Android 应用工程师的 Binder 原理剖析

一：Binder的全面介绍

1.binder的出现

```
Binder`是`Android`中最重要的一种进程间通信机制，基于开源的`OpenBinder
```

George Hoffman当时任**Be**公司的工程师，他启动了一个名为`OpenBinder`的项目，在**Be**公司被**ParmSource**公司收购后，`OpenBinder`由**Dinnie Hackborn**继续开发，后来成为管理`ParmOS6 Cobalt OS`的进程的基础。在**Hackborn**加入谷歌后，他在`OpenBinder`的基础上开发出了`Android Binder`(以下简称`Binder`)，用来完成`Android`的进程通信。

2.为什么需要学习binder

作为一名Android开发，我们每天都在和Binder打交道，虽然可能有的时候不会注意到，譬如：

- startActivity的时候，会获取AMS服务，调用AMS服务的startActivity方法
- startActivity传递的对象为什么需要序列化
- bindService为什么回调的是一个Ibinder对象
- 多进程应用，各个进程之间如何通信
- AIDL的使用
- ...

它们都和`Binder`有着莫切关系，当碰到上面的场景，或者一些疑难问题的时候，理解`Binder`机制是非常有必要的。我们知道 Android 应用程序是由 Activity、Service、Broadcast Receiver 和 Content Provide 四大组件中的一个或者多个组成的。有时这些组件运行在同一进程，有时运行在不同的进程。这些进程间的通信就依赖于 Binder IPC 机制。不仅如此，Android 系统对应用层提供的各种服务如：ActivityManagerService、PackageManagerService 等都是基于 Binder IPC 机制来实现的。Binder 机制在 Android 中的位置非常重要，毫不夸张的说理解 Binder 是迈向 Android 高级工程的第一步。

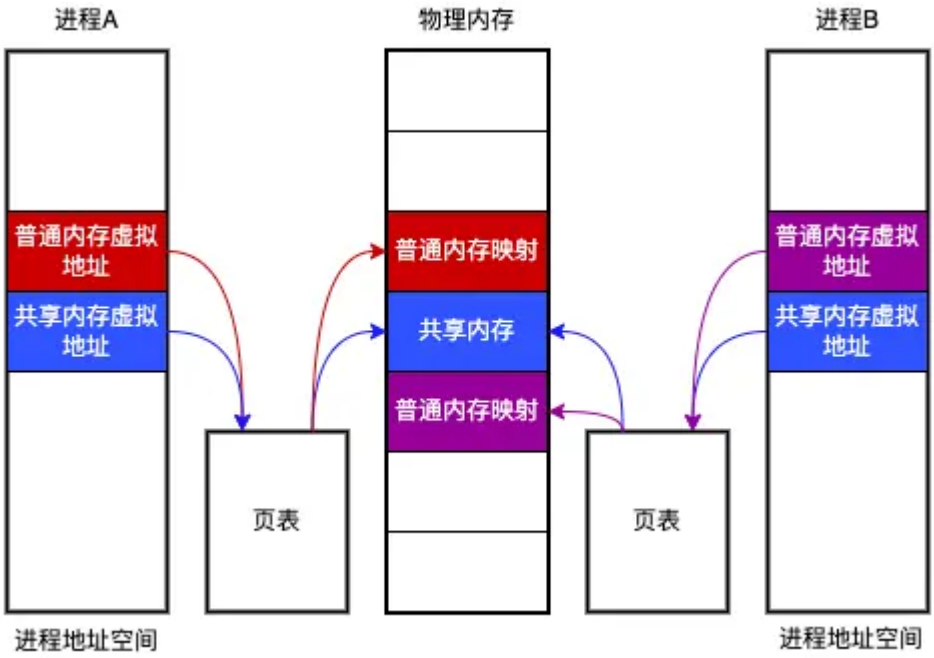
3.为什么Android选择Binder

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

3.1 常见进程间通信

共享内存

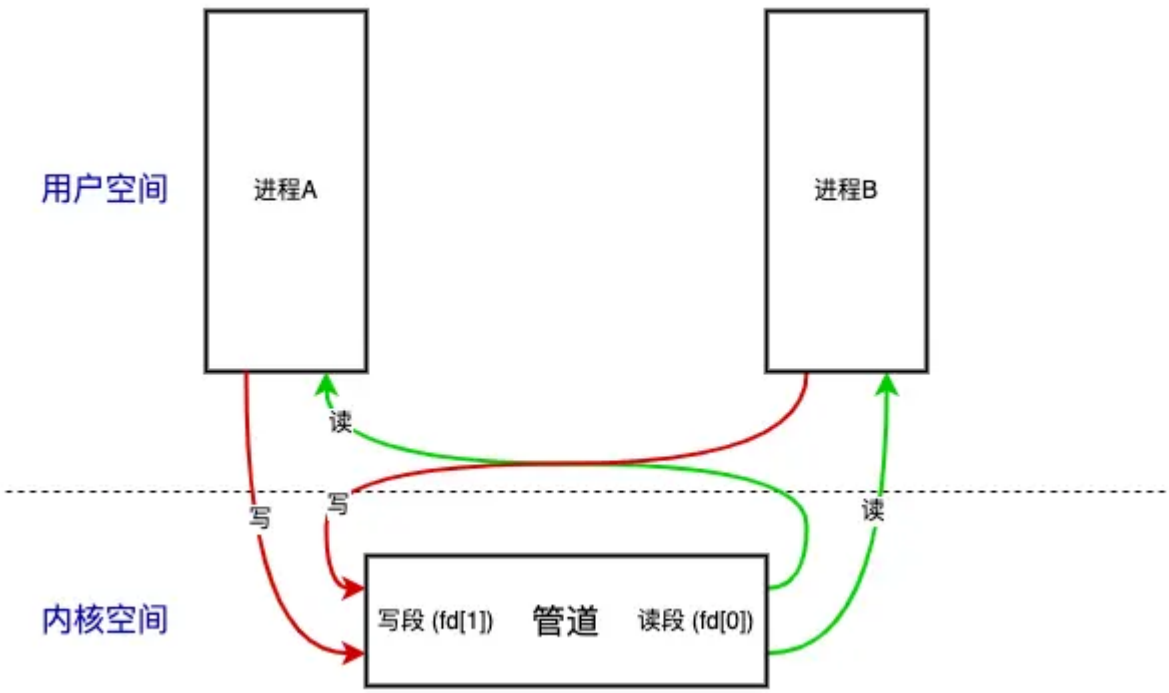
共享内存是进程间通信中最简单的方式之一，共享内存允许两个或更多进程访问同一块内存，当一个进程改变了这块地址中的内容的时候，其它进程都会察觉到这个更改，它的原理如下图所示：



因为共享内存是访问同一块内存，所以数据不需要进行任何复制，是IPC几种方式中最快，性能最好的方式。但相对应的，共享内存未提供同步机制，需要我们手动控制内存间的互斥操作，较容易发生问题。同时共享内存由于能任意的访问和修改内存中的数据，如果有恶意程序去针对某个程序设计代码，很可能导致隐私泄漏或者程序崩溃，所以安全性较差。

管道

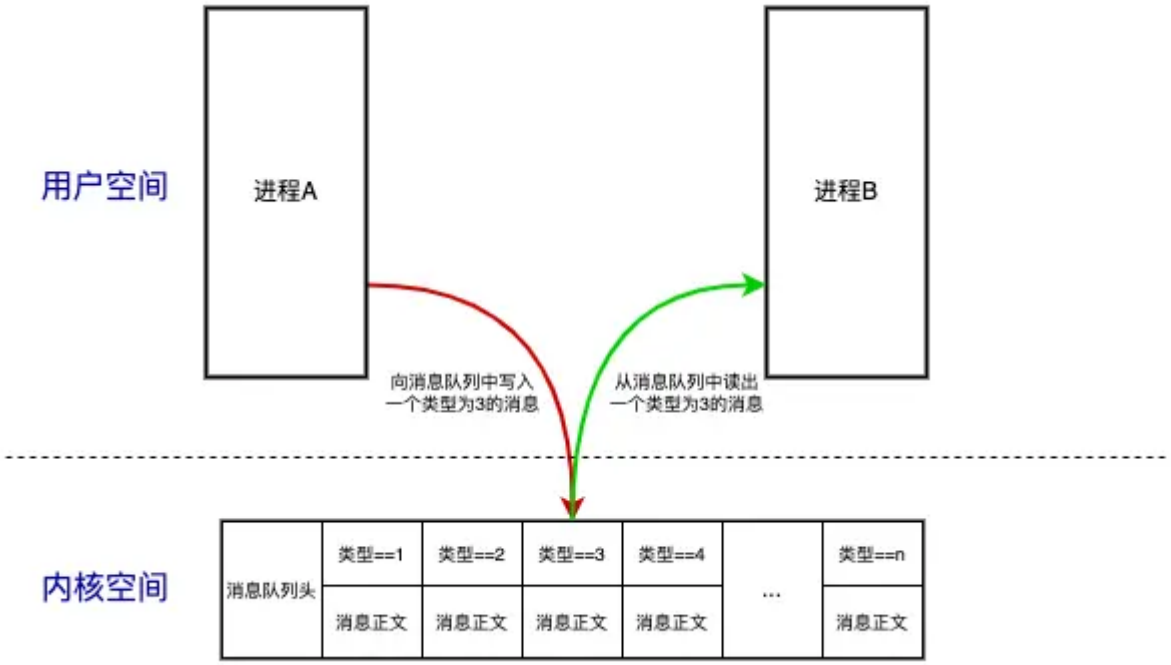
管道分为命名管道和无名管道，它是以一种特殊的文件作为中间介质，我们称为管道文件，它具有固定的读端和写端，写进程通过写段向管道文件里写入数据，读进程通过读段从读进程中读出数据，构成一条数据传递的流水线，它的原理如下图所示：



管道一次通信需要经历2次数据复制（进程A -> 管道文件，管道文件 -> 进程B）。管道的读写分阻塞和非阻塞，管道创建会分配一个缓冲区，而这个缓冲区是有限的，如果传输的数据大小超过缓冲区上限，或者在阻塞模式下没有安排好数据的读写，会出现阻塞的情况。管道所传送的是无格式字节流，这就要求管道的读出方和写入方必须事先约定好数据的格式。

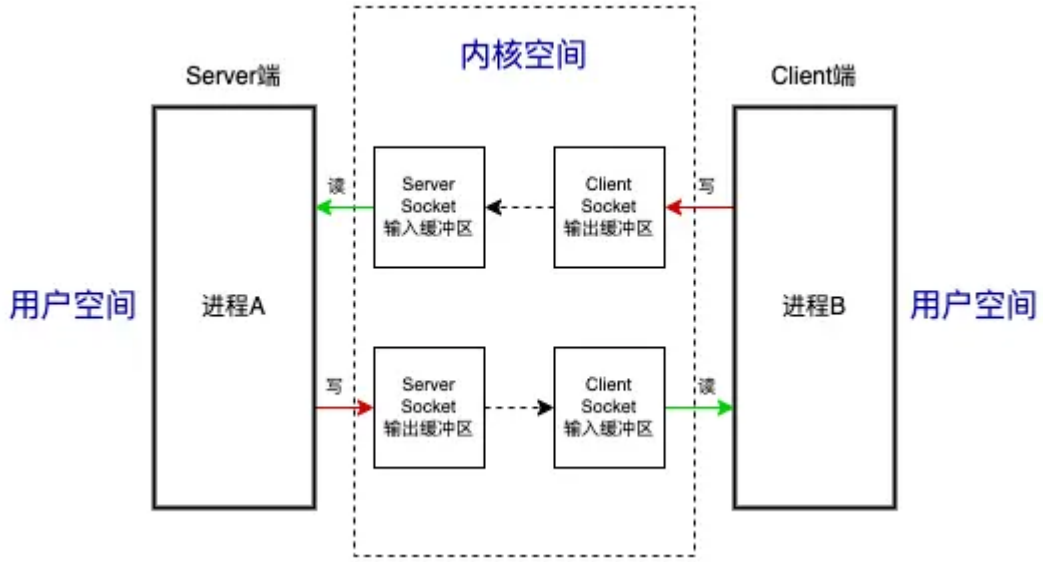
消息队列

消息队列是存放在内核中的消息链表，每个消息队列由消息队列标识符表示。消息队列允许多个进程同时读写消息，发送方与接收方要约定好消息体的数据类型与大小。消息队列克服了信号承载信息量少、管道只能承载无格式字节流等缺点，消息队列一次通信同样需要经历2次数据复制（进程A -> 消息队列，消息队列 -> 进程B），它的原理如下图所示：



Socket

`Socket`原本是为了网络设计的，但也可以通过本地回环地址(`127.0.0.1`)进行进程间通信，后来在`Socket`的框架上更是发展出一种IPC机制，名叫`UNIX Domain Socket`。`Socket`是一种典型的`C/S`架构，一个`Socket`会拥有两个缓冲区，一读一写，由于发送/接收消息需要将一个`Socket`缓冲区中的内容拷贝至另一个`Socket`缓冲区，所以`Socket`一次通信也是需要经历2次数据复制，它的原理如下图所示：



