Binder对于Android开发而言，是极为重要的一个专业术语。从最直观的角度看，它是Android的一个类，继承了Binder接口。从IPC的角度而言，Binder是一种进程间通信机制，Binder还可以理解为一种虚拟的物理设备。Binder驱动，是一个动态内核可加载模块。负责各个用户进程通过Binder实现通信的内核模块叫Binder驱动。从Framework层面和应用层面，Binder是连接其各部分的桥梁。

多进程基础

在将Binder之前先让我们看看Android多进程概念。在Android中对于单个进程有着内存限制（

> 1、每个进程分配一个JVM虚拟机，更加的安全。
>
> 2、实际上由于手机的屏幕显示内容有限，内容足够即可。
>
> 3、使开发者对内存使用更为合理。避免一个应用占用过多内存，而导致其他进程无法正常运行。）所以Android引入了多进程的概念，允许同一应用内，为了分担主进程压力，将占用内存的某些页面单独开一个线程。

在Android中进程有这几个等级

> 前台进程，
>
> 可见进程，
>
> 服务进程，
>
> 后台/缓存进程
>
> 空进程。

进程的创建较为简单，在AndroidManifest.xml的声明四大组件的标签中增加”android:process”属性即可

1为什么需要Binder机制

在传统的Linux上，有着很多选择可以实现进程间通信，管道，socket，信号量，共享空间等，为什么要新建一个Binder的进程间通信机制呢?

性能角度

1. **管道：** 消息队列 套接字都需要两次数据拷贝，共享内存方式一次拷贝都不需要，Binder也只需要拷贝一次，其性能仅次于共享内存稳定性角度 Binder基于CS架构，Server与Client断相对独立，
2. **稳定性较好：** 。共享内存实现方式复杂且没有客户与服务端之别，需要充分考虑到访问临界并发同步问题。安全角度传统的IPC对于通信双方的身份没有做出严格的验证，使用传统IPC机制，只能由客户在数据包内填入UID/PID,不可靠.Android为每个安装好的应用程序分配了自己的UID，故进程的UID是鉴别进程身份的重要标志，Android系统只向外暴露Client端，Client端将任务发送给Server端，根据权限控制策略，判断PID与UID是否符合权限。
3. **语言角度：** Linux是基于C语言编写的（面向过程的语言），Android基于Java语言（面向对象），对于Binder恰恰符合面向对象的思想，将进程间通信转化为通过对Binder对象的引用调用该对象

> 的方法，而其独特之处在于Binder对象是一个可以跨进程引用的对象，它的实体位于一个进程中，而它的引用却遍布于系统的各个进程之中。可以从一个进程传给其它进程，让大家都能访问同一Server，就像将一个对象或引用赋值给另一个引用一样。Binder模糊了进程边界，淡化了进程间通信过程，整个系统仿佛运行于同一个面向对象的程序之中。从语言层面，Binder更适合基于面向对象语言的Android系统，对于Linux系统可能会有点"水土不服"综合以上四点，就是为什么谷歌要创造一个Binder机制的原因了。其中个人感觉最重要的是第三点，安全问题。

2 Binder跨进程通信原理

2.1 Linux内核的基础知识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：

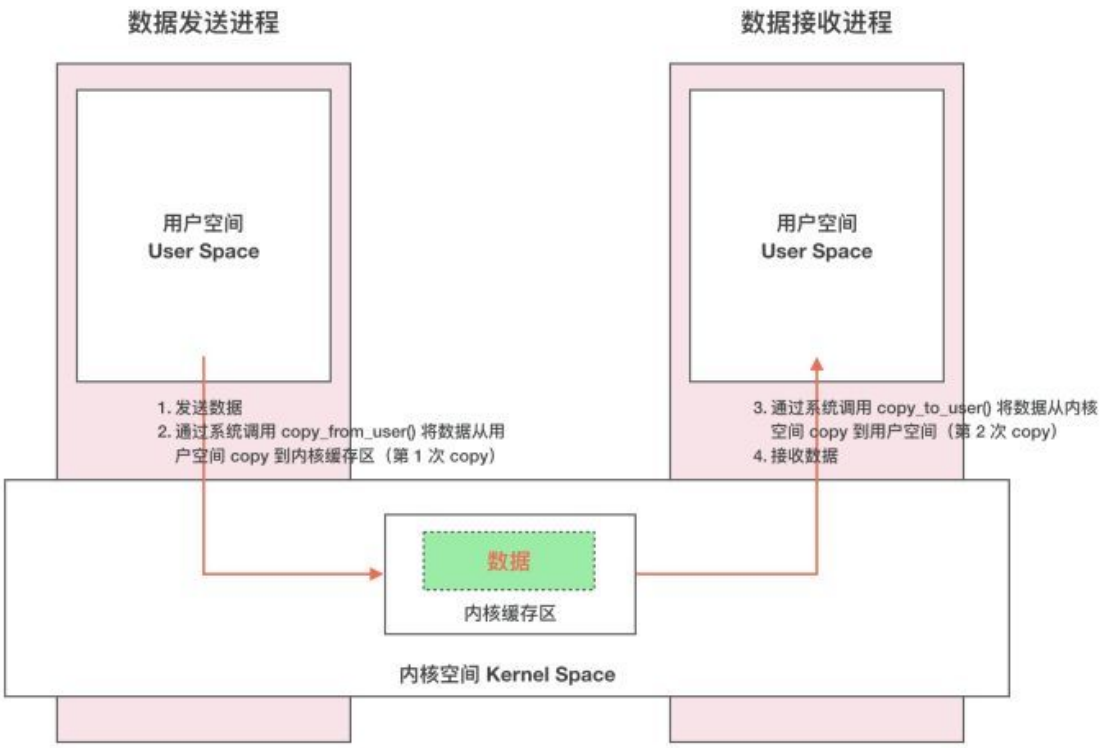
进程间通信（IPC）。

进程空间划分：用户空间/内核空间用户空间指的是用户程序所运行的空间，内核空间是 Linux 内核的运行空间，为了安全，它们是隔离的，即使用户的程序崩溃了，内核也不受影响。

