binder的工作机制，看下面一个例子，它的结构还是很清晰的：



**DESCRIPTOR**

Binder的唯一标识，一般用当前Binder的类名表示。

**asInterface(android.os.IBinder obj)**

在客户端中用于将服务端的Binder对象转换成客户端所需要的AIDL类型的Binder对象，关键点就obj.queryLocalInterface(DESCRIPTOR)这个方法，如果客户端和服务端在同一进程那么obj.queryLocalInterface(DESCRIPTOR)返回的就是obj自身，最终asInterface方法返回的也是obj这个对象；如果不在同一进程obj.queryLocalInterface(DESCRIPTOR)则返回的就是null，最终asInterface方法会new一个服务端Binder的代理对象返回。

**asBinder**

返回当前Binder对象。这个方法是IInterface接口的唯一方法，Binder实现类Stub和Binder代理类Proxy都实现了IInterface接口，并且asBinder返回的永远是一个Binder对象。

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

处理客户端跨进程通信发送的请求。这个方法运行在服务端的Binder线程池中，当客户端调用它本地的代理类跨进程请求服务端时，客户端的请求会经过系统底层的封装交由服务端的onTransact方法处理。当服务端接收到客户端的请求时，首先通过code判断具体请求的目标方法，如果请求携带了参数则从data中取出参数，并调用和传递给服务端的目标方法，如果目标方法有返回值，当目标方法执行完后将目标方法的返回值写入到reply中返回给客户端。另外onTransact方法的返回值是boolean类型，返回true调用服务端方法成功，返回false调用服务端方法无效。可以通过这个特性来做客户端请求的权限验证。

它接受参数如下：

code：确定客户端请求的具体是什么方法。代表这些方法的键值在Stub中定义。

data：客户端传递进来的参数。

reply：当服务端执行完客户端请求的方法时，如果客户端需要返回参数，则将返回参数写入到reply对象中返回给客户端。

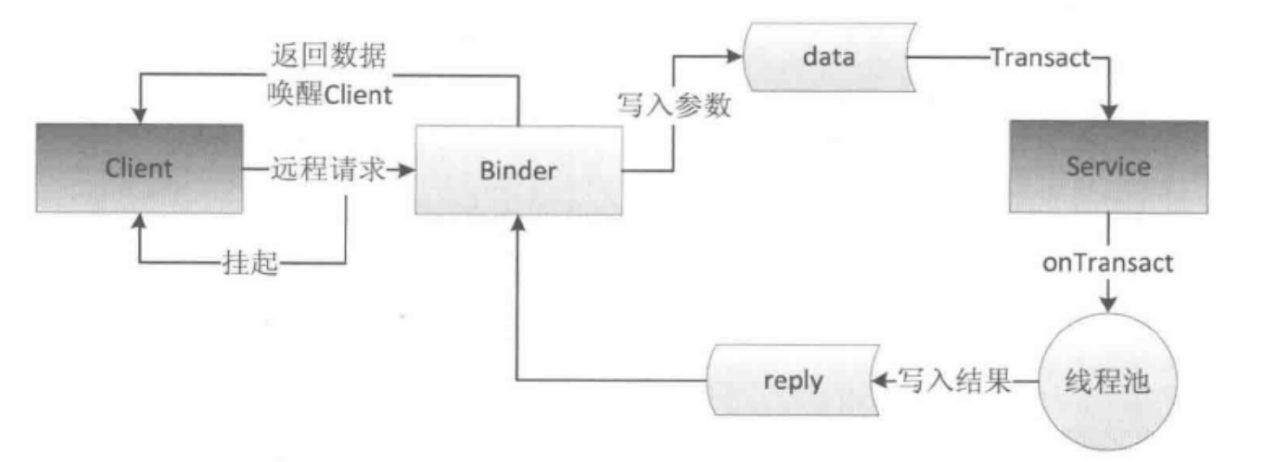
flags：由AIDL生成的java接口中默认是0，并且没有使用到这个参数。

**Binder代理类Proxy**

客户端跨进程请求服务端会在本地生成一个服务端Binder的代理类。代理类实现了与服务端同样java接口，即也同时实现了IInterface接口，代理类中的方法运行在客户端，主要是封装客户端的请求然后调用transact方法并将参数传递给它发起远程过程调用请求，同时挂起当前线程。接着服务端的onTransact方法会被调用并执行目标方法，执行完后返回个客户端执行结果，客户端接受到执行结果后继续执行当前线程，并从\_reply中取出服务端返回的结果，并通过AIDL接口中定义方法的形参修饰符定向tag(in/out/inout)是否将客户端传递的参数与服务端进行同步。