Binder

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

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

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

这就不得不通道 Linux 下的另一个概念：**内存映射**。

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

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

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

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

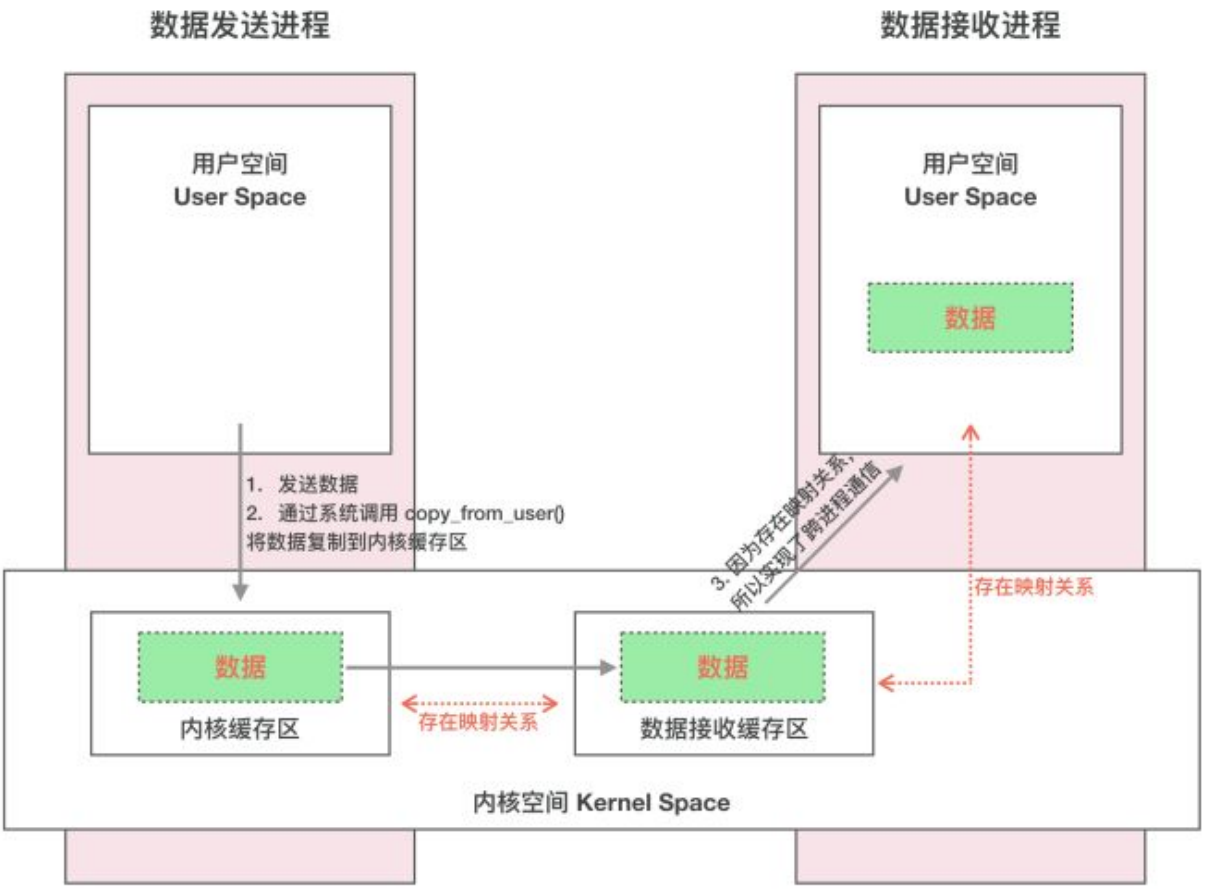
而 Binder 并不存在物理介质，因此 Binder 驱动使用 mmap() 并不是为了在物理介质和用户空间之间建立映射，而是用来在内核空间创建数据接收的缓存空间。

一次完整的 Binder IPC 通信过程通常是这样：

1. 首先 Binder 驱动在内核空间创建一个数据接收缓存区；

2. 接着在内核空间开辟一块内核缓存区，建立**内核缓存区**和**内核中数据接收缓存区**之间的映射关系，以及**内核中数据接收缓存区**和**接收进程用户空间地址**的映射关系；
3. 发送方进程通过系统调用 copy*from*user() 将数据 copy 到内核中的**内核缓存区**，由于内核缓存区和接收进程的用户空间存在内存映射，因此也就相当于把数据发送到了接收进程的用户空间，这样便完成了一次进程间的通信。

如下图：



小结

性能

首先说说性能上的优势。Socket 作为一款通用接口，其传输效率低，开销大，主要用在跨网络的进程间通信和本机上进程间的低速通信。消息队列和管道采用存储-转发方式，即数据先从发送方缓存区拷贝到内核开辟的缓存区中，然后再从内核缓存区拷贝到接收方缓存区，至少有两次拷贝过程。共享内存虽然无需拷贝，但控制复杂，难以使用。Binder 只需要一次数据拷贝，性能上仅次于共享内存。

稳定性

再说说稳定性，Binder 基于 C/S 架构，客户端（Client）有什么需求就丢给服务端（Server）去完成，架构清晰、职责明确又相互独立，自然稳定性更好。共享内存虽然无需拷贝，但是控制负责，难以使用。从稳定性的角度讲，Binder 机制是优于内存共享的。

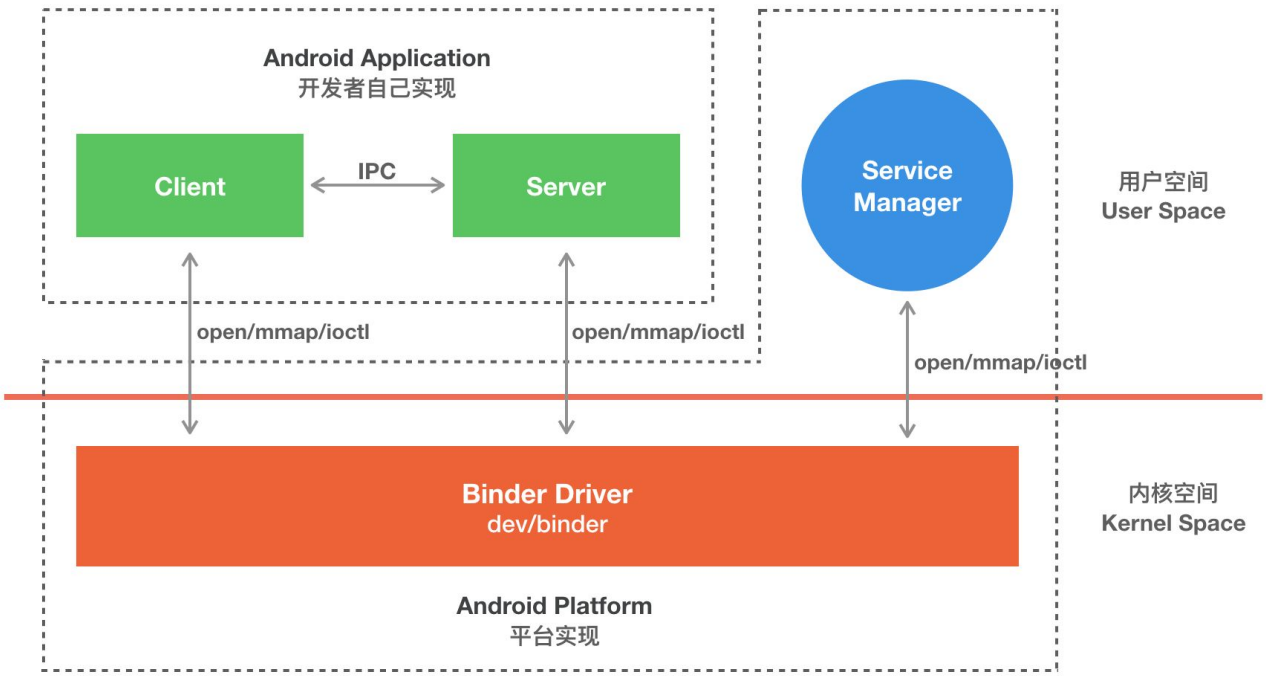
安全性

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

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

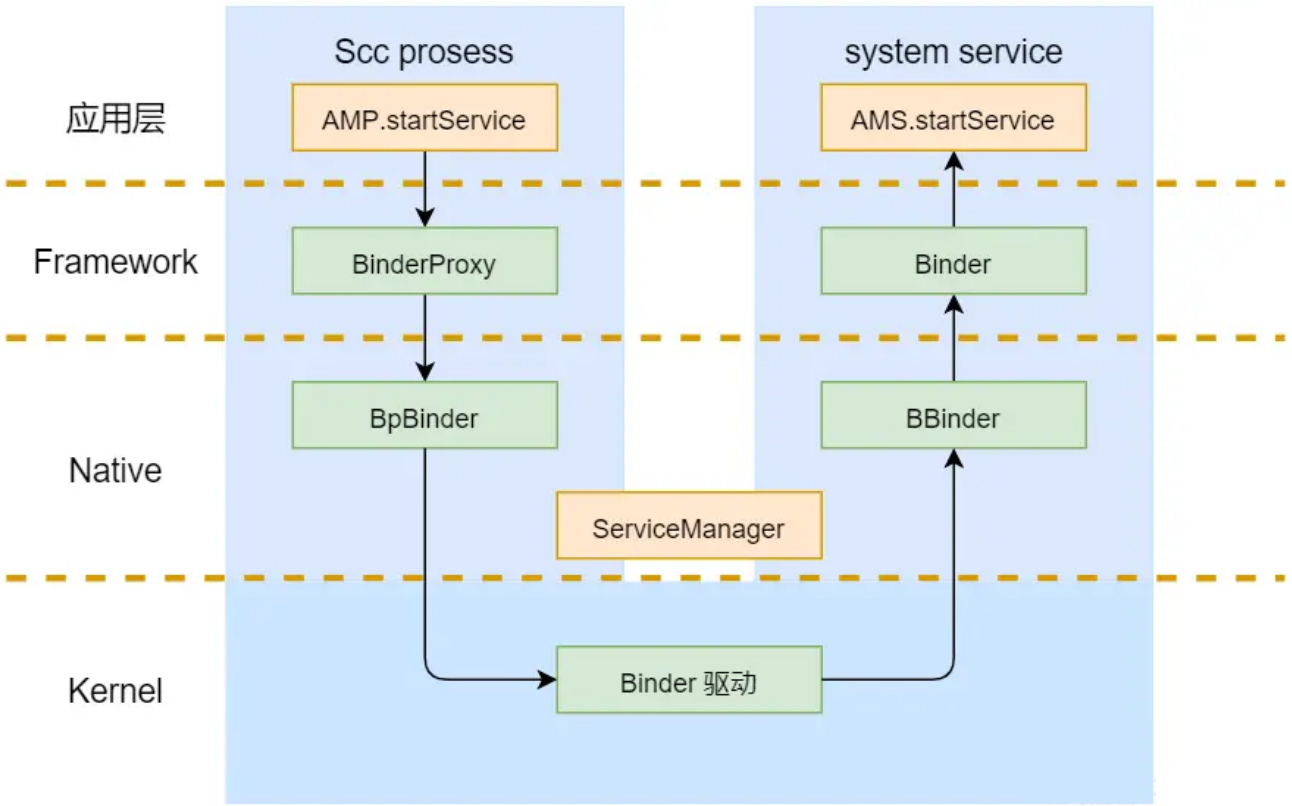
4.Binder架构

Binder 是基于 C/S 架构的。由一系列的组件组成，包括 Client、Server、ServiceManager、Binder 驱动。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。

Client 先去ServiceManager中拿到Server的binder（BpBinder），然后Client再通过这个binder来“**调用**”Server端的代码。当然这个“**调用**”是跨进程的过程，需要通过ioctl来支持，也就是需要Binder驱动支持。具体的流程如下图所示。



5.总结

上面章节整体分析了一下IPC通信，那么很多关于binder通信的细节，可能大家还会比较懵逼，那么这些不熟悉的或者大家懵逼的理论，我们将在接下来的内容里面进行分析。

