|  |
| --- |
| 1、Binder是android中一种特有的跨进程通信方式，你使用binder，就能拿到别的进程的服务。  应用场景：媒体播放、音视频捕获、智能传感器（加速度、方位、温度、亮度等），甚至是startActivity、startService都用到了binder。  存在一个Server： 提供服务，并负责一些管理。  而APP应用程序作为一个client客户端：主要与这些server建立连接，就可以轻松的使用服务。  2、AIDL是一种架构，能够为我们使用binder的时候，更加的轻便，简单。使用AIDL(android接口定义语言)，专门为binder设计的，是一种框架。  IPC，进程间通信(Inter Process Communication)。  Binder是一种进程间通信的方式。  3、Linux 已经提供了管道、消息队列、共享内存和 Socket 等 IPC 机制。  所以：提出需求--> 高效率、安全高的进程间通信。  就是Binder了，是C/S 模型，只需要一次的内存copy。  性能很高，仅次于内存共享，并且有PID、UID的校验。  但是，Binder的优势：  优势 描述  性能 只需要一次数据拷贝，性能上仅次于共享内存  稳定性 基于 C/S 架构，职责明确、架构清晰，因此稳定性好  安全性 为每个 APP 分配 UID，进程的 UID 是鉴别进程身份的重要标志  4、动态内核可加载模块 && 内存映射。  Binder IPC 正是基于内存映射（mmap）来实现的，  内存映射简单的讲就是将用户空间的一块内存区域映射到内核空间。映射关系建立后，用户对这块内存区域的修改可以直接反应到内核空间；反之内核空间对这段区域的修改也能直接反应到用户空间。  5、IBinder 代表了跨进程通信的能力，接口  IInterface 代表server提供的服务  Binder java层的binder代理类 BinderProxy  stub 编译生成的静态内部类，继承与Binder,又实现了IInterface  6、Binder通信流程  1）client打包数据  2）发送数据client--> binder驱动  3）发送数据binder驱动--> server  4）server数据解包  5）server执行目标方法  6）返回结果server--> binder驱动  7）返回结果binder驱动-->client  7.编码实现跨进程调用  （1）可以通过手写实现Binder IPC.  （2）也可以借助AIDL实现Binder通信。  1）客户端：定义一个aidl文件，里面是服务的接口。  2）客户端：IDE同步之后，AS会自动生成一个aidl对应的java类  3）服务端：同样执行上面的1、2步骤。然后创建一个Service。  需要重写onBind()方法，返回一个IBinde实现对象，即ILoginInterface.Stub。  再重写你提供的服务，即定义的接口。  比如，这里 login() 内启动一个登录页面。  4）客户端：在客户端启动服务（绑定状态的）  a.绑定服务bindService(); //绑定的服务是服务端的服务。  b.上面的绑定服务需要传一个参数服务连接：ServiceConnection对象。  需要重写方法onServiceConnected，这里初始化ILoginInterface iLogin;（就是IBinder对象）  c.用户点击事件就可以调用iLogin的方法，login()登录了，就开始跨进程服务了。 |

Binder 原理剖析https://juejin.im/post/6844903589635162126

# 1.Binder进程间通信机制

Binder 是一种进程间通信机制，基于开源的 OpenBinder 实现；OpenBinder 起初由 Be Inc. 开发，后由 Plam Inc. 接手。从字面上来解释 Binder 有胶水、粘合剂的意思，顾名思义就是粘和不同的进程，使之实现通信。

IPC，进程间通信(Inter Process Communication)。

Binder是一种进程间通信的方式。

使用AIDL(接口定义语言)，专门为binder设计的，是一种框架。

# 2.为什么是 Binder ?

Android 系统是基于 Linux 内核的，Linux 已经提供了管道、消息队列、共享内存和 Socket 等 IPC 机制。那为什么 Android 还要提供 Binder 来实现 IPC 呢？主要是基于性能、稳定性和安全性几方面的原因。

