Binder机制

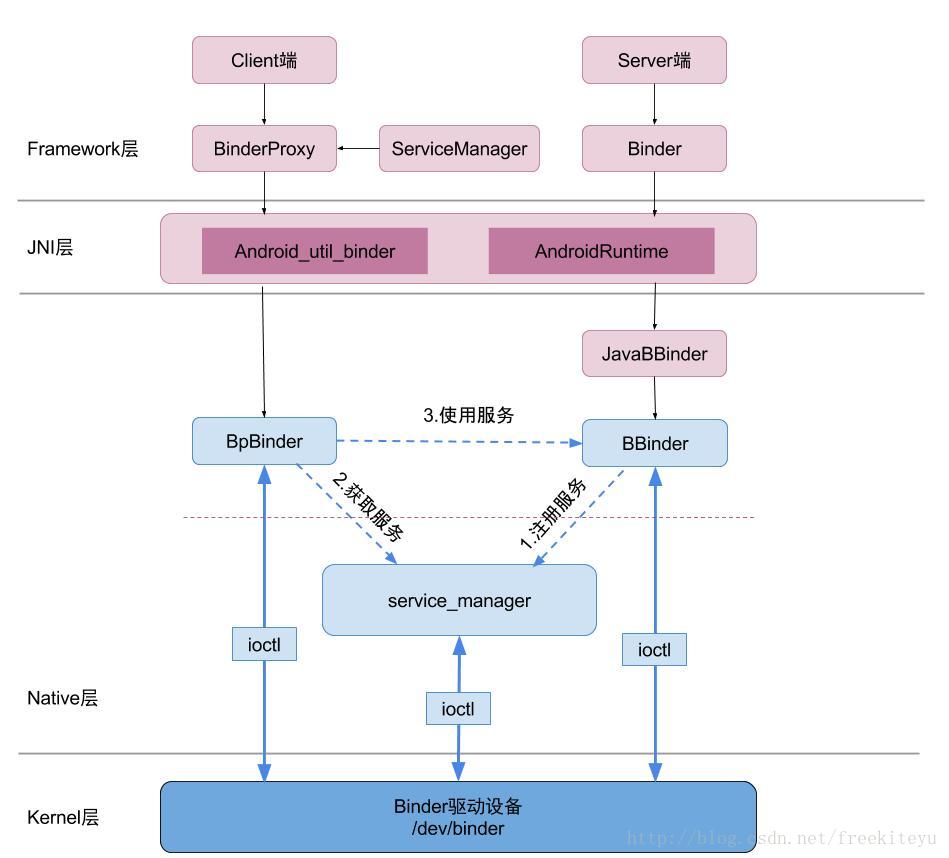
# Android-Binder进程间通讯机制

# 1.什么是 Binder？

Binder是Android系统中进程间通讯（IPC）的一种方式，也是Android系统中最重要的特性之一。Android中的四大组件Activity，Service，Broadcast，ContentProvider，不同的App等都运行在不同的进程中，它是这些进程间通讯的桥梁。正如其名“粘合剂”一样，它把系统中各个组件粘合到了一起，是各个组件的桥梁。