二：如何设计一个binder

1.Binder的设计方案

在理解Binder架构前，我们来考虑下，如果是你，该如何设计一个Binder的进程间通信机制。

要实现一个IPC通信那么需要**几个核心要素**：

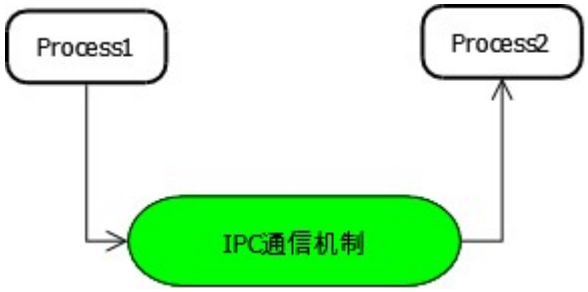
1)发起端：肯定包括发起端所从属的进程，以及实际执行传输动作的线程

2)接收端：接收发送端的数据。

3)待传输的数据

4)内存映射，内核态

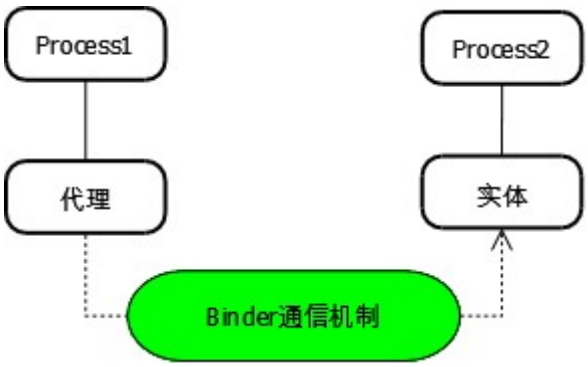
首先先画一个最简单的IPC通信图：



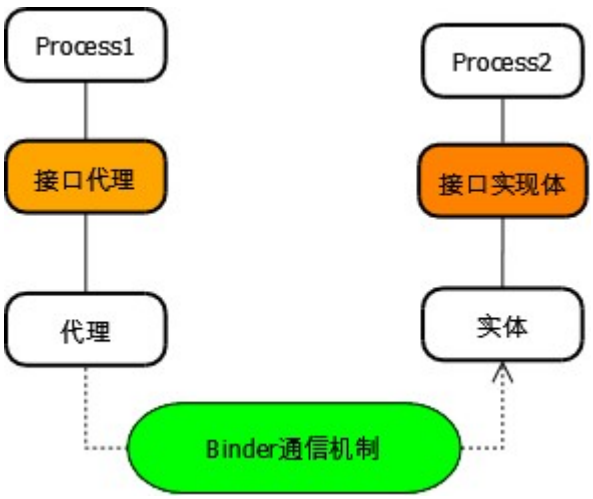
进程Process1和进程Process2 通过IPC通信机制进行通信。

再进行扩展调整，把IPC机制换成Binder机制，那么就变成如下的图形：

由于Android存在进程隔离，那么两个进程之间是不能直接传输数据的，Process1需要得到Process2的代理，Process2需要一个实体。



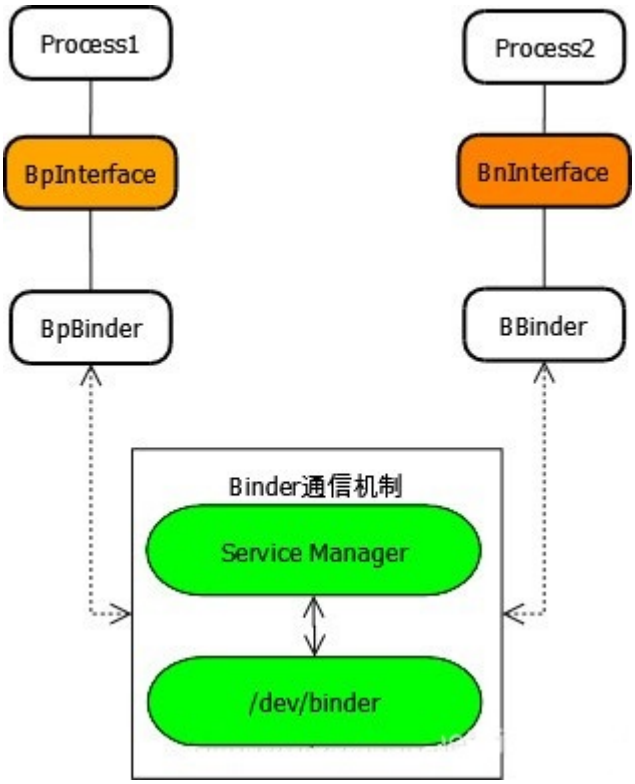
为了实现RPC，我们的代理都是提供接口，称为“接口代理”，实体需要提供“接口实体”，如下图所示：



我们把代理改成BpBinder，实体改成BBinder，接口代理改成BpInterface，接口实现体改成BnInterface。

我们都知道两个进程的数据共享，需要陷入内核态，那就需要一个驱动设备“/dev/binder”，同时需要一个守护进行来进行service管理，我们成为ServiceManager。

进一步演变为：



假如我们想要把通过Process1 的微信信息发送给Process2的微信，我们需要做下面几步：

0)Process2在腾讯服务器中进行注册(包括微信名称、当前活动的IP地址等)

1)Process1从朋友列表中中查找到 Process2的名称，这就是Process2的别名："service_name"

2)Process1 编写消息消息内容，点击发送。 这里的消息内容就是IPC数据

3)数据会发送到腾讯的服务器，服务器理解为Binder驱动

4)服务器从数据库中解析出IPC数据，找到Process2信息，转到Process2注册的地址， 数据库理解为ServiceManager

5)把数据发给Process2，完成Process1和Process2的通信

我们可以简单的把上面的顺序内容进行转换：

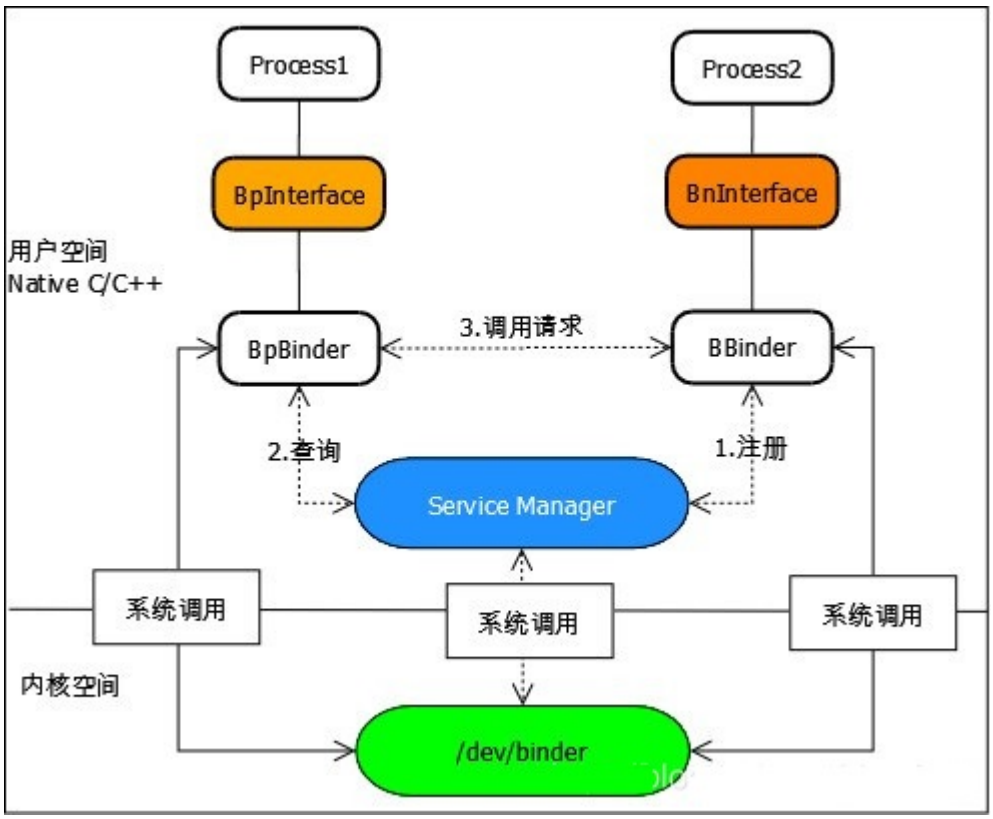
1)Binder驱动---腾讯服务器

2)数据库--ServiceManager

3)Service_name: Process2的微信名称

4)IPC数据：Process1 发送的微信消息

Native C/C++和内核进行通信需要通过系统调用，ServiecManager的主要用来对Service管理，提供了add\find\list等操作。Native进程的数据直接可以通过系统调用陷入内核态，进入图像转换，变为如下:



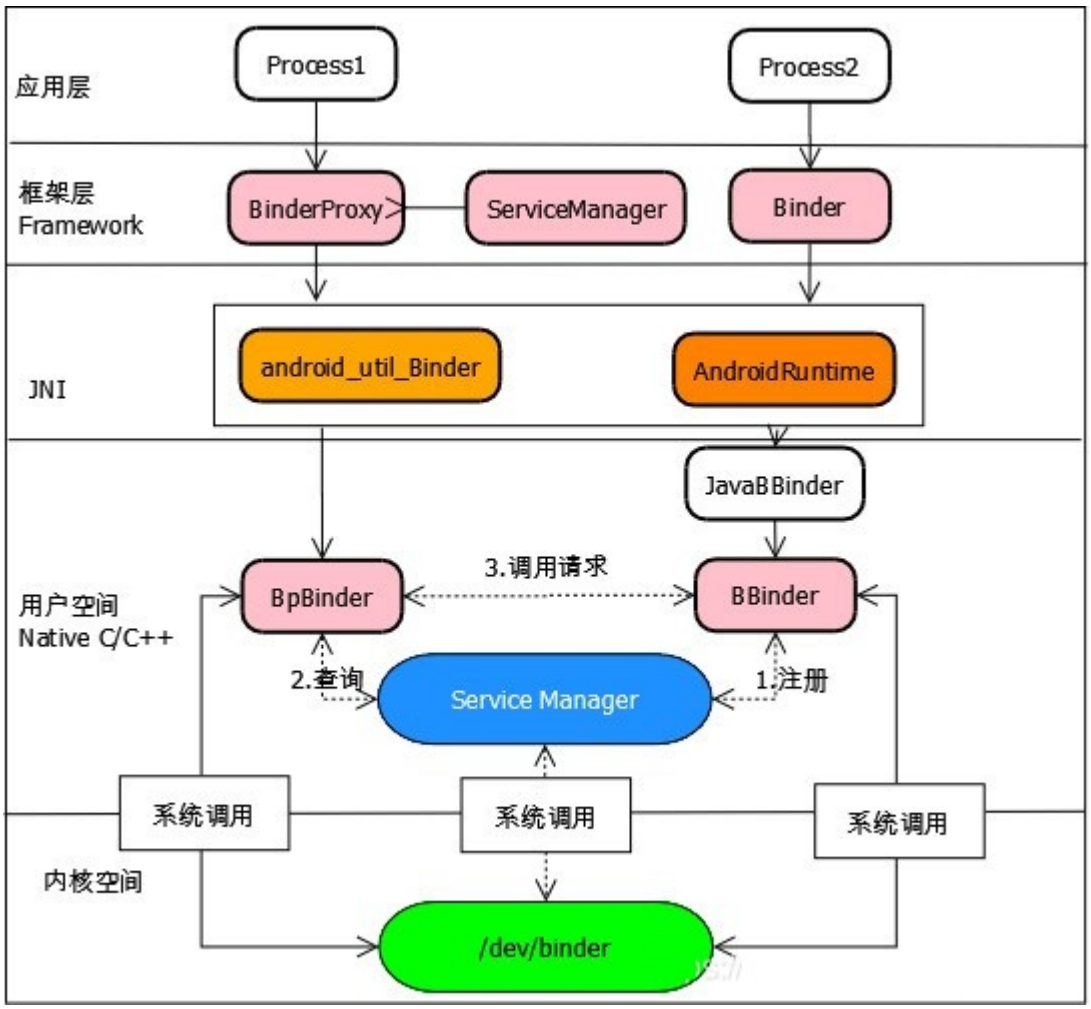
上面列举的是Native C/C++空间的进程进行Binder通信机制，那么JAVA层是如何通信的呢，Native层的Binder提供的是libbinder.so，那么从JAVA到Native，需要经过JNI、Framework层的封装，

JNI层的命名通常为android_util_xxx，我们这里是binder机制，那么JNI层的文件为 android_util_binder，同时Native的BBinder不能直接传给JAVA层，在JNI里面转换了一个JavaBBinder对象。

Framework层给应用层提供时，其实提供的也是一个代理，我们也称之为BinderProxy。

在JAVA侧要对应一个Binder的实体，称之为Binder。

JAVA侧的服务进行也需要一个管理者，类似于Native，创建了JAVA的ServiceManager，那么设计如下：



相信到现阶段，大家已经对binder的运行机制有了一点的理解了，但是仍旧会有很多懵逼的地方，大家如果有这种感觉，不要急，回过头再看一遍Binder通信模型就理解了。我们接下来分析一下AIDL对binder的一个封装。

2.Binder何时初始化

我们已经讲了这么多关于binder的通信的案例了，那么Binder到底是在什么时候初始化呢?