**注意：**客户端跨进程请求服务端后客户端当前的线程会被挂起，即当前线程会被阻塞，直到远程服务返回数据，所以假如客户端在主线程请求了一个很耗时的方法，最好另起一个线程来执行这个操作。由于服务端的Binder方法运行在Binder线程池中，所以当Binder的方法都应该同步的方式去实现，因为他已经运行在一个线程中了。

了解了Binder的实现类Stub和代理类Proxy之后用户空间的Binder的运行机制大体就了解了，下面参考《android开发艺术探索》的一张图总结Binder的运行机制：



**IPC机制**

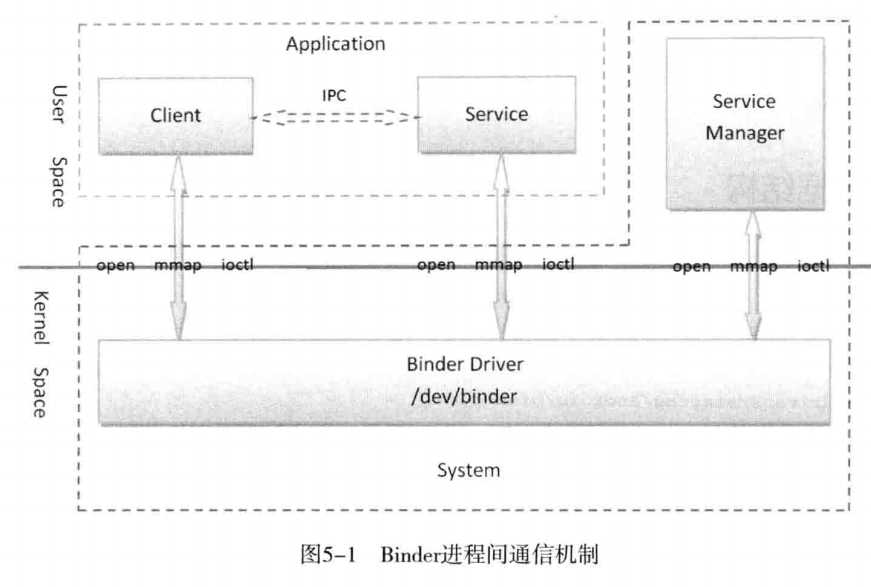
Android系统是基于Linux内核开发的，Linux内核提供了丰富的进程间通信机制，管道（Pipe）、信号（Signal）、消息队列（Message）、共享内存（ShareMemory）、插口（Socket）等。然而Android系统并没有采用这些传统的进程间通信机制，而是开发了一套新的通信机制——Binder。

IPC的意思是进程间通信，即两个进程之间**相互交换数据**的过程。其中提供服务的进程叫Service进程，而访问服务的进程叫Client进程。Binder间通信机制的每一个Service进程和Client进程各自都维护了一个Binder线程池来处理两个进程之间相互通信的请求。因此它们也可以并发的提供和访问服务。

Service进程和Client进程需要通过内核层Binder**驱动程序**来进行通信。Binder驱动程序向用户空间暴露了一个设备文件/dev/binder，是的进程之间可以间接的通过它来建立通信通道。

Service进程可以提供多个组件来向Client进程提供服务，这些组件被称为Service组件，Service组件在启动的时候会将自己注册到ServiceManager组件中，以便Client组件可以通过Service组件找到它，因此ServiceManager组件也是Binder通信机制中的上下文管理者，这里ServiceManager也需要与Client进程和Service进程进行通信，即也需要进行IPC，此时ServiceManager是它们的服务端。

下面借鉴一张图来描述IPC机制：



从上图中可以看出Client、Service、ServiceManager运行在用户空间，Binder驱动程序运行在内核空间，其中ServiceManager和Binder驱动程序由系统负责，而Client和Service由应用程序来实现。

* **内核空间的Binder驱动**

Server进程和Client进程要依靠内核空间的Binder驱动程序来进行通信，Binder驱动驱动程序向用户空间暴露了一个设备文件/dev/binder，使应用程序可以间接的通过它来进行进程间通信。

* **ServiceManager**

Service进程在启动时会将自己注册到一个ServiceManager中以便Client进程可以找到它。