（1）性能

首先说说性能上的优势。

Socket 作为一款通用接口，其传输效率低，开销大，主要用在跨网络的进程间通信和本机上进程间的低速通信。

消息队列和管道采用存储-转发方式，即数据先从发送方缓存区拷贝到内核开辟的缓存区中，然后再从内核缓存区拷贝到接收方缓存区，至少有两次拷贝过程。

共享内存虽然无需拷贝，但控制复杂，难以使用。

Binder 只需要一次数据拷贝，性能上仅次于共享内存。

（2）稳定性

再说说稳定性，Binder 基于 C/S 架构，客户端（Client）有什么需求就丢给服务端（Server）去完成，架构清晰、职责明确又相互独立，自然稳定性更好。

共享内存虽然无需拷贝，但是控制负责，难以使用。

从稳定性的角度讲，Binder 机制是优于内存共享的。

（3）安全性

另一方面就是安全性。Android 作为一个开放性的平台，市场上有各类海量的应用供用户选择安装，因此安全性对于 Android 平台而言极其重要。作为用户当然不希望我们下载的 APP 偷偷读取我的通信录，上传我的隐私数据，后台偷跑流量、消耗手机电量。传统的 IPC 没有任何安全措施，完全依赖上层协议来确保。首先传统的 IPC 接收方无法获得对方可靠的进程用户ID/进程ID（UID/PID），从而无法鉴别对方身份。Android 为每个安装好的 APP 分配了自己的 UID，故而进程的 UID 是鉴别进程身份的重要标志。传统的 IPC 只能由用户在数据包中填入 UID/PID，但这样不可靠，容易被恶意程序利用。可靠的身份标识只有由 IPC 机制在内核中添加。其次传统的 IPC 访问接入点是开放的，只要知道这些接入点的程序都可以和对端建立连接，不管怎样都无法阻止恶意程序通过猜测接收方地址获得连接。

同时 Binder 既支持实名 Binder，又支持匿名 Binder，安全性高。

基于上述原因，Android 需要建立一套新的 IPC 机制来满足系统对稳定性、传输性能和安全性方面的要求，这就是 Binder。

最后用一张表格来总结下 Binder 的优势：

优势 描述

性能 只需要一次数据拷贝，性能上仅次于共享内存

稳定性 基于 C/S 架构，职责明确、架构清晰，因此稳定性好

安全性 为每个 APP 分配 UID，进程的 UID 是鉴别进程身份的重要标志

# 3. Linux 下传统的进程间通信原理

了解 Linux IPC 相关的概念和原理有助于我们理解 Binder 通信原理。因此，在介绍 Binder 跨进程通信原理之前，我们先聊聊 Linux 系统下传统的进程间通信是如何实现。

3.1 基本概念介绍

这里我们先从 Linux 中进程间通信涉及的一些基本概念开始介绍，然后逐步展开，向大家说明传统的进程间通信的原理。

Liunx 中跨进程通信涉及到的一些基本概念：

进程隔离

进程空间划分：用户空间(User Space)/内核空间(Kernel Space)

系统调用：用户态/内核态

进程隔离简单的说就是操作系统中，进程与进程间内存是不共享的。两个进程就像两个平行的世界，A 进程没法直接访问 B 进程的数据，这就是进程隔离的通俗解释。A 进程和 B 进程之间要进行数据交互就得采用特殊的通信机制：进程间通信（IPC）。

进程空间划分：用户空间(User Space)/内核空间(Kernel Space)

现在操作系统都是采用的虚拟存储器，对于 32 位系统而言，它的寻址空间（虚拟存储空间）就是 2 的 32 次方，也就是 4GB。操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也可以访问底层硬件设备的权限。为了保护用户进程不能直接操作内核，保证内核的安全，操作系统从逻辑上将虚拟空间划分为用户空间（User Space）和内核空间（Kernel Space）。针对 Linux 操作系统而言，将最高的 1GB 字节供内核使用，称为内核空间；较低的 3GB 字节供各进程使用，称为用户空间。

简单的说就是，内核空间（Kernel）是系统内核运行的空间，用户空间（User Space）是用户程序运行的空间。为了保证安全性，它们之间是隔离的。

系统调用：用户态与内核态

虽然从逻辑上进行了用户空间和内核空间的划分，但不可避免的用户空间需要访问内核资源，比如文件操作、访问网络等等。为了突破隔离限制，就需要借助系统调用来实现。系统调用是用户空间访问内核空间的唯一方式，保证了所有的资源访问都是在内核的控制下进行的，避免了用户程序对系统资源的越权访问，提升了系统安全性和稳定性。