理解Binder对于理解整个Android系统有着非常重要的作用，如果对Binder不了解，就很难对Android系统机制有更深入的理解。

# 1. Binder 架构

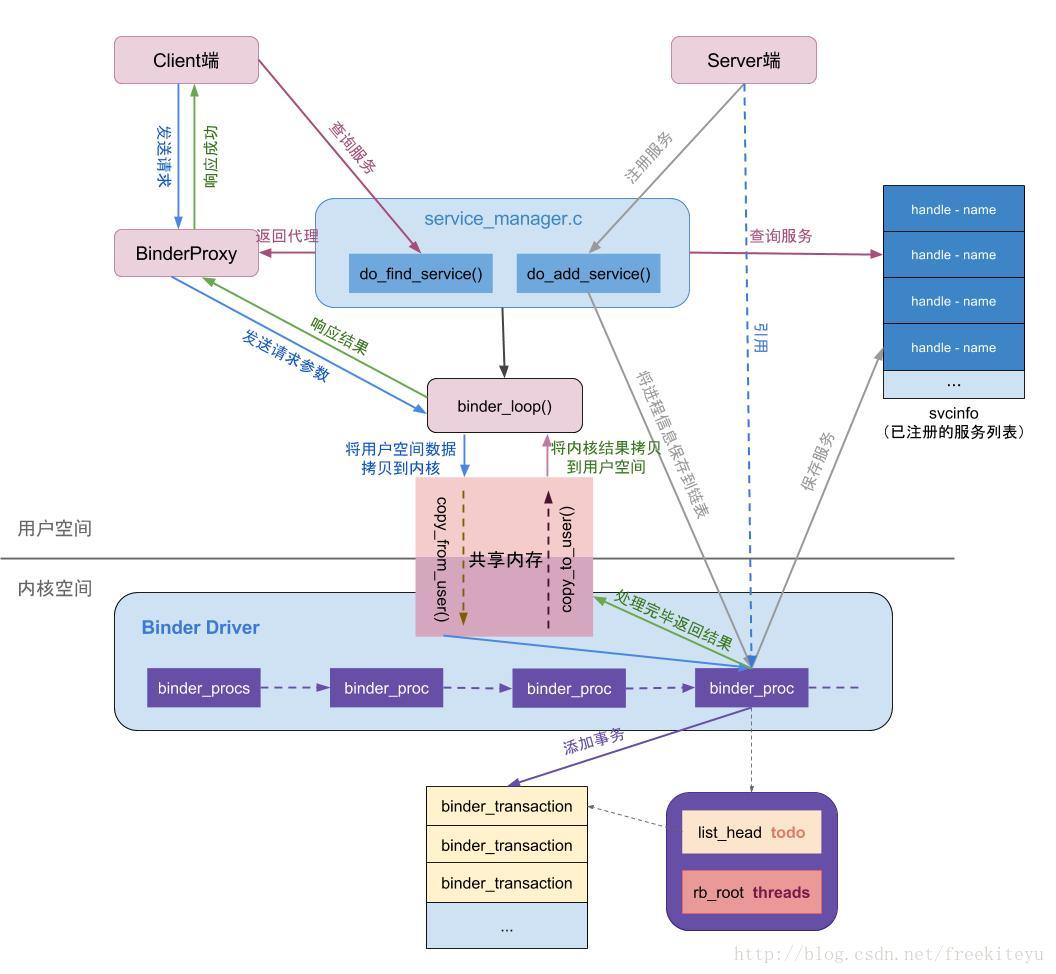


Binder 通信采用 C/S 架构，从组件视角来说，包含 Client、 Server、 ServiceManager 以及 Binder 驱动，其中 ServiceManager 用于管理系统中的各种服务。

Binder 在 framework 层进行了封装，通过 JNI 技术调用 Native（C/C++）层的 Binder 架构。

Binder 在 Native 层以 ioctl 的方式与 Binder 驱动通讯。

# 2. Binder 机制



首先需要注册服务端，只有注册了服务端，客户端才有通讯的目标，服务端通过 ServiceManager 注册服务，注册的过程就是向 Binder 驱动的全局链表 binder\_procs 中插入服务端的信息（binder\_proc 结构体，每个 binder\_proc 结构体中都有 todo 任务队列），然后向 ServiceManager 的 svcinfo 列表中缓存一下注册的服务。

有了服务端，客户端就可以跟服务端通讯了，通讯之前需要先获取到服务，拿到服务的代理，也可以理解为引用。比如下面的代码：

//获取WindowManager服务引用

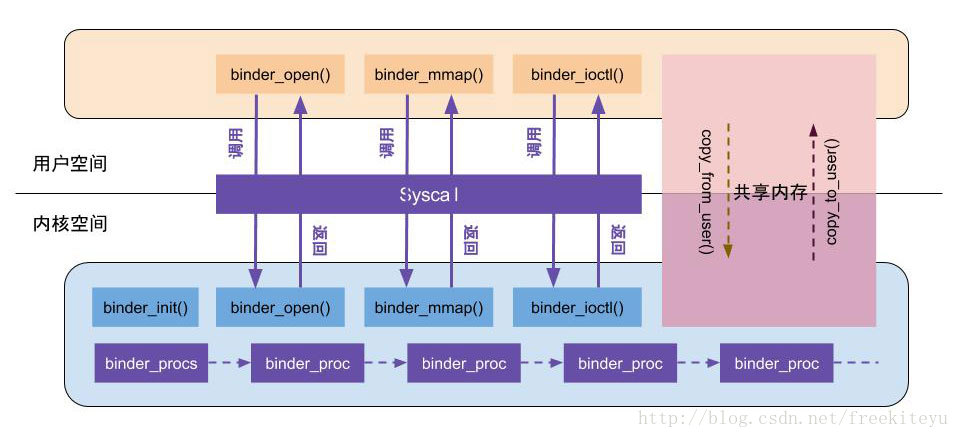
WindowManager wm = (WindowManager)getSystemService(getApplication().WINDOW\_SERVICE);