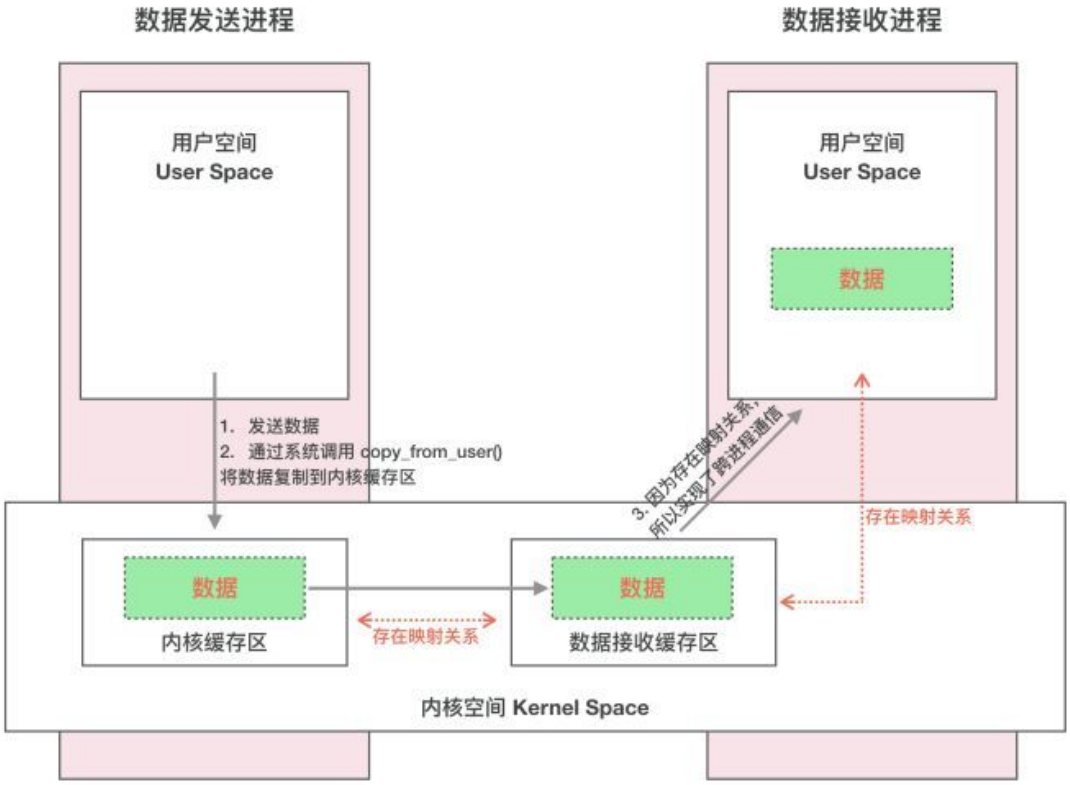
系统调用：用户态和内核态虽然从逻辑上进行了用户空间和内核空间的划分，但不可避免的用户空间需要访问内核资源，比如文件操作、访问网络等等。为了突破隔离限制，就需要借助**系统调用**来实现。

