Android Binder原理（一）学习Binder前必须要了解的知识点

前言**

Binder原理是掌握系统底层原理的基石，也是进阶高级工程师的必备知识点，这篇文章不会过多介绍Binder原理，而是讲解学习Binder前需要的掌握的知识点。

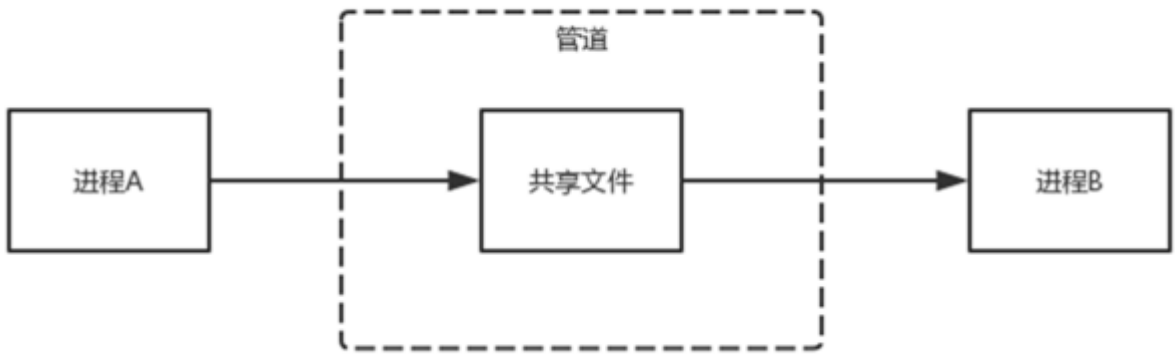
1.Linux和Android的IPC机制种类

IPC全名为inter-Process Communication，含义为进程间通信，是指两个进程之间进行数据交换的过程。在Android和Linux中都有各自的IPC机制，这里分别来介绍下。

1.1 Linux中的IPC机制种类

Linux中提供了很多进程间通信机制，主要有管道（pipe）、信号（sinal）、信号量（semophore）、消息队列（Message）、共享内存（Share Memory)、套接字（Socket）等。

管道 管道是Linux由Unix那里继承过来的进程间的通信机制，它是Unix早期的一个重要通信机制。管道的主要思想是，在内存中创建一个共享文件，从而使通信双方利用这个共享文件来传递信息。这个共享文件比较特殊，它不属于文件系统并且只存在于内存中。另外还有一点，管道采用的是半双工通信方式的，数据只能在一个方向上流动。 简单的模型如下所示。



信号 信号是软件层次上对中断机制的一种模拟，是一种异步通信方式，进程不必通过任何操作来等待信号的到达。信号可以在用户空间进程和内核之间直接交互，内核可以利用信号来通知用户空间的进程发生了哪些系统事件。信号不适用于信息交换，比较适用于进程中断控制。 **信号量** 信号量是一个计数器，用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。主要作为进程间以及同一进程内不同线程之间的同步手段。 **消息队列** 消息队列是消息的链表，具有特定的格式，存放在内存中并由消息队列标识符标识，并且允许一个或多个进程向它写入与读取消息。信息会复制两次，因此对于频繁或者信息量大的通信不宜使用消息队列。

共享内存 多个进程可以直接读写的一块内存空间，是针对其他通信机制运行效率较低而设计的。 为了在多个进程间交换信息，内核专门留出了一块内存区，可以由需要访问的进程将其映射到自己的私有地址空间。进程就可以直接读写这一块内存而不需要进行数据的拷贝，从而大大的提高效率。

套接字 套接字是更为基础的进程间通信机制，与其他方式不同的是，套接字可用于不同机器之间的进程间通信。

1.2 Android中的IPC机制

Android系统是基于Linux内核的，在Linux内核基础上，又拓展出了一些IPC机制。Android系统除了支持套接字，还支持序列化、Messenger、AIDL、Bundle、文件共享、ContentProvider、Binder等。Binder会在后面介绍，先来了解前面的IPC机制。 **序列化** 序列化指的是Serializable/Parcelable，Serializable是Java提供的一个序列化接口，是一个空接口，为对象提供标准的序列化和反序列化操作。Parcelable接口是Android中的序列化方式，更适合在Android平台上使用，用起来比较麻烦，效率很高。 **Messenger** Messenger在Android应用开发中的使用频率不高，可以在不同进程中传递Message对象，在Message中加入我们想要传的数据就可以在进程间的进行数据传递了。Messenger是一种轻量级的IPC方案并对AIDL进行了封装。

AIDL AIDL全名为Android interface definition Language，即Android接口定义语言。Messenger是以串行的方式来处理客户端发来的信息，如果有大量的消息发到服务端，服务端仍然一个一个的处理再响应客户端显然是不合适的。另外还有一点，Messenger用来进程间进行数据传递但是却不能满足跨进程的方法调用，这个时候就需要使用AIDL了。

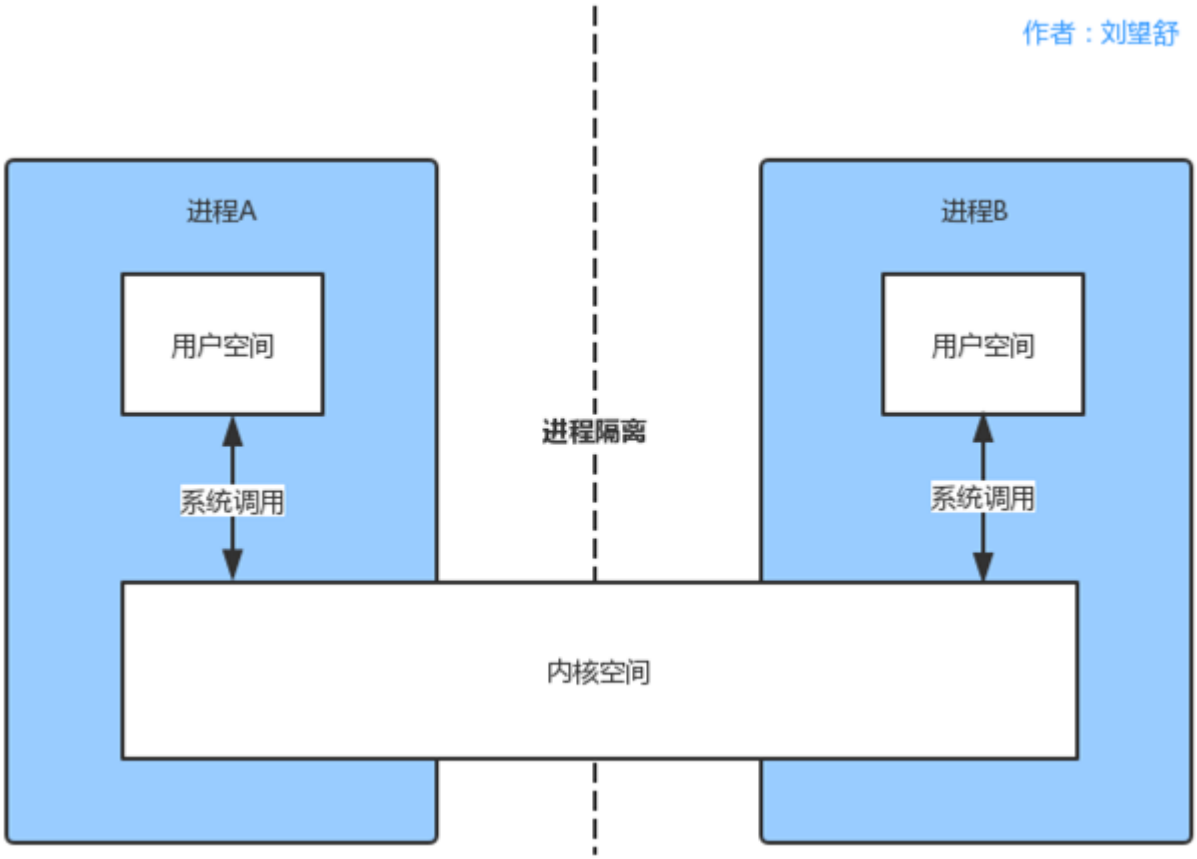
Bundle Bundle实现了Parcelable接口，所以它可以方便的在不同的进程间传输。Acitivity、Service、Receiver都是在Intent中通过Bundle来进行数据传递。

文件共享 两个进程通过读写同一个文件来进行数据共享，共享的文件可以是文本、XML、JOSN。文件共享适用于对数据同步要求不高的进程间通信。

ContentProvider ContentProvider为存储和获取数据了提供统一的接口，它可以在不同的应用程序之间共享数据，本身就是适合进程间通信的。ContentProvider底层实现也是Binder，但是使用起来比AIDL要容易许多。系统中很多操作都采用了ContentProvider，例如通讯录，音视频等，这些操作本身就是跨进程进行通信。

2.Linux和Binder的IPC通信原理

在讲到Linux的进程通信原理之前，我们需要先了解Liunx中的几个概念。



内核空间和用户空间 当我们接触到Liunx时，免不了听到两个词，User space（用户空间）和 Kernel space（内核空间），那么它们的含义是什么呢？ 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和内核空间。Linux 操作系统将最高的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。

内核空间是Linux内核的运行空间，用户空间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，内核也不会受到影响。内核空间的数据是可以进程间共享的，而用户空间则不可以。比如在上图进程A的用户空间是不能和进程B的用户空间共享的。

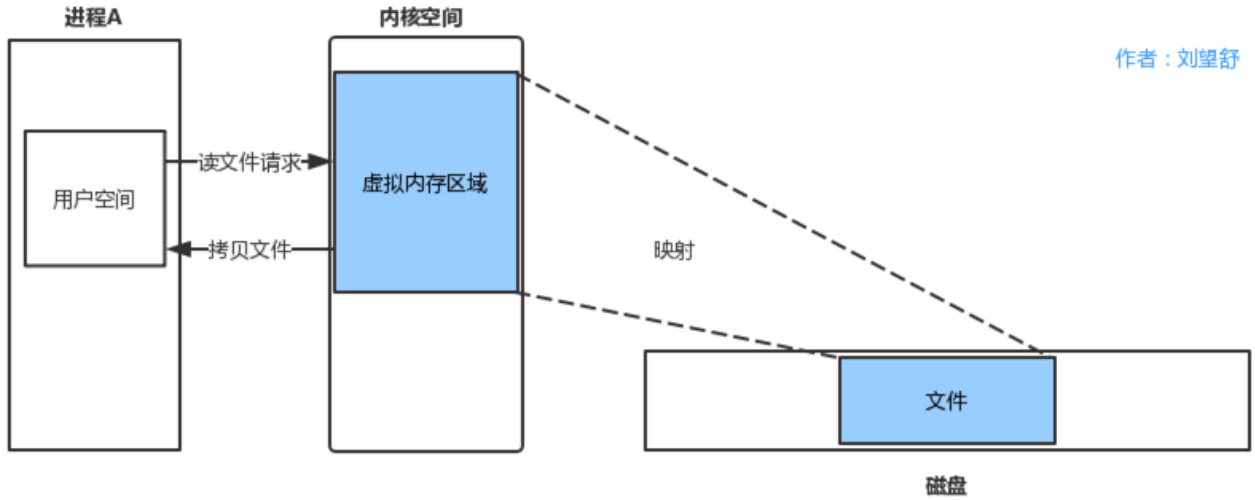
进程隔离 进程隔离指的是，一个进程不能直接操作或者访问另一个进程。也就是进程A不可以直接访问进程B的数据。

系统调用 用户空间需要访问内核空间，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。

进程A和进程B的用户空间可以通过如下系统函数和内核空间进行交互。

- copy_from_user：将用户空间的数据拷贝到内核空间。
- copy_to_user：将内核空间的数据拷贝到用户空间。

内存映射 由于应用程序不能直接操作设备硬件地址，所以操作系统提供了一种机制：内存映射，把设备地址映射到进程虚拟内存区。 举个例子，如果用户空间需要读取磁盘的文件，如果不采用内存映射，那么就需要在内核空间建立一个页缓存，页缓存去拷贝磁盘上的文件，然后用户空间拷贝页缓存的文件，这就需要两次拷贝。 采用内存映射，如下图所示。

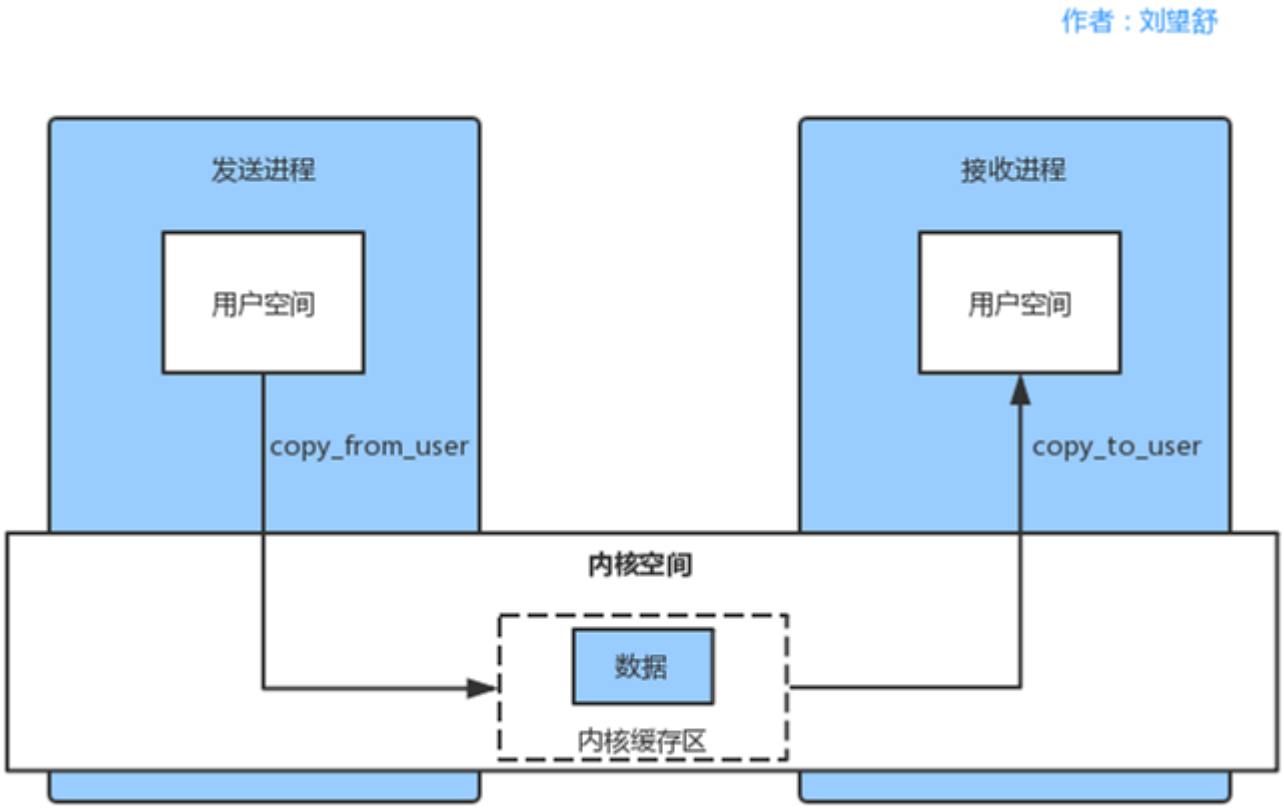


由于新建了虚拟内存区域，那么磁盘文件和虚拟内存区域就可以直接映射，少了一次拷贝。

内存映射全名为Memory Map，在Linux中通过系统调用函数mmap来实现内存映射。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，反之亦然。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。

2.1 Linux的IPC通信原理

了解Liunx中的几个概念后，就可以学习Linux的IPC通信原理了，如下图所示。



内核程序在内核空间分配内存并开辟一块内核缓存区，发送进程通过copy_from_user函数将数据拷贝到到内核空间的缓冲区中。同样的，接收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。这样数据发送进程和数据接收进程完成了一次数据传输，也就是一次进程间通信。

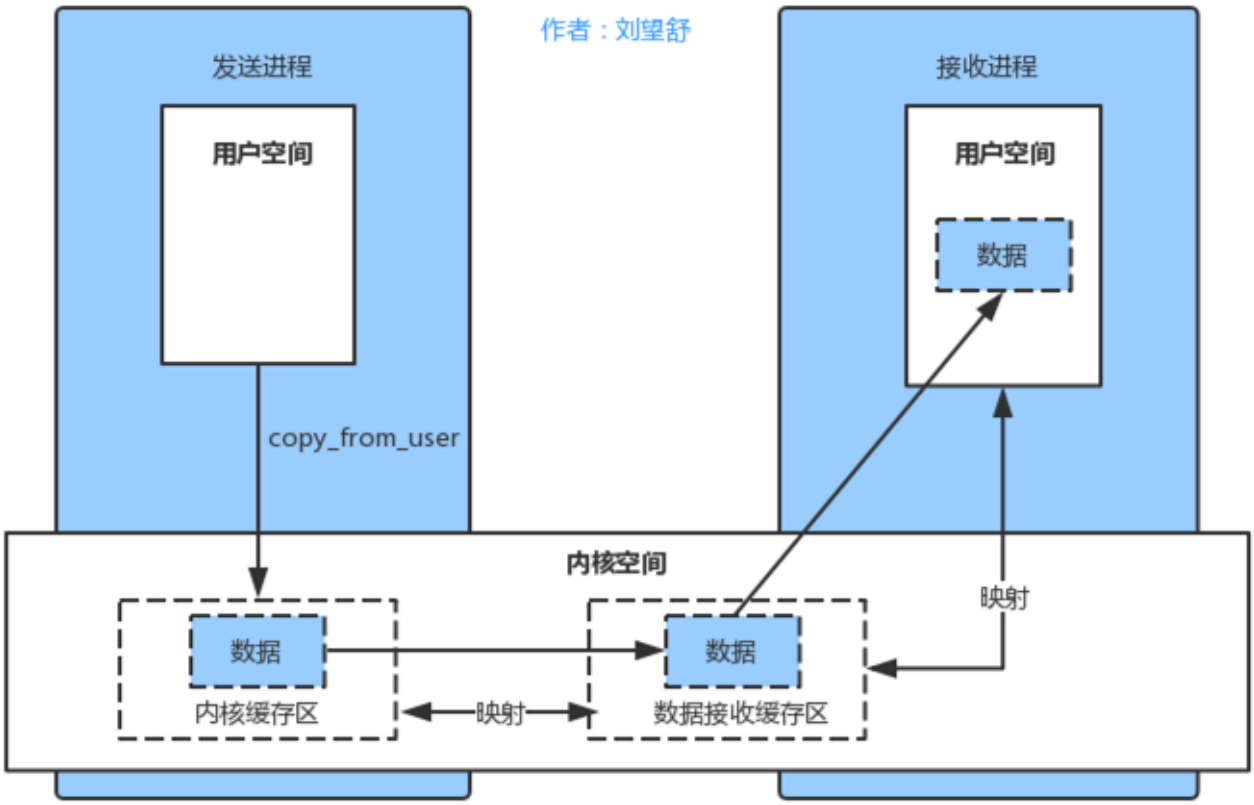
Linux的IPC通信原理有两个问题：

1. 一次数据传递需要经历：用户空间 -> 内核缓存区 -> 用户空间，需要2次数据拷贝，这样效率不高。
2. 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，浪费了空间或者时间。