Binder初始化一般是指binder驱动的初始化，大家在使用binder的过程中，我们从来没有执行过new Binder的方式来实现Binder初始化，原因很简单：binder初始化有它自身独立的特点。

每一个应用进程启动的时候，都是通过zygote fork产生的，所以，当fork产生进程后app进程的代码就开始执行，就开始运行的地方如下：

```
    public static final Runnable zygoteInit(int targetSdkVersion,
                                            long[] disabledCompatChanges,
                                            String[] argv, ClassLoader classLoader) {
        if (RuntimeInit.DEBUG) {
            Slog.d(RuntimeInit.TAG, "RuntimeInit: Starting application from zygote");
        }

        Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "ZygoteInit");
        RuntimeInit.redirectLogStreams();

        RuntimeInit.commonInit();//初始化运行环境
        ZygoteInit.nativeZygoteInit();//启动Binder ，方法在 androidRuntime.cpp中注册
```

```
        // 通过反射创建程序入口函数的 Method 对象，并返回 Runnable 对象
        //ActivityThread.main();
        return RuntimeInit.applicationInit(targetSdkVersion, disabledCompatChanges, argv,
                classLoader);
    }
```

大家可以看到，会执行ZygoteInit.nativeZygoteInit()函数，而nativeZygoteInit函数执行appRuntime的onZygoteInit代码，也就是App_main.cpp中的 onZygoteInit()函数，函数如下：

```
virtual void onZygoteInit()
{
    sp<ProcessState> proc = ProcessState::self();
    ALOGV("App process: starting thread pool.\n");
    proc->startThreadPool();
}
```

在ProcessState的self函数里面就会初始化ProcessState（），而这个初始化的一个非常重要的动作就是启动binder驱动和并构建binder的Map映射。具体代码如下：

```
ProcessState::ProcessState(const char *driver)
    : mDriverName(String8(driver))
    , mDriverFD(open_driver(driver)) //打开binder的虚拟驱动
    , mVMStart(MAP_FAILED)
    , mThreadCountLock(PTHREAD_MUTEX_INITIALIZER)
    , mThreadCountDecrement(PTHREAD_COND_INITIALIZER)
    , mExecutingThreadsCount(0)
    , mMaxThreads(DEFAULT_MAX_BINDER_THREADS)
    , mStarvationStartTimeMs(0)
    , mBinderContextCheckFunc(nullptr)
    , mBinderContextUserData(nullptr)
    , mThreadPoolStarted(false)
    , mThreadPoolSeq(1)
    , mCallRestriction(CallRestriction::NONE)
{

// TODO(b/139016109): enforce in build system
#if defined(__ANDROID_APEX__)
    LOG_ALWAYS_FATAL("Cannot use libbinder in APEX (only system.img libbinder) since it is
not stable.");
#endif

    if (mDriverFD >= 0) {

        //调用mmap接口向Binder驱动中申请内核空间的内存
        // mmap the binder, providing a chunk of virtual address space to receive
transactions.
        mVMStart = mmap(nullptr, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE,
mDriverFD, 0);
        if (mVMStart == MAP_FAILED) {
            // *sigh*
            ALOGE("Using %s failed: unable to mmap transaction memory.\n",
mDriverName.c_str());
```

```
            close(mDriverFD);
            mDriverFD = -1;
            mDriverName.clear();
        }
    }

#ifdef __ANDROID__
    LOG_ALWAYS_FATAL_IF(mDriverFD < 0, "Binder driver '%s' could not be opened.
Terminating.", driver);
#endif
}
```

所以，总的来说，Binder的初始化是在进程已创建就完成了。创建进程后会第一时间为这个进程打开一个binder驱动，并调用mmap接口向Binder驱动中申请内核空间的内存。

三：Binder通信模型

1.Binder通信模型

要讲解Binder的通信模型，我们首先需要了解网络的通信模型。

1.1网络请求的通信模型

通常我们访问一个网页的步骤是这样的：首先在浏览器输入一个地址，如 http://www.google.com 然后按下回车键。但是并没有办法通过域名地址直接找到我们要访问的服务器，因此需要首先访问 DNS 域名服务器，域名服务器中保存了 http://www.google.com 对应的 ip 地址 10.249.23.13，然后通过这个 ip 地址才能放到到 http://www.google.com 对应的服务器。

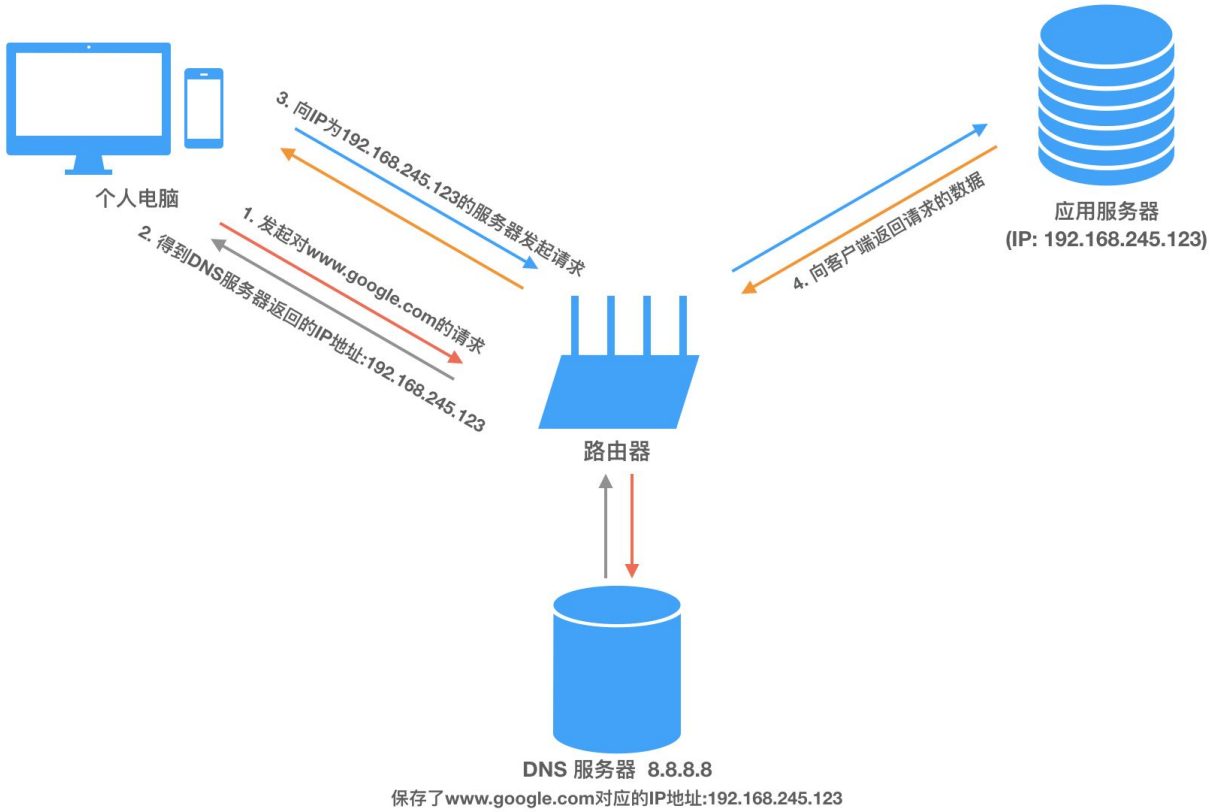
它的通信模型是这样的：

1) Server端向DNS服务器注册自己的域名和IP，因此NDS服务器中存储了所有的域名和IP的对于的信息表；

2) Client端先通过域名去访问网络服务，但是域名却不能给client端打通服务，因为访问服务是通过IP进行；

3）这个时候为了通过域名找到IP地址，所以首先会通过路由器先去访问DNS服务器，通过DNS服务器的域名解析拿到IP地址；

4）client端在拿到DNS给的IP地址后，通过IP地址由路由器去访问服务器，完成网络请求。



1.2 切换到binder中的说明

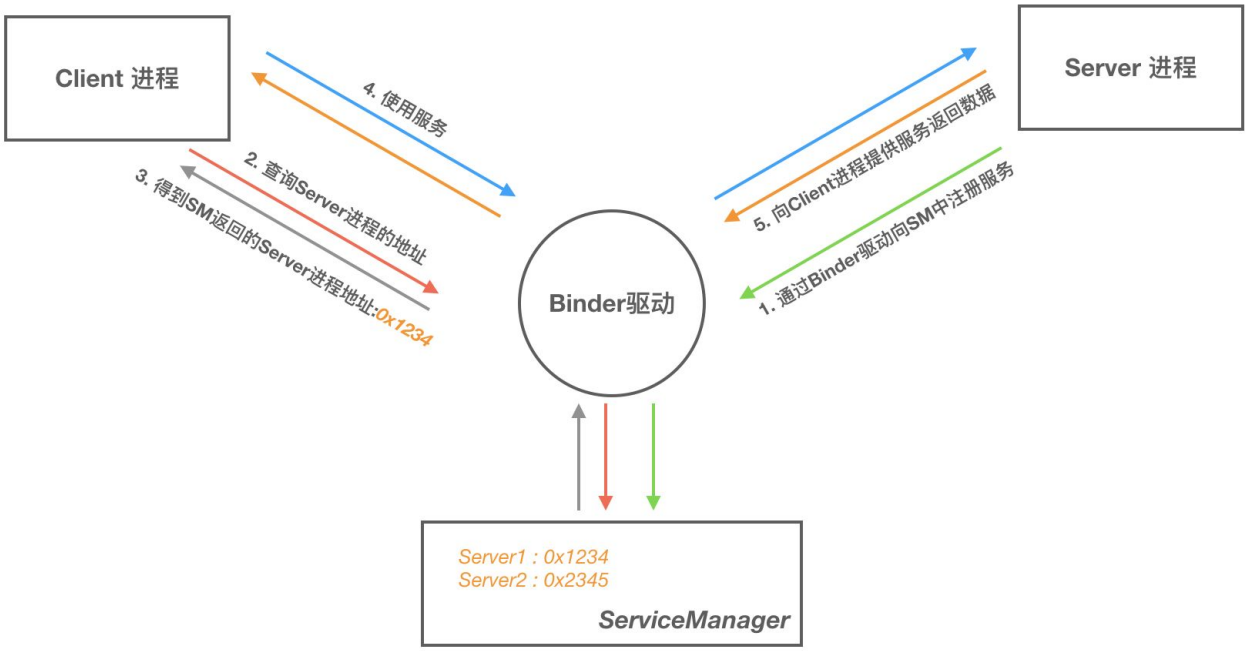
Binder通信的模型就和上面的网络通信模型有着异曲同工之妙。在角色方面，Client进程就等同于网络请求的Client端，服务器就是Server进程，路由器相当于Binder驱动，而DNS服务器相当于ServiceManager进程。 **Binder 驱动**就如同路由器一样，是整个通信的核心；驱动负责进程之间 Binder 通信的建立，Binder 在进程之间的传递，Binder 引用计数管理，数据包在进程之间的传递和交互等一系列底层支持。

ServiceManager 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，就像网站一样除了有 IP 地址以外还有自己的网址（域名）。Server 创建了 Binder，并为它起一个可读易记得名字，比如AMS的binder，我们就取了一个名字叫做“activity”，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“activity”的 Binder，它位于某个 Server 中。而ServcieManager中就存储了一个表格，这个表格中就有名字和binder引用对应的信息item。

Client 获得实名 Binder 的引用

在这个代码 ServiceManager.getService(Context.ACTIVITY_SERVICE) //String ACTIVITY_SERVICE = "activity",这段代码是Client去ServcieManager中获取server进程的binder的代码，这个地方获取的是AMS的binder实体的代码。代码的设计逻辑就是去ServcieManager的查找表容器中去获取名字为activity的binder实体。

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。



2.Binder通信的流程

Binder框架定义了四个角色：Server，Client，ServiceManager（以后简称SMgr）以及Binder驱动。其中Server，Client，SMgr运行于用户空间，驱动运行于内核空间。这四个角色的关系和互联网类似：Server是服务器，Client是客户终端，SMgr是域名服务器（DNS），驱动是路由器。

在网络通信中域名服务器的地址是一个固定的地址，所以很方便的通过这个固定地址拿到。那么在Binder中，SMgr是一个进程，Server是另一个进程，Server向SMgr注册Binder必然会涉及进程间通信。当前实现的是进程间通信却又要用到进程间通信，这就好象蛋可以孵出鸡前提却是要找只鸡来孵蛋。Binder的实现比较巧妙：预先创造一只鸡来孵蛋：SMgr和其它进程同样采用Binder通信，SMgr是Server端，有自己的Binder对象（实体），其它进程都是Client，需要通过这个Binder的引用来实现Binder的注册，查询和获取。SMgr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SMgr时Binder驱动会自动为它创建Binder实体（这就是那只预先造好的鸡）。其次这个Binder的引用在所有Client中都固定为0而无须通过其它手段获得。也就是说，一个Server若要向SMgr注册自己Binder就必需通过0这个引用号和SMgr的Binder通信。类比网络通信，0号引用就好比域名服务器的地址，你必须预先手工或动态配置好。要注意这里说的Client是相对SMgr而言的，一个应用程序可能是个提供服务的Server，但对SMgr来说它仍然是个Client。

有了以上的知识铺垫，我们可以比较清晰的把BInder通信的流程梳理出来：

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。
4. 通过这个Binder实体引用，Client实现和 Server 进程的通信。