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*2.2 Linux下的传统IPC通信原理

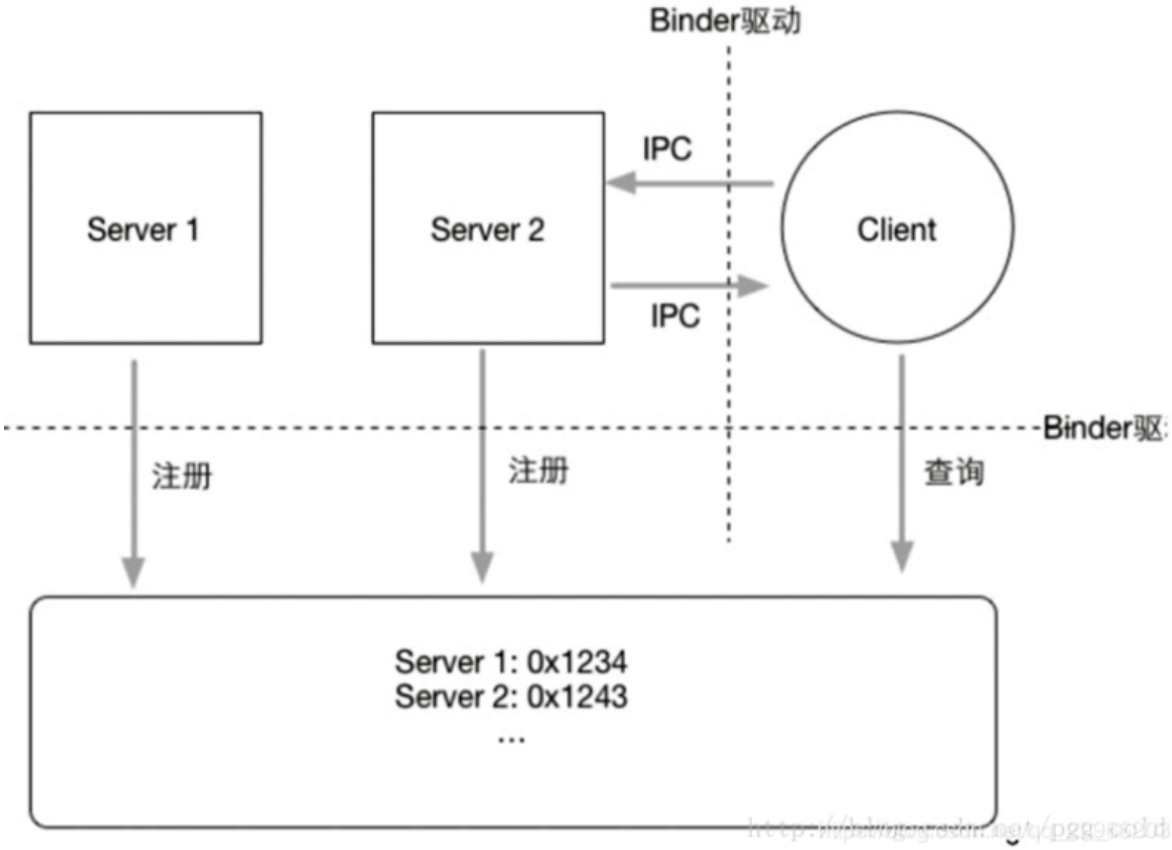


通常的做法是消息发送方将要发送的数据存放在内存缓存区中，通过系统调用进入内核态。然后内核程序在内核空间分配内存，开辟一块内核缓存区，调用 copy*from*user() 函数将数据从用户空间的内存缓存区拷贝到内核空间的内核缓存区中。同样的，接收方进程在接收数据时在自己的用户空间开辟一块内存缓存区，然后内核程序调用 copy*to*user() 函数将数据从内核缓存区拷贝到接收进程的内存缓存区。这样数据发送方进程和数据接收方进程就完成了一次数据传输，我们称完成了一次进程间通信。这种传统的 IPC 通信方式有两个问题：性能低下，一次数据传递需要经历：内存缓存区 --> 内核缓存区 --> 内存缓存区，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。2.3 Binder跨进程通信原理动态内核可加载模块 && 内存映射这个动态内核可加载模块，就是Binder驱动，运行在内核空间，使各个用户进程通过Binder实现通信。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涉及到的内存映射通过mmap（）实现。Binder实现原理



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存缓存区，建立**内核缓冲区**和**内核中数据接收缓存缓存区**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**的映射关系发送方通过系统调用copyfromuser（）将数据copy到内核中的内核缓冲区，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。3 Binder通信模型进程间通信必然包含两个进程，通常我们将通信的双方分别为客户端进程（Client）和服务端进程（Server），由于进程隔离机智存在，通信双方必然需要借助Binder来实现。3.1 **Client/Server/ServiceManager/驱动**Binder是C/S架构，有Client，Server，ServiceManager，Binder驱动四部分组成。其中 Client、Server、Service Manager 运行在用户空间，Binder 驱动运行在内核空间。其中 Service Manager 和 Binder 驱动由系统提供，而 Client、Server 由应用程序来实现。Client、Server 和 ServiceManager 均是通过系统调用 open、mmap 和 ioctl 来访问设备文件

/dev/binder，从而实现与 Binder 驱动的交互来间接的实现跨进程通信。



- Client、Server、ServiceManager、Binder 驱动这几个组件在通信过程中扮演的角色就如同互联网中服务器（Server）、客户端（Client）、DNS域名服务器（ServiceManager）以及路由器（Binder 驱动）之前的关系。
- - 通常访问网络的过程：1、客户端输入地址，2、通过DNS找到所要搜寻的客户端的IP地址，3、通过IP地址来找到服务器端（ps：路由起到一个中转的重要作用）
-

对各个部分的描述，Android Binder 设计与实现*一文中对 Client、Server、ServiceManager、Binder 驱动有很详细的描述，以下是部分摘录：*

> **Binder 驱动**
>
> Binder 驱动就如同路由器一样，是整个通信的核心；驱动负责进程之间 Binder 通信的建立，Binder 在进程之间的传递，Binder 引用计数管理，数据包在进程之间的传递和交互等一系列底层支持。
>
> **ServiceManager 与实名 Binder**
>
> ServiceManager 和 DNS 类似，作用是将字符形式的 Binder 名字转化成 Client 中对该 Binder 的引用，使得 Client 能够通过 Binder 的名字获得对 Binder 实体的引用。注册了名字的 Binder 叫实名 Binder，就像网站一样除了除了有 IP 地址以外还有自己的网址。Server 创建了 Binder，并为它起一个字符形式，可读易记得名字，将这个 Binder 实体连同名字一起以数据包的形式通过 Binder 驱动发送给 ServiceManager ，通知 ServiceManager 注册一个名为“张三”的 Binder，它位于某个 Server 中。**驱动**为这个穿越进程边界的 Binder 创建位于内核中的实体节点以及 ServiceManager 对实体的引用，将名字以及新建的引用打包传给 ServiceManager。ServiceManger 收到数据后从中取出**名字和引用填入查找表。**
>
> 细心的读者可能会发现，ServierManager 是一个进程，Server 是另一个进程，Server 向 ServiceManager 中注册 Binder 必然涉及到进程间通信。当前实现进程间通信又要用到进程间通信，这就好像蛋可以孵出鸡的前提却是要先找只鸡下蛋！Binder 的实现比较巧妙，就是预先创造一只鸡来下蛋。ServiceManager 和其他进程同样采用 Bidner 通信，ServiceManager 是 Server 端，有自己的 Binder 实体，其他进程都是 Client，需要通过这