获取服务端的方式就是通过 ServiceManager 向 svcinfo 列表中查询一下返回服务端的代理，svcinfo 列表就是所有已注册服务的通讯录，保存了所有注册的服务信息。

有了服务端的引用我们就可以向服务端发送请求了，通过 BinderProxy 将我们的请求参数发送给 ServiceManager，通过共享内存的方式使用内核方法 copy\_from\_user() 将我们的参数先拷贝到内核空间，这时我们的客户端进入等待状态，然后 Binder 驱动向服务端的 todo 队列里面插入一条事务，执行完之后把执行结果通过 copy\_to\_user() 将内核的结果拷贝到用户空间（这里只是执行了拷贝命令，并没有拷贝数据，binder只进行一次拷贝），唤醒等待的客户端并把结果响应回来，这样就完成了一次通讯。

# 3. Binder 驱动

我们先来了解下用户空间与内核空间是怎么交互的。



## 3.1用户空间/内核空间

详细解释可以参考 Kernel Space Definition； 简单理解如下：

Kernel space 是 Linux 内核的运行空间，User space 是用户程序的运行空间。 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。

Kernel space 可以执行任意命令，调用系统的一切资源； User space 只能执行简单的运算，不能直接调用系统资源，必须通过系统接口（又称 system call），才能向内核发出指令。

总结：为了安全，用户控件和内核控件是相互隔离的,这样，即使用户程序崩溃，内核也不会受到影响。内核空间可以执行任何命令，调用系统的一切资源;而用户空间只能执行简单的运算，不能直接调用系统资源，必须通过系统接口（System call）,才能向内核发出指令。

## 3.2系统调用/内核态/用户态

虽然从逻辑上抽离出用户空间和内核空间；但是不可避免的的是，总有那么一些用户空间需要访问内核的资源；比如应用程序访问文件，网络是很常见的事情，怎么办呢？

Kernel space can be accessed by user processes only through the use of system calls.

用户空间访问内核空间的唯一方式就是系统调用；通过这个统一入口接口，所有的资源访问都是在内核的控制下执行，以免导致对用户程序对系统资源的越权访问，从而保障了系统的安全和稳定。用户软件良莠不齐，要是它们乱搞把系统玩坏了怎么办？因此对于某些特权操作必须交给安全可靠的内核来执行。

当一个任务（进程）执行系统调用而陷入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）此时处理器处于特权级最高的（0级）内核代码中执行。当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）。即此时处理器在特权级最低的（3级）用户代码中运行。处理器在特权等级高的时候才能执行那些特权CPU指令。

总结：

系统调用：用户空间访问内核空间（如：访问文件，网络）通过系统调用：在内核的控制下执行系统调用。这是为了，将某些特权的操作必须交给安全可靠地内核执行，避免用户程序对系统资源越权访问，保证系统的安全和稳定。

内核态/用户态：内核态：当一个进程执行系统调用而陷入内核代码，此时该进程处于内核运行态（内核态），此时CPU执行特权级（0级）最高的内核代码。

用户态：进程在执行用户代码，处于用户运行态，此时处理器执行特权级别最低的（3级）的用户代码。

## 3.3内核模块/驱动

通过系统调用，用户空间可以访问内核空间，那么如果一个用户空间想与另外一个用户空间进行通信怎么办呢？很自然想到的是让操作系统内核添加支持；传统的 Linux 通信机制，比如 Socket，管道等都是内核支持的；但是 Binder 并不是 Linux 内核的一部分，它是怎么做到访问内核空间的呢？ Linux 的动态可加载内核模块（Loadable Kernel Module，LKM）机制解决了这个问题；模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被链接到内核作为内核的一部分在内核空间运行。这样，Android系统可以通过添加一个内核模块运行在内核空间，用户进程之间的通过这个模块作为桥梁，就可以完成通信了。

在 Android 系统中，这个运行在内核空间的，负责各个用户进程通过 Binder 通信的内核模块叫做 Binder 驱动;

驱动程序：一般指的是设备驱动程序（Device Driver），是一种可以使计算机和设备通信的特殊程序。相当于硬件的接口，操作系统只有通过这个接口，才能控制硬件设备的工作；