Linux 使用两级保护机制：0 级供系统内核使用，3 级供用户程序使用。

当一个任务（进程）执行系统调用而陷入内核代码中执行时，称进程处于内核运行态（内核态）。此时处理器处于特权级最高的（0级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。

当进程在执行用户自己的代码的时候，我们称其处于用户运行态（用户态）。此时处理器在特权级最低的（3级）用户代码中运行。

系统调用主要通过如下两个函数来实现：

copy\_from\_user() //将数据从用户空间拷贝到内核空间

copy\_to\_user() //将数据从内核空间拷贝到用户空间

3.2 Linux 下的传统 IPC 通信原理

理解了上面的几个概念，我们再来看看传统的 IPC 方式中，进程之间是如何实现通信的。

通常的做法是消息发送方将要发送的数据存放在内存缓存区中，通过系统调用进入内核态。然后内核程序在内核空间分配内存，开辟一块内核缓存区，调用 copy\_from\_user() 函数将数据从用户空间的内存缓存区拷贝到内核空间的内核缓存区中。同样的，接收方进程在接收数据时在自己的用户空间开辟一块内存缓存区，然后内核程序调用 copy\_to\_user() 函数将数据从内核缓存区拷贝到接收进程的内存缓存区。这样数据发送方进程和数据接收方进程就完成了一次数据传输，我们称完成了一次进程间通信。如下图：

这种传统的 IPC 通信方式有两个问题：

性能低下，一次数据传递需要经历：内存缓存区 --> 内核缓存区 --> 内存缓存区，需要 2 次数据拷贝；

接收数据的缓存区由数据接收进程提供，但是接收进程并不知道需要多大的空间来存放将要传递过来的数据，因此只能开辟尽可能大的内存空间或者先调用 API 接收消息头来获取消息体的大小，这两种做法不是浪费空间就是浪费时间。

# 4. Binder 跨进程通信原理

理解了 Linux IPC 相关概念和通信原理，接下来我们正式介绍下 Binder IPC 的原理。

动态内核可加载模块 && 内存映射

正如前面所说，跨进程通信是需要内核空间做支持的。传统的 IPC 机制如管道、Socket 都是内核的一部分，因此通过内核支持来实现进程间通信自然是没问题的。但是 Binder 并不是 Linux 系统内核的一部分，那怎么办呢？这就得益于 Linux 的动态内核可加载模块（Loadable Kernel Module，LKM）的机制；模块是具有独立功能的程序，它可以被单独编译，但是不能独立运行。它在运行时被链接到内核作为内核的一部分运行。这样，Android 系统就可以通过动态添加一个内核模块运行在内核空间，用户进程之间通过这个内核模块作为桥梁来实现通信。

在 Android 系统中，这个运行在内核空间，负责各个用户进程通过 Binder 实现通信的内核模块就叫 Binder 驱动（Binder Dirver）。

那么在 Android 系统中用户进程之间是如何通过这个内核模块（Binder 驱动）来实现通信的呢？难道是和前面说的传统 IPC 机制一样，先将数据从发送方进程拷贝到内核缓存区，然后再将数据从内核缓存区拷贝到接收方进程，通过两次拷贝来实现吗？显然不是，否则也不会有开篇所说的 Binder 在性能方面的优势了。

这就不得不通道 Linux 下的另一个概念：内存映射。

Binder IPC 机制中涉及到的内存映射通过 mmap() 来实现，mmap() 是操作系统中一种内存映射的方法。内存映射简单的讲就是将用户空间的一块内存区域映射到内核空间。映射关系建立后，用户对这块内存区域的修改可以直接反应到内核空间；反之内核空间对这段区域的修改也能直接反应到用户空间。

内存映射能减少数据拷贝次数，实现用户空间和内核空间的高效互动。两个空间各自的修改能直接反映在映射的内存区域，从而被对方空间及时感知。也正因为如此，内存映射能够提供对进程间通信的支持。