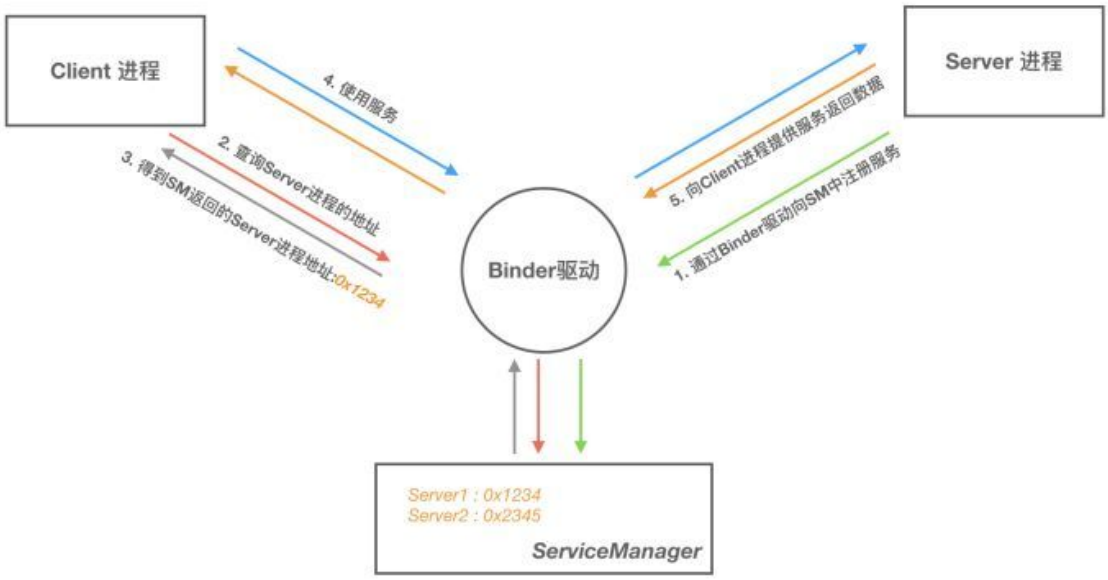
> 个 Binder 的引用来实现 Binder 的注册，查询和获取。ServiceManager 提供的 Binder 比较特殊，它没有名字也不需要注册。当一个进程使用 BINDER*SET*CONTEXT_MGR 命令将自己注册成 ServiceManager 时 Binder 驱动会自动为它创建 Binder 实体（**这就是那只预先造好的那只鸡**）。其次这个 Binder 实体的引用在所有 Client 中都固定为 0 而无需通过其它手段获得。也就是说，一个 Server 想要向 ServiceManager 注册自己的 Binder 就必须通过这个 **0 号引用**和 ServiceManager 的 Binder 通信。类比互联网，0 号引用就好比是域名服务器的地址，你必须预先动态或者手工配置好。要注意的是，这里说的 Client 是相对于 ServiceManager 而言的，一个进程或者应用程序可能是提供服务的 Server，但对于 ServiceManager 来说它仍然是个 Client。
>
> **Client 获得实名 Binder 的引用**
>
> Server 向 ServiceManager 中注册了 Binder 以后， Client 就能通过名字获得 Binder 的引用了。Client 也利用保留的 0 号引用向 ServiceManager 请求访问某个 Binder: 我申请访问名字叫张三的 Binder 引用。ServiceManager 收到这个请求后从请求数据包中取出 Binder 名称，在查找表里找到对应的条目，取出对应的 Binder 引用作为回复发送给发起请求的 Client。从面向对象的角度看，Server 中的 Binder 实体现在有两个引用：一个位于 ServiceManager 中，一个位于发起请求的 Client 中。如果接下来有更多的 Client 请求该 Binder，系统中就会有更多的引用指向该 Binder ，就像 Java 中一个对象有多个引用一样。

3.2 Binder通信过程

至此，我们大致能总结出 Binder 通信过程：

1. 首先，一个进程使用 BINDER*SET*CONTEXT_MGR 命令通过 Binder 驱动将自己注册成为 ServiceManager；
2. Server 通过驱动向 ServiceManager 中注册 Binder（Server 中的 Binder 实体），表明可以对外提供服务。驱动为这个 Binder 创建位于内核中的实体节点以及 ServiceManager 对实体的引用，将名字以及新建的引用打包传给 ServiceManager，ServiceManger 将其填入查找表。
3. Client 通过名字，在 Binder 驱动的帮助下从 ServiceManager 中获取到对 Binder 实体的引用，通过这个引用就能实现和 Server 进程的通信。