驱动就是操作硬件的接口，为了支持Binder通信过程，Binder 使用了一种“硬件”，因此这个模块被称之为驱动。

熟悉了上面这些概念，我们再来看下上面的图，用户空间中 binder\_open(), binder\_mmap(), binder\_ioctl() 这些方法通过 system call 来调用内核空间 Binder 驱动中的方法。内核空间与用户空间共享内存通过 copy\_from\_user(), copy\_to\_user() 内核方法来完成用户空间与内核空间内存的数据传输。 Binder驱动中有一个全局的 binder\_procs 链表保存了服务端的进程信息。

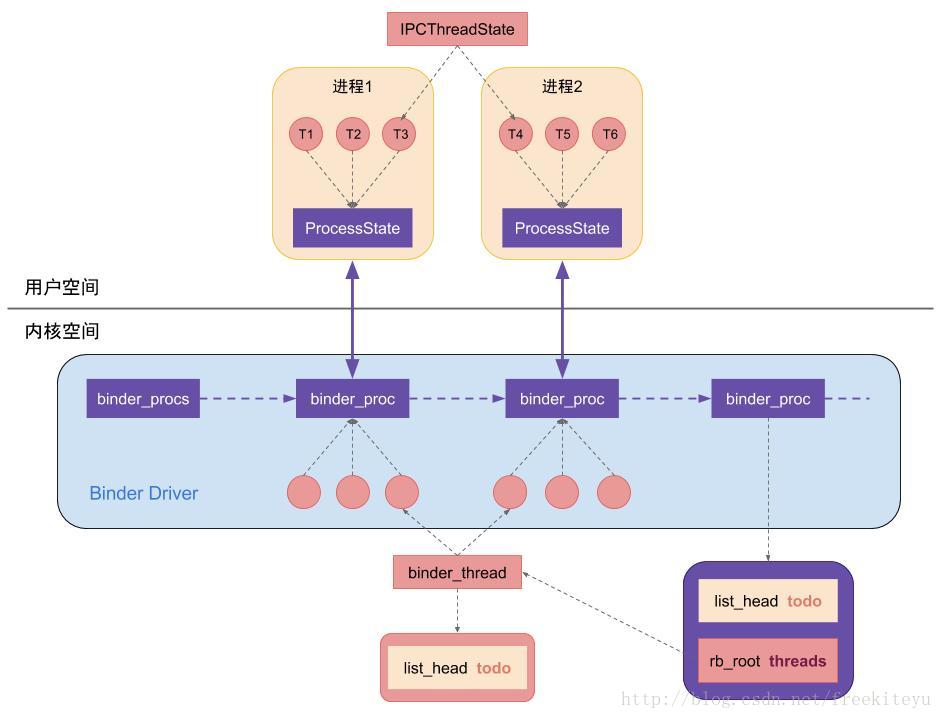
总结：如何让一个用户空间访问另外一个用户空间：1. 操作系统内核添加支持，如：socket，管道

2.Linux内核可以动态加载内核模块（LKM），如Binder(并不属于Linux内核，可以单独编译，但不能独立运行) 它在运行时，会被链接到内核中，作为内核的一部分在内核空间中运行。不同的应用程序通过这个模块完成通信。

其中运行在内核空间，负责Binder通信的内核模块叫做Binder驱动。

驱动程序：一般指的是设备驱动程序（Device Driver），是一种可以使计算机和设备通信的特殊程序。相当于硬件的接口，操作系统只有通过这个接口，才能控制硬件设备的工作。

# 4. Binder 进程与线程



对于底层Binder驱动，通过 binder\_procs 链表记录所有创建的 binder\_proc 结构体，binder 驱动层的每一个 binder\_proc 结构体都与用户空间的一个用于 binder 通信的进程一一对应，且每个进程有且只有一个 ProcessState 对象，这是通过单例模式来保证的。在每个进程中可以有很多个线程，每个线程对应一个 IPCThreadState 对象，IPCThreadState 对象也是单例模式，即一个线程对应一个 IPCThreadState 对象，在 Binder 驱动层也有与之相对应的结构，那就是 Binder\_thread 结构体。在 binder\_proc 结构体中通过成员变量 rb\_root threads，来记录当前进程内所有的 binder\_thread。