# Binder IPC 实现原理

Binder IPC 正是基于内存映射（mmap）来实现的，但是 mmap() 通常是用在有物理介质的文件系统上的。

比如进程中的用户区域是不能直接和物理设备打交道的，如果想要把磁盘上的数据读取到进程的用户区域，需要两次拷贝（磁盘-->内核空间-->用户空间）；通常在这种场景下 mmap() 就能发挥作用，通过在物理介质和用户空间之间建立映射，减少数据的拷贝次数，用内存读写取代I/O读写，提高文件读取效率。

而 Binder 并不存在物理介质，因此 Binder 驱动使用 mmap() 并不是为了在物理介质和用户空间之间建立映射，而是用来在内核空间创建数据接收的缓存空间。

一次完整的 Binder IPC 通信过程通常是这样：

首先 Binder 驱动在内核空间创建一个数据接收缓存区；

接着在内核空间开辟一块内核缓存区，建立内核缓存区和内核中数据接收缓存区之间的映射关系，以及内核中数据接收缓存区和接收进程用户空间地址的映射关系；

发送方进程通过系统调用 copy\_from\_user() 将数据 copy 到内核中的内核缓存区，由于内核缓存区和接收进程的用户空间存在内存映射，因此也就相当于把数据发送到了接收进程的用户空间，这样便完成了一次进程间的通信。

# Binder 通信模型

介绍完 Binder IPC 的底层通信原理，接下来我们看看实现层面是如何设计的。

一次完整的进程间通信必然至少包含两个进程，通常我们称通信的双方分别为客户端进程（Client）和服务端进程（Server），由于进程隔离机制的存在，通信双方必然需要借助 Binder 来实现。

从进程间通信的角度看，Binder 是一种进程间通信的机制；

从 Server 进程的角度看，Binder 指的是 Server 中的 Binder 实体对象；

从 Client 进程的角度看，Binder 指的是对 Binder 代理对象，是 Binder 实体对象的一个远程代理。

从传输过程的角度看，Binder 是一个可以跨进程传输的对象；Binder 驱动会对这个跨越进程边界的对象对一点点特殊处理，自动完成代理对象和本地对象之间的转换。

# 编码实现跨进程调用

1. 可以通过手写实现Binder IPC.
2. 也可以借助AIDL实现Binder通信。

1）客户端：定义一个aidl文件，里面是服务的接口：

interface ILoginInterface {

// 登录

void login();

// 登录返回

void loginCallback(boolean loginStatus, String loginUser);

}

2）客户端：IDE同步之后，AS会自动生成一个aidl对应的java类

public interface ILoginInterface extends android.os.IInterface {

public static abstract class Stub extends android.os.Binder implements

com.netease.binder.ILoginInterface {

private static class Proxy implements com.netease.binder.ILoginInterface{

......

}

}

}

1. 服务端：同样执行上面的1、2步骤。然后创建一个Service。

需要重写onBind()方法，返回一个IBinde实现对象，即ILoginInterface.Stub。

再重写你提供的服务，即定义的接口。

比如，这里 login() 内启动一个登录页面。

@Override

public IBinder onBind(Intent intent) {

return new ILoginInterface.Stub() {

@Override

public void login() throws RemoteException {

Log.e("netease >>> ", "BinderB\_MyService");

// 单项通信有问题，真实项目双向通信，双服务绑定 ！！！

//A-->B A调起B进行操作

//有结果后，是B再调起A的服务

serviceStartActivity();

}

@Override

public void loginCallback(boolean loginStatus, String loginUser) throws RemoteException {

}

};

}

1. 客户端：在客户端启动服务（绑定状态的）
2. 绑定服务bindService(); //绑定的服务是服务端的服务。
3. 上面的绑定服务需要传一个参数服务连接：ServiceConnection对象。

需要重写方法onServiceConnected，这里初始化ILoginInterface iLogin;（就是IBinder对象）

// 服务连接

private ServiceConnection conn = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

//这样IBinder 转化为了 aidl对应的类

iLogin = ILoginInterface.Stub.asInterface(service);

}

@Override

public void onServiceDisconnected(ComponentName name) {

}

};

1. 用户点击事件就可以调用iLogin的方法，login()登录了，就开始跨进程服务了。