2.2 Binder的通信原理

Binder是基于开源的OpenBinder实现的，OpenBinder最早并不是由Google公司开发的，而是Be Inc公司开发的，接着由Palm, Inc.公司负责开发。后来OpenBinder的作者Dianne Hackborn加入了Google公司，并负责Android平台的开发工作，顺便把这项技术也带进了Android。

Binder是基于内存映射来实现的，在前面我们知道内存映射通常是用在有物理介质的文件系统上的，Binder没有物理介质，它使用内存映射是为了跨进程传递数据。



Binder通信的步骤如下所示。 1.Binder驱动在内核空间创建一个数据接收缓存区。 2.在内核空间开辟一块内核缓存区，建立内核缓存区和数据接收缓存区之间的映射关系，以及数据接收缓存区和接收进程用户空间地址的映射关系。 3.发送方进程通过copy_from_user()函数将数据拷贝 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。

整个过程只使用了1次拷贝，不会因为不知道数据的大小而浪费空间或者时间，效率更高。

3.为什么要使用Binder

Android是基于Linux内核的 ，Linux提供了很多IPC机制，而Android却自己设计了Binder来进行通信，主要是因为以下几点。 **性能方面** 性能方面主要影响的因素是拷贝次数，管道、消息队列、Socket的拷贝次书都是两次，性能不是很好，共享内存不需要拷贝，性能最好，Binder的拷贝次书为1次，性能仅次于内存拷贝。 **稳定性方面** Binder是基于C/S架构的，这个架构通常采用两层结构，在技术上已经很成熟了，稳定性是没有问题的。共享内存没有分层，

难以控制，并发同步访问临界资源时，可能还会产生死锁。从稳定性的角度讲，Binder是优于共享内存的。 **安全方面** Android是一个开源的系统，并且拥有开放性的平台，市场上应用来源很广，因此安全性对于Android 平台而言极其重要。 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），无法鉴别对方身份。Android 为每个安装好的APP分配了自己的UID，通过进程的UID来鉴别进程身份。另外，Android系统中的Server端会判断UID/PID是否满足访问权限，而对外只暴露Client端，加强了系统的安全性。 **语言方面** Linux是基于C语言，C语言是面向过程的，Android应用层和Java Framework是基于Java语言，Java语言是面向对象的。Binder本身符合面向对象的思想，因此作为Android的通信机制更合适不过。

从这四方面来看，Linux提供的大部分IPC机制根本无法和Binder相比较，而共享内存只在性能方面优于Binder，其他方面都劣于Binder，这些就是为什么Android要使用Binder来进行进程间通信，当然系统中并不是所有的进程通信都是采用了Binder，而是根据场景选择最合适的，比如Zygote进程与AMS通信使用的是Socket，Kill Process采用的是信号。

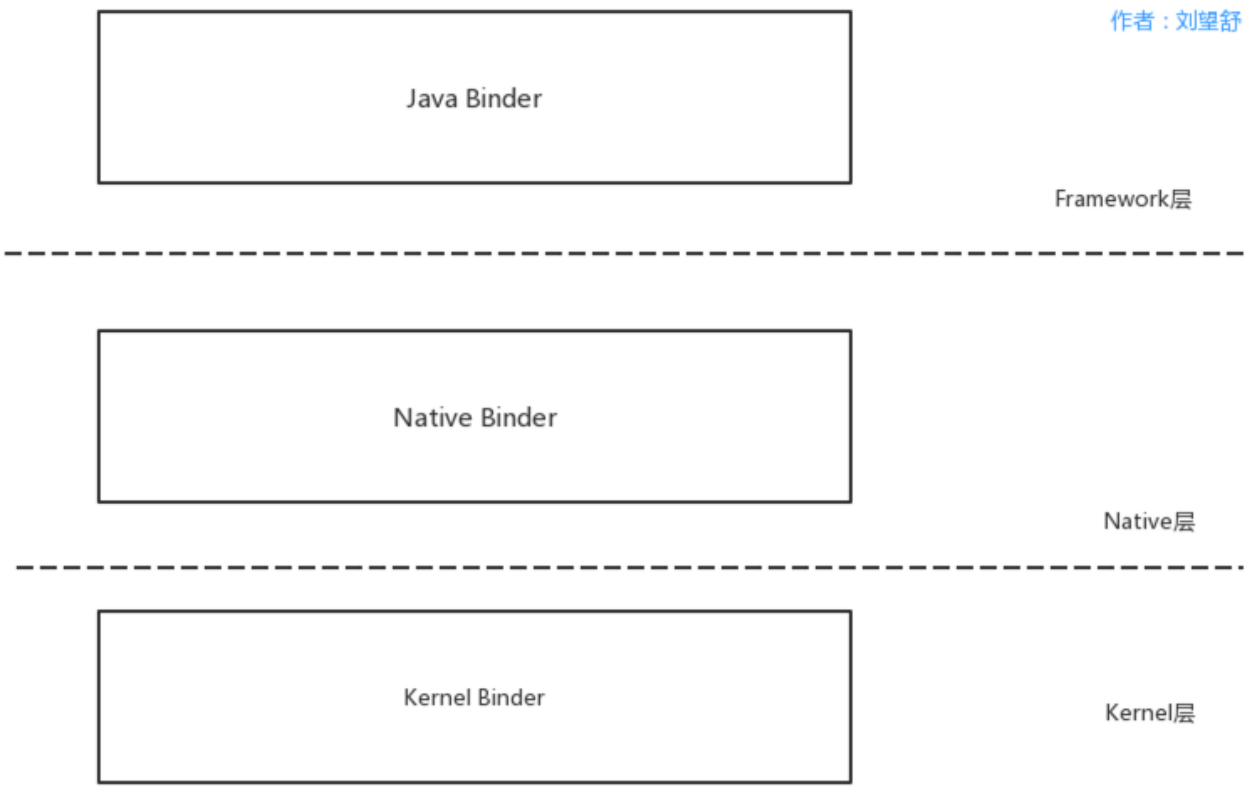
4.为什么要学习Binder?

Binder机制在Android中的地位举足轻重，我们需要掌握的很多原理都和Binder有关：

1. 系统中的各个进程是如何通信的？
2. Android系统启动过程
3. AMS、PMS的原理
4. 四大组件的原理，比如Activity是如何启动的？
5. 插件化原理
6. 系统服务的Client端和Server端是如何通信的？（比如MediaPlayer和MeidaPlayerService)

上面只是列了一小部分，简单来说说，比如系统在启动时，SystemServer进程启动后会创建Binder线程池，目的是通过Binder，使得在SystemServer进程中的服务可以和其他进程进行通信了。再比如我们常说的AMS、PMS都是基于Binder来实现的，拿PMS来说，PMS运行在SystemServer进程，如果它想要和DefaultContainerService通信（是用于检查和复制可移动文件的系统服务），就需要通过Binder，因为DefaultContainerService运行在com.android.defcontainer进程。 还有一个比较常见的C/S架构间通信的问题，Client端的MediaPlayer和Server端的MeidaPlayerService不是运行在一个进程中的，同样需要Binder来实现通信。

可以说Binder机制是掌握系统底层原理的基石。根据Android系统的分层，Binder机制主要分为以下几个部分。



上图并没有给出Binder机制的具体的细节，而是先给出了一个概念，根据系统的Android系统的分层，我将Binder机制分为了Java Binder、Native Binder、Kernel Binder，实际上Binder的内容非常多，完全可以写一本来介绍，但是对于应用开发来说，并不需要掌握那么多的知识点，因此本系列主要会讲解Java Binder和Native Binder。

感谢 https://mp.weixin.qq.com/s/NBm5lh8_ZLfodOXT8Ph5iA https://www.zhihu.com/question/39440766/answer/89210950 https://blog.csdn.net/carson_ho/article/details/73560642

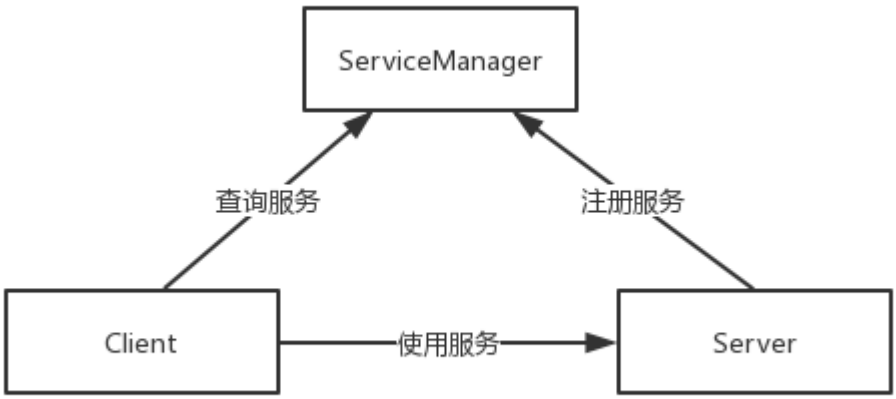
Android Binder原理（二）ServiceManager中的Binder机制

前言

在上一篇文章中，我们了解了学习Binder前必须要了解的知识点，其中有一点就是Binder机制的三个部分：Java Binder、Native Binder、Kernel Binder，其中Java Binder和Native Binder都是应用开发需要掌握的。Java Binder是需要借助Native Binder来工作的，因此需要先了解Native Binder，Native Binder架构的原型就是基于Binder通信的C/S架构，因此我们先从它开始入手。源码是基于Android 9.0。

1.基于Binder通信的C/S架构

在Android系统中，Binder进程间的通信的使用是很普遍的，在Android进阶三部曲第一部的最后一章，我讲解了MediaPlayer框架，这个框架基于C/S架构，并采用Binder来进行进程间通信，如下图所示。

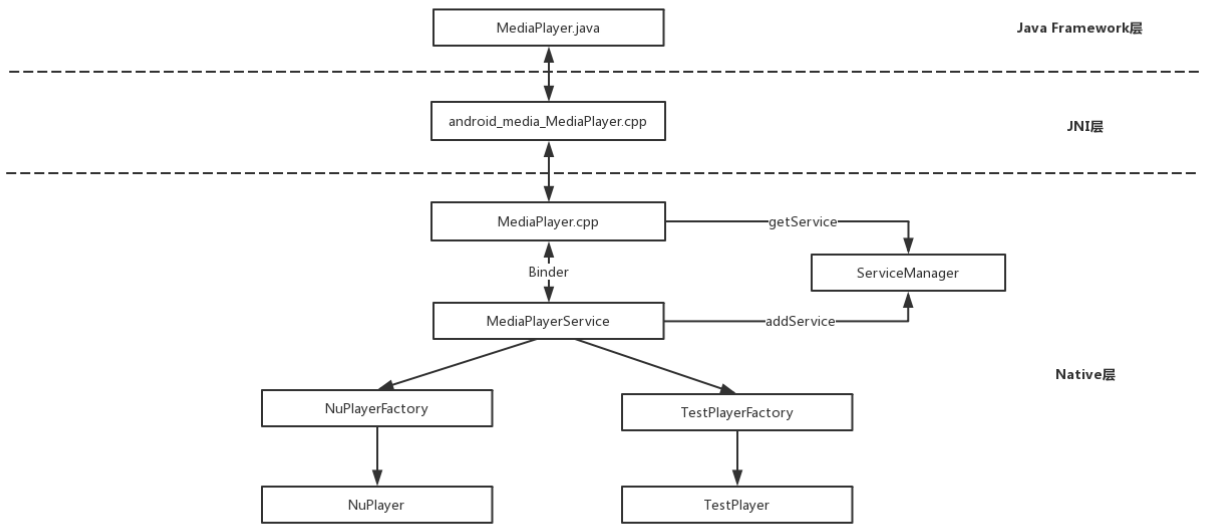


从图中可以看出，除了常规C/S架构的Client端和Server端，还包括了ServiceManager，它用于管理系统中的服务。首先Server进程会注册一些Service到ServiceManager中，Client要使用某个Service，则需要先到ServiceManager查询Service的相关信息，然后根据Service的相关信息与Service所在的Server进程建立通信通路，这样Client就可以使用Service了。

2.MediaServer的main函数

Client、Server、ServiceManager三者的交互都是基于Binder通信的，那么任意两者的交互都可以说明Binder的通信的原理，可以说Native Binder的原理的核心就是ServiceManager的原理，为了更好的了解ServiceManager，这里拿MediaPlayer框架来举例，它也是学习多媒体时必须要掌握的知识点。

MediaPlayer框架的简单框架图如下所示。



可以看到，MediaPlayer和MediaPlayerService是通过Binder来进行通信的，MediaPlayer是Client端，MediaPlayerService是Server端，MediaPlayerService是系统多媒体服务的一种，系统多媒体服务是由一个叫做MediaServer的服务进程提供的，它是一个可执行程序，在Android系统启动时，MediaServer也被启动，它的入口函数如下所示。 **frameworks/av/media/mediaserver/main_mediaserver.cpp**

```
CPP
int main(int argc __unused, char **argv __unused)
{
    signal(SIGPIPE, SIG_IGN);
    //获取ProcessState实例
    sp<ProcessState> proc(ProcessState::self());//1
    sp<IServiceManager> sm(defaultServiceManager());//2
    ALOGI("ServiceManager: %p", sm.get());
    InitializeIcuOrDie();
    //注册MediaPlayerService
    MediaPlayerService::instantiate();//3
    ResourceManagerService::instantiate();
    registerExtensions();
    //启动Binder线程池
    ProcessState::self()->startThreadPool();
    //当前线程加入到线程池
    IPCThreadState::self()->joinThreadPool();
}
```

注释1处用于获取ProcessState实例，在这一过程中会打开/dev/binder设备，并使用mmap为Binder驱动分配一个虚拟地址空间用来接收数据。 注释2处用来得到一个IServiceManager，通过这个IServiceManager，其他进程就可以和当前的ServiceManager进行交互，这里就用到了Binder通信。 注释3处用来注册MediaPlayerService。 除了注释3处的知识点在下一篇文章进行介绍，注释1和注释2处的内容，本篇文章会分别来进行介绍，先看ProcessState实例。

3.每个进程唯一的ProcessState

ProcessState从名称就可以看出来，用于代表进程的状态，先来查看上一小节的ProcessState的self函数。
frameworks/native/libs/binder/ProcessState.cpp

```
CPP
sp<ProcessState> ProcessState::self()
{
    Mutex::Autolock _l(gProcessMutex);
    if (gProcess != NULL) {
        return gProcess;
    }
    gProcess = new ProcessState("/dev/binder");//1
    return gProcess;
}
```

这里采用了单例模式，确保每个进程只有一个ProcessState实例。注释1处用于创建一个ProcessState实例，参数为/dev/binder。接着来查看ProcessState的构造函数，代码如下所示。
frameworks/native/libs/binder/ProcessState.cpp

```
CPP
ProcessState::ProcessState(const char *driver)
    : mDriverName(String8(driver))
    , mDriverFD(open_driver(driver))//1
    , mVMStart(MAP_FAILED)
    , mThreadCountLock(PTHREAD_MUTEX_INITIALIZER)
```

```
    , mThreadCountDecrement(PTHREAD_COND_INITIALIZER)
    , mExecutingThreadsCount(0)
    , mMaxThreads(DEFAULT_MAX_BINDER_THREADS)
    , mStarvationStartTimeMs(0)
    , mManagesContexts(false)
    , mBinderContextCheckFunc(NULL)
    , mBinderContextUserData(NULL)
    , mThreadPoolStarted(false)
    , mThreadPoolSeq(1)
{
    if (mDriverFD >= 0) {
        mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE,
mDriverFD, 0);//2
        if (mVMStart == MAP_FAILED) {
            // *sigh*
            ALOGE("Using %s failed: unable to mmap transaction memory.\n",
mDriverName.c_str());
            close(mDriverFD);
            mDriverFD = -1;
            mDriverName.clear();
        }
    }
    LOG_ALWAYS_FATAL_IF(mDriverFD < 0, "Binder driver could not be opened.  Terminating.");
}
```