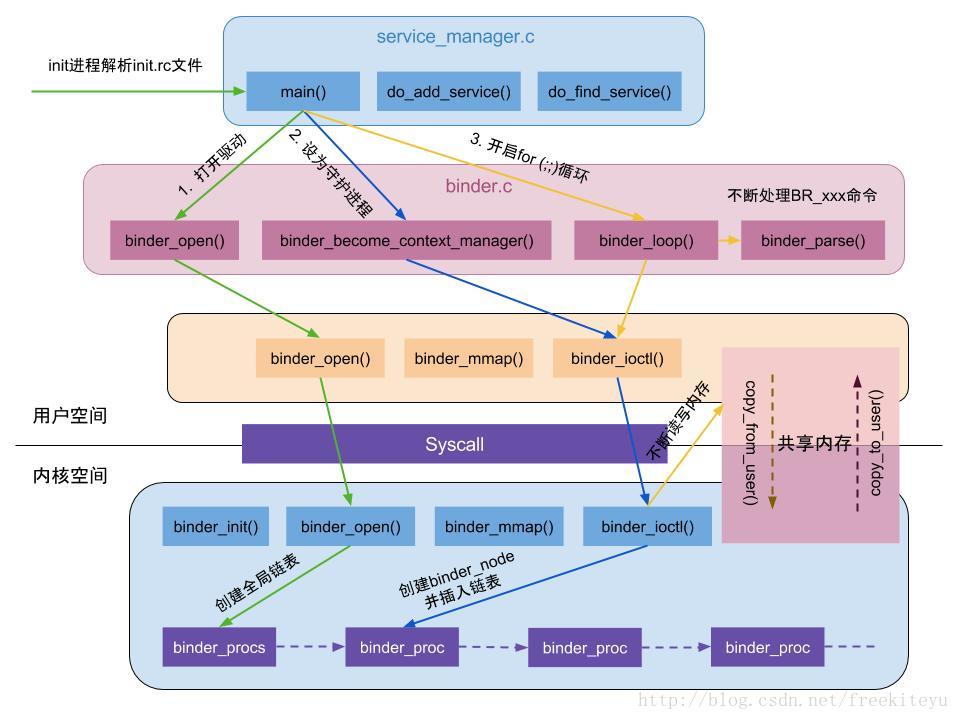
Binder 线程池：每个 Server 进程在启动时创建一个 binder 线程池，并向其中注册一个 Binder 线程；之后 Server 进程也可以向 binder 线程池注册新的线程，或者 Binder 驱动在探测到没有空闲 binder 线程时主动向 Server 进程注册新的的 binder 线程。对于一个 Server 进程有一个最大 Binder 线程数限制，默认为16个 binder 线程，例如 Android 的 system\_server 进程就存在16个线程。对于所有 Client 端进程的 binder 请求都是交由 Server 端进程的 binder 线程来处理的。

总结：用户空间，每一个应用程序对应一个进程，每个进程都都有且只有一个ProcessState对象，同时，每个线程也有且只有一个IPCThreadState对象；在Binder驱动层，binder\_proc结构体中的变量rb\_root——threads,记录当前进程中的所有binder\_thread。

Binder线程池，每个Service进程在创建时，就会创建一个Binder线程池，同时向里面注册一个Binder线程;之后Service进程向binder线程池中注册新线程，或者Binder驱动探测到没有空闲binder线程时，也回主动向Service进程注册新的binder线程。而一个Service进程最大的Binder线程数是16。所有的Client端进程的binder请求都会交由Server端进程的binder线程处理。

# 5. ServiceManager 启动

了解了 Binder 驱动，怎么与 Binder 驱动进行通讯呢？那就是通过 ServiceManager，好多文章称 ServiceManager 是 Binder 驱动的守护进程，大管家，其实 ServiceManager 的作用很简单就是提供了查询服务和注册服务的功能。下面我们来看一下 ServiceManager 启动的过程。



- ServiceManager 分为 framework 层和 native 层，framework 层只是对 native 层进行了封装方便调用，图上展示的是 native 层的 ServiceManager 启动过程。

ServiceManager 的启动是系统在开机时，init 进程解析 init.rc 文件调用 service\_manager.c 中的 main() 方法入口启动的。 native 层有一个 binder.c 封装了一些与 Binder 驱动交互的方法。