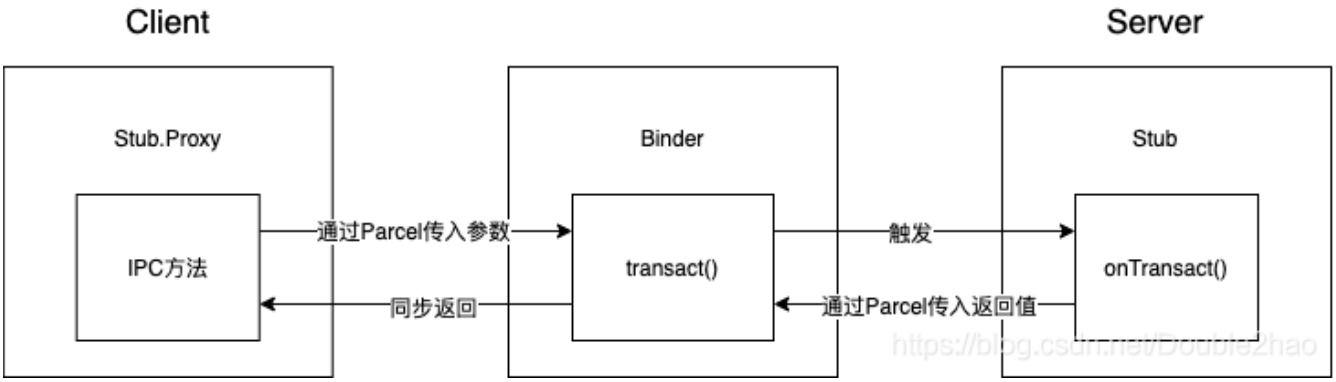
四：AIDL

先给大家上一个新闻，AIDL可以生成C++ 文件，哈哈，不仅仅是可以生成java文件哦，当然，要在android的比较新的版本上面才能够实现这点，目前需要在api-29之后。

概述

aidl是常用的android IPC方式，本文将根据一个demo来解析下AIDL的原理。

为了便于读者理解，本文不会探究Binder的实现细节，可以认为Binder在此文的分析中被看做是一个“黑盒”。有一定经验的读者可以直接到文末看总结，最终流程图如下：



Demo讲解一下AIDL

demo实现内容： MainActivity通过AIDL实现IPC来调用另一个进程中RemoteTestService的一个方法。 主要有以下几个文件：

1. MainActivity.java

   自定义ServiceConnection，然后将ServiceConnection传入bindService，获取到IBinder后实现远程调用。
2. RemoteTestService.java 在ITestServer.Stub中实现需要远程调用的方法testFunction()，在onBind中返回。
3. IServer.aidl 定义一个aidl文件和需要远程调用的方法
4. AndroidManifest.xml 在此设置RemoteTestService在remote进程。

先看MainActivity.java

```
public class MainActivity extends AppCompatActivity {

  private IServer iServer;
  private ServiceConnection serviceConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
      iServer = IServer.Stub.asInterface(service);
      System.out.println("onServiceConnected");

      try {
        int a = iServer.testFunction("test string");
        Log.i("test", "after testFunction a:" + a);
```

```
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
      Log.i("test", "onServiceDisconnected");
    }
  };

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Intent intent = new Intent(MainActivity.this, RemoteTestService.class);
    bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);
  }
}
```

接下来看RemoteTestService.java

```
public class RemoteTestService extends Service {

  private IServer.Stub serverBinder = new IServer.Stub() {
    @Override public int testFunction(String s) throws RemoteException {
      Log.i("test","testFunction s= " + s);
      return 0;
    }
  };

  @Nullable @Override
  public IBinder onBind(Intent intent) {
    return serverBinder;
  }
}
```

再看IServer.aidl

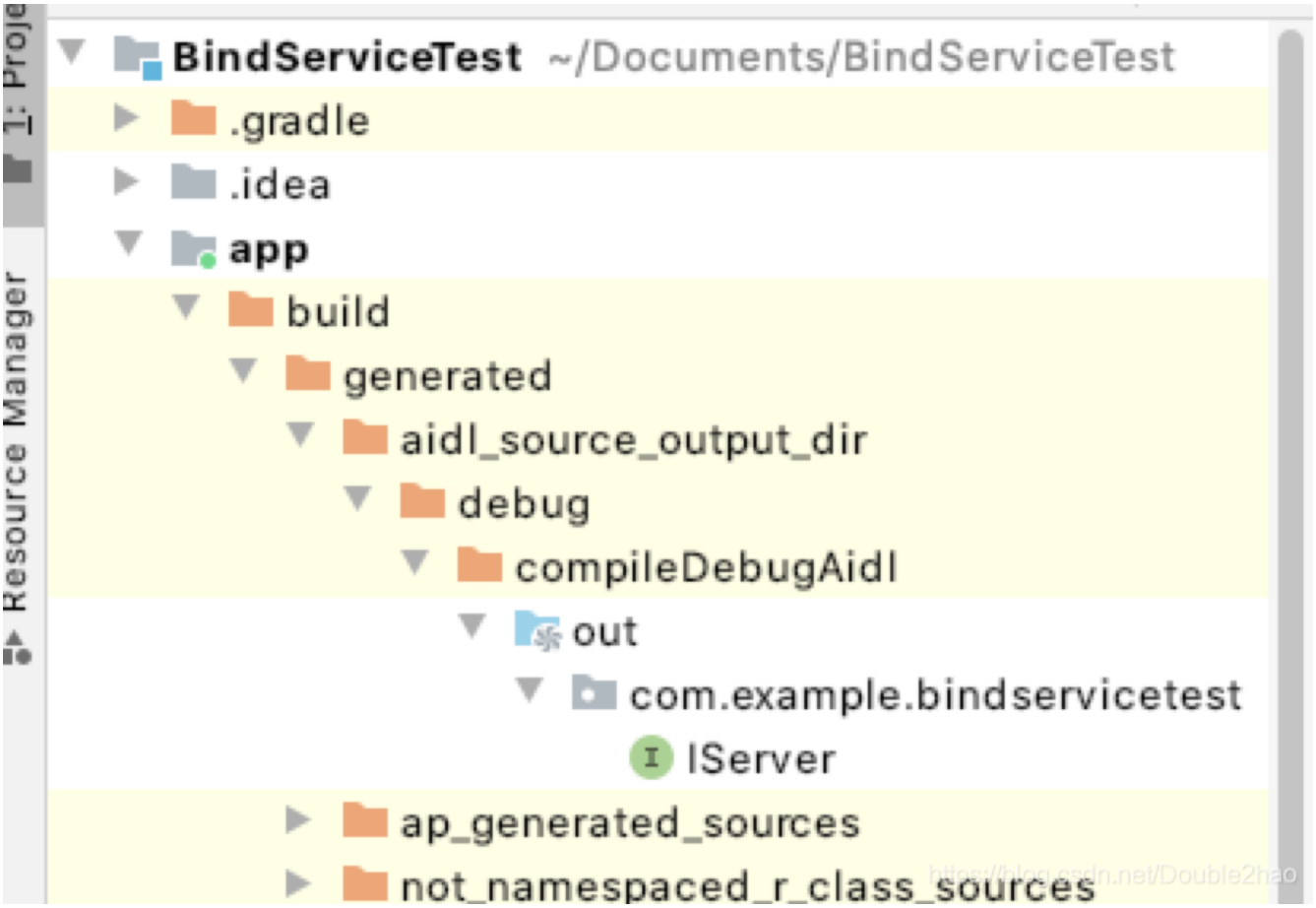
```
interface IServer {
    int testFunction(String s);
}
```

再看 AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.bindservicetest">
```

```
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
      <activity android:name=".MainActivity">
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />

          <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>

      <service
          android:name=".RemoteTestService"
          android:process=":remote" />
    </application>
</manifest>
```

aidl自动生成的文件 定义完aidl文件，编译会自动生成一个java接口文件。 这个接口文件在build目录下，具体路径如下：



打开文件，我们就可以看到aidl自动生成的代码。

```
public interface IServer extends android.os.IInterface
{
  /** Default implementation for IServer. */
  public static class Default implements com.example.bindservicetest.IServer
  {
    @Override public int testFunction(java.lang.String s) throws android.os.RemoteException
    {
      return 0;
    }
    @Override
    public android.os.IBinder asBinder() {
      return null;
    }
  }
  /** Local-side IPC implementation stub class. */
  public static abstract class Stub extends android.os.Binder implements
com.example.bindservicetest.IServer
  {
    private static final java.lang.String DESCRIPTOR =
"com.example.bindservicetest.IServer";
    /** Construct the stub at attach it to the interface. */
    public Stub()
    {
      this.attachInterface(this, DESCRIPTOR);
    }
    /**
     * Cast an IBinder object into an com.example.bindservicetest.IServer interface,
     * generating a proxy if needed.
     */
    public static com.example.bindservicetest.IServer asInterface(android.os.IBinder obj)
    {
      if ((obj==null)) {
        return null;
      }
      android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
      if (((iin!=null)&&(iin instanceof com.example.bindservicetest.IServer))) {
        return ((com.example.bindservicetest.IServer)iin);
      }
      return new com.example.bindservicetest.IServer.Stub.Proxy(obj);
    }
    @Override public android.os.IBinder asBinder()
    {
      return this;
    }
    @Override public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel
reply, int flags) throws android.os.RemoteException
    {
      java.lang.String descriptor = DESCRIPTOR;
      switch (code)
      {
        case INTERFACE_TRANSACTION:
        {
          reply.writeString(descriptor);
```

```
        return true;
      }
      case TRANSACTION_testFunction:
      {
        data.enforceInterface(descriptor);
        java.lang.String _arg0;
        _arg0 = data.readString();
        int _result = this.testFunction(_arg0);
        reply.writeNoException();
        reply.writeInt(_result);
        return true;
      }
      default:
      {
        return super.onTransact(code, data, reply, flags);
      }
    }
  }
  private static class Proxy implements com.example.bindservicetest.IServer
  {
    private android.os.IBinder mRemote;
    Proxy(android.os.IBinder remote)
    {
      mRemote = remote;
    }
    @Override public android.os.IBinder asBinder()
    {
      return mRemote;
    }
    public java.lang.String getInterfaceDescriptor()
    {
      return DESCRIPTOR;
    }
    @Override
    public int testFunction(java.lang.String s) throws android.os.RemoteException
    {
      android.os.Parcel _data = android.os.Parcel.obtain();
      android.os.Parcel _reply = android.os.Parcel.obtain();
      int _result;
      try {
        _data.writeInterfaceToken(DESCRIPTOR);
        _data.writeString(s);
        boolean _status = mRemote.transact(Stub.TRANSACTION_testFunction, _data,
                                           _reply, 0);
        if (!_status && getDefaultImpl() != null) {
          return getDefaultImpl().testFunction(s);
        }
        _reply.readException();
        _result = _reply.readInt();
      }
      finally {
        _reply.recycle();
        _data.recycle();
```

```
        }
        return _result;
      }
      public static com.example.bindservicetest.IServer sDefaultImpl;
    }
    static final int TRANSACTION_testFunction =
                        (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
    public static boolean setDefaultImpl(com.example.bindservicetest.IServer impl) {
      if (Stub.Proxy.sDefaultImpl == null && impl != null) {
        Stub.Proxy.sDefaultImpl = impl;
        return true;
      }
      return false;
    }
    public static com.example.bindservicetest.IServer getDefaultImpl() {
      return Stub.Proxy.sDefaultImpl;
    }
  }
  public int testFunction(java.lang.String s) throws android.os.RemoteException;
}
```

上面的代码有些乱七八糟的，毕竟是aidl工具生成的，大家看核心部分就好了，那些try，那些if else 大可不必关注。所以核心代码就在下面：

Stub 里面的关键函数1-asInterface：

```
public static com.example.testservice.IServer asInterface(android.os.IBinder obj)
{
    if ((obj==null)) {
        return null;
    }
    android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
    if (((iin!=null)&&(iin instanceof com.example.testservice.IServer))) {
        return ((com.example.testservice.IServer)iin);
    }
    return new com.example.testservice.IServer.Stub.Proxy(obj);//核心代码 1
}
```

该核心代码就是创建一个Proxy对象，同时将IBinder对象的obj传值给Proxy。

Stub类中的关键函数2-onTransact():

```
public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply,
    int flags) throws android.os.RemoteException
    {
      java.lang.String descriptor = DESCRIPTOR;
      switch (code)
      {
        case INTERFACE_TRANSACTION:
        {
          reply.writeString(descriptor);
          return true;
        }
```

```
        case TRANSACTION_testFunction:
        {
          data.enforceInterface(descriptor);
          java.lang.String _arg0;
          _arg0 = data.readString();
          int _result = this.testFunction(_arg0); //核心代码2
          reply.writeNoException();
          reply.writeInt(_result);
          return true;
        }
        default:
        {
          return super.onTransact(code, data, reply, flags);
        }
      }
    }
```

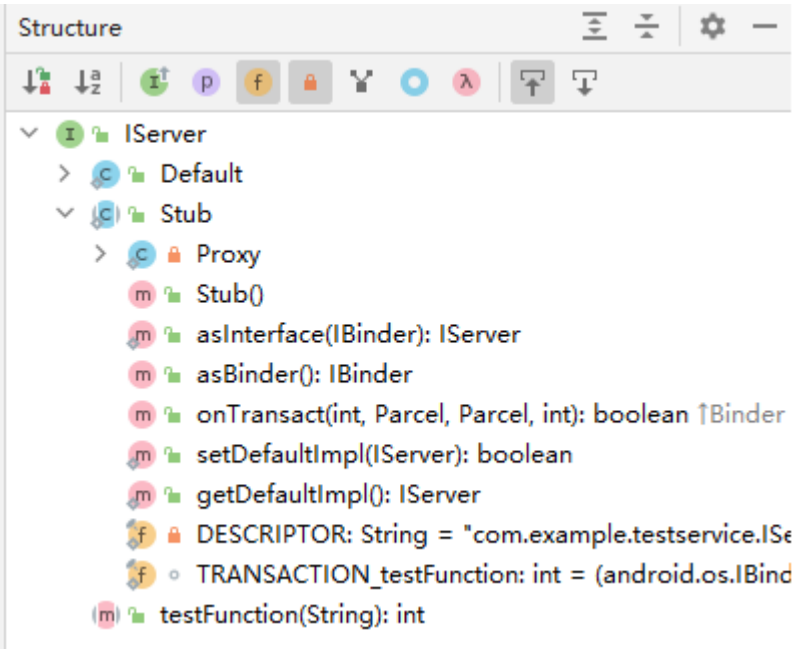
onTransact函数功能主要是接收来自binder底层的调用，通过binder底层调用onTransact函数，将函数执行到核心代码2，具体的流程请看后面的小结。

Prox类里面的关键函数1 testFunction：