ProcessState的构造函数中调用了很多函数，需要注意的是注释1处，它用来打开/dev/binder设备。 注释2处的mmap函数，它会在内核虚拟地址空间中申请一块与用户虚拟内存相同大小的内存，然后再申请物理内存，将同一块物理内存分别映射到内核虚拟地址空间和用户虚拟内存空间，实现了内核虚拟地址空间和用户虚拟内存空间的数据同步操作，也就是内存映射。 mmap函数用于对Binder设备进行内存映射，除了它还有open、ioctl函数，来看看它们做了什么。 注释1处的open_driver函数的代码如下所示。 **frameworks/native/libs/binder/ProcessState.cpp**

```
CPP
static int open_driver(const char *driver)
{
    int fd = open(driver, O_RDWR | O_CLOEXEC);//1
    if (fd >= 0) {
        ...
        size_t maxThreads = DEFAULT_MAX_BINDER_THREADS;
        result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);//2
        if (result == -1) {
            ALOGE("Binder ioctl to set max threads failed: %s", strerror(errno));
        }
    } else {
        ALOGW("Opening '%s' failed: %s\n", driver, strerror(errno));
    }
    return fd;
}
```

注释1处用于打开/dev/binder设备并返回文件操作符fd，这样就可以操作内核的Binder驱动了。注释2处的ioctl函数的作用就是和Binder设备进行参数的传递，这里的ioctl函数用于设定binder支持的最大线程数为15（maxThreads的值为15）。最终open_driver函数返回文件操作符fd。

ProcessState就分析倒这里，总的来说它做了以下几个重要的事： 1.打开/dev/binder设备并设定Binder最大的支持线程数。 2.通过mmap为binder分配一块虚拟地址空间，达到内存映射的目的。

4.ServiceManager中的Binder机制

回到第一小节的MediaServer的入口函数，在注释2处调用了defaultServiceManager函数。
frameworks/native/libs/binder/IServiceManager.cpp

```
CPP
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;

    {
        AutoMutex _l(gDefaultServiceManagerLock);
        while (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));//1
            if (gDefaultServiceManager == NULL)
                sleep(1);
        }
    }

    return gDefaultServiceManager;
}
```

从IServiceManager所在的文件路径就可以知道，ServiceManager中不仅仅使用了Binder通信，它自身也是属于Binder体系的。defaultServiceManager中同样使用了单例，注释1处的interface_cast函数生成了gDefaultServiceManager，其内部调用了ProcessState的getContextObject函数，代码如下所示。
frameworks/native/libs/binder/ProcessState.cpp

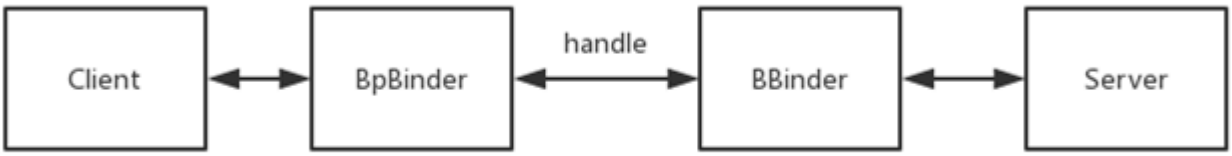
```
CPP
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& /*caller*/)
{
    return getStrongProxyForHandle(0);
}

sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;
    AutoMutex _l(mLock);
    handle_entry* e = lookupHandleLocked(handle);//1
    if (e != NULL) {
        IBinder* b = e->binder;
        if (b == NULL || !e->refs->attemptIncWeak(this)) {
            if (handle == 0) {
                Parcel data;
                status_t status = IPCThreadState::self()->transact(
                        0, IBinder::PING_TRANSACTION, data, NULL, 0);
                if (status == DEAD_OBJECT)
                   return NULL;
            }
```

```
            b = BpBinder::create(handle);//2
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
            result = b;
        } else {
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }

    return result;
}
```

getContextObject函数中直接调用了getStrongProxyForHandle函数，注意它的参数的值为0，那么handle的值就为0，handle是一个资源标识。注释1处查询这个资源标识对应的资源（handle_entry）是否存在，如果不存在就会在注释2处新建BpBinder，并在注释3处赋值给 handle_entry的binder。最终返回的result的值为BpBinder。

4.1 BpBinder和BBinder

说到BpBinder，不得不提到BBinder，它们是Binder通信的"双子星"，都继承了IBinder。BpBinder是Client端与Server交互的代理类，而BBinder则代表了Server端。BpBinder和BBinder是一一对应的，BpBinder会通过handle来找到对应的BBinder。 我们知道在ServiceManager中创建了BpBinder，通过handle(值为0)可以找到对应的BBinder。



分析完了ProcessState的getContextObject函数，回到interface_cast函数：

```
CPP
gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
```

interface_cast具体实现如下所示。 **frameworks/native/libs/binder/include/binder/IInterface.h**

```
CPP
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```

当前的场景中，INTERFACE的值为IServiceManager，那么替换后代码如下所示。

```CPP
inline sp<IServiceManager> interface_cast(const sp<IBinder>& obj)
{
    return IServiceManager::asInterface(obj);
}
```

我们接着来分析IServiceManager。

4.2 解密IServiceManager

BpBinder和BBinder负责Binder的通信，而IServiceManager用于处理ServiceManager的业务，IServiceManager是C++代码，因此它的定义在IServiceManager.h中。
frameworks/native/libs/binder/include/binder/IServiceManager.h

```CPP
class IServiceManager : public IInterface
{
public:
    DECLARE_META_INTERFACE(ServiceManager)//1
    ...
    //一些操作Service的函数
    virtual sp<IBinder>         getService( const String16& name) const = 0;
    virtual sp<IBinder>         checkService( const String16& name) const = 0;
    virtual status_t addService(const String16& name, const sp<IBinder>& service,
                                bool allowIsolated = false,
                                int dumpsysFlags = DUMP_FLAG_PRIORITY_DEFAULT) = 0;
    virtual Vector<String16> listServices(int dumpsysFlags = DUMP_FLAG_PRIORITY_ALL) = 0;
    enum {
        GET_SERVICE_TRANSACTION = IBinder::FIRST_CALL_TRANSACTION,
        CHECK_SERVICE_TRANSACTION,
        ADD_SERVICE_TRANSACTION,
        LIST_SERVICES_TRANSACTION,
    };
};
```

可以看到IServiceManager继承了IInterface，其内部定义了一些常量和一些操作Service的函数，在注释1处调用了DECLARE_META_INTERFACE宏，它的定义在IInterface.h中。
frameworks/native/libs/binder/include/binder/IInterface.h

```CPP
#define DECLARE_META_INTERFACE(INTERFACE)                               \
    static const ::android::String16 descriptor;                        \
    static ::android::sp<I##INTERFACE> asInterface(                     \
            const ::android::sp<::android::IBinder>& obj);              \
    virtual const ::android::String16& getInterfaceDescriptor() const;  \
    I##INTERFACE();                                                     \
    virtual ~I##INTERFACE();
```

其中INTERFACE的值为ServiceManager，那么经过替换后的代码如下所示。

```cpp
    static const ::android::String16 descriptor;
```

```
    //定义asInterface函数
    static ::android::sp<IServiceManager> asInterface(
            const ::android::sp<::android::IBinder>& obj);
    virtual const ::android::String16& getInterfaceDescriptor() const;
    //定义IServiceManager构造函数
    IServiceManager();
    //定义IServiceManager析构函数
    virtual ~IServiceManager();
```
从DECLARE_META_INTERFACE宏的名称和上面的代码中，可以发现它主要声明了一些函数和一个变量。那么这些函数和变量的实现在哪呢？答案还是在IInterface.h中，叫做IMPLEMENT_META_INTERFACE宏，代码如下所示/
frameworks/native/libs/binder/include/binder/IInterface.h
```cpp
#define IMPLEMENT_META_INTERFACE(INTERFACE, NAME)                       \
    const ::android::String16 I##INTERFACE::descriptor(NAME);           \
    const ::android::String16&                                          \
            I##INTERFACE::getInterfaceDescriptor() const {              \
        return I##INTERFACE::descriptor;                                \
    }                                                                   \
    ::android::sp<I##INTERFACE> I##INTERFACE::asInterface(              \
            const ::android::sp<::android::IBinder>& obj)               \
    {                                                                   \
        ::android::sp<I##INTERFACE> intr;                               \
        if (obj != NULL) {                                              \
            intr = static_cast<I##INTERFACE*>(                          \
                obj->queryLocalInterface(                               \
                        I##INTERFACE::descriptor).get());               \
            if (intr == NULL) {                                         \
                intr = new Bp##INTERFACE(obj);                          \
            }                                                           \
        }                                                               \
        return intr;                                                    \
    }                                                                   \
    I##INTERFACE::I##INTERFACE() { }                                    \
    I##INTERFACE::~I##INTERFACE() { }                                   \
```

DECLARE_META_INTERFACE宏和IMPLEMENT_META_INTERFACE宏是配合使用的，很多系统服务都使用了它们，IServiceManager使用IMPLEMENT_META_INTERFACE宏只有一行代码，如下所示。
frameworks/native/libs/binder/IServiceManager.cpp

```
CPP
IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager");
```

IMPLEMENT_META_INTERFACE宏的INTERFACE值为ServiceManager，NAME值为”android.os.IServiceManager”，进行替换后的代码如下所示。

```
CPP
const ::android::String16 IServiceManager::descriptor("android.os.IServiceManager");
const ::android::String16&
        IServiceManager::getInterfaceDescriptor() const {
    return IServiceManager::descriptor;
}
```

```
 //实现了asInterface函数
::android::sp<IServiceManager> IServiceManager::asInterface(
        const ::android::sp<::android::IBinder>& obj)
{
    ::android::sp<IServiceManager> intr;
    if (obj != NULL) {
        intr = static_cast<IServiceManager>(
            obj->queryLocalInterface(
                    IServiceManager::descriptor).get());
        if (intr == NULL) {
            intr = new BpServiceManager(obj);//1
        }
    }
    return intr;
}
IServiceManager::IServiceManager() { }
IServiceManager::~IServiceManager() { }
```

关键的点就在于注释1处，新建了一个BpServiceManager，传入的参数obj的值为BpBinder。看到这里，我们也就明白了，asInterface函数就是用BpBinder为参数创建了BpServiceManager，从而推断出interface_cast函数创建了BpServiceManager，再往上推断，IServiceManager的defaultServiceManager函数返回的就是BpServiceManager。 BpServiceManager有什么作用呢，先从BpServiceManager的构造函数看起。
frameworks/native/libs/binder/IServiceManager.cpp

```
CPP
class BpServiceManager : public BpInterface<IServiceManager>
{
public:
    explicit BpServiceManager(const sp<IBinder>& impl)
        : BpInterface<IServiceManager>(impl)
    {
    }
...
}
```

impl的值其实就是BpBinder，BpServiceManager的构造函数调用了基类BpInterface的构造函数。
frameworks/native/libs/binder/include/binder/IInterface.h

```
CPP
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
...
};
```

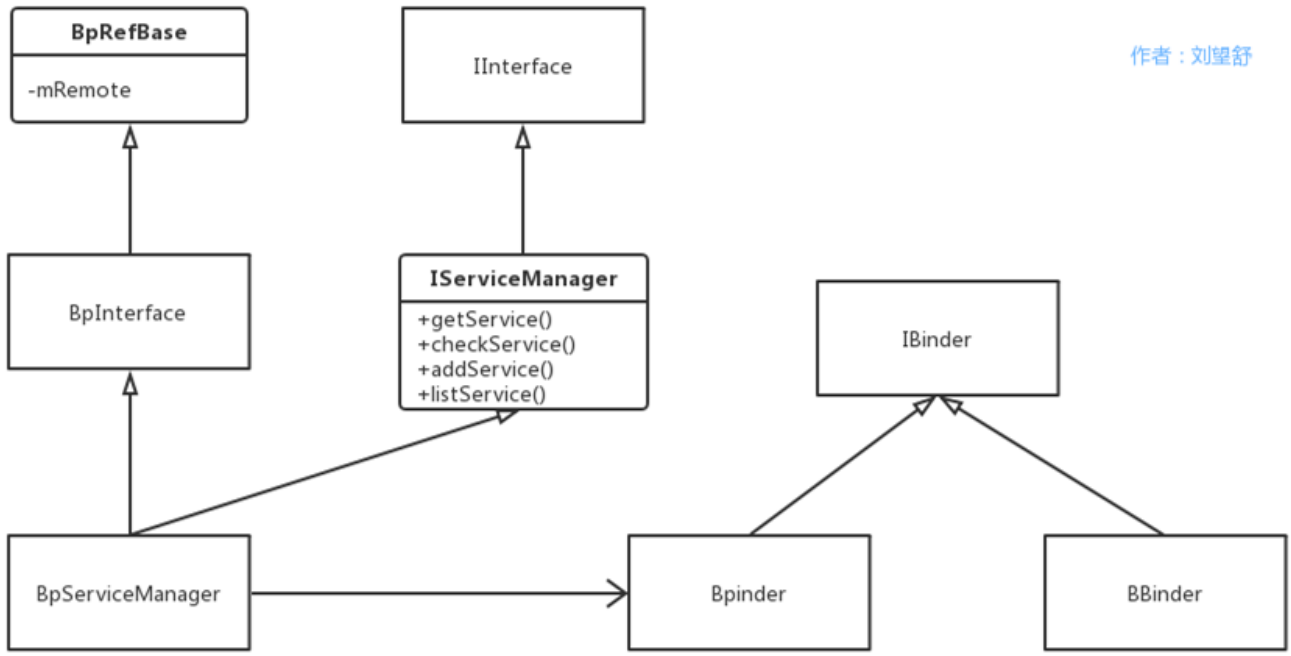
BpInterface继承了BpRefBase，BpRefBase的实现如下所示。 **frameworks/native/libs/binder/Binder.cpp**

```
CPP
BpRefBase::BpRefBase(const sp<IBinder>& o)
    : mRemote(o.get()), mRefs(NULL), mState(0)
{
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);

    if (mRemote) {
        mRemote->incStrong(this);
        mRefs = mRemote->createWeak(this);
    }
}
```

mRemote是一个IBinder* 指针，它最终的指向为BpBinder，也就是说BpServiceManager的mRemote指向了BpBinder。那么BpServiceManager的作用也就知道了，就是它实现了IServiceManager，并且通过BpBinder来实现通信。

4.3 IServiceManager家族

可能上面讲的会让你有些头晕，这是因为对各个类的关系不大明确，通过下图也许你就会豁然开朗。



1.BpBinder和BBinder都和通信有关，它们都继承自IBinder。 2.BpServiceManager派生自IServiceManager，它们都和业务有关。 3.BpRefBase包含了mRemote，通过不断的派生，BpServiceManager也同样包含mRemote，它指向了BpBinder，通过BpBinder来实现通信。

5.小节

本篇文章我们学到了Binder通信的C/S架构，也知道了Native Binder的原理的核心其实就是ServiceManager的原理，为了讲解ServiceManager的原理，我们需要一个框架来举例，那就是MediaPlayer框架。在讲解MediaServer的入口函数时，我们遇到了三个问题，其中前两个问题相关的知识点ProcessState和IServiceManager都讲解到了，下一篇文章会讲解第三个问题，MediaPlayerService是如何注册的。

Android Binder原理（三）系统服务的注册过程

前言

在上一篇文章中，我们学习了ServiceManager中的Binder机制，有一个问题由于篇幅问题没有讲完，那就是MediaPlayerService是如何注册的。通过了解MediaPlayerService是如何注册的，可以得知系统服务的注册过程。

1.从调用链角度说明MediaPlayerService是如何注册的

我们先来看MediaServer的入口函数，代码如下所示。
frameworks/av/media/mediaserver/main_mediaserver.cpp

```
CPP
int main(int argc __unused, char **argv __unused)
{
    signal(SIGPIPE, SIG_IGN);
    //获取ProcessState实例
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm(defaultServiceManager());
    ALOGI("ServiceManager: %p", sm.get());
    InitializeIcuOrDie();
    //注册MediaPlayerService
    MediaPlayerService::instantiate();//1
    ResourceManagerService::instantiate();
    registerExtensions();
    //启动Binder线程池
    ProcessState::self()->startThreadPool();
    //当前线程加入到线程池
    IPCThreadState::self()->joinThreadPool();
}
```

这段代码中的很多内容都在上一篇文章介绍过了，接着分析注释1处的代码。

frameworks/av/media/libmediaplayerservice/MediaPlayerService.cpp

```
CPP
void MediaPlayerService::instantiate() {
    defaultServiceManager()->addService(
            String16("media.player"), new MediaPlayerService，());
}
```

defaultServiceManager返回的是BpServiceManager，不清楚的看[Android Binder原理（二）ServiceManager中的Binder机制][1]这篇文章。参数是一个字符串和MediaPlayerService，看起来像是Key/Value的形式来完成注册，接着看addService函数。

frameworks/native/libs/binder/IServiceManager.cpp

```
CPP
virtual status_t addService(const String16& name, const sp<IBinder>& service,
                                bool allowIsolated, int dumpsysPriority) {
       Parcel data, reply;//数据包
       data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
       data.writeString16(name); //name值为"media.player"
       data.writeStrongBinder(service); //service值为MediaPlayerService
       data.writeInt32(allowIsolated ? 1 : 0);
       data.writeInt32(dumpsysPriority);
       status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);//1
       return err == NO_ERROR ? reply.readExceptionCode() : err;
   }
```

data是一个数据包，后面会不断的将数据写入到data中， 注释1处的remote()指的是mRemote，也就是BpBinder。addService函数的作用就是将请求数据打包成data，然后传给BpBinder的transact函数，代码如下所示。
frameworks/native/libs/binder/BpBinder.cpp

```
CPP
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}
```

BpBinder将逻辑处理交给IPCThreadState，先来看IPCThreadState::self()干了什么？
frameworks/native/libs/binder/IPCThreadState.cpp