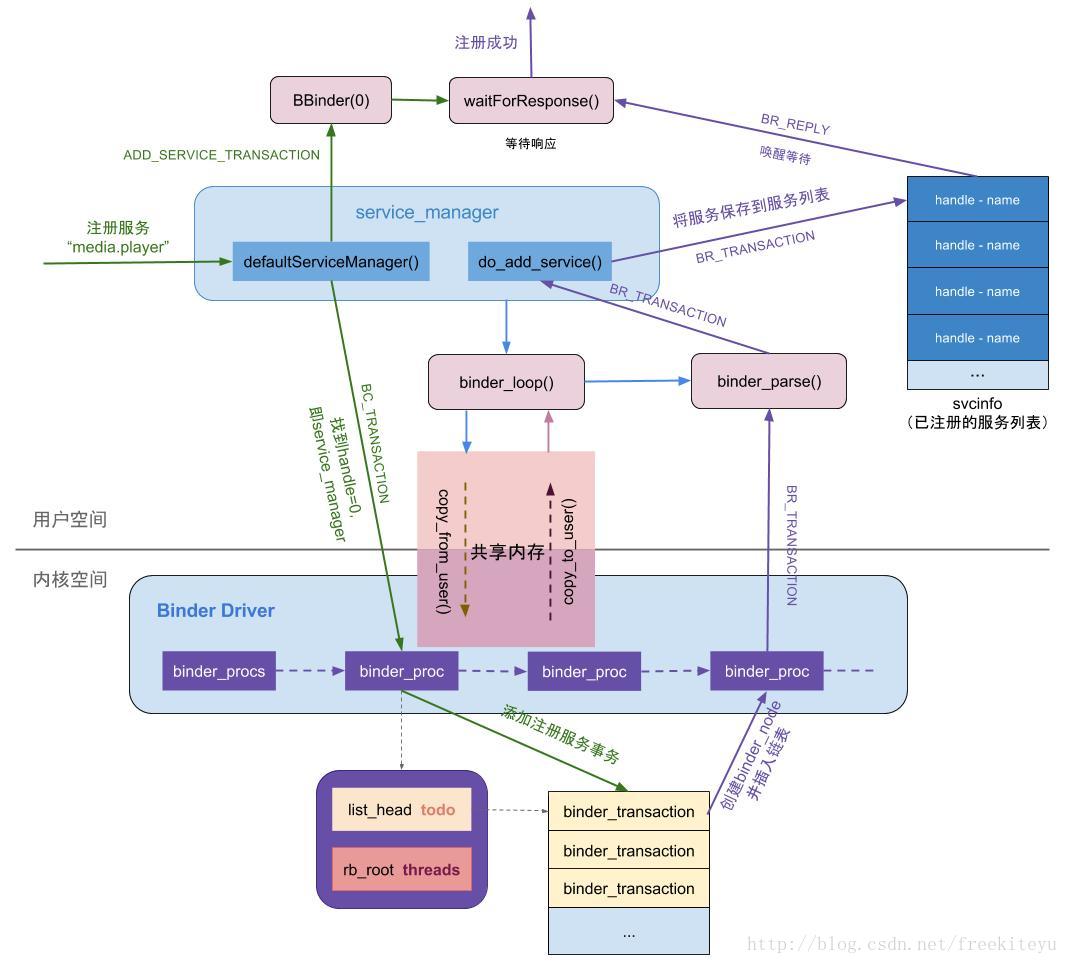
ServiceManager 的启动分为三步，首先打开驱动创建全局链表 binder\_procs，然后将自己当前进程信息保存到 binder\_procs 链表，最后开启 loop 不断的处理共享内存中的数据，并处理 BR\_xxx 命令（ioctl 的命令，BR 可以理解为 binder reply 驱动处理完的响应）。

总结:

1.ServiceManager的启动：开机时，init进程解析init.rc文件调用service\_manager.c中的main()方法启动servicesManager

2.ServiceManager的启动分三步，首先打开驱动创建全局链表binder\_proc, 然后将自己当前进程信息保存到binder\_procs链表中，最后开启loop不断处理共享内存中数据，并处理BR\_xxx命令。