我们看到整个通信过程都需要 Binder 驱动的接入。下图能更加直观的展现整个通信过程：

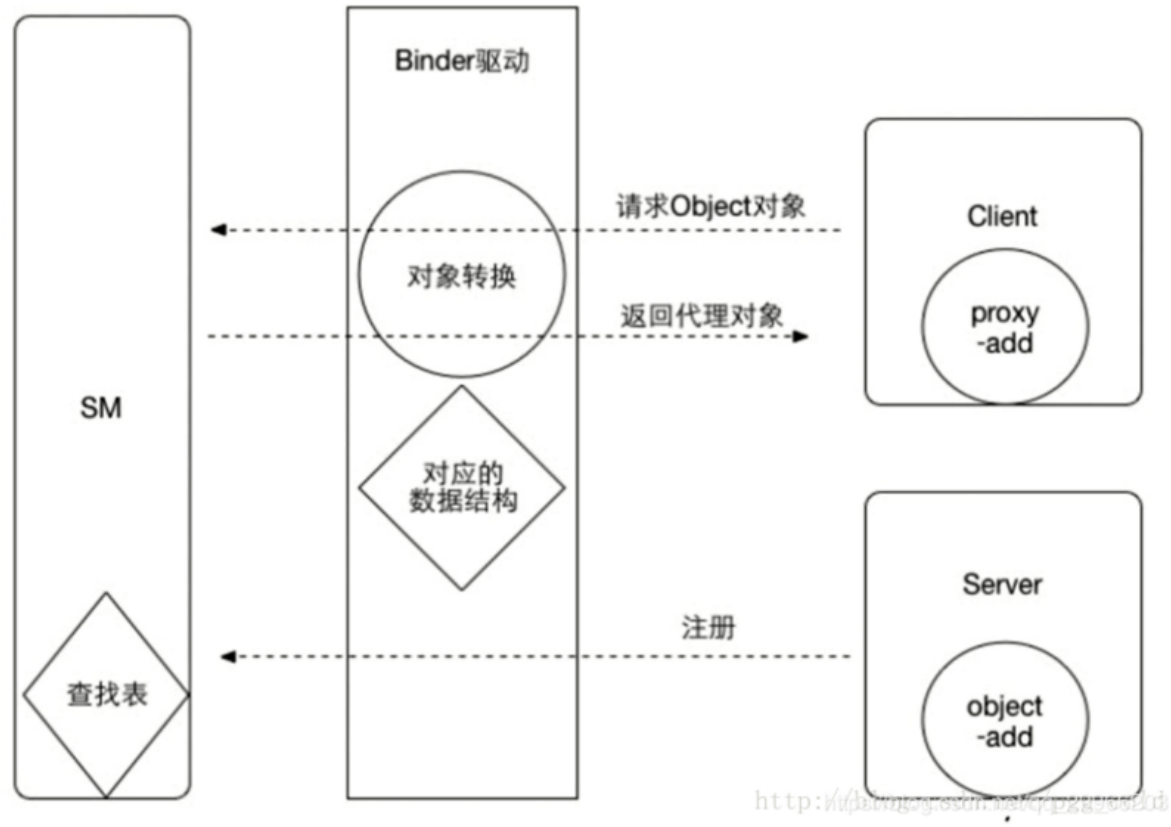


3.3 Binder通信中的代理模式

A 进程想要 B 进程中某个对象（object）是如何实现的呢？它们分属不同的进程，A 进程没法直接使用 B 进程中的 object。

前面我们介绍过跨进程通信的过程都有 Binder 驱动的参与，因此在数据流经 Binder 驱动的时候驱动会对数据做一层转换。当 A 进程想要获取 B 进程中的 object 时，驱动并不会真的把 object 返回给 A，而是返回了一个跟 object 看起来一模一样的代理对象 objectProxy，这个 objectProxy 具有和 object 一摸一样的方法，但是这些方法并没有 B 进程中 object 对象那些方法的能力，这些方法只需要把把请求参数交给驱动即可。对于 A 进程来说和直接调用 object 中的方法是一样的。

当 Binder 驱动接收到 A 进程的消息后，发现这是个 objectProxy 就去查询自己维护的表单，一查发现这是 B 进程 object 的代理对象。于是就会去通知 B 进程调用 object 的方法，并要求 B 进程把返回结果发给自己。当驱动拿到 B 进程的返回结果后就会转发给 A 进程，一次通信就完成了。