```
CPP
IPCThreadState* IPCThreadState::self()
{
    //首次进来gHaveTLS的值为false
    if (gHaveTLS) {
restart:
        const pthread_key_t k = gTLS;//1
        IPCThreadState* st = (IPCThreadState*)pthread_getspecific(k);//2
        if (st) return st;
        return new IPCThreadState;//3
    }
    ...
    pthread_mutex_unlock(&gTLSMutex);
    goto restart;
}
```

注释1处的TLS的全称为Thread local storage，指的是线程本地存储空间，在每个线程中都有TLS，并且线程间不共享。注释2处用于获取TLS中的内容并赋值给IPCThreadState*指针。注释3处会新建一个IPCThreadState，这里可以得知IPCThreadState::self()实际上是为了创建IPCThreadState，它的构造函数如下所示。
frameworks/native/libs/binder/IPCThreadState.cpp

```
CPP
IPCThreadState::IPCThreadState()
    : mProcess(ProcessState::self()),
      mStrictModePolicy(0),
      mLastTransactionBinderFlags(0)
{
    pthread_setspecific(gTLS, this);//1
    clearCaller();
    mIn.setDataCapacity(256);
    mOut.setDataCapacity(256);
}
```

注释1处的pthread_setspecific函数用于设置TLS，将IPCThreadState::self()获得的TLS和自身传进去。IPCThreadState中还包含mIn、一个mOut，其中mIn用来接收来自Binder驱动的数据，mOut用来存储发往Binder驱动的数据，它们默认大小都为256字节。 知道了IPCThreadState的构造函数，再回来查看IPCThreadState的transact函数。 **frameworks/native/libs/binder/IPCThreadState.cpp**

```
CPP
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    status_t err;

    flags |= TF_ACCEPT_FDS;
    ...
    err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);//1

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
       ...
        if (reply) {
            err = waitForResponse(reply);//2
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
       ...
    } else {
       //不需要等待reply的分支
        err = waitForResponse(NULL, NULL);
    }
```

```
    return err;
}
```

调用BpBinder的transact函数实际上就是调用IPCThreadState的transact函数。注释1处的writeTransactionData函数用于传输数据，其中第一个参数BC_TRANSACTION代表向Binder驱动发送命令协议，向Binder设备发送的命令协议都以BC*开头，而Binder驱动返回的命令协议以BR*开头。这个命令协议我们先记住，后面会再次提到他。

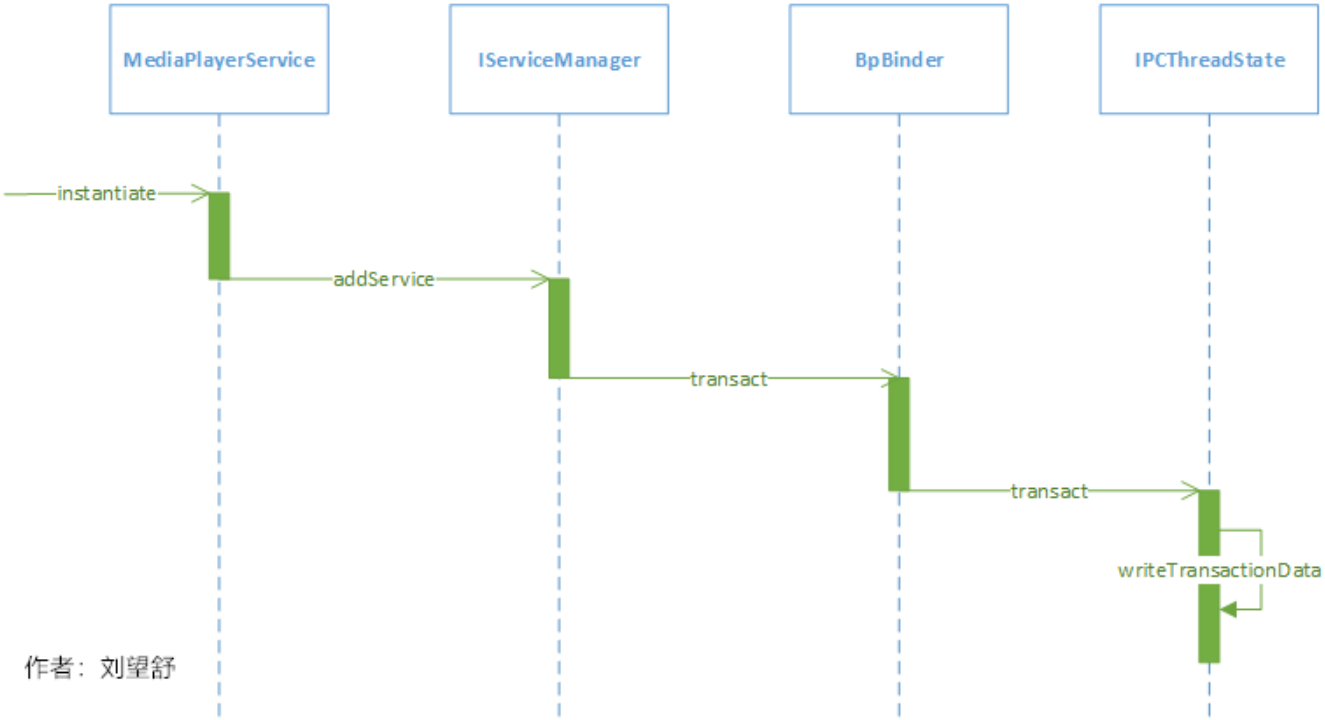
现在分别来分析注释1的writeTransactionData函数和注释2处的waitForResponse函数。

1.1 writeTransactionData函数分析

frameworks/native/libs/binder/IPCThreadState.cpp

```
CPP
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
    int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
{
    binder_transaction_data tr;//1

    tr.target.ptr = 0;
    tr.target.handle = handle;//2
    tr.code = code;  //code=ADD_SERVICE_TRANSACTION
    tr.flags = binderFlags;
    tr.cookie = 0;
    tr.sender_pid = 0;
    tr.sender_euid = 0;

    const status_t err = data.errorCheck();//3
    if (err == NO_ERROR) {
        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        tr.offsets_size = data.ipcObjectsCount()*sizeof(binder_size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    } else if (statusBuffer) {
        tr.flags |= TF_STATUS_CODE;
        *statusBuffer = err;
        tr.data_size = sizeof(status_t);
        tr.data.ptr.buffer = reinterpret_cast<uintptr_t>(statusBuffer);
        tr.offsets_size = 0;
        tr.data.ptr.offsets = 0;
    } else {
        return (mLastError = err);
    }

    mOut.writeInt32(cmd);  //cmd=BC_TRANSACTION
    mOut.write(&tr, sizeof(tr));

    return NO_ERROR;
}
```

注释1处的binder_transaction_data结构体(tr结构体）是向Binder驱动通信的数据结构，注释2处将handle传递给target的handle，用于标识目标，这里的handle的值为0，代表了ServiceManager。 注释3处对数据data进行错误检查，如果没有错误就将数据赋值给对应的tr结构体。最后会将BC_TRANSACTION和tr结构体写入到mOut中。 上面代码调用链的时序图如下所示。



1.2 waitForResponse函数分析

接着回过头来查看waitForResponse函数做了什么，waitForResponse函数中的case语句很多，这里截取部分代码。
frameworks/native/libs/binder/IPCThreadState.cpp

```
CPP
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    uint32_t cmd;
    int32_t err;
    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;//1
        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;
        cmd = (uint32_t)mIn.readInt32();
        IF_LOG_COMMANDS() {
            alog << "Processing waitForResponse Command: "
                << getReturnString(cmd) << endl;
        }
        switch (cmd) {
        case BR_TRANSACTION_COMPLETE:
            if (!reply && !acquireResult) goto finish;
            break;

        case BR_DEAD_REPLY:
            err = DEAD_OBJECT;
            goto finish;
```

```
       ...
        default:
            //处理各种命令协议
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
            break;
        }
}
finish:
    ...
    return err;
}
```

注释1处的talkWithDriver函数的内部通过ioctl与Binder驱动进行通信，代码如下所示。
frameworks/native/libs/binder/IPCThreadState.cpp

```
CPP
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    if (mProcess->mDriverFD <= 0) {
        return -EBADF;
    }
    //和Binder驱动通信的结构体
    binder_write_read bwr; //1
    //mIn是否有可读的数据，接收的数据存储在mIn
    const bool needRead = mIn.dataPosition() >= mIn.dataSize();
    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;
    bwr.write_size = outAvail;
    bwr.write_buffer = (uintptr_t)mOut.data();//2
    //这时doReceive的值为true
    if (doReceive && needRead) {
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (uintptr_t)mIn.data();//3
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }
   ...
    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;
    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
    status_t err;
    do {
        IF_LOG_COMMANDS() {
            alog << "About to read/write, write size = " << mOut.dataSize() << endl;
        }
#if defined(__ANDROID__)
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)//4
            err = NO_ERROR;
        else
            err = -errno;
#else
        err = INVALID_OPERATION;
```

```
#endif
     ...
    } while (err == -EINTR);
    ...
    return err;
}
```

注释1处的 binder_write_read是和Binder驱动通信的结构体，在注释2和3处将mOut、mIn赋值给binder_write_read的相应字段，最终通过注释4处的ioctl函数和Binder驱动进行通信，这一部分涉及到Kernel Binder的内容 了，就不再详细介绍了，只需要知道在Kernel Binder中会记录服务名和handle，用于后续的服务查询。

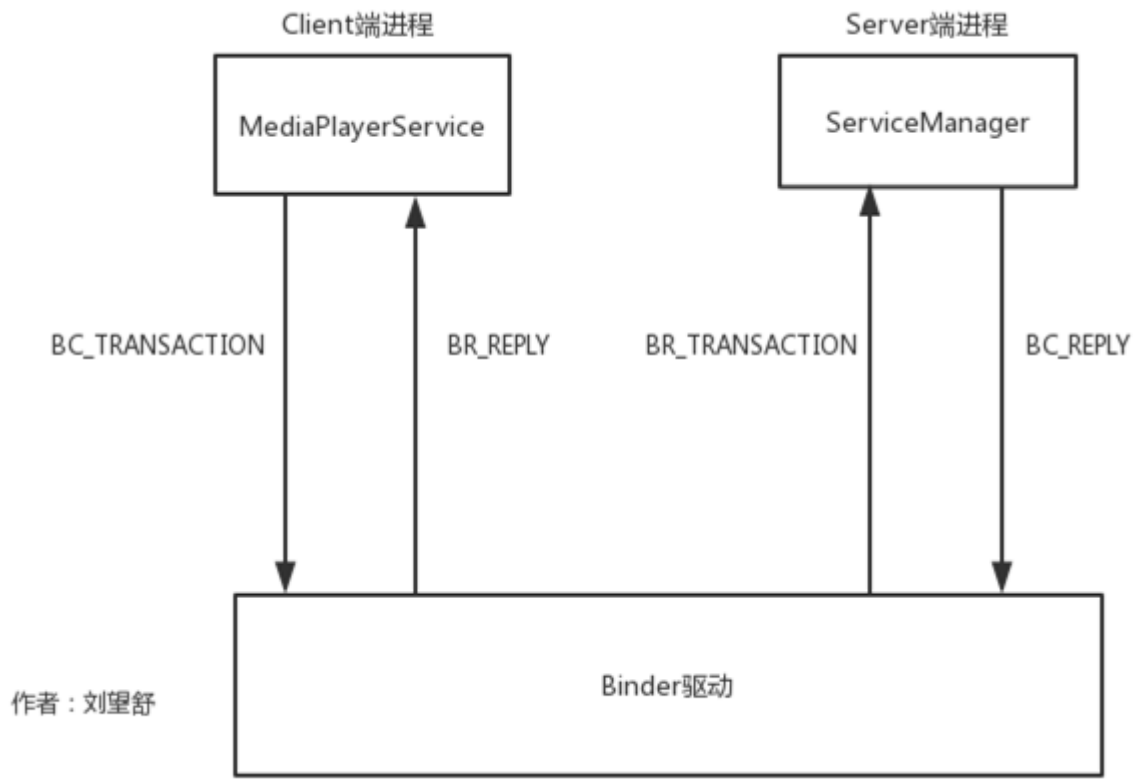
1.3 小节

从调用链的角度来看，MediaPlayerService是如何注册的貌似并不复杂，因为这里只是简单的介绍了一个调用链分支，可以简单的总结为以下几个步骤：

1. addService函数将数据打包发送给BpBinder来进行处理。
2. BpBinder新建一个IPCThreadState对象，并将通信的任务交给IPCThreadState。
3. IPCThreadState的writeTransactionData函数用于将命令协议和数据写入到mOut中。
4. IPCThreadState的waitForResponse函数主要做了两件事，一件事是通过ioctl函数操作mOut和mIn来与Binder驱动进行数据交互，另一件事是处理各种命令协议。

2.从进程角度说明MediaPlayerService是如何注册的

实际上MediaPlayerService的注册还涉及到了进程，如下图所示。



从图中看出是以C/S架构为基础，addService是在MediaPlayerService进行的，它是Client端，用于请求添加系统服务。而Server端则是指的是ServiceManager，用于完成系统服务的添加。 Client端和Server端分别运行在两个进程中，通过向Binder来进行通信。更详细点描述，就是两端通过向Binder驱动发送命令协议来完成系统服务的添加。这其中命令协议非常多，过程也比较复杂，这里对命令协议进行了简化，只涉及到了四个命令协议，其中BC_TRANSACTION和BR_TRANSACTION过程是一个完整的事务，BC_REPLY和BR_REPLY是一个完整的事务。 Client端和Server端向Binder驱动发送命令协议以BC开头，而Binder驱动向Client端和Server端返回的命令协议以BR_开头。

步骤如下所示： 1.Client端向Binder驱动发送BC_TRANSACTION命令。 2.Binder驱动接收到请求后生成BR_TRANSACTION命令，唤醒Server端的线程后将BR_TRANSACTION命令发送给ServiceManager。 3.Server端中的服务注册完成后，生成BC_REPLY命令发送给Binder驱动。 4.Binder驱动生成BR_REPLY命令，唤醒Client端的线程后将BR_REPLY命令发送个Client端。

通过这些协议命令来驱动并完成系统服务的注册。

3.总结

本文分别从调用链角度和进程角度来讲解MediaPlayerService是如何注册的，间接的得出了服务是如何注册的 。这两个角度都比较复杂，因此这里分别对这两个角度做了简化，作为应用开发，我们不需要注重太多的过程和细节，只需要了解大概的步骤即可。

Android Binder原理（四）ServiceManager的启动过程

前言

在上一篇文章中，我们以MediaPlayerService为例，讲解了系统服务是如何注册的（addService），既然有注册就势必要有获取，但是在了解获取服务前，我们最好先了解ServiceManager的启动过程，这样更有助于理解系统服务的注册和获取的过程。

另外还有一点需要说明的是，要想了解ServiceManager的启动过程，需要查看Kernel Binder部分的源码，这部分代码在内核源码中，AOSP源码是不包括内核源码的，因此需要单独下载，见Android AOSP基础（二）AOSP源码和内核源码下载这篇文章。

1.ServiceManager的入口函数

ServiceManager是init进程负责启动的，具体是在解析init.rc配置文件时启动的，init进程是在系统启动时启动的，因此ServiceManager亦是如此，不理解init进程和init.rc的可以看Android系统启动流程（一）解析init进程启动过程这篇文章。 rc文件内部由Android初始化语言编写（Android Init Language）编写的脚本，它主要包含五种类型语句：Action、Commands、Services、Options和Import。 在Android 7.0中对init.rc文件进行了拆分，每个服务一个rc文件。ServiceManager的启动脚本在servicemanager.rc中：
frameworks/native/cmds/servicemanager/servicemanager.rc

```
CPP
service servicemanager /system/bin/servicemanager
    class core animation
    user system  //1
    group system readproc
```

```
    critical //2
    onrestart restart healthd
    onrestart restart zygote
    onrestart restart audioserver
    onrestart restart media
    onrestart restart surfaceflinger
    onrestart restart inputflinger
    onrestart restart drm
    onrestart restart cameraserver
    onrestart restart keystore
    onrestart restart gatekeeperd
    writepid /dev/cpuset/system-background/tasks
    shutdown critical
```

service用于通知init进程创建名为servicemanager的进程，这个servicemanager进程执行程序的路径为/system/bin/servicemanager。 注释1的关键字user说明servicemanager是以用户system的身份运行的，注释2处的critical说明servicemanager是系统中的关键服务，关键服务是不会退出的，如果退出了，系统就会重启，当系统重启时就会启动用onrestart关键字修饰的进程，比如zygote、media、surfaceflinger等等。

servicemanager的入口函数在service_manager.c中:
frameworks/native/cmds/servicemanager/service_manager.c

```
CPP
int main(int argc, char** argv)
{
    struct binder_state *bs;//1
    union selinux_callback cb;
    char *driver;

    if (argc > 1) {
        driver = argv[1];
    } else {
        driver = "/dev/binder";
    }

    bs = binder_open(driver, 128*1024);//2
    ...
    if (binder_become_context_manager(bs)) {//3
        ALOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }
    ...
    if (getcon(&service_manager_context) != 0) {
        ALOGE("SELinux: Failed to acquire service_manager context. Aborting.\n");
        abort();
    }
    binder_loop(bs, svcmgr_handler);//4

    return 0;
}
```

注释1处的binder_state结构体用来存储binder的三个信息：

```
CPP
struct binder_state
{
    int fd; //binder设备的文件描述符
    void *mapped; //binder设备文件映射到进程的地址空间
    size_t mapsize; //内存映射后，系统分配的地址空间的大小，默认为128KB
};
```

main函数主要做了三件事： 1.注释2处调用binder_open函数用于打开binder设备文件，并申请128k字节大小的内存空间。 2.注释3处调用binder_become_context_manager函数，将servicemanager注册成为Binder机制的上下文管理者。 3.注释4处调用binder_loop函数，循环等待和处理client端发来的请求。

现在对这三件事分别进行讲解。

1.1 打开binder设备

binder_open函数用于打开binder设备文件，并且将它映射到进程的地址空间，如下所示。

frameworks/native/cmds/servicemanager/binder.c