# 6. ServiceManager 注册服务



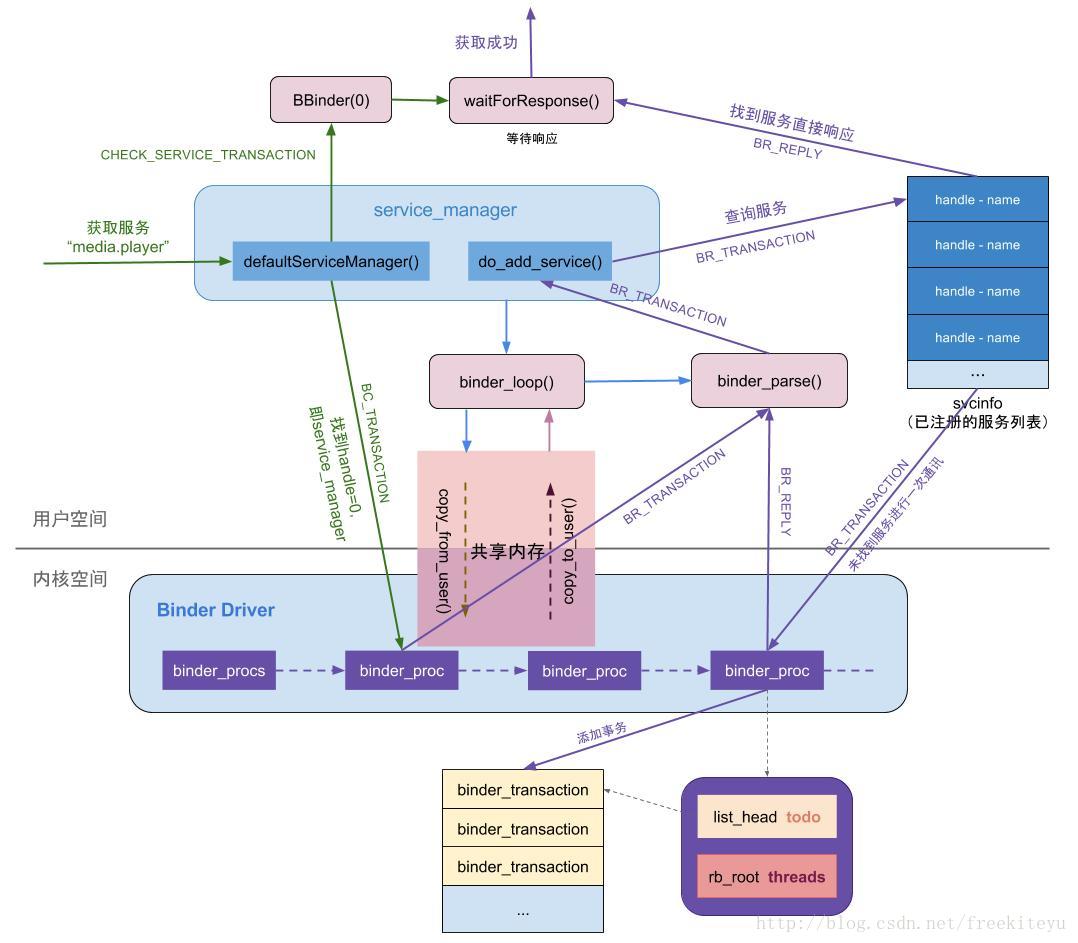
注册 MediaPlayerService 服务端，我们通过 ServiceManager 的 addService() 方法来注册服务。

首先 ServiceManager 向 Binder 驱动发送 BC\_TRANSACTION 命令（ioctl 的命令，BC 可以理解为 binder client 客户端发过来的请求命令）携带 ADD\_SERVICE\_TRANSACTION 命令，同时注册服务的线程进入等待状态 waitForResponse()。 Binder 驱动收到请求命令向 ServiceManager 的 todo 队列里面添加一条注册服务的事务。事务的任务就是创建服务端进程 binder\_node 信息并插入到 binder\_procs 链表中。

事务处理完之后发送 BR\_TRANSACTION 命令，ServiceManager 收到命令后向 svcinfo 列表中添加已经注册的服务。最后发送 BR\_REPLY 命令唤醒等待的线程，通知注册成功。