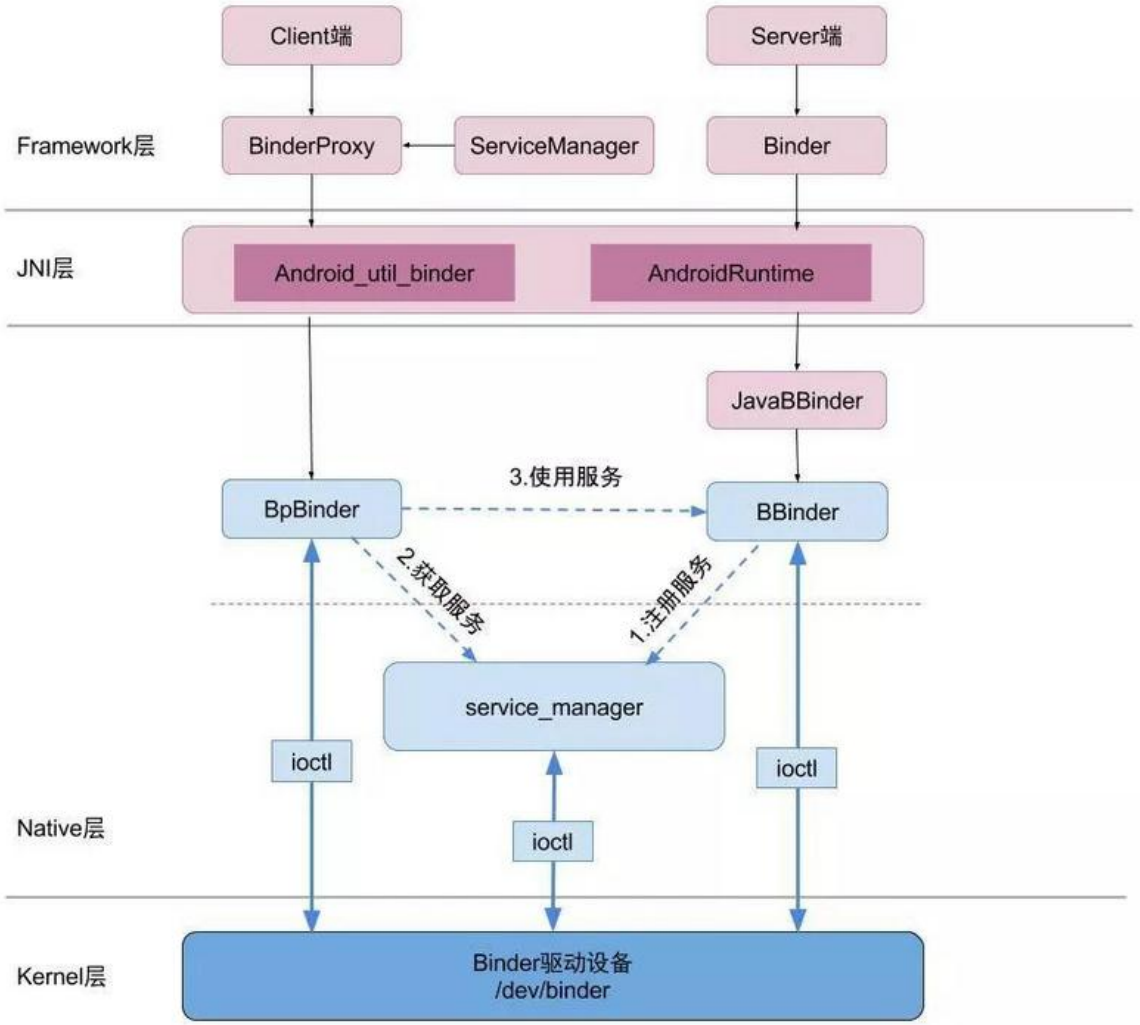


3.3 Binder 具体使用

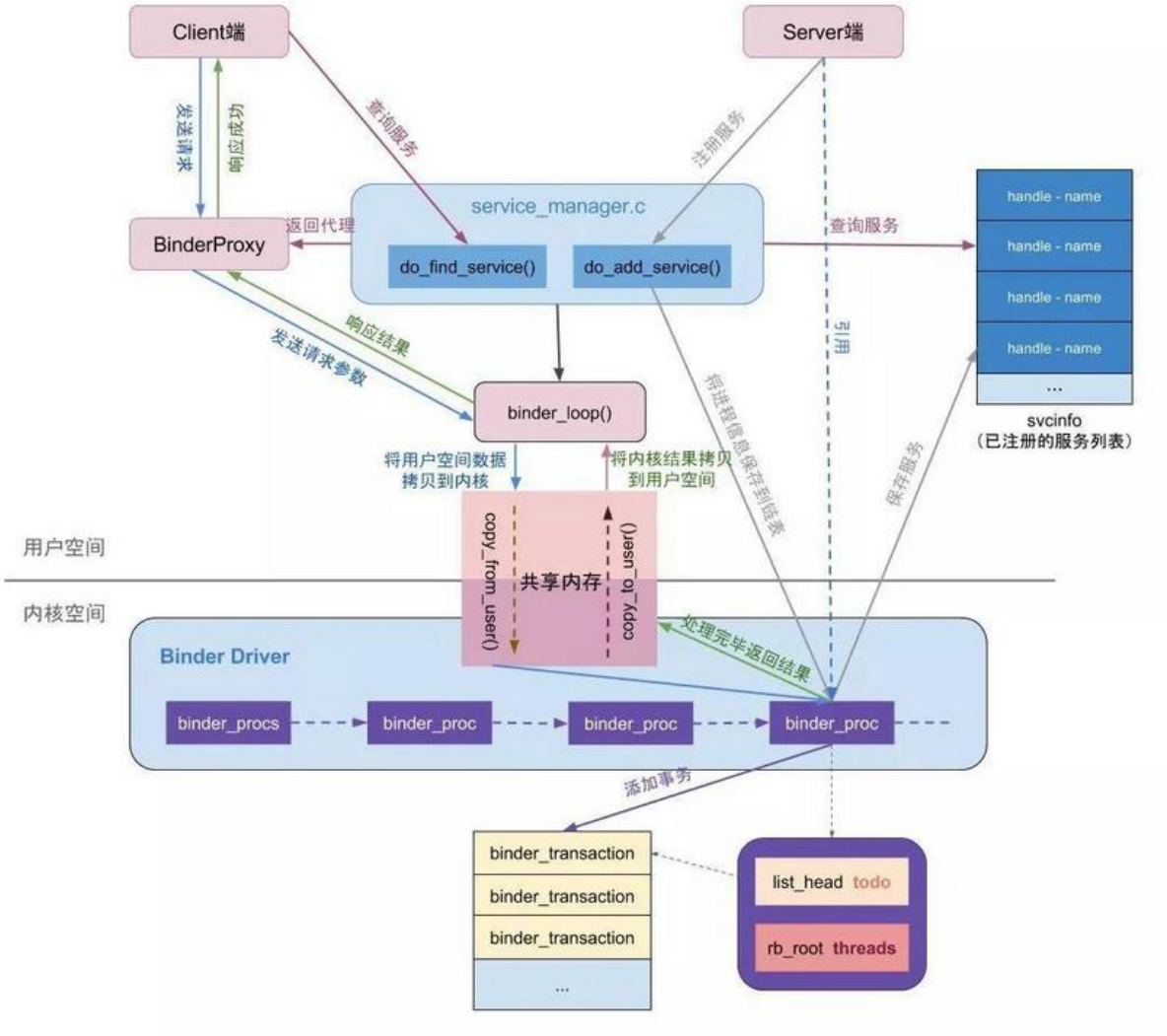
这部需要结合AIDL来进行，下次再来分析。

3.4 Binder导图

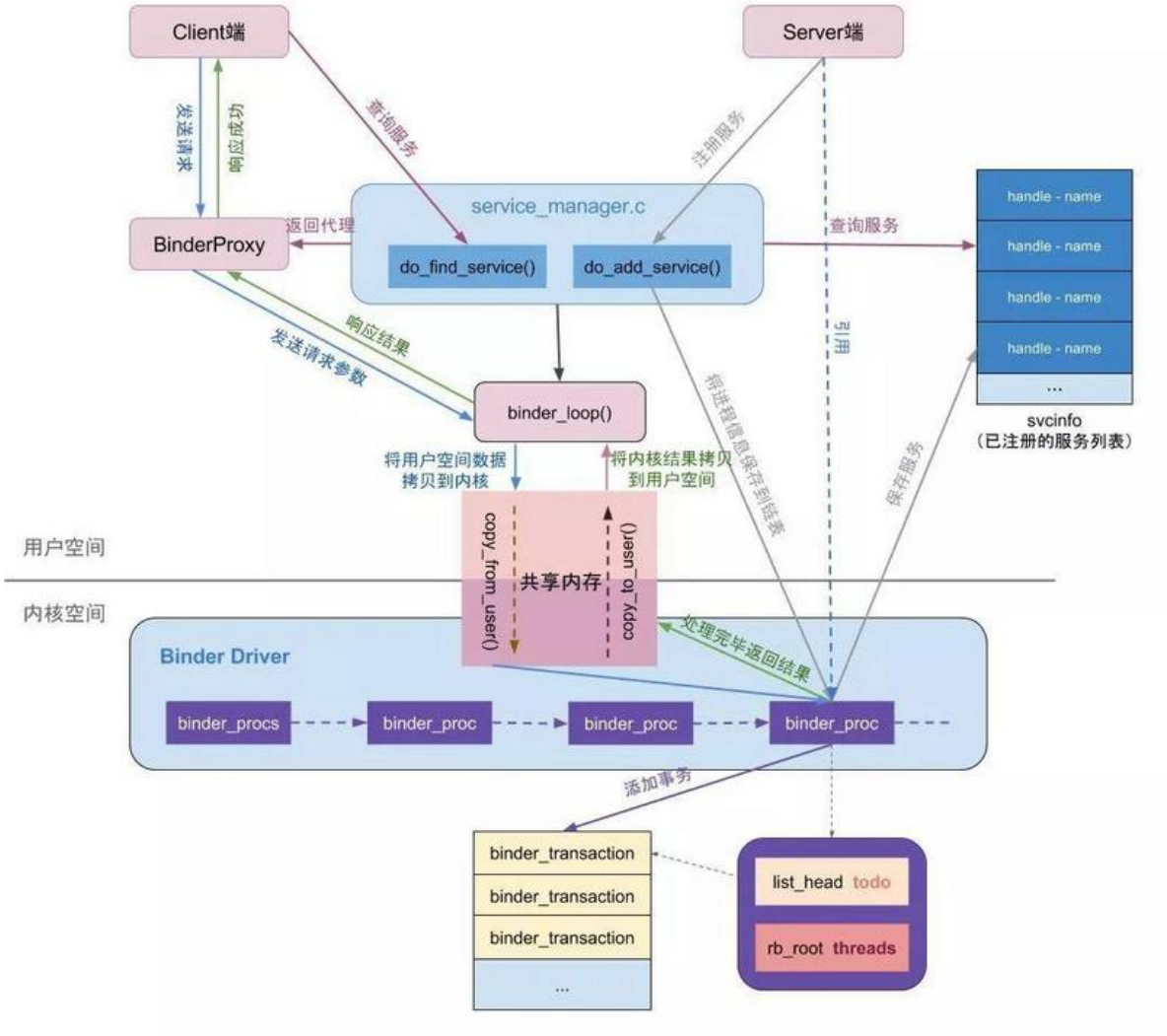
1、binder架构


Client端
Server端
Framework层
BinderProxy
ServiceManager
Binder
JNI层
Android_util_binder
AndroidRuntime
JavaBBinder
3.使用服务
BpBinder
BBinder
2.获取服务
1.注册服务
service_manager
ioctl
ioctl
ioctl
Native层
Kernel层
Binder驱动设备
/dev/binder

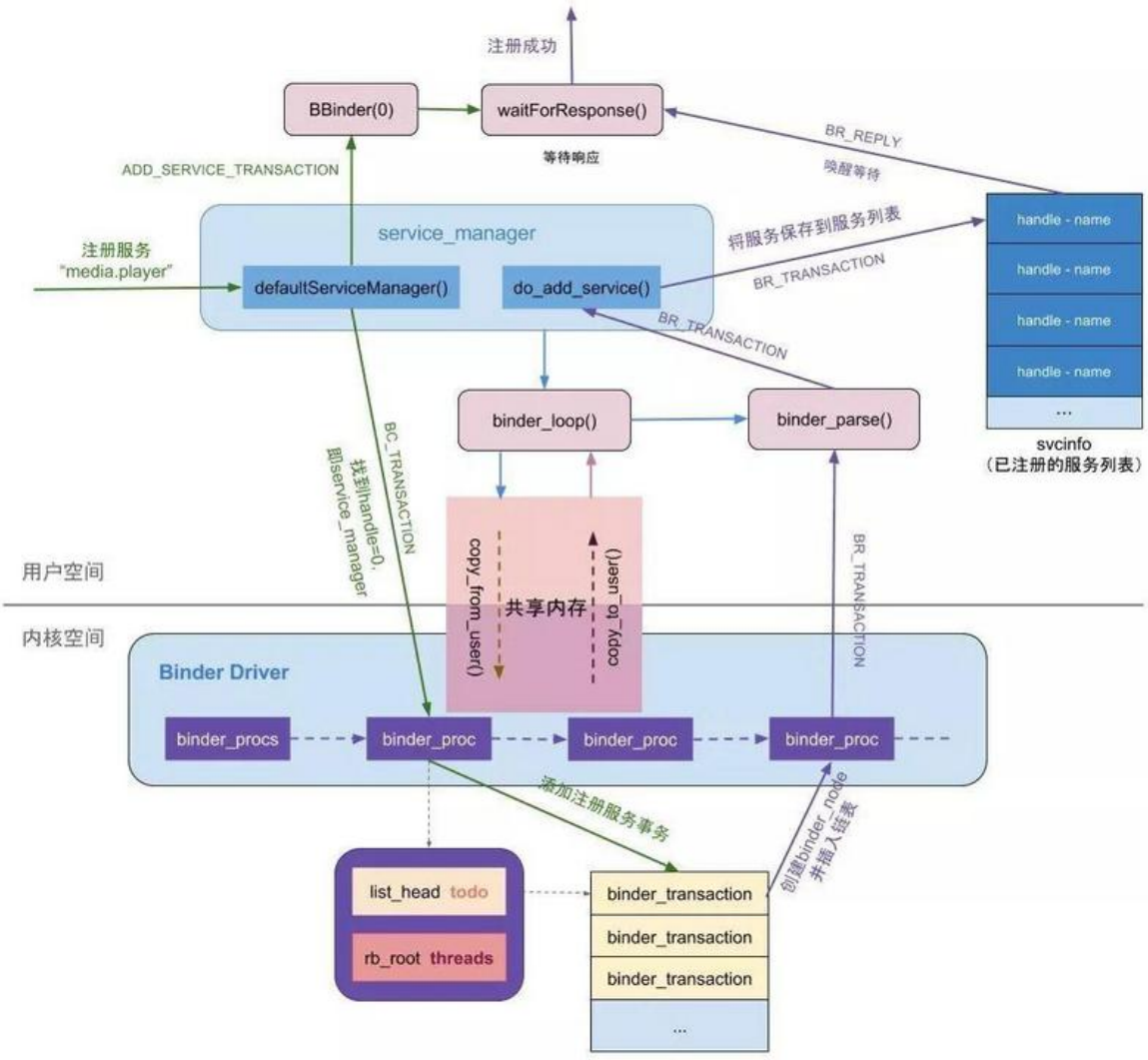
2、binder机制


Client端
Server端
发送请求
响应
查询服务
注册服务
service_manager.c
BinderProxy
返回代理
do_find_service()
do_add_service()
查询服务
handle - name
handle - name
handle - name
handle - name
...
svcinfo
（已注册的服务列表）
引用
响应结果
发送请求参数
binder_loop()
将进程信息保存到链表
保存服务
将用户空间数据
拷贝到内核
将内核结果拷贝
到用户空间
用户空间
内核空间
copy_from_user()
共享内存
copy_to_user()
处理完毕返回结果
Binder Driver
binder_procs
binder_proc
binder_proc
binder_proc
添加事务
binder_transaction
binder_transaction
binder_transaction
...
list_head todo
rb_root threads