```
CPP
struct binder_state *binder_open(const char* driver, size_t mapsize)
{
    struct binder_state *bs;
    struct binder_version vers;

    bs = malloc(sizeof(*bs));
    if (!bs) {
        errno = ENOMEM;
        return NULL;
    }

    bs->fd = open(driver, O_RDWR | O_CLOEXEC);//1
    if (bs->fd < 0) {
        fprintf(stderr,"binder: cannot open %s (%s)\n",
                driver, strerror(errno));
        goto fail_open;
    }
    //获取Binder的version
    if ((ioctl(bs->fd, BINDER_VERSION, &vers) == -1) ||
        (vers.protocol_version != BINDER_CURRENT_PROTOCOL_VERSION)) {//2
        fprintf(stderr,
                "binder: kernel driver version (%d) differs from user space version (%d)\n",
                vers.protocol_version, BINDER_CURRENT_PROTOCOL_VERSION);
        goto fail_open;
    }

    bs->mapsize = mapsize;
    bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);//3
    if (bs->mapped == MAP_FAILED) {
        fprintf(stderr,"binder: cannot map device (%s)\n",
                strerror(errno));
        goto fail_map;
```

```
    }
    return bs;

fail_map:
    close(bs->fd);
fail_open:
    free(bs);
    return NULL;
}
```

注释1处用于打开binder设备文件，后面会进行分析。 注释2处的ioctl函数用于获取Binder的版本，如果获取不到或者内核空间和用户空间的binder不是同一个版本就会直接goto到fail_open标签，释放binder的内存空间。 注释3处调用mmap函数进行内存映射，通俗来讲就是将binder设备文件映射到进程的地址空间，地址空间的大小为mapsize，也就是128K。映射完毕后会将地址空间的起始地址和大小保存在binder_state结构体中的mapped和mapsize变量中。

这里着重说一下open函数，它会调用Kernel Binder部分的binder_open函数，这部分源码位于内核源码中，这里展示的代码版本为goldfish3.4。

用户态和内核态 临时插入一个知识点:用户态和内核态 Intel的X86架构的CPU提供了0到3四个特权级，数字越小，权限越高，Linux操作系统中主要采用了0和3两个特权级，分别对应的就是内核态与用户态。用户态的特权级别低，因此进程在用户态下不经过系统调用是无法主动访问到内核空间中的数据的，这样用户无法随意的进入所有进程共享的内核空间，起到了保护的作用。下面来介绍下什么是用户态和内核态。 当一个进程在执行用户自己的代码时处于用户态，比如open函数，它运行在用户空间，当前的进程处于用户态。 当一个进程因为系统调用进入内核代码中执行时就处于内核态，比如open函数通过系统调用（__open()函数），查找到了open函数在Kernel Binder对应的函数为binder_open，这时binder_open运行在内核空间，当前的进程由用户态切换到内核态。

kernel/goldfish/drivers/staging/android/binder.c

```
CPP
static int binder_open(struct inode *nodp, struct file *filp)
{   //代表Binder进程
    struct binder_proc *proc;//1
    binder_debug(BINDER_DEBUG_OPEN_CLOSE, "binder_open: %d:%d\n",
             current->group_leader->pid, current->pid);
    //分配内存空间
    proc = kzalloc(sizeof(*proc), GFP_KERNEL);//2
    if (proc == NULL)
        return -ENOMEM;
    get_task_struct(current);
    proc->tsk = current;
    INIT_LIST_HEAD(&proc->todo);
    init_waitqueue_head(&proc->wait);
    proc->default_priority = task_nice(current);
    //binder同步锁
    binder_lock(__func__);

    binder_stats_created(BINDER_STAT_PROC);
    hlist_add_head(&proc->proc_node, &binder_procs);
    proc->pid = current->group_leader->pid;
    INIT_LIST_HEAD(&proc->delivered_death);
    filp->private_data = proc;//3
```

```
    //binder同步锁释放
    binder_unlock(__func__);
    ...
    return 0;
}
```

注释1处的binder_proc结构体代表binder进程，用于管理binder的各种信息。注释2处用于为binder_proc分配内存空间。注释3处将binder_proc赋值给file指针的private_data变量，后面的1.2小节会再次提到这个private_data变量。

1.2 注册成为Binder机制的上下文管理者

binder_become_context_manager函数用于将servicemanager注册成为Binder机制的上下文管理者，这个管理者在整个系统只有一个，代码如下所示。 **frameworks/native/cmds/servicemanager/binder.c**

```
CPP
int binder_become_context_manager(struct binder_state *bs)
{
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

ioctl函数会调用Binder驱动的binder_ioctl函数，binder_ioctl函数代码比较多，这里截取BINDER_SET_CONTEXT_MGR的处理部分，代码如下所示。 **kernel/goldfish/drivers/staging/android/binder.c**

```
CPP
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    int ret;
    struct binder_proc *proc = filp->private_data; //1
    struct binder_thread *thread;
    unsigned int size = _IOC_SIZE(cmd);
    void __user *ubuf = (void __user *)arg;
    trace_binder_ioctl(cmd, arg);

    ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
    if (ret)
        goto err_unlocked;

    binder_lock(__func__);
    thread = binder_get_thread(proc);//2
    if (thread == NULL) {
        ret = -ENOMEM;
        goto err;
    }

    switch (cmd) {
    ...
    case BINDER_SET_CONTEXT_MGR:
        if (binder_context_mgr_node != NULL) {//3
            printk(KERN_ERR "binder: BINDER_SET_CONTEXT_MGR already set\n");
            ret = -EBUSY;
            goto err;
```

```
        }
        ret = security_binder_set_context_mgr(proc->tsk);
        if (ret < 0)
            goto err;
        if (binder_context_mgr_uid != -1) {//4
            if (binder_context_mgr_uid != current->cred->euid) {//5
                printk(KERN_ERR "binder: BINDER_SET_"
                       "CONTEXT_MGR bad uid %d != %d\n",
                       current->cred->euid,
                       binder_context_mgr_uid);
                ret = -EPERM;
                goto err;
            }
        } else
            binder_context_mgr_uid = current->cred->euid;//6
        binder_context_mgr_node = binder_new_node(proc, NULL, NULL);//7
        if (binder_context_mgr_node == NULL) {
            ret = -ENOMEM;
            goto err;
        }
        binder_context_mgr_node->local_weak_refs++;
        binder_context_mgr_node->local_strong_refs++;
        binder_context_mgr_node->has_strong_ref = 1;
        binder_context_mgr_node->has_weak_ref = 1;
        break;
 ...
err_unlocked:
    trace_binder_ioctl_done(ret);
    return ret;
}
```

注释1处将file指针中的private_data变量赋值给binder_proc，这个private_data变量在binder_open函数中讲过，是一个binder_proc结构体。注释2处的binder_get_thread函数用于获取binder_thread，binder_thread结构体指的是binder线程，binder_get_thread函数内部会从传入的参数binder_proc中查找binder_thread，如果查询到直接返回，如果查询不到会创建一个新的binder_thread并返回。 注释3处的全局变量binder_context_mgr_node代表的是Binder机制的上下文管理者对应的一个Binder对象，如果它不为NULL，说明此前自身已经被注册为Binder的上下文管理者了，Binder的上下文管理者是不能重复注册的，因此会goto到err标签。 注释4处的全局变量binder_context_mgr_uid代表注册了Binder机制上下文管理者的进程的有效用户ID，如果它的值不为-1，说明此前已经有进程注册Binder的上下文管理者了，因此在注释5处判断当前进程的有效用户ID是否等于binder_context_mgr_uid，不等于就goto到err标签。 如果不满足注释4的条件，说明此前没有进程注册Binder机制的上下文管理者，就会在注释6处将当前进程的有效用户ID赋值给全局变量binder_context_mgr_uid，另外还会在注释7处调用binder_new_node函数创建一个Binder对象并赋值给全局变量binder_context_mgr_node。

1.3 循环等待和处理client端发来的请求

servicemanager成功注册成为Binder机制的上下文管理者后，servicemanager就是Binder机制的“总管”了，它需要在系统运行期间处理client端的请求，由于client端的请求不确定何时发送，因此需要通过无限循环来实现，实现这一需求的函数就是binder_loop。 **frameworks/native/cmds/servicemanager/binder.c**

```
CPP
void binder_loop(struct binder_state *bs, binder_handler func)
{
```

```
    int res;
    struct binder_write_read bwr;
    uint32_t readbuf[32];

    bwr.write_size = 0;
    bwr.write_consumed = 0;
    bwr.write_buffer = 0;

    readbuf[0] = BC_ENTER_LOOPER;
    binder_write(bs, readbuf, sizeof(uint32_t));//1

    for (;;) {
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (uintptr_t) readbuf;

        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);//2

        if (res < 0) {
            ALOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
            break;
        }

        res = binder_parse(bs, 0, (uintptr_t) readbuf, bwr.read_consumed, func);//3
        if (res == 0) {
            ALOGE("binder_loop: unexpected reply?!\n");
            break;
        }
        if (res < 0) {
            ALOGE("binder_loop: io error %d %s\n", res, strerror(errno));
            break;
        }
    }
}
```

注释1处将BC_ENTER_LOOPER指令通过binder_write函数写入到Binder驱动中，这样当前线程（ServiceManager的主线程）就成为了一个Binder线程，这样就可以处理进程间的请求了。 在无限循环中不断的调用注释2处的ioctl函数，它不断的使用BINDER_WRITE_READ指令查询Binder驱动中是否有新的请求，如果有就交给注释3处的binder_parse函数处理。如果没有，当前线程就会在Binder驱动中睡眠，等待新的进程间请求。

由于binder_write函数的调用链中涉及到了内核空间和用户空间的交互，因此这里着重讲解下。

frameworks/native/cmds/servicemanager/binder.c

```
CPP
int binder_write(struct binder_state *bs, void *data, size_t len)
{
    struct binder_write_read bwr;//1
    int res;

    bwr.write_size = len;
    bwr.write_consumed = 0;
    bwr.write_buffer = (uintptr_t) data;//2
```

```
    bwr.read_size = 0;
    bwr.read_consumed = 0;
    bwr.read_buffer = 0;
    res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);//3
    if (res < 0) {
        fprintf(stderr,"binder_write: ioctl failed (%s)\n",
                strerror(errno));
    }
    return res;
}
```

注释1处定义binder_write_read结构体，接下来的代码对bwr进行赋值，其中需要注意的是，注释2处的data的值为BC_ENTER_LOOPER。注释3处的ioctl函数将会bwr中的数据发送给binder驱动，我们已经知道了ioctl函数在Kernel Binder中对应的函数为binder_ioctl，此前分析过这个函数，这里截取BINDER_WRITE_READ命令处理部分。

kernel/goldfish/drivers/staging/android/binder.c

```
CPP
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    ...
    void __user *ubuf = (void __user *)arg;
    ...
    switch (cmd) {
    case BINDER_WRITE_READ: {
        struct binder_write_read bwr;
        if (size != sizeof(struct binder_write_read)) {
            ret = -EINVAL;
            goto err;
        }
        if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {//1
            ret = -EFAULT;
            goto err;
        }
        binder_debug(BINDER_DEBUG_READ_WRITE,
                "binder: %d:%d write %ld at %08lx, read %ld at %08lx\n",
                proc->pid, thread->pid, bwr.write_size, bwr.write_buffer,
                bwr.read_size, bwr.read_buffer);

        if (bwr.write_size > 0) {//2
            ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer,
bwr.write_size, &bwr.write_consumed);//3
            trace_binder_write_done(ret);
            if (ret < 0) {
                bwr.read_consumed = 0;
                if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                    ret = -EFAULT;
                goto err;
            }
        }
        ...
        binder_debug(BINDER_DEBUG_READ_WRITE,
                "binder: %d:%d wrote %ld of %ld, read return %ld of %ld\n",
```

```
                proc->pid, thread->pid, bwr.write_consumed, bwr.write_size,
                bwr.read_consumed, bwr.read_size);
        if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {//4
            ret = -EFAULT;
            goto err;
        }
        break;
    }
   ...
    return ret;
}
```

注释1处的copy_from_user函数，在本系列的第一篇文章Android Binder原理（一）学习Binder前必须要了解的知识点提过。在这里，它用于将把用户空间数据ubuf拷贝出来保存到内核数据bwr（binder_write_read结构体）中。 注释2处，bwr的输入缓存区有数据时，会调用注释3处的binder_thread_write函数来处理BC_ENTER_LOOPER协议，其内部会将目标线程的状态设置为BINDER_LOOPER_STATE_ENTERED，这样目标线程就是一个Binder线程。 注释4处通过copy_to_user函数将内核空间数据bwr拷贝到用户空间。

2.总结

ServiceManager的启动过程实际上就是分析ServiceManager的入口函数，在入口函数中主要做了三件事，本篇文章深入到内核源码来对这三件逐一进行分析，由于涉及的函数比较多，这篇文章只介绍了我们需要掌握的，剩余大家可以自行阅读源码，比如binder_thread_write、copy_to_user函数。

Android Binder原理（五）系统服务的获取过程

前言

在本系列的此前文章中，以MediaPlayerService为例，讲解了系统服务是如何注册的（addService），既然有注册那肯定也要有获取，本篇文章仍旧以MediaPlayerService为例，来讲解系统服务的获取过程（getService）。文章会分为两个部分进行讲解，分别是客户端MediaPlayerService请求获取服务和服务端ServiceManager处理请求，先来学习第一部分。

1.客户端MediaPlayerService请求获取服务

要想获取MediaPlayerService，需要先调用getMediaPlayerService函数，如下所示。
frameworks/av/media/libmedia/IMediaDeathNotifier.cpp

```
CPP
IMediaDeathNotifier::getMediaPlayerService()
{
    ALOGV("getMediaPlayerService");
    Mutex::Autolock _l(sServiceLock);
    if (sMediaPlayerService == 0) {
        sp<IServiceManager> sm = defaultServiceManager();//1
        sp<IBinder> binder;
        do {
            binder = sm->getService(String16("media.player"));//2
            if (binder != 0) {//3
                break;
```

```
            }
            ALOGW("Media player service not published, waiting...");
            usleep(500000); //4
        } while (true);

        if (sDeathNotifier == NULL) {
            sDeathNotifier = new DeathNotifier();
        }
        binder->linkToDeath(sDeathNotifier);
        sMediaPlayerService = interface_cast<IMediaPlayerService>(binder);//5
    }
    ALOGE_IF(sMediaPlayerService == 0, "no media player service!?");
    return sMediaPlayerService;
}
```

注释1处的defaultServiceManager返回的是BpServiceManager，注释2处获取名为"media.player"的系统服务（MediaPlayerService），返回的值为BpBinder。由于这个时候MediaPlayerService可能还没有向ServiceManager注册，那么就不能满足注释3的条件，在注释4处休眠0.5s后继续调用getService函数，直到获取服务对应的为止。 注释5处的interface_cast函数用于将BpBinder转换成BpMediaPlayerService，其原理就是通过BpBinder的handle来找到对应的服务，即BpMediaPlayerService。

注释2处的获取服务是本文的重点，BpServiceManager的getService函数如下所示。
frameworks/native/libs/binder/IServiceManager.cpp::BpServiceManager

```
CPP
virtual sp<IBinder> getService(const String16& name) const
   {
      ...
       int n = 0;
       while (uptimeMillis() < timeout) {
           n++;
           if (isVendorService) {
               ALOGI("Waiting for vendor service %s...", String8(name).string());
               CallStack stack(LOG_TAG);
           } else if (n%10 == 0) {
               ALOGI("Waiting for service %s...", String8(name).string());
           }
           usleep(1000*sleepTime);

           sp<IBinder> svc = checkService(name);//1
           if (svc != NULL) return svc;
       }
       ALOGW("Service %s didn't start. Returning NULL", String8(name).string());
       return NULL;
   }
```

getService函数中主要做的事就是循环的查询服务是否存在，如果不存在就继续查询，查询服务用到了注释1处的checkService函数，代码如下所示。
frameworks/native/libs/binder/IServiceManager.cpp::BpServiceManager

```
CPP
virtual sp<IBinder> checkService( const String16& name) const
{
    Parcel data, reply;//1
    data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
    data.writeString16(name);//2
    remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);//3
    return reply.readStrongBinder();
}
```

注释1处的data，看过上一篇文章的同学应该很熟悉，它出现在BpServiceManager的addService函数中，data是一个数据包，后面会不断的将数据写入到data中。注释2处将字符串”media.player”写入到data中。 注释3处的remote()指的是mRemote，也就是BpBinder，BpBinder的transact函数如下所示。

frameworks/native/libs/binder/BpBinder.cpp

```
CPP
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}
```

BpBinder将逻辑处理交给IPCThreadState，后面的调用链在=Android Binder原理（三）系统服务的注册过程中讲过，这里再次简单的过一遍，IPCThreadState::self()会创建创建IPCThreadState，IPCThreadState的transact函数如下所示。 **frameworks/native/libs/binder/IPCThreadState.cpp**

```
CPP
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    status_t err;

    flags |= TF_ACCEPT_FDS;
    ...
    err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);//1

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
```

```
       ...
        if (reply) {
            err = waitForResponse(reply);//2
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
       ...
    } else {
       //不需要等待reply的分支
        err = waitForResponse(NULL, NULL);
    }

    return err;
}
```