总结：

7. ServiceManager 获取服务

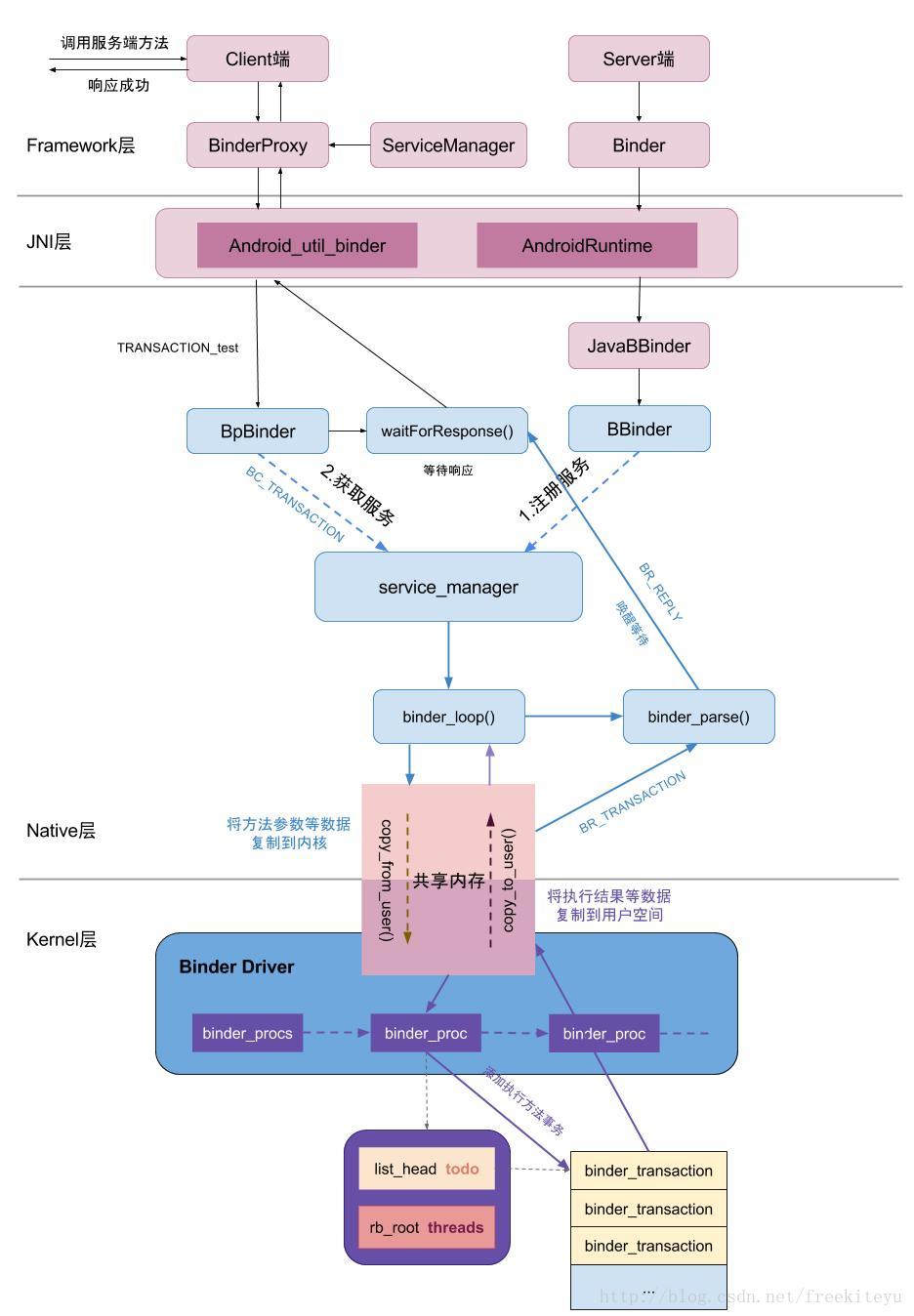


获取服务的过程与注册类似，相反的过程。通过 ServiceManager 的 getService() 方法来注册服务。

首先 ServiceManager 向 Binder 驱动发送 BC\_TRANSACTION 命令携带 CHECK\_SERVICE\_TRANSACTION 命令，同时获取服务的线程进入等待状态 waitForResponse()。

Binder 驱动收到请求命令向 ServiceManager 的发送 BC\_TRANSACTION 查询已注册的服务，查询到直接响应 BR\_REPLY 唤醒等待的线程。若查询不到将与 binder\_procs 链表中的服务进行一次通讯再响应。

8. 进行一次完整通讯



我们在使用 Binder 时基本都是调用 framework 层封装好的方法，AIDL 就是 framework 层提供的傻瓜式是使用方式。假设服务已经注册完，我们来看看客户端怎么执行服务端的方法。

首先我们通过 ServiceManager 获取到服务端的 BinderProxy 代理对象，通过调用 BinderProxy 将参数，方法标识（例如：TRANSACTION\_test，AIDL中自动生成）传给 ServiceManager，同时客户端线程进入等待状态。

ServiceManager 将用户空间的参数等请求数据复制到内核空间，并向服务端插入一条执行执行方法的事务。事务执行完通知 ServiceManager 将执行结果从内核空间复制到用户空间，并唤醒等待的线程，响应结果，通讯结束。

网上资料很多，个人觉得比较好的如下：

1. Bander设计与实现

2. 老罗的 Android进程间通信（IPC）机制Binder简要介绍和学习计划 系列

3. Innost的 深入理解Binder 系列

4. Gityuan的 Binder系列 (基于 Android 6.0)

5. Binder学习指南

参考资料

Binder系列

Binder学习指南

更多文章：

https://github.com/jeanboydev/Android-ReadTheFuckingSourceCode