```
public int testFunction(java.lang.String s) throws android.os.RemoteException
      {
        android.os.Parcel _data = android.os.Parcel.obtain();
        android.os.Parcel _reply = android.os.Parcel.obtain();
        int _result;
        try {
          _data.writeInterfaceToken(DESCRIPTOR);
          _data.writeString(s);
          boolean _status = mRemote.transact(Stub.TRANSACTION_testFunction, _data,
                            _reply, 0); // 核心代码3
          if (!_status && getDefaultImpl() != null) {
            return getDefaultImpl().testFunction(s);
          }
          _reply.readException();
          _result = _reply.readInt();
        }
        finally {
          _reply.recycle();
          _data.recycle();
        }
        return _result;
      }
```

这个函数是和aidl文件里面定义的函数名字是一样的，后面进行跨进程通信会引用到这个函数，这个的功能是创建一个Parcel 数据_data，为什么必须是Parcel数据了，因为android跨进程传递的数据必须序列化，而序列化所采用的方式就是Parcel。大家如果研究过Parcel序列化，你们会发现它的主要实现是在native层，而Binder通信最主要的部分是在native层。另外，如果跨进程的调用testFunction有返回值，那么这个返回值讲以 _replay来存储和传递。

类的结构图如下：



以上是aidl生成的java代码的介绍，其实在android api 29之后，aidl可以生成C++的代码，具体的生成的代码的逻辑和java代码的逻辑一模一样，只是语言有修改而已。

小结

在运用AIDL通信的过程中，首先Client端（某个acitivty）先会发起bindService的请求，此时Server端（某service）会将自己的binder给Client端，也就是如下代码：

首先，Activity会执行下面的代码：

```
Intent intent = new Intent(MainActivity.this, RemoteTestService.class);
bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);//核心代码
```

bindServcie会去绑定服务"RemoteTestService"，在执行bindService的时候会回调到connection中去，而connection的代码如下：

```
private ServiceConnection connection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            myService =  IServer.Stub.asInterface(service);//proxy
            Log.d(TAG, "onServiceConnected: ");
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {
//            unbindService(connection);
            Log.d(TAG, "onServiceDisconnected: ");
        }
    };
```

当绑定成功就会执行onServiceConnected 回调函数，回调函数就会有一个IBinder对象service，此时通过asInterface函数，会将这个IBinder对象（service）转换成为一个Proxy对象。所以，我们Client去进行跨进程函数调用的时候就是使用这个proxy对象进行。

总结

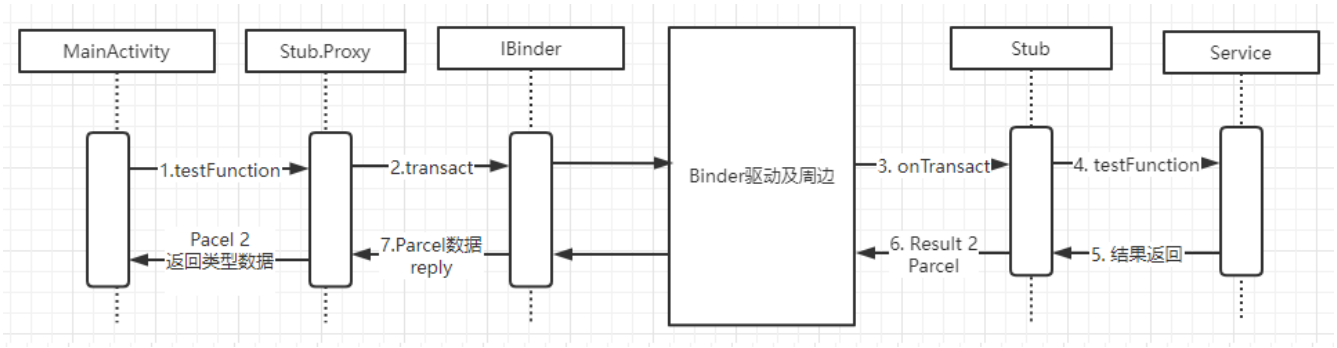
binder基于AIDL的通信流程图如下：

Client端：MainActivity；

Server端：Service；

Service的对外aidl接口如上面的案例所示，

MainActivity已经通过BindService拿到了Service的IBinder对象。



aidl通信的基本步骤如下：

1. Client通过ServiceConnection获取到Server的Binder，并且封装成一个Proxy。
2. 通过Proxy来同步调用IPC方法（testFunction），同时通过Parcel将参数传给Binder，最终触发Binder的transact方法。
3. Binder的transact方法最终会触发到Server上Stub的onTransact方法。
4. Server上Stub的onTransact方法中，会先从Parcel中解析中参数，然后将参数带入真正的方法中执行，然后将结果写入Parcel后传回。
5. 请注意：Client的Ipc方法中，执行Binder的transact时，是阻塞等待的，一直到Server逻辑执行结束后才会继续执行。当然，如果IPC方法是oneWay的方式，那么就是非阻塞的等待。
6. 当Server返回结果后，Client从Parcel中取出返回值，于是实现了一次IPC调用。

所以，aidl 文件会生成一个java文件，这个java文件的意义在于将核心的binder驱动封装成为java层可以直接调用的代码，同时也处理了将java层的数据格式转换为Parcel格式数据进行跨进程传递的一个功能。所以，aidl是一个使用binder的标准方案，该方案的代码同样的可以通过用户自己编写的方式完成。

五：进程间通信之bindService流程

之所以在这里分析bindService的流程是为了给大家展示一个app进程和另外一个进程的Service实现跨进程通信的过程，这个过程将给大家全面的展示开发中最常见的跨进程通信时Client端拿到Server端Binder的总流程。

基于android API 30 代码的执行流程如下：

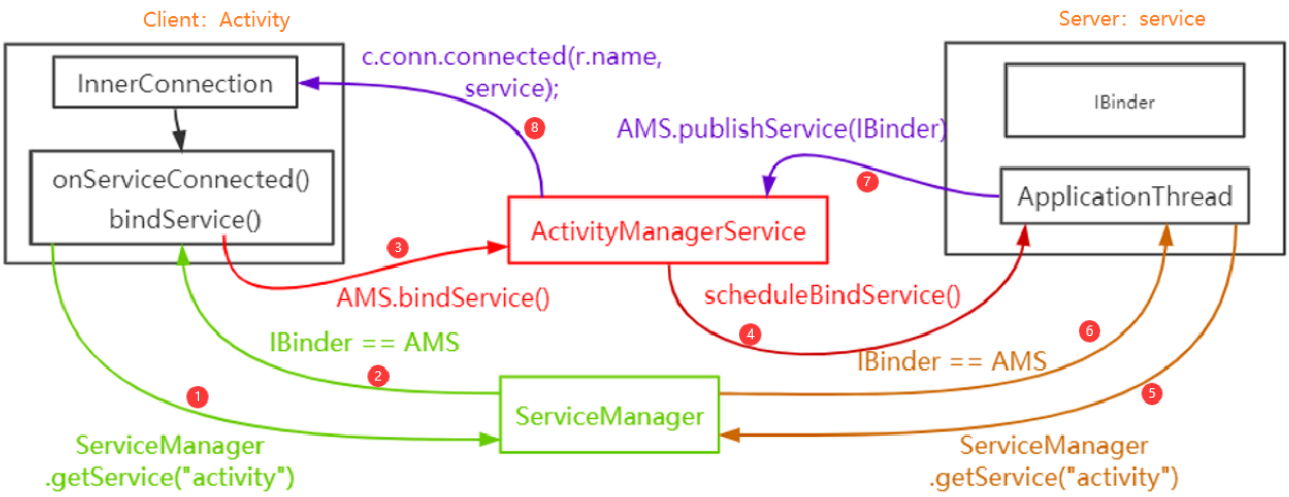
```
ContextWrapper.bindService()
ContextImpl.bindService()
ContextImpl. bindServiceCommon()
ActivityManagerService.bindService()
ActiveServices.bindServiceLocked()
ActiveServices.requestServiceBindingLocked()
```

```
ActivityThread.ApplicationThread.scheduleBindService()
ActivityThread.sendMessage()
ActivityThread.handleBindService()
ActivityManagerService.publishService()
ActiveService.publishServiceLocked()
LoadedApk.ServiceDispatcher.InnerConnection.connected()
LoadedApk.ServiceDispatcher.connected()
ActivityThread.post()
LoadedApk.ServiceDispatcher. RunConnection.run()
LoadedApk.ServiceDispatcher.doConnected()
ServiceConnection.onServiceConnected()
```

我们最重要的是分析清楚，跨进程的流程。

上面会经历几个进程：1）App进程A，发起bindService(); 2）AMS所在的systemServer进程；3）Service所在的进程B；4）还有一个隐藏在背后支撑binder通信的ServiceManager。

上面的代码是具体的bindService的过程，我们可以梳理出一下的图，通过图片来分析会更加清晰。



具体流程如下：

1）Activity作为Client发起bindService，最终会调度到AMS 去执行bindService。在这个过程中，Client要去调用AMS的代码，所以此时就会涉及到跨进程调度，基于第三章的Binder通信模型我们不难知道，Client会先和ServiceManager通信，从ServiceManager中拿到AMS的IBinder。

2）Activity拿到AMS的IBinder后，跨进程执行AMS的BindService函数；

3）由于AMS管理所有的应用进程，因此AMS中持有了应用进程的Binder，所以此时AMS可以发起第4步也就是跨进程调度scheduleBindService();

4)Server端会在收到AMS的bindService的请求后，会将自己的IBinder发送给client，但是Server必须通过AMS才能将Binder对象传过去，所以此时需要跨进程从ServiceManager中去拿到AMS的binder；

5）Server端通过AMS的binder直接调用AMS的代码publishService(),将service的Binder发送给AMS；

6）经过层层调用，最终AMS讲Server端的binder通过回调connect函数传递给了Client端的Activity；

以上就是bindService的全流程，这个流程主要的目的是将Server端的Binder对象发送给Client端。从此以后，Client端就可以通过Server端的binder与Server端像调用自己的代码一样完成跨进程通信了。

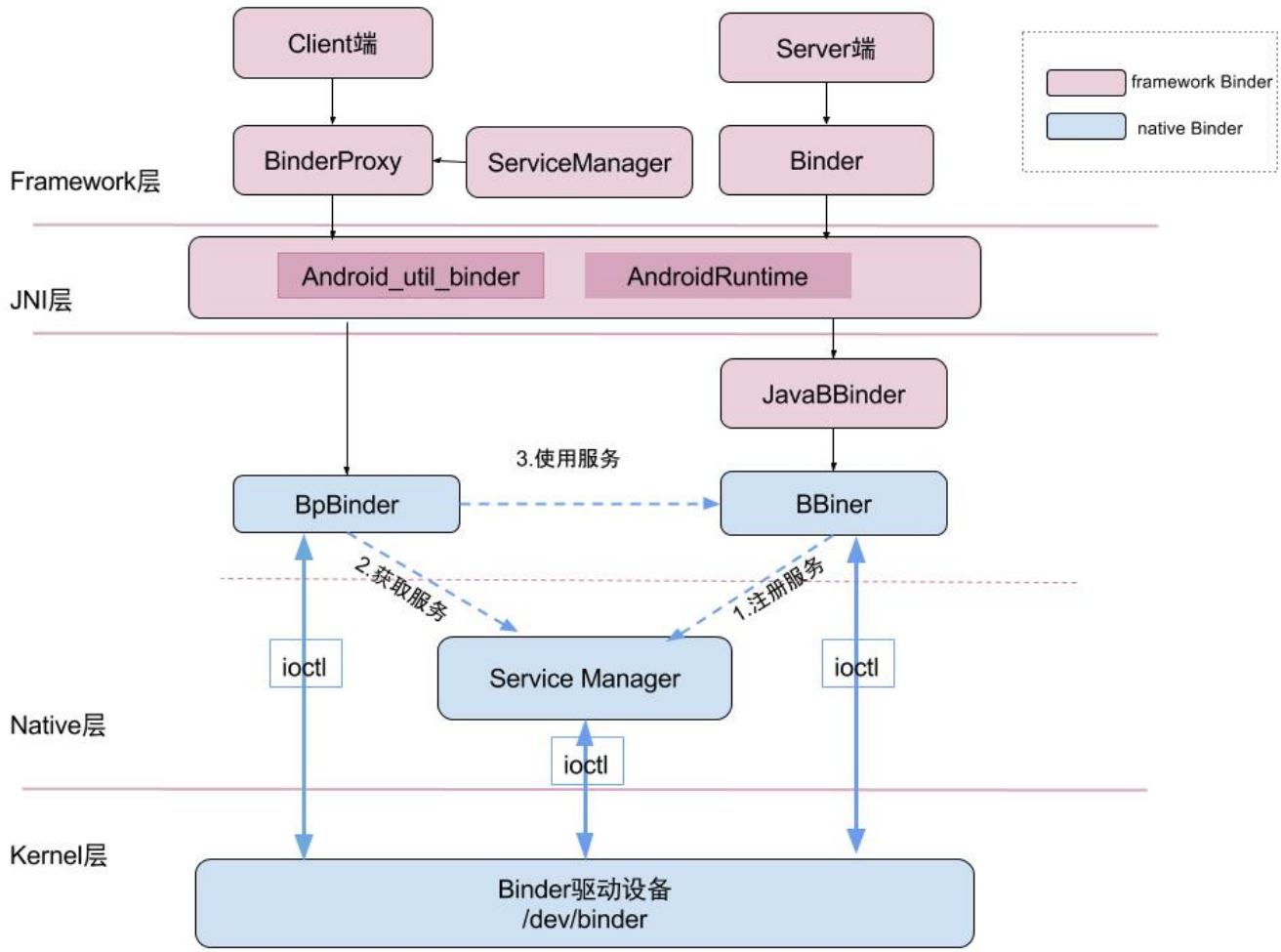
六：Java& Native层面 Binder相互转化

1. Java与Native 通信的基本流程

前面的一~五章已经基本将Binder通信的上层逻辑解释清楚了，接下来的重点是分析binder的上层 是如何走到native层的，换一句话说，native层如何被封装成为上层的。也就是Binder的 Java和Native层的相互转化。

当讲到Binder的java&Native层的通信，天生最好研究的就是ServiceManager类了，因为，它是运用最广泛的也是大家最熟悉的使用了Binder通信的场景。

大家不妨先来看这个图：



通过上图，我们不难发现，在Binder通信必然需要有JNI层的支撑，缺乏JNI层，将无法完成java到Native层的调用，因此我们必须研究一下Java framework层与Native层之间的相互调动。

1）Client端通过ServiceManager 拿到Server端服务的Binder 代理，也就是BinderProxy(是Server端Binder的一个代理)；

2）这个BinderProxy的访问需要经过JNI层的Android_util_binder类将请求转交给native的BpBinder（p代表代理的意思）；

3）BpBinder会通过ioctl将请求转交给Binder驱动设备；

4）在服务端注册了一个监听请求的回调函数，一旦驱动层收到BpBinder 的调用，就会回调BBInder注册的回调函数，于是，就将请求转给了BBinder；

5）BBinder拿到请求后，会进行一些数据的处理，然后通过JNI将请求转交给了java类；

6）java层会通过aidl中的函数将请求发送给Server端的实现者，由Server端通过stub 去调用相关的执行代码，并将结果通过类似的路径返回。

以上只是一个大概的流程，具体的实现流程我们将在后面的分析中更详细的介绍。然而要全面的分析这个流程最好的案例便是ServiceManager。

2. ServcieManager场景

在跨进程通信过程中，比如我们与AMS通信，首先需要拿到AMS的binder，然而，AMS的binder往往是通过ServcieManager获取的，因此会有代码：