调用BpBinder的transact函数实际上就是调用IPCThreadState的transact函数。注释1处的writeTransactionData函数用于传输数据，其中第一个参数BC_TRANSACTION代表向Binder驱动发送命令协议。 注释1处的writeTransactionData用于准备发送的数据，其内部会将BC_TRANSACTION和binder_transaction_data结构体写入到mOut中。 接着查看waitForResponse函数做了什么，代码如下所示。

frameworks/native/libs/binder/IPCThreadState.cpp

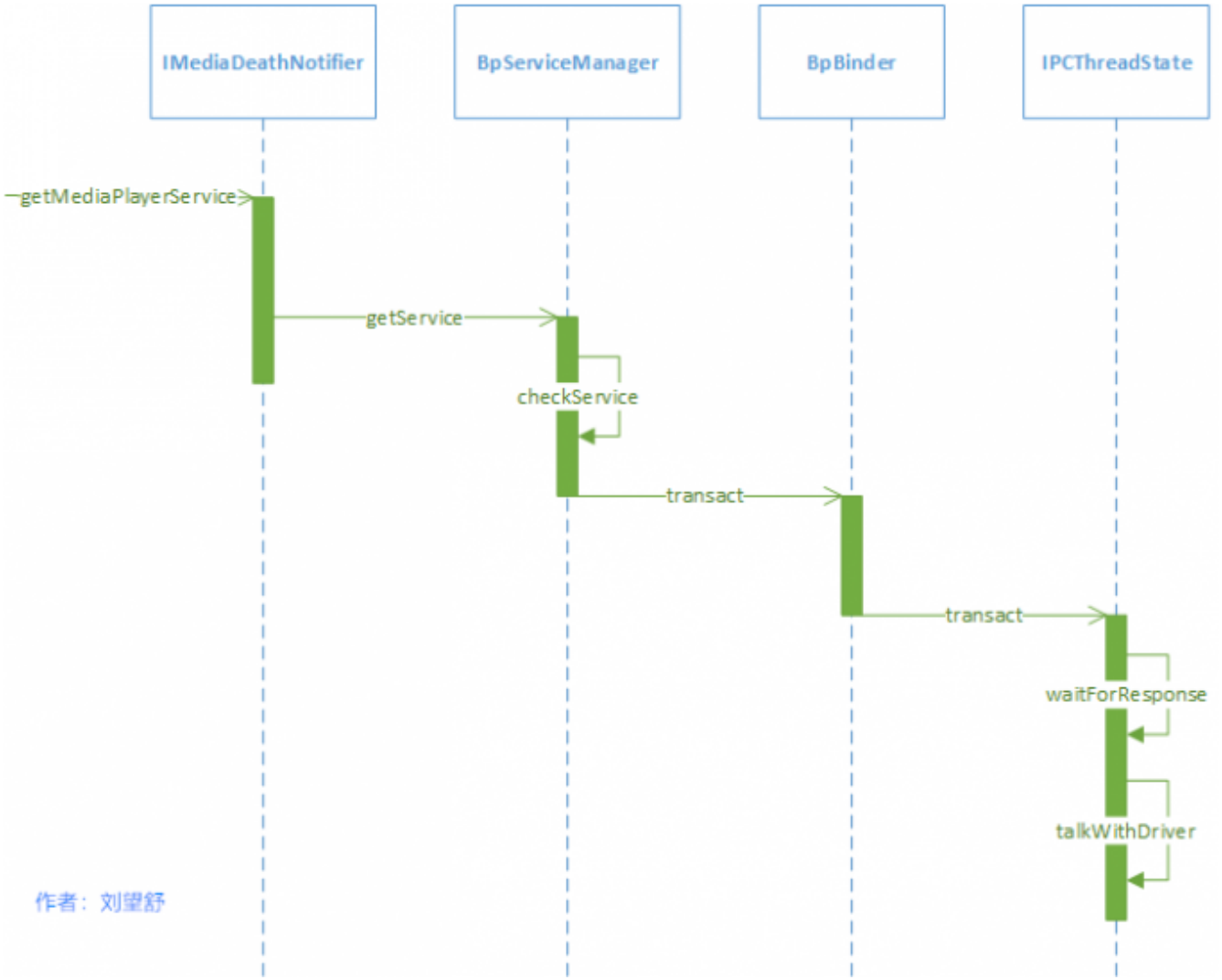
```
CPP
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    uint32_t cmd;
    int32_t err;
    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;//1
        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;
        cmd = (uint32_t)mIn.readInt32();
        IF_LOG_COMMANDS() {
            alog << "Processing waitForResponse Command: "
                << getReturnString(cmd) << endl;
        }
        switch (cmd) {
        case BR_TRANSACTION_COMPLETE:
            if (!reply && !acquireResult) goto finish;
            break;
       ...
        default:
            //处理各种命令协议
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
            break;
        }
}
finish:
    ...
```

```
    return err;
}
```

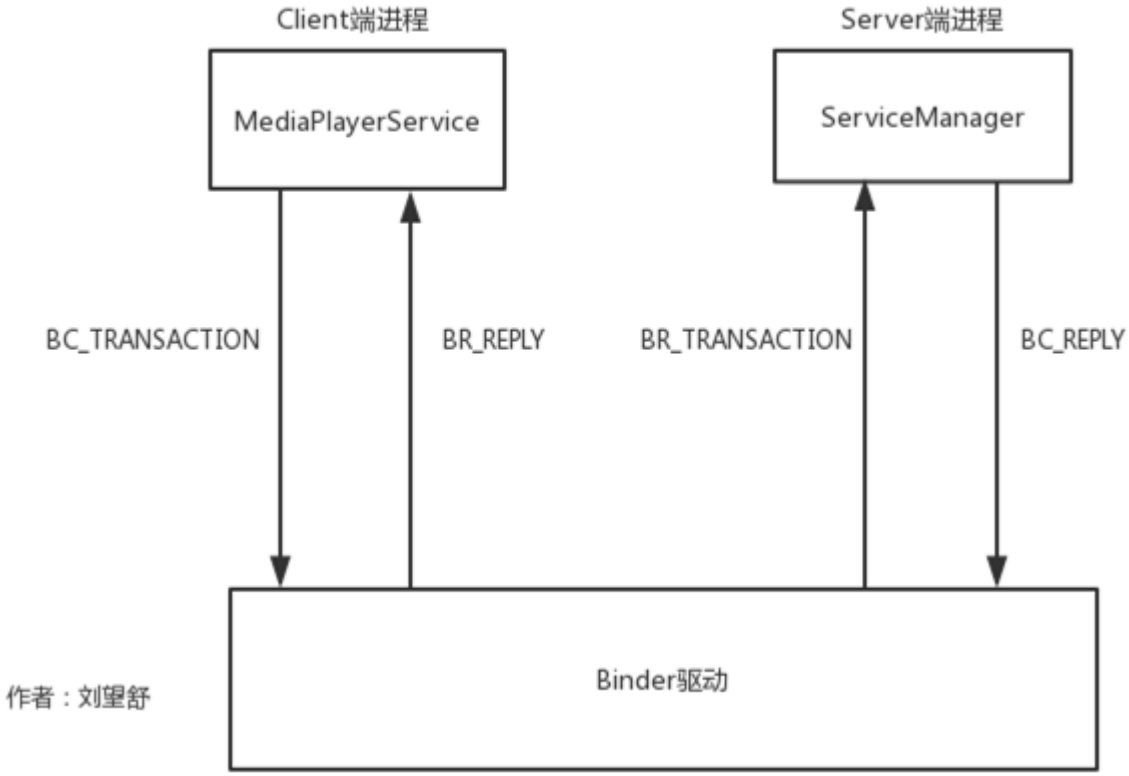
注释1处的talkWithDriver函数的内部通过ioctl与Binder驱动进行通信，代码如下所示。
frameworks/native/libs/binder/IPCThreadState.cpp

```
CPP
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    if (mProcess->mDriverFD <= 0) {
        return -EBADF;
    }
    //和Binder驱动通信的结构体
    binder_write_read bwr; //1
    //mIn是否有可读的数据，接收的数据存储在mIn
    const bool needRead = mIn.dataPosition() >= mIn.dataSize();
    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;
    bwr.write_size = outAvail;
    bwr.write_buffer = (uintptr_t)mOut.data();//2
    //这时doReceive的值为true
    if (doReceive && needRead) {
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (uintptr_t)mIn.data();//3
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }
   ...
    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;
    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
    status_t err;
    do {
        IF_LOG_COMMANDS() {
            alog << "About to read/write, write size = " << mOut.dataSize() << endl;
        }
#if defined(__ANDROID__)
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)//4
            err = NO_ERROR;
        else
            err = -errno;
#else
        err = INVALID_OPERATION;
#endif
     ...
    } while (err == -EINTR);
    ...
    return err;
}
```

注释1处的 binder_write_read是和Binder驱动通信的结构体，在注释2和3处将mOut、mIn赋值给binder_write_read的相应字段，最终通过注释4处的ioctl函数和Binder驱动进行通信。这一过程的时序图如下所示。



这时我们需要再次查看Android Binder原理（三）系统服务的注册过程这篇文章第2小节给出的图。



从这张简化的流程图可以看出，我们当前分析的是客户端进程的流程，当MediaPlayerService向Binder驱动发送BC_TRANSACTION命令后，Binder驱动会向ServiceManager发送BR_TRANSACTION命令，接下来我们来查看服务端ServiceManager是如何处理获取服务这一请求的。

2.服务端ServiceManager处理请求

说到服务端ServiceManager处理请求，不得不说到ServiceManager的启动过程，具体的请看Android Binder原理（四）ServiceManager的启动过程 这篇文章。 这里简单回顾servicemanager的入口函数，如下所示。

frameworks/native/cmds/servicemanager/service_manager.c

```
CPP
int main(int argc, char** argv)
{
    ...
    bs = binder_open(driver, 128*1024);
    ...
    if (binder_become_context_manager(bs)) {
        ALOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }
    ...
    if (getcon(&service_manager_context) != 0) {
        ALOGE("SELinux: Failed to acquire service_manager context. Aborting.\n");
        abort();
    }
    binder_loop(bs, svcmgr_handler);//1
```

```
    return 0;
}
```

main函数主要做了三件事，其中最后一件事就是调用binder_loop函数，这里需要注意，它的第二个参数为svcmgr_handler，后面会再次提到svcmgr_handler。 binder_loop函数如下所示。
frameworks/native/cmds/servicemanager/binder.c

```
CPP
void binder_loop(struct binder_state *bs, binder_handler func)
{
...
    for (;;) {
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (uintptr_t) readbuf;

        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);

        if (res < 0) {
            ALOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
            break;
        }
        res = binder_parse(bs, 0, (uintptr_t) readbuf, bwr.read_consumed, func);
        if (res == 0) {
            ALOGE("binder_loop: unexpected reply?!\n");
            break;
        }
        if (res < 0) {
            ALOGE("binder_loop: io error %d %s\n", res, strerror(errno));
            break;
        }
    }
}
```

在无限循环中不断的调用ioctl函数，它不断的使用BINDER_WRITE_READ指令查询Binder驱动中是否有新的请求，如果有就交给binder_parse函数处理。如果没有，当前线程就会在Binder驱动中睡眠，等待新的进程间通信请求。binder_parse函数如下所示。 **frameworks/native/cmds/servicemanager/binder.c**

```
CPP
int binder_parse(struct binder_state *bs, struct binder_io *bio,
                 uintptr_t ptr, size_t size, binder_handler func)
{
    int r = 1;
    uintptr_t end = ptr + (uintptr_t) size;

    while (ptr < end) {
        uint32_t cmd = *(uint32_t *) ptr;
        ptr += sizeof(uint32_t);
#if TRACE
        fprintf(stderr,"%s:\n", cmd_name(cmd));
#endif
        switch(cmd) {
```

```
        ...
        case BR_TRANSACTION: {
            struct binder_transaction_data *txn = (struct binder_transaction_data *) ptr;
            if ((end - ptr) < sizeof(*txn)) {
                ALOGE("parse: txn too small!\n");
                return -1;
            }
            binder_dump_txn(txn);
            if (func) {
                unsigned rdata[256/4];
                struct binder_io msg;
                struct binder_io reply;
                int res;

                bio_init(&reply, rdata, sizeof(rdata), 4);
                bio_init_from_txn(&msg, txn);
                res = func(bs, txn, &msg, &reply);//1
                if (txn->flags & TF_ONE_WAY) {
                    binder_free_buffer(bs, txn->data.ptr.buffer);
                } else {
                    binder_send_reply(bs, &reply, txn->data.ptr.buffer, res);
                }
            }
            ptr += sizeof(*txn);
            break;
        }
        ...
    }

    return r;
}
```

这里截取了BR_TRANSACTION命令的处理部分，注释1出的func通过一路传递指向的是svcmgr_handler，svcmgr_handler函数如下所示。 **frameworks/native/cmds/servicemanager/service_manager.c**

```
CPP
int svcmgr_handler(struct binder_state *bs,
                   struct binder_transaction_data *txn,
                   struct binder_io *msg,
                   struct binder_io *reply)
{
    ...
    switch(txn->code) {
    case SVC_MGR_GET_SERVICE:
    case SVC_MGR_CHECK_SERVICE:
        s = bio_get_string16(msg, &len);
        if (s == NULL) {
            return -1;
        }
        handle = do_find_service(s, len, txn->sender_euid, txn->sender_pid);
        if (!handle)
            break;
        bio_put_ref(reply, handle);
```

```
        return 0;

   ...
    default:
        ALOGE("unknown code %d\n", txn->code);
        return -1;
    }

    bio_put_uint32(reply, 0);
    return 0;
}
```

当要获取服务时，会调用do_find_service函数，代码如下所示。
frameworks/native/cmds/servicemanager/service_manager.c

```
CPP
uint32_t do_find_service(const uint16_t *s, size_t len, uid_t uid, pid_t spid)
{
    struct svcinfo *si = find_svc(s, len);//1

    if (!si || !si->handle) {
        return 0;
    }

    if (!si->allow_isolated) {
        uid_t appid = uid % AID_USER;
        if (appid >= AID_ISOLATED_START && appid <= AID_ISOLATED_END) {
            return 0;
        }
    }
    if (!svc_can_find(s, len, spid, uid)) {
        return 0;
    }

    return si->handle;
}
```

注释1处的find_svc函数用于查询服务，返回的svcinfo是一个结构体，其内部包含了服务的handle值，最终会返回服务的handle值。接着来看find_svc函数： **frameworks/native/cmds/servicemanager/service_manager.c**

```
CPP
struct svcinfo *find_svc(const uint16_t *s16, size_t len)
{
    struct svcinfo *si;

    for (si = svclist; si; si = si->next) {
        if ((len == si->len) &&
            !memcmp(s16, si->name, len * sizeof(uint16_t))) {
            return si;
        }
    }
    return NULL;
}
```

系统服务的注册流程中，在Kernel Binder中会调用do_add_service函数，其内部会将包含服务名和handle值的svcinfo保存到svclist列表中。同样的，在获取服务的流程中，find_svc函数中会遍历svclist列表，根据服务名查找对应服务是否已经注册，如果已经注册就会返回对应的svcinfo，如果没有注册就返回NULL。

总结

这篇文章将系统服务的获取过程分为两个部分，代码涉及到了Native Binder和Kernel Binder。在下一篇文章中会继续学习Java Binder相关的内容。