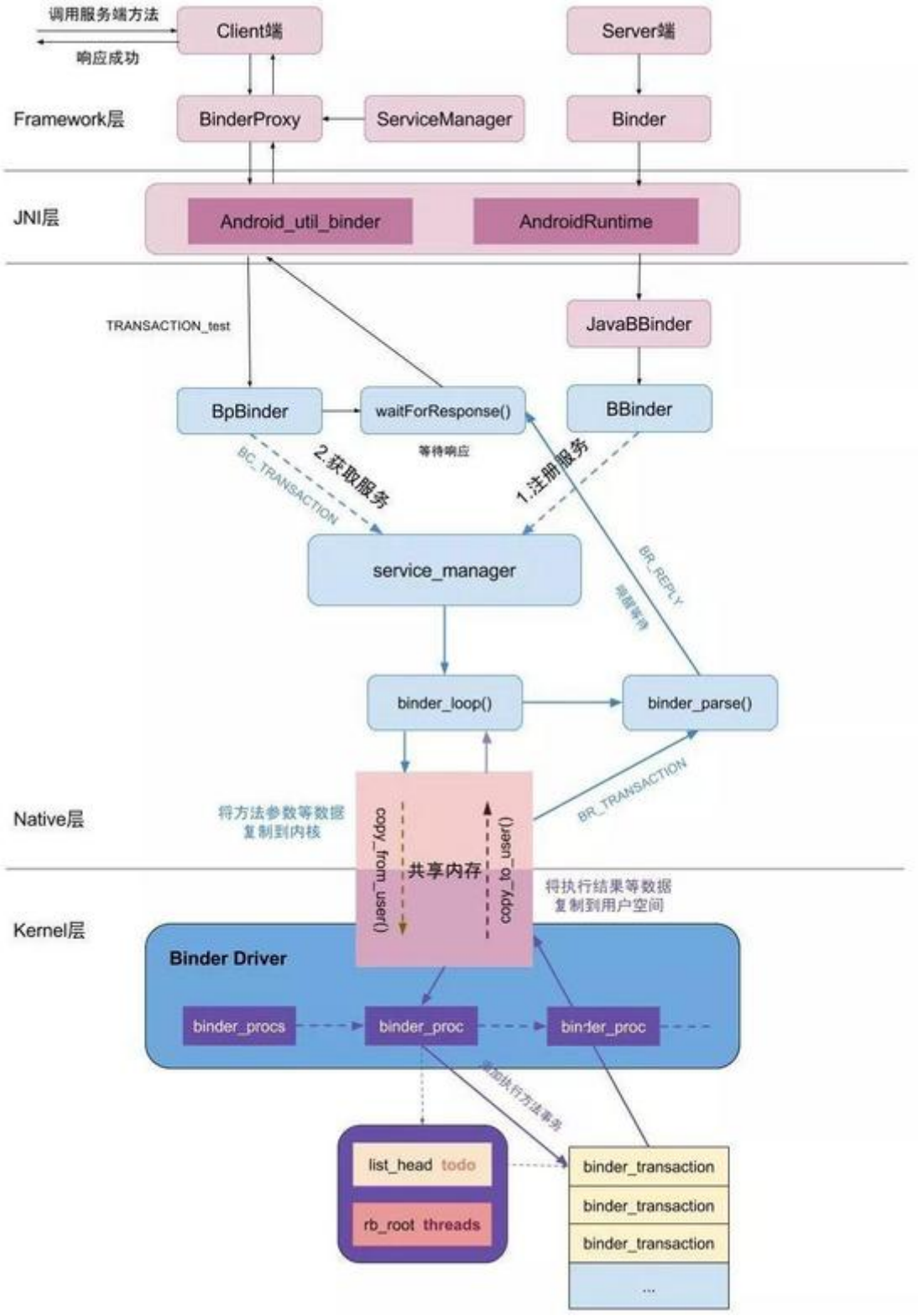
3、注册服务的流程


Client端
Server端
发送请求
响应
查询服务
注册服务
service_manager.c
返回代理
BinderProxy
do_find_service()
do_add_service()
查询服务
handle - name
handle - name
handle - name
handle - name
...
svcinfo
（已注册的服务列表）
响应结果
发送请求参数
binder_loop()
引用
将进程信息保存到链表
保存服务
将用户空间数据
拷贝到内核
将内核结果拷贝
到用户空间
用户空间
内核空间
copy_from_user()
共享内存
copy_to_user()
处理完毕返回结果
Binder Driver
binder_procs
binder_proc
binder_proc
binder_proc
添加事务
binder_transaction
binder_transaction
binder_transaction
...
list_head todo
rb_root threads

4、获取服务的流程


注册成功
BBinder(0)
waitForResponse()
等待响应
BR_REPLY
唤醒等待
ADD_SERVICE_TRANSACTION
service_manager
注册服务
"media.player"
defaultServiceManager()
do_add_service()
将服务保存到服务列表
BR_TRANSACTION
BR_TRANSACTION
handle - name
handle - name
handle - name
handle - name
...
svcinfo
（已注册的服务列表）
binder_loop()
binder_parse()
BC_TRANSACTION
找到handle=0,
即service_manager
copy_from_user()
共享内存
copy_to_user()
BR_TRANSACTION
用户空间
内核空间
Binder Driver
binder_procs
binder_proc
binder_proc
binder_proc
添加注册服务事务
创建binder_node
并插入链表
list_head todo
rb_root threads
binder_transaction
binder_transaction
binder_transaction
...

5、一次完整的响应流程


调用服务端方法
响应成功
Client端
Server端
Framework层
BinderProxy
ServiceManager
Binder
JNI层
Android_util_binder
AndroidRuntime
TRANSACTION_test
JavaBBinder
BpBinder
waitForResponse()
BBinder
等待响应
BC_TRANSACTION
2.获取服务
1.注册服务
BR_REPLY
service_manager
binder_loop()
binder_parse()
BR_TRANSACTION
Native层
将方法参数等数据
复制到内核
copy_from_user()
共享内存
copy_to_user()
将执行结果等数据
复制到用户空间
Kernel层
Binder Driver
binder_procs
binder_proc
binder_proc
list_head todo
rb_root threads
binder_transaction
binder_transaction
binder_transaction
...