```
ServiceManager.getService(Context.ACTIVITY_SERVICE)
```

以上代码将调用下面的代码

```
public static IBinder getService(String name) {
    try {
        IBinder service = sCache.get(name);
        if (service != null) {
            return service;
        } else {
            return Binder.allowBlocking(rawGetService(name));
        }
    } catch (RemoteException e) {
        Log.e(TAG, "error in getService", e);
    }
    return null;
}
```

关键函数：Binder.allowBlocking(rawGetService(name))

```
public static IBinder allowBlocking(IBinder binder) {
    try {
        if (binder instanceof BinderProxy) {
            ((BinderProxy) binder).mWarnOnBlocking = false;
        } else if (binder != null && binder.getInterfaceDescriptor() != null
                && binder.queryLocalInterface(binder.getInterfaceDescriptor()) == null) {
            Log.w(TAG, "Unable to allow blocking on interface " + binder);
        }
    } catch (RemoteException ignored) {
    }
    return binder;
}
```

关键函数中 Binder.allowBlocking并没有什么实质性的有价值的代码，所以所有的核心代码在rawGetService中。

```
private static IBinder rawGetService(String name) throws RemoteException {
    ------ 省略
    final IBinder binder = getIServiceManager().getService(name);

    ------ 省略
    return binder;
}
```

这里的逻辑是，先通过getIServiceManager()获取到IServiceManager对象，然后再通过这个IServiceManager对象获取到一个IBinder。

注意，此处有两个IPC

1. 获取到IServiceManager
2. 通过IServiceManager获取到该name的Service的Binder的IPC

由于本文本文重在探索java到native的逻辑，我们且只看第一个。

关于第二个： 源码中实现IServiceManager的类是ServiceManagerNative 有兴趣的读者可以自己探索下，或者关注笔者后面的文章

ServiceManager.getIServiceManager()

```
private static IServiceManager getIServiceManager() {
    if (sServiceManager != null) {
        return sServiceManager;
    }
    // Find the service manager
    sServiceManager = ServiceManagerNative
            .asInterface(Binder.allowBlocking(BinderInternal.getContextObject()));
    return sServiceManager;
}
```

到这，可以看到返回的其实是ServiceManagerNative.asInterface()的返回值。

对AIDL有所了解就知道，asInterface()的内容大概如下：

这个方法属于aidl接口的内部类 Stub。 在同一进程中，就会直接返回Stub，如果在另一个进程中调用，就会返回将这个ibinder封装好的Proxy对象。 对于上面的逻辑不太清楚的可以先学习前面的章节AIDL。于是，虽然没有看ServiceManagerNative的源码，但是其逻辑已经很清晰了。我们只需要关注IBinder的来源就可以了，也就是BinderInternal.getContextObject()。

BinderInternal.getContextObject()代码如下：

```
public static final native IBinder getContextObject();
```

至此，可以发现IBinder对象其实是从native层获取到的。

JNI层代码如下：

android_util_Binder.android_os_BinderInternal_getContextObject()代码如下

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

这里先通过ProcessState获取到了一个native 层的IBinder强引用，也就是一个BpBinder。 然后将这个native层的IBinder强引用传入javaObjectForIBinder()方法，最终封装成java层的IBinder然后返回。此处先不深究ProcessState的逻辑，整个native层的binder有自己的一整套的逻辑，后面的文章会继续探索。我们可以先稍微看下javaObjectForIBinder()的大概逻辑。 android_util_Binder.javaObjectForIBinder()

```
// If the argument is a JavaBBinder, return the Java object that was used to create it.
// Otherwise return a BinderProxy for the IBinder. If a previous call was passed the
// same IBinder, and the original BinderProxy is still alive, return the sameBinderProxy.
// 将一个BpBinder对象(这是native中的类型)转换成java中的类型
jobject javaObjectForIBinder(JNIEnv* env, const sp<IBinder>& val)
{
    // N.B. This function is called from a @FastNative JNI method, so don't take locks
    //around calls to Java code or block the calling thread for a long time for any
    //reason.

    if (val == NULL) return NULL;

    //JavaBBinder返回true，其他类均返回flase
    if (val->checkSubclass(&gBinderOffsets)) {
        // It's a JavaBBinder created by ibinderForJavaObject. Already has Java object.
        jobject object = static_cast<JavaBBinder*>(val.get())->object();
        LOGDEATH("objectForBinder %p: it's our own %p!\n", val.get(), object);
        return object;
    }

    BinderProxyNativeData* nativeData = new BinderProxyNativeData();
    nativeData->mOrgue = new DeathRecipientList;
    nativeData->mObject = val;

    // 核心代码：运用反射创建一个BinderProxy对象
    jobject object = env->CallStaticObjectMethod(gBinderProxyOffsets.mClass,
            gBinderProxyOffsets.mGetInstance, (jlong) nativeData, (jlong) val.get());
    if (env->ExceptionCheck()) {
        // In the exception case, getInstance still took ownership of nativeData.
        return NULL;
    }
    BinderProxyNativeData* actualNativeData = getBPNativeData(env, object);
    //如果object是刚刚新建出来的BinderProxy
    if (actualNativeData == nativeData) {
         //处理proxy计数
        // Created a new Proxy
        uint32_t numProxies = gNumProxies.fetch_add(1, std::memory_order_relaxed);
        uint32_t numLastWarned = gProxiesWarned.load(std::memory_order_relaxed);
        if (numProxies >= numLastWarned + PROXY_WARN_INTERVAL) {
            // Multiple threads can get here, make sure only one of them gets to
            // update the warn counter.
```

```
            if (gProxiesWarned.compare_exchange_strong(numLastWarned,
                        numLastWarned + PROXY_WARN_INTERVAL, std::memory_order_relaxed))
            {
                ALOGW("Unexpectedly many live BinderProxies: %d\n", numProxies);
            }
        }
    } else {
        delete nativeData;
    }

    return object;  //object 是反射参数的java的 BinderProxy
}
```

上面的函数看起来复杂，其实功能就是将一个BpBinder对象(这是native中的类型)转换成java中的类型，中间采用了反射技术而已。

核心代码进行说明：

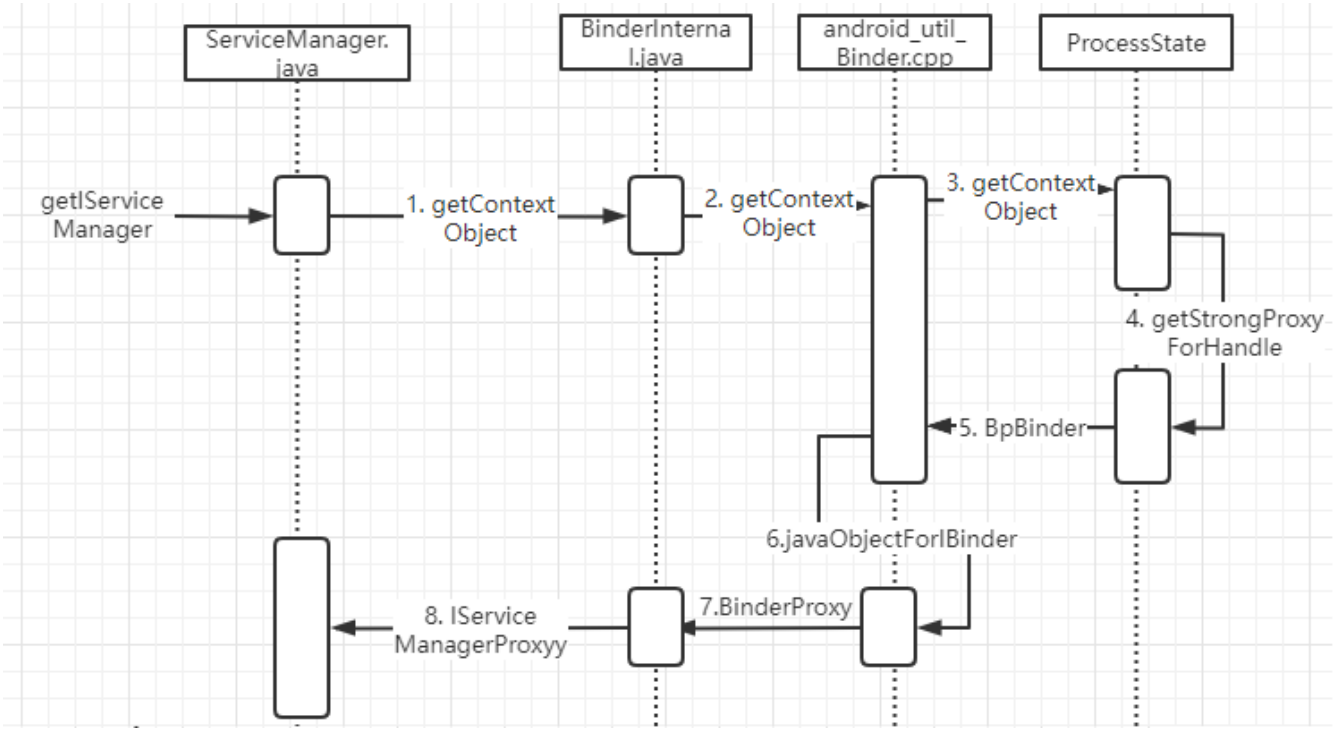
```
jobject object = env->CallStaticObjectMethod(gBinderProxyOffsets.mClass,
            gBinderProxyOffsets.mGetInstance, (jlong) nativeData, (jlong) val.get());
```

`gBinderProxyOffsets`是什么?

```
static struct binderproxy_offsets_t
{
    // Class state.
    jclass mClass;
    jmethodID mGetInstance;
    jmethodID mSendDeathNotice;

    // Object state.
    //指向BinderProxyNativeData的指针
    jfieldID mNativeData;  // Field holds native pointer to BinderProxyNativeData.
} gBinderProxyOffsets;

const char* const kBinderProxyPathName = "android/os/BinderProxy";

static int int_register_android_os_BinderProxy(JNIEnv* env)
{
    ...
    jclass clazz = FindClassOrDie(env, kBinderProxyPathName);
    gBinderProxyOffsets.mClass = MakeGlobalRefOrDie(env, clazz);
    gBinderProxyOffsets.mGetInstance = GetStaticMethodIDOrDie(env, clazz, "getInstance",
            "(JJ)Landroid/os/BinderProxy;");
    gBinderProxyOffsets.mSendDeathNotice =
            GetStaticMethodIDOrDie(env, clazz, "sendDeathNotice",
                                   "
(Landroid/os/IBinder$DeathRecipient;Landroid/os/IBinder;)V");
    gBinderProxyOffsets.mNativeData = GetFieldIDOrDie(env, clazz, "mNativeData", "J");
    ...
}
```

可以看到，`gBinderProxyOffsets`实际上是一个用来记录一些`java`中对应类、方法以及字段的结构体，用于从`native`层调用`java`层代码，而通过int_register_android_os_BinderProxy，我们知道，binderproxy_offsets_t中的mClass字段就是 BInderProxy，而mGetInstance 就是BInderProxy.java 中getInstance方法。因此核心代码创建的是一个BinderProxy对象。

具体的执行流程如下图所示：



第一大步：为了获取ServcieManager 的IServiceManager，首先要ServcieManager进程创建一个底层的Binder，所以会有android_os_BinderInternal_getContextObject也就是第2步，第2步会在ProcessState::self()中初始化Binder驱动，然后再执行第3步；

第二大步：上图中第3步会调用到第4步，在第4步getStrongProxyForHandle代码如下：

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;

    AutoMutex _l(mLock);

    //查找或建立handle对应的handle_entry
    handle_entry* e = lookupHandleLocked(handle);

    if (e != nullptr) {
        // We need to create a new BpBinder if there isn't currently one, OR we
        // are unable to acquire a weak reference on this current one.  The
        // attemptIncWeak() is safe because we know the BpBinder destructor will always
        // call expungeHandle(), which acquires the same lock we are holding now.
        // We need to do this because there is a race condition between someone
        // releasing a reference on this BpBinder, and a new reference on its handle
        // arriving from the driver.
        IBinder* b = e->binder;
        if (b == nullptr || !e->refs->attemptIncWeak(this)) {
            if (handle == 0) {
```

```
                // Special case for context manager...
                // The context manager is the only object for which we create
                // a BpBinder proxy without already holding a reference.
                // Perform a dummy transaction to ensure the context manager
                // is registered before we create the first local reference
                // to it (which will occur when creating the BpBinder).
                // If a local reference is created for the BpBinder when the
                // context manager is not present, the driver will fail to
                // provide a reference to the context manager, but the
                // driver API does not return status.
                //
                // Note that this is not race-free if the context manager
                // dies while this code runs.
                //
                // TODO: add a driver API to wait for context manager, or
                // stop special casing handle 0 for context manager and add
                // a driver API to get a handle to the context manager with
                // proper reference counting.
                //当handle为ServiceManager的特殊情况
                //需要确保在创建Binder引用之前，context manager已经被binder注册
                //需要先确保邋ServcieManager活着
                Parcel data;
                status_t status = IPCThreadState::self()->transact(
                        0, IBinder::PING_TRANSACTION, data, nullptr, 0);
                if (status == DEAD_OBJECT)
                   return nullptr;
            }

            //创建BpBinder并保存下来以便后面再次查找
            b = BpBinder::create(handle);
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
            result = b;
        } else {
            // This little bit of nastyness is to allow us to add a primary
            // reference to the remote proxy when this team doesn't have one
            // but another team is sending the handle to us.
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }
    return result;
}
```

其中会执行一个BpBinder::create(handle)，此处会创建一个BpBinder对象，并且将BpBinder对象赋值给result对象并返回result，也就是返回BpBinder对象。这个BpBinder对象一路返回，最终在android_util_Binder.cpp中的android_os_BinderInternal_getContextObject函数中会执行下面的代码：

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

也就是说b的值是BpBinder，然后BpBinder赋值给了javaObjectForIBinder作为参数。

第三大步：BpBinder需要返回给java层Client端使用，所以此时的封装就是将BpBinder封装成为Java层的BinderProxy对象。因此在Client端得到的IServiceManager 其实是BinderProxy类的子类的对象。

所以，BinderInternal.getContextObject()，返回的是一个层层封装的类的实例，具体来说，是Native层的BpBinder对象被封装成为BinderProxy对象并返回。

3. 出现了几个陌生的概念：BinderProxy，BpBinder，BBinder，IBinder

下面我们一一的介绍这些概念。

3.1 IBinder

```
// IBinder从Refbase继承而来，一提供强弱指针计数能力
class IBinder : public virtual RefBase
{
public:
    enum {
        FIRST_CALL_TRANSACTION  = 0x00000001,
        LAST_CALL_TRANSACTION   = 0x00ffffff,

        PING_TRANSACTION        = B_PACK_CHARS('_','P','N','G'),
        DUMP_TRANSACTION        = B_PACK_CHARS('_','D','M','P'),
        INTERFACE_TRANSACTION   = B_PACK_CHARS('_', 'N', 'T', 'F'),
        SYSPROPS_TRANSACTION    = B_PACK_CHARS('_', 'S', 'P', 'R'),

        // Corresponds to TF_ONE_WAY -- an asynchronous call.
        FLAG_ONEWAY             = 0x00000001
    };

                          IBinder();

    // 根据descriptor查询相应的IInterface对象
    virtual sp<IInterface>  queryLocalInterface(const String16& descriptor);

    // 获取descriptor描述符
    virtual const String16& getInterfaceDescriptor() const = 0;

    virtual bool            isBinderAlive() const = 0;
    virtual status_t        pingBinder() = 0;
    virtual status_t        dump(int fd, const Vector<String16>& args) = 0;

    // transact binder通信函数
    virtual status_t        transact(   uint32_t code,
                                        const Parcel& data,
                                        Parcel* reply,
```

```
                                        uint32_t flags = 0) = 0;

    // 死亡通知相应类
    class DeathRecipient : public virtual RefBase
    {
    public:
        virtual void binderDied(const wp<IBinder>& who) = 0;
    };


    // 如其名，用于注册Binder用的
virtual status_t        linkToDeath(const sp<DeathRecipient>& recipient,
                                    void* cookie = NULL,
                                    uint32_t flags = 0) = 0;

    // 撤销用之前注册的死亡通知函数
virtual status_t        unlinkToDeath(  const wp<DeathRecipient>& recipient,
                                        void* cookie = NULL,
                                        uint32_t flags = 0,
                                        wp<DeathRecipient>* outRecipient = NULL) = 0;

    virtual bool            checkSubclass(const void* subclassID) const;

    typedef void (*object_cleanup_func)(const void* id, void* obj, void* cleanupCookie);

    virtual void            attachObject(   const void* objectID,
                                            void* object,
                                            void* cleanupCookie,
                                            object_cleanup_func func) = 0;
    virtual void*           findObject(const void* objectID) const = 0;
    virtual void            detachObject(const void* objectID) = 0;

    // 返回服务端的binder引用
virtual BBinder*        localBinder();
    // 放回客户端的binder引用
virtual BpBinder*       remoteBinder();

protected:
    virtual          ~IBinder();

private:
};
```

对于IBinder中的方法，基本都是没有实现的，这些方法的实现都交给继承它的子类来实现，那下面直接看BpBinder的内容。

3.2 BpBinder

BpBinder和BBinder都是Android中与Binder通信相关的代表，他们都是从IBinder中继承而来的。其中BpBinder是客户端用来与Server交互的代理类，BBinder则是和proxy相对的一端，它是proxy交互的目的端。如果说Proxy代表客户端，那么BBinder就代表这服务端。这里BpBinder和BBinder是一一对应的，即某个BpBinder只能和对应的BBinder交互。

```
class BpBinder : public IBinder
{
public:
                        BpBinder(int32_t handle);

    inline  int32_t     handle() const { return mHandle; }

    virtual const String16&    getInterfaceDescriptor() const;
    virtual bool        isBinderAlive() const;
    virtual status_t    pingBinder();
    virtual status_t    dump(int fd, const Vector<String16>& args);

    virtual status_t    transact(   uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);

    virtual status_t    linkToDeath(const sp<DeathRecipient>& recipient,
                                    void* cookie = NULL,
                                    uint32_t flags = 0);
    virtual status_t    unlinkToDeath(  const wp<DeathRecipient>& recipient,
                                        void* cookie = NULL,
                                        uint32_t flags = 0,
                                        wp<DeathRecipient>* outRecipient = NULL);

    virtual void        attachObject(   const void* objectID,
                                        void* object,
                                        void* cleanupCookie,
                                        object_cleanup_func func);
    virtual void*       findObject(const void* objectID) const;
    virtual void        detachObject(const void* objectID);

    virtual BpBinder*   remoteBinder();

            status_t    setConstantData(const void* data, size_t size);
            void        sendObituary();

    // 对象管理
    class ObjectManager
    {
    public:
                    ObjectManager();
                    ~ObjectManager();

        void        attach( const void* objectID,
                            void* object,
                            void* cleanupCookie,
                            IBinder::object_cleanup_func func);
        void*       find(const void* objectID) const;
        void        detach(const void* objectID);

        void        kill();
```

```
    private:
                    ObjectManager(const ObjectManager&);
        ObjectManager& operator=(const ObjectManager&);

        struct entry_t
        {
            void* object;
            void* cleanupCookie;
            IBinder::object_cleanup_func func;
        };

        KeyedVector<const void*, entry_t> mObjects;
    };

protected:
    virtual             ~BpBinder();
    virtual void        onFirstRef();
    virtual void        onLastStrongRef(const void* id);
    virtual bool        onIncStrongAttempted(uint32_t flags, const void* id);

private:
    const   int32_t             mHandle;

    struct Obituary {
        wp<DeathRecipient> recipient;
        void* cookie;
        uint32_t flags;
    };

            void                reportOneDeath(const Obituary& obit);
            bool                isDescriptorCached() const;

    mutable Mutex               mLock;
            // BpBinder代理端是否还存活
            volatile int32_t    mAlive;
            // 是否已发送过死亡通知
            volatile int32_t    mObitsSent;
            // 用Vector保存死亡通知信息
            Vector<Obituary>*   mObituaries;
            ObjectManager       mObjects;
            Parcel*             mConstantData;
    mutable String16            mDescriptorCache;
};
```

BpBinder的构造函数

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle)  // 将传入的handle值保存到mHandle成员变量里
    , mAlive(1)        // mAlive设置为1
    , mObitsSent(0)
    , mObituaries(NULL)
{
    ALOGV("Creating BpBinder %p handle %d\n", this, mHandle);

    extendObjectLifetime(OBJECT_LIFETIME_WEAK);
    IPCThreadState::self()->incWeakHandle(handle);
}
```

首先调用extendObjectLifetime将对象改为弱引用控制，通过IPCThreadState的incWeakHandle增加Binder Service的如引用计数值。IncWeakHandle如下：

```
void IPCThreadState::incWeakHandle(int32_t handle)
{
    LOG_REMOTEREFS("IPCThreadState::incWeakHandle(%d)\n", handle);
    mOut.writeInt32(BC_INCREFS);
    mOut.writeInt32(handle);
}
```

这一步只是将BC_INCREFS请求写到mOut中，还没有发送出去。

除了incWeakHandle函数外还有decWeakHandle，incStrongHandle和decStrongHandle与Binder协议中的其他命令对应起来。这些细节我们就先不分析了。

另外，transact函数是核心，因为，Client是需要通过transact将请求传递给Binder驱动的，所以transact是BpBinder的核心函数之一。

3.3 BBinder

```
class BBinder : public IBinder
{
public:
                        BBinder();

    virtual const String16& getInterfaceDescriptor() const;
    virtual bool        isBinderAlive() const;
    virtual status_t    pingBinder();
    virtual status_t    dump(int fd, const Vector<String16>& args);

    virtual status_t    transact(   uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);

    virtual status_t    linkToDeath(const sp<DeathRecipient>& recipient,
                                    void* cookie = NULL,
                                    uint32_t flags = 0);

    virtual status_t    unlinkToDeath(  const wp<DeathRecipient>& recipient,
```

```
                                        void* cookie = NULL,
                                        uint32_t flags = 0,
                                        wp<DeathRecipient>* outRecipient = NULL);

    virtual void        attachObject(   const void* objectID,
                                        void* object,
                                        void* cleanupCookie,
                                        object_cleanup_func func);
    virtual void*       findObject(const void* objectID) const;
    virtual void        detachObject(const void* objectID);

    virtual BBinder*    localBinder();

protected:
    virtual             ~BBinder();

    virtual status_t    onTransact( uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);

private:
                        BBinder(const BBinder& o);
            BBinder&    operator=(const BBinder& o);

    class Extras;

    atomic_uintptr_t    mExtras;  // should be atomic<Extras *>
            void*       mReserved0;
};
```

其中，isBinderAlive函数只是返回true，这很好理解，因为BBinder代表的是Binder通信中的服务端，服务端是否存活，服务端自己当然是知道的。同时linkToDeath和unlinkToDeath这两个函数都只是返回INVALID_OPERATION，表明这两个函数在服务端是不需要做什么的，毕竟这两个函数是用来注册死亡通知用的。

由于是服务端，所以是Client端向服务