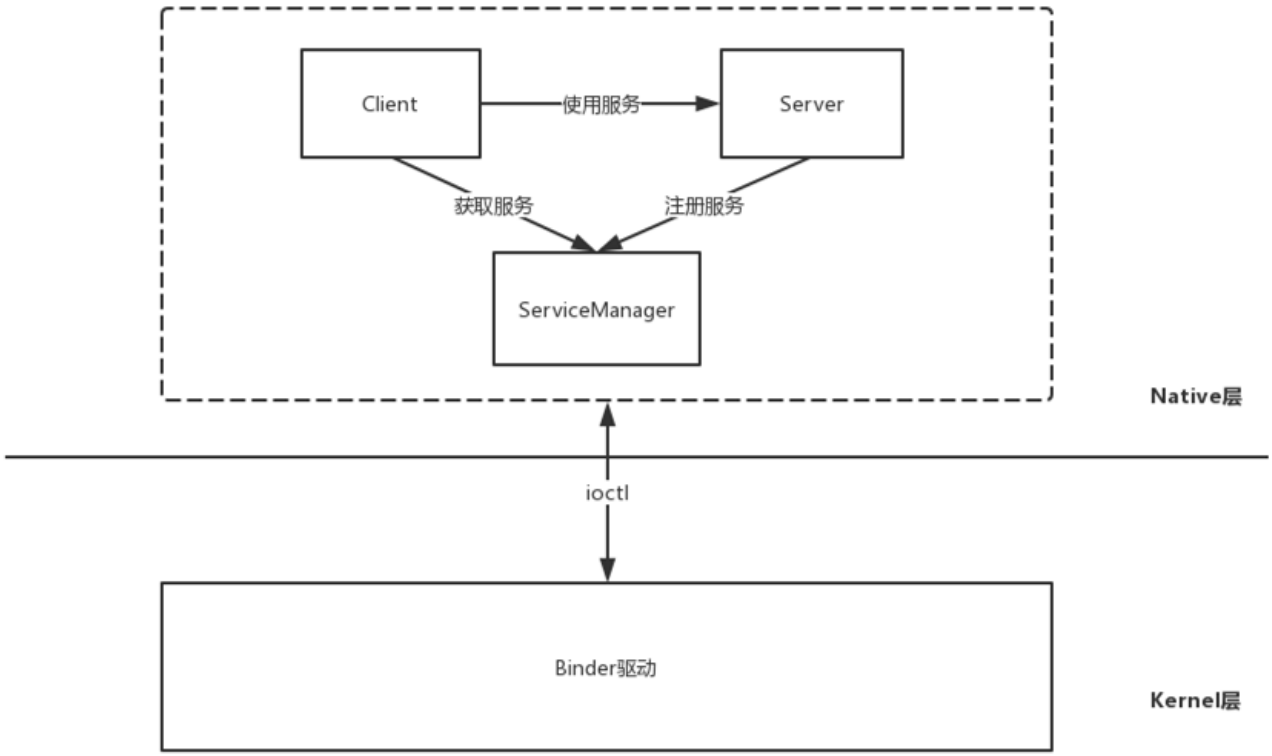
Android Binder原理（六）Java Binder的初始化

前言

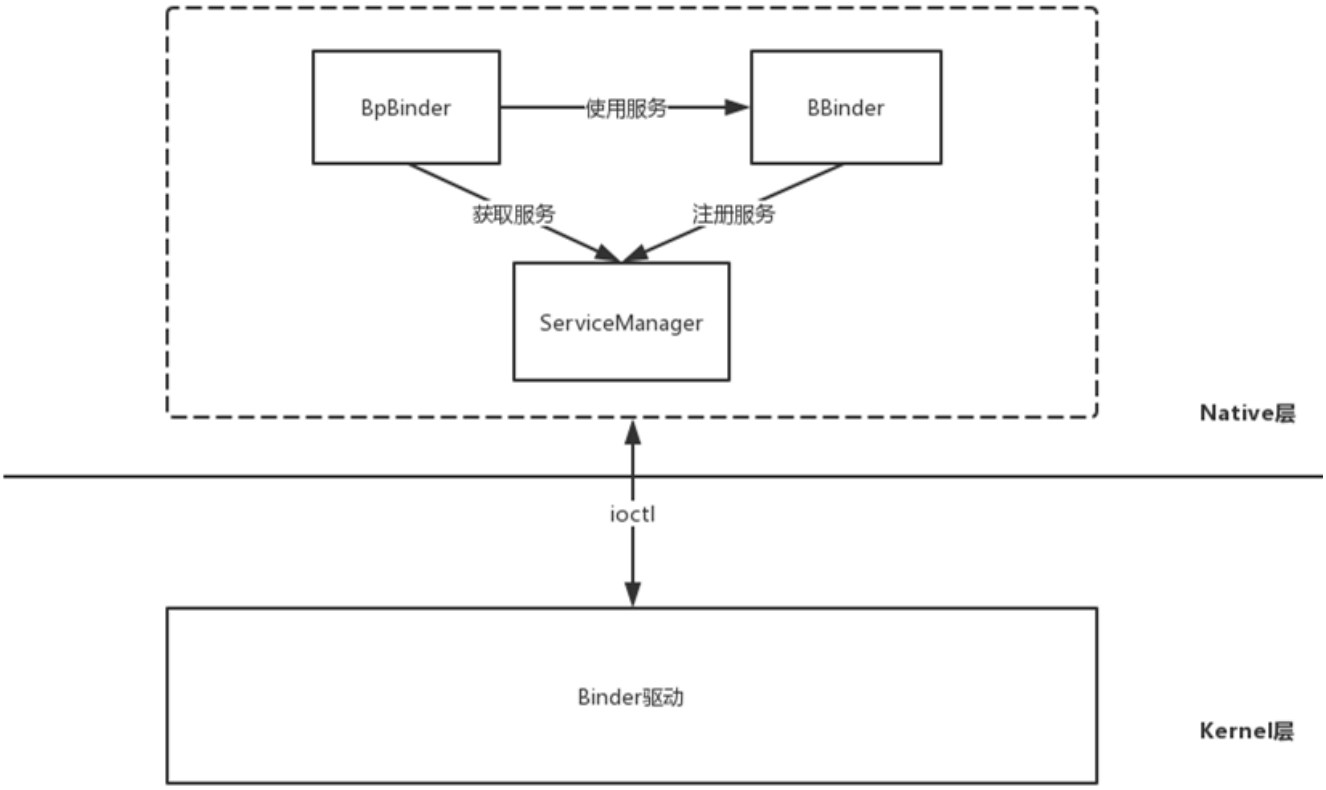
在Android Binder原理（一）学习Binder前必须要了解的知识点这篇文章中，我根据Android系统的分层，将Binder机制分为了三层：

1. Java Binder (对应Framework层的Binder)
2. Native Binder(对应Native层的Binder)
3. Kernel Binder(对应Kernel层的Binder)

在此前的文章中，我一直都在介绍Native Binder和Kernel Binder的内容，它们的架构简单总结为下图。



在Android Binder原理（二）ServiceManager中的Binder机制这篇文章中，我讲过BpBinder是Client端与Server交互的代理类，而BBinder则代表了Server端，那么上图就可以改为：



从上图可以看到，Native Binder实际是基于C/S架构，Bpinder是Client端，BBinder是Server端，在Android Binder原理（四）ServiceManager的启动过程这篇文章中，我们得知Native Binder通过ioctl函数和Binder驱动进行数据交互。 Java Binder是需要借助Native Binder来进行工作的，因此Java Binder在设计上也是一个C/S架构，可以说Java

Binder是Native Binder的一个镜像，所以在学习Java Binder前，最好先要学习此前文章讲解的Native Binder的内容。本篇文章先来讲解Java Binder是如何初始化的，即Java Binder的JNI注册。

1.Java Binder的JNI注册

Java Binder要想和Native Binder进行通信，需要通过JNI，JNI的注册是在Zygote进程启动过程中注册的，代码如下所示。 **frameworks/base/core/jni/AndroidRuntime.cpp**

```
CPP
void AndroidRuntime::start(const char* className, const Vector<String8>& options, bool
zygote)
{
    ...
    JniInvocation jni_invocation;
    jni_invocation.Init(NULL);
    JNIEnv* env;
    if (startVm(&mJavaVM, &env, zygote) != 0) {//1
        return;
    }
    onVmCreated(env);
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }
   ...
}
```

注释1处用于启动Java虚拟机，注释2处startReg函数用于完成虚拟机的JNI注册，关于AndroidRuntime的start函数的具体分析见Android系统启动流程（二）解析Zygote进程启动过程这篇文章。 startReg函数如下所示。
frameworks/base/core/jni/AndroidRuntime.cpp

```
CPP
/*static*/ int AndroidRuntime::startReg(JNIEnv* env)
{
    ATRACE_NAME("RegisterAndroidNatives");
    androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThreadEtc);

    ALOGV("--- registering native functions ---\n");
    env->PushLocalFrame(200);

    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {//1
        env->PopLocalFrame(NULL);
        return -1;
    }
    env->PopLocalFrame(NULL);
    return 0;
}
```

注释1处的register_jni_procs函数的作用就是循环调用gRegJNI数组的成员所对应的方法，如下所示。

```
CPP
static int register_jni_procs(const RegJNIRec array[], size_t count, JNIEnv* env)
{
    for (size_t i = 0; i < count; i++) {
        if (array[i].mProc(env) < 0) {
#ifndef NDEBUG
            ALOGD("----------!!! %s failed to load\n", array[i].mName);
#endif
            return -1;
        }
    }
    return 0;
}
```

gRegJNI数组中有100多个成员变量：

```
CPP
static const RegJNIRec gRegJNI[] = {
    REG_JNI(register_com_android_internal_os_RuntimeInit),
    REG_JNI(register_com_android_internal_os_ZygoteInit_nativeZygoteInit),
    REG_JNI(register_android_os_SystemClock),
    ...
    REG_JNI(register_android_os_Binder),//1
   ...
};
```

其中REG_JNI是一个宏定义：

```
CODE
#define REG_JNI(name)      { name }
struct RegJNIRec {
    int (*mProc)(JNIEnv*);
};
```

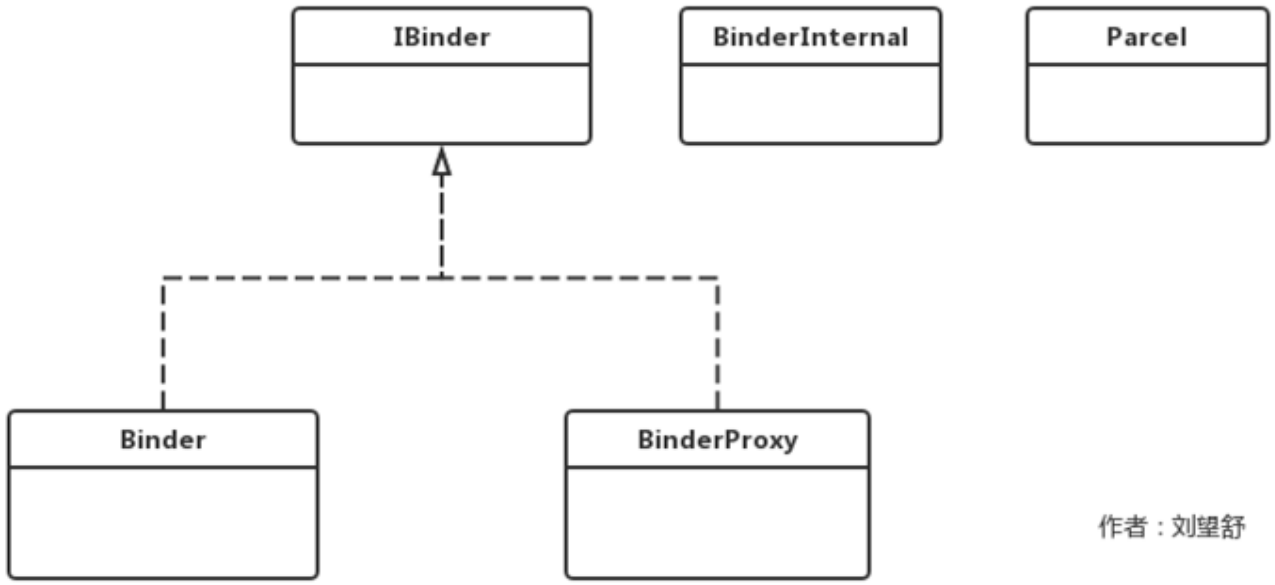
实际上就是调用参数名所对应的函数。负责Java Binder和Native Binder通信的函数为注释1处的register_android_os_Binder，代码如下所示。 **frameworks/base/core/jni/android_util_Binder.cpp**

```
CPP
int register_android_os_Binder(JNIEnv* env)
{
    //注册Binder类
    if (int_register_android_os_Binder(env) < 0)
        return -1;
    //注册BinderInternal类
    if (int_register_android_os_BinderInternal(env) < 0)
        return -1;
    //注册BinderProxy类
    if (int_register_android_os_BinderProxy(env) < 0)
        return -1;
    ...
```

```
    return 0;
}
```

register_android_os_Binder函数做了三件事，分别是: 1.注册Binder类 2.注册BinderInternal类 3.注册BinderProxy类

它们是Java Binder关联类的一小部分，它们的关系如下图所示。



- IBinder接口中定义了很多整型的变量，其中定义一个叫做`FLAG_ONEWAY`的整形变量。客户端发起调用时，客户端一般会阻塞，直到服务端返回结果。设置`FLAG_ONEWAY`后，客户端只需要把请求发送到服务端就可以立即返回，而不需要等待服务端的结果，这是一种非阻塞方式。
- Binder和BinderProxy实现了IBinder接口，Binder是服务端的代表，而BinderProxy是客户端的代表。
- BinderInternal只是在Binder框架中被使用，其内部类GcWatcher用于处理和Binder的垃圾回收。
- Parcel是一个数据包装器，它可以在进程间进行传递，Parcel既可以传递基本数据类型也可以传递Binder对象，Binder通信就是通过Parcel来进行客户端与服务端数据交互。Parcel的实现既有Java部分，也有Native部分，具体实现在Native部分中。

下面分别对Binder、BinderInternal这两个类的注册进行分析。

1.1 Binder类的注册

调用int_register_android_os_Binder函数来完成Binder类的注册，代码如下所示。
frameworks/base/core/jni/android_util_Binder.cpp

```
CPP
static const JNINativeMethod gBinderMethods[] = {
     /* name, signature, funcPtr */
    { "getCallingPid", "()I", (void*)android_os_Binder_getCallingPid },
    { "getCallingUid", "()I", (void*)android_os_Binder_getCallingUid },
    { "clearCallingIdentity", "()J", (void*)android_os_Binder_clearCallingIdentity },
    { "restoreCallingIdentity", "(J)V", (void*)android_os_Binder_restoreCallingIdentity },
    { "setThreadStrictModePolicy", "(I)V",
(void*)android_os_Binder_setThreadStrictModePolicy },
    { "getThreadStrictModePolicy", "()I", (void*)android_os_Binder_getThreadStrictModePolicy
```

```
},
    { "flushPendingCommands", "()V", (void*)android_os_Binder_flushPendingCommands },
    { "getNativeBBinderHolder", "()J", (void*)android_os_Binder_getNativeBBinderHolder },
    { "getNativeFinalizer", "()J", (void*)android_os_Binder_getNativeFinalizer },
    { "blockUntilThreadAvailable", "()V", (void*)android_os_Binder_blockUntilThreadAvailable
}
};
const char* const kBinderPathName = "android/os/Binder";//1
static int int_register_android_os_Binder(JNIEnv* env)
{
    jclass clazz = FindClassOrDie(env, kBinderPathName);//2

    gBinderOffsets.mClass = MakeGlobalRefOrDie(env, clazz);//3
    gBinderOffsets.mExecTransact = GetMethodIDOrDie(env, clazz, "execTransact", "
(IJJI)Z");//4
    gBinderOffsets.mObject = GetFieldIDOrDie(env, clazz, "mObject", "J");

    return RegisterMethodsOrDie(
        env, kBinderPathName,
        gBinderMethods, NELEM(gBinderMethods));
}
```

注释1处的kBinderPathName的值为"android/os/Binder"，这是Binder在Java Binder中的全路径名。 注释2处根据这个路径名获取Binder的Class对象，并赋值给jclass类型的变量clazz，clazz是Java层Binder在JNI层的代表。 注释3处通过MakeGlobalRefOrDie函数将本地引用clazz转变为全局引用并赋值给gBinderOffsets.mClass。 注释4处用于找到Java层的Binder的成员方法execTransact并赋值给gBinderOffsets.mExecTransact。 注释5处用于找到Java层的Binder的成员变量mObject并赋值给gBinderOffsets.mObject。 最后一行通过RegisterMethodsOrDie函数注册gBinderMethods中定义的函数，其中gBinderMethods是JNINativeMethod类型的数组，里面存储的是Binder的Native方法（Java层）与JNI层函数的对应关系。

gBinderMethods的定义如下所示。

```
CPP
static struct bindernative_offsets_t
{
    jclass mClass;
    jmethodID mExecTransact;
    jfieldID mObject;

} gBinderOffsets;
```

使用gBinderMethods来保存变量和方法有两个原因： 1.为了效率考虑，如果每次调用相关的方法时都需要查询方法和变量，显然效率比较低。 2.这些成员变量和方法都是本地引用，在int int_register_android_os_Binder函数返回时，这些本地引用会被自动释放，因此用gBinderOffsets来保存，以便于后续使用。

对于JNI不大熟悉的同学可以看Android深入理解JNI（二）类型转换、方法签名和JNIEnv这篇文章。

1.2 BinderInternal类的注册

调用int_register_android_os_BinderInternal函数来完成BinderInternal类的注册，代码如下所示。
frameworks/base/core/jni/android_util_Binder.cpp

```
CPP
const char* const kBinderInternalPathName = "com/android/internal/os/BinderInternal";
static int int_register_android_os_BinderInternal(JNIEnv* env)
{
    jclass clazz = FindClassOrDie(env, kBinderInternalPathName);

    gBinderInternalOffsets.mClass = MakeGlobalRefOrDie(env, clazz);
    gBinderInternalOffsets.mForceGc = GetStaticMethodIDOrDie(env, clazz, "forceBinderGc", "
()V");
    gBinderInternalOffsets.mProxyLimitCallback = GetStaticMethodIDOrDie(env, clazz,
"binderProxyLimitCallbackFromNative", "(I)V");

    jclass SparseIntArrayClass = FindClassOrDie(env, "android/util/SparseIntArray");
    gSparseIntArrayOffsets.classObject = MakeGlobalRefOrDie(env, SparseIntArrayClass);
    gSparseIntArrayOffsets.constructor = GetMethodIDOrDie(env,
gSparseIntArrayOffsets.classObject,
                                                          "<init>", "()V");
    gSparseIntArrayOffsets.put = GetMethodIDOrDie(env, gSparseIntArrayOffsets.classObject,
"put",
                                                  "(II)V");

    BpBinder::setLimitCallback(android_os_BinderInternal_proxyLimitcallback);

    return RegisterMethodsOrDie(
        env, kBinderInternalPathName,
        gBinderInternalMethods, NELEM(gBinderInternalMethods));
}
```

和int_register_android_os_Binder函数的实现类似，主要做了三件事： 1.获取BinderInternal在JNI层的代表clazz。 2.将BinderInternal类中有用的成员变量和方法存储到gBinderInternalOffsets中。 3.注册BinderInternal类的Native方法对应的JNI函数。

还有一个BinderProxy类的注册，它和Binder、BinderInternal的注册过程差不多，这里就不再赘述了，有兴趣的读者可以自行去看源码。

Android Binder原理（七）Java Binder中系统服务的注册过程

前言

在Android Binder原理（三）系统服务的注册过程这篇文章中，我介绍的是Native Binder中的系统服务的注册过程，这一过程的核心是ServiceManager，而在Java Binder中，也有一个ServiceManager，只不过这个ServiceManager是Java文件。 既然要将系统服务注册到ServiceManager，那么需要选择一个系统服务为例，这里以常见的AMS为例。

1.将AMS注册到ServiceManager

在AMS的setSystemProcess方法中，会调用ServiceManager的addService方法，如下所示。
frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
CPP
public void setSystemProcess() {
    try {
        ServiceManager.addService(Context.ACTIVITY_SERVICE, this, /* allowIsolated= */ true,
                DUMP_FLAG_PRIORITY_CRITICAL | DUMP_FLAG_PRIORITY_NORMAL |
DUMP_FLAG_PROTO);//1
       ....
    } catch (PackageManager.NameNotFoundException e) {
        throw new RuntimeException(
                "Unable to find android system package", e);
    }
 ...
}
```

注释1处的`Context.ACTIVITY_SERVICE`的值为"activity"，作用就是将AMS注册到ServiceManager中。接着来看ServiceManager的addService方法。 **frameworks/base/core/java/android/os/ServiceManager.java**

```
CPP
public static void addService(String name, IBinder service, boolean allowIsolated,
        int dumpPriority) {
    try {
        getIServiceManager().addService(name, service, allowIsolated, dumpPriority);
    } catch (RemoteException e) {
        Log.e(TAG, "error in addService", e);
    }
}
```

主要分析getIServiceManager方法返回的是什么，代码如下所示。
frameworks/base/core/java/android/os/ServiceManager.java

```
CPP
private static IServiceManager getIServiceManager() {
     if (sServiceManager != null) {
         return sServiceManager;
     }
     sServiceManager = ServiceManagerNative
             .asInterface(Binder.allowBlocking(BinderInternal.getContextObject()));
     return sServiceManager;
 }
```

讲到这里，已经积累了几个点需要分析，分别是：

- BinderInternal.getContextObject()
- ServiceManagerNative.asInterface()
- getIServiceManager().addService()

现在我们来各个击破它们。

1.1 BinderInternal.getContextObject()

Binder.allowBlocking的作用是将BinderProxy的sWarnOnBlocking值置为false。主要来分析BinderInternal.getContextObject()做了什么，这个方法是一个Native方法，找到它对应的函数：
frameworks/base/core/jni/android_util_Binder.cpp

```
CPP
static const JNINativeMethod gBinderInternalMethods[] = {
    { "getContextObject", "()Landroid/os/IBinder;",
(void*)android_os_BinderInternal_getContextObject },
   ...
};
```

对应的函数为android_os_BinderInternal_getContextObject：
frameworks/base/core/jni/android_util_Binder.cpp

```
CPP
static jobject android_os_BinderInternal_getContextObject(JNIEnv* env, jobject clazz)
{
    sp<IBinder> b = ProcessState::self()->getContextObject(NULL);//1
    return javaObjectForIBinder(env, b);
}
```

ProcessState::self()的作用是创建ProcessState，注释1处最终返回的是BpBinder，不理解的可以查看Android Binder原理（二）ServiceManager中的Binder机制这篇文章。

BpBinder是Native Binder中的Client端，这说明Java层的ServiceManager需要Native层的BpBinder，但是这个BpBinder在Java层是无法直接使用，那么就需要传入javaObjectForIBinder函数来做处理，其内部会创建一个BinderProxy对象，这样我们得知 BinderInternal.getContextObject()最终得到的是BinderProxy。 在Android Binder原理（六）Java Binder的初始化这篇文章我们讲过，BinderProxy是Java Binder的客户端的代表。 需要注意的一点是，这个传入的BpBinder会保存到BinderProxy的成员变量mObject中，后续会再次提到这个点。

1.2 ServiceManagerNative.asInterface()

说到asInterface方法，在Native Binder中也有一个asInterface函数。在Android Binder原理（二）ServiceManager中的Binder机制这篇文章中讲过IServiceManager的asInterface函数，它的作用是用BpBinder做为参数创建BpServiceManager。那么在Java Binder中的asInterface方法的作用又是什么？
frameworks/base/core/java/android/os/ServiceManagerNative.java

```
CPP
static public IServiceManager asInterface(IBinder obj)
 {
     if (obj == null) {
         return null;
     }
     IServiceManager in =
         (IServiceManager)obj.queryLocalInterface(descriptor);
     if (in != null) {
         return in;
     }

     return new ServiceManagerProxy(obj);
 }
```

根据1.1小节，我们得知obj的值为BinderProxy，那么asInterface方法的作用就是用BinderProxy作为参数创建ServiceManagerProxy。 BinderProxy和BpBinder分别在Jave Binder和Native Binder作为客户端的代表，BpServiceManager通过BpBinder来实现通信，同样的，ServiceManagerProxy也会将业务的请求交给BinderProxy来处理。 分析到这里，那么：

```
CPP
sServiceManager = ServiceManagerNative
        .asInterface(Binder.allowBlocking(BinderInternal.getContextObject()));
```

可以理解为：

```
CPP
    sServiceManager = new ServiceManagerProxy(BinderProxy);
}
```

1.3 getIServiceManager().addService()

根据1.2节的讲解，getIServiceManager()返回的是ServiceManagerProxy，ServiceManagerProxy是ServiceManagerNative的内部类，它实现了IServiceManager接口。

来查看ServiceManagerProxy的addService方法，
frameworks/base/core/java/android/os/ServiceManagerNative.java::ServiceManagerProxy

```
CPP
public void addService(String name, IBinder service, boolean allowIsolated, int
dumpPriority)
        throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IServiceManager.descriptor);
    data.writeString(name);
    data.writeStrongBinder(service);//1
    data.writeInt(allowIsolated ? 1 : 0);
    data.writeInt(dumpPriority);
    mRemote.transact(ADD_SERVICE_TRANSACTION, data, reply, 0);//2
    reply.recycle();
    data.recycle();
}
```

注释1处的data.writeStrongBinder很关键，后续会进行分析。这里又看到了Parcel，它是一个数据包装器，将请求数据写入到Parcel类型的对象data中，通过注释1处的mRemote.transact发送出去，mRemote实际上是BinderProxy，BinderProxy.transact是native函数，实现的函数如下所示。
frameworks/base/core/jni/android_util_Binder.cpp

```
CPP
static jboolean android_os_BinderProxy_transact(JNIEnv* env, jobject obj,
        jint code, jobject dataObj, jobject replyObj, jint flags) // throws RemoteException
{
    if (dataObj == NULL) {
```

```
        jniThrowNullPointerException(env, NULL);
        return JNI_FALSE;
    }
    Parcel* data = parcelForJavaObject(env, dataObj);//1
    if (data == NULL) {
        return JNI_FALSE;
    }
    Parcel* reply = parcelForJavaObject(env, replyObj);//2
    if (reply == NULL && replyObj != NULL) {
        return JNI_FALSE;
    }
    IBinder* target = getBPNativeData(env, obj)->mObject.get();//3
    if (target == NULL) {
        jniThrowException(env, "java/lang/IllegalStateException", "Binder has been
finalized!");
        return JNI_FALSE;
    }
   ...
    status_t err = target->transact(code, *data, reply, flags);//4
    return JNI_FALSE;
}
```

注释1和注释2处，将Java层的Parcel对象转化成为Native层的Parcel对象。在1.1小节中，我们得知BpBinder会保存到BinderProxy的成员变量mObject中，因此在注释3处，从BinderProxy的成员变量mObject中获取BpBinder。最终会在注释4处调用BpBinder的transact函数，向Binder驱动发送数据，可以看出Java Binder是需要Native Binder支持的，最终的目的就是向Binder驱动发送和接收数据。

2.引出JavaBBinder

接着回过头来分析1.3小节遗留下来的data.writeStrongBinder(service)，代码如下所示。
frameworks/base/core/java/android/os/Parcel.java

```
CPP
public final void writeStrongBinder(IBinder ll) {
      nativeWriteStrongBinder(mNativePtr, val);
  }
```

nativeWriteStrongBinder是Native方法，实现的函数为android_os_Parcel_writeStrongBinder：
frameworks/base/core/jni/android_os_Parcel.cpp

```
CPP
static void android_os_Parcel_writeStrongBinder(JNIEnv* env, jclass clazz, jlong nativePtr,
jobject object)
{
    Parcel* parcel = reinterpret_cast<Parcel*>(nativePtr);
    if (parcel != NULL) {
        const status_t err = parcel->writeStrongBinder(ibinderForJavaObject(env,
object));//1
        if (err != NO_ERROR) {
            signalExceptionForError(env, clazz, err);
        }
    }
}
```

接着查看注释1处ibinderForJavaObject函数： **frameworks/base/core/jni/android_util_Binder.cpp**

```
CPP
sp<IBinder> ibinderForJavaObject(JNIEnv* env, jobject obj)
{
    if (obj == NULL) return NULL;
    if (env->IsInstanceOf(obj, gBinderOffsets.mClass)) {//1
        JavaBBinderHolder* jbh = (JavaBBinderHolder*)
            env->GetLongField(obj, gBinderOffsets.mObject);
        return jbh->get(env, obj);//2
    }
    if (env->IsInstanceOf(obj, gBinderProxyOffsets.mClass)) {
        return getBPNativeData(env, obj)->mObject;
    }

    ALOGW("ibinderForJavaObject: %p is not a Binder object", obj);
    return NULL;
}
```

注释2处，如果obj是Java层的BinderProxy类，则返回BpBinder。 注释1处，如果obj是Java层的Binder类，那么先获取JavaBBinderHolder对象，然后在注释2处调用JavaBBinderHolder的get函数，代码如下所示。
frameworks/base/core/jni/android_util_Binder.cpp::JavaBBinderHolder

```
CPP
class JavaBBinderHolder
{
public:
    sp<JavaBBinder> get(JNIEnv* env, jobject obj)
    {
        AutoMutex _l(mLock);
        sp<JavaBBinder> b = mBinder.promote();//1
        if (b == NULL) {
            //obj是一个Java层Binder对象
            b = new JavaBBinder(env, obj);//2
            mBinder = b;
            ALOGV("Creating JavaBinder %p (refs %p) for Object %p, weakCount=%" PRId32 "\n",
                 b.get(), b->getWeakRefs(), obj, b->getWeakRefs()->getWeakCount());
```

```
        }
        return b;
    }
    sp<JavaBBinder> getExisting()
    {
        AutoMutex _l(mLock);
        return mBinder.promote();
    }
private:
    Mutex           mLock;
    wp<JavaBBinder> mBinder;
};
```

成员变量mBinder是`wp<JavaBBinder>`类型的弱引用，在注释1处得到`sp<JavaBBinder>`类型的强引用b，在注释2处创建JavaBBinder并赋值给b。那么，JavaBBinderHolder的get函数返回的是JavaBBinder。

data.writeStrongBinder(service)在本文中等价于：

```
CODE
data.writeStrongBinder(new JavaBBinder(env , Binder))。
```

讲到这里可以得知ServiceManager.addService()传入的并不是AMS本身，而是JavaBBinder。

3.解析JavaBBinder

接着来分析JavaBBinder，查看它的构造函数：
frameworks/base/core/jni/android_util_Binder.cpp::JavaBBinderHolder::JavaBBinder

```
CPP
class JavaBBinder : public BBinder
{
public:
    JavaBBinder(JNIEnv* env, jobject /* Java Binder */ c)
        : mVM(jnienv_to_javavm(env)), mObject(env->NewGlobalRef(object))
    {
        ALOGV("Creating JavaBBinder %p\n", this);
        gNumLocalRefsCreated.fetch_add(1, std::memory_order_relaxed);
        gcIfManyNewRefs(env);
    }
...
```

可以发现JavaBBinder继承了BBinder，那么JavaBBinder的作用是什么呢？当Binder驱动得到客户端的请求，紧接着会将响应发送给JavaBBinder，这时会调用JavaBBinder的onTransact函数，代码如下所示。
frameworks/base/core/jni/android_util_Binder.cpp::JavaBBinderHolder::JavaBBinder

```
CPP
virtual status_t onTransact(
       uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0)
   {
       JNIEnv* env = javavm_to_jnienv(mVM);
       ALOGV("onTransact() on %p calling object %p in env %p vm %p\n", this, mObject, env,
```

```
mVM);
       IPCThreadState* thread_state = IPCThreadState::self();
       const int32_t strict_policy_before = thread_state->getStrictModePolicy();
       jboolean res = env->CallBooleanMethod(mObject, gBinderOffsets.mExecTransact,
           code, reinterpret_cast<jlong>(&data), reinterpret_cast<jlong>(reply), flags);//1

       ...
       return res != JNI_FALSE ? NO_ERROR : UNKNOWN_TRANSACTION;
   }
```

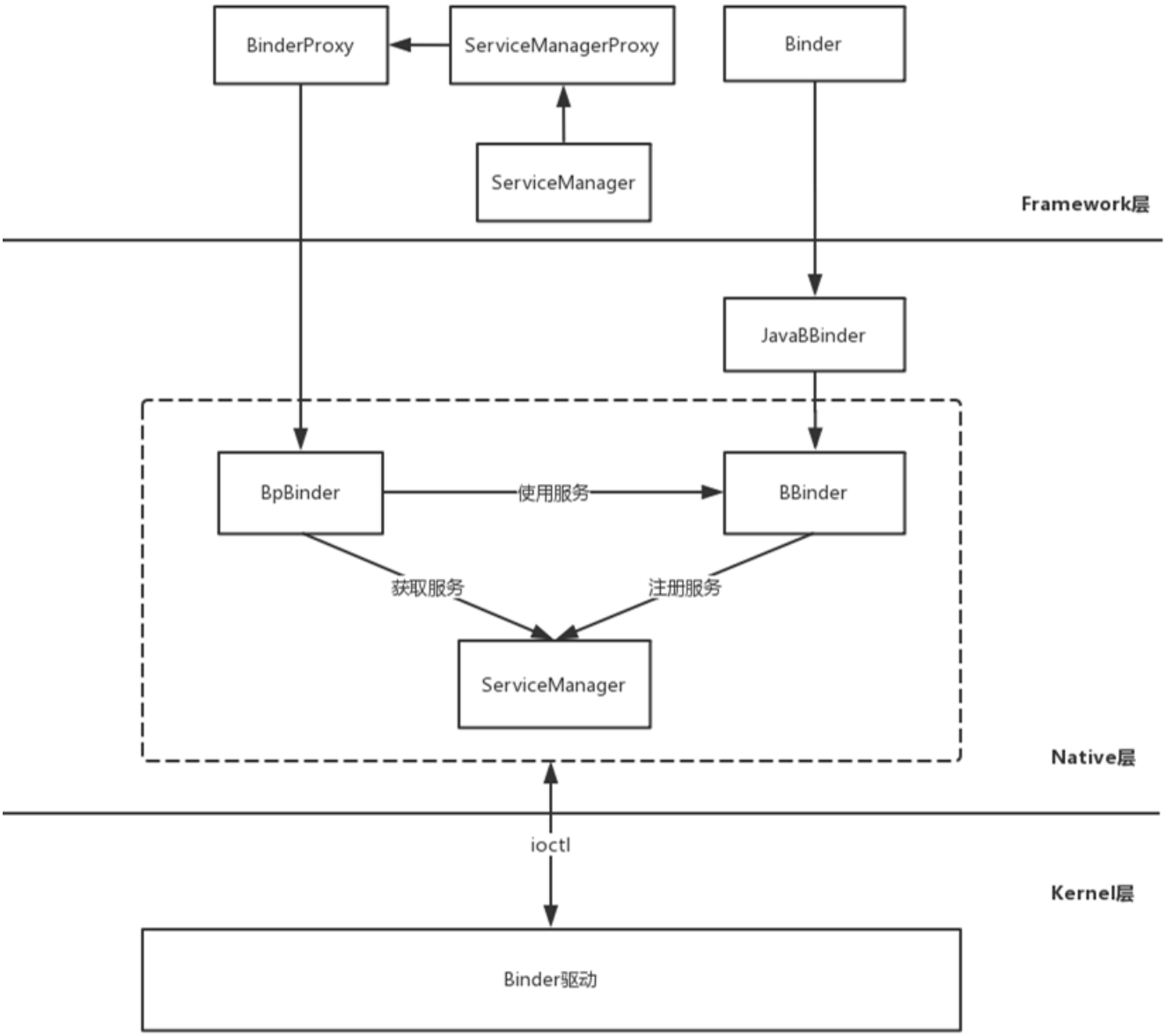
在注释1处会调用Java层Binder的execTransact函数： **frameworks/base/core/java/android/os/Binder.java**

```
CPP
    private boolean execTransact(int code, long dataObj, long replyObj,
            int flags) {
...
        try {
            if (tracingEnabled) {
                Trace.traceBegin(Trace.TRACE_TAG_ALWAYS, getClass().getName() + ":" + code);
            }
            res = onTransact(code, data, reply, flags);//1
        } catch (RemoteException|RuntimeException e) {
           ...
        }
       ...
        return res;
    }
```

关键点是注释1处的onTransact函数，AMS实现了onTransact函数，从而完成业务实现。 从这里可有看出，JavaBBinder并没有实现什么业务，当它接收到请求时，会调用Binder类的execTransact函数，execTransact函数内部又调用了onTransact函数，系统服务会重写onTransact函数来实现自身的业务功能。

4.Java Binder架构

Binder架构如下图所示。



Native Binder的部分在此前的文章已经讲过，这里主要来说说Java Binder部分，从图中可以看到： 1.Binder是服务端的代表，JavaBBinder继承BBinder，JavaBBinder通过mObject变量指向Binder。 2.BinderProxy是客户端的代表，ServiceManager的addService等方法会交由ServiceManagerProxy处理。 3.ServiceManagerProxy的成员变量mRemote指向BinderProxy对象，所以ServiceManagerProxy的addService等方法会交由BinderProxy来处理。 4.BinderProxy的成员变量mObject指向BpBinder对象，因此BinderProxy可以通过BpBinder和Binder驱动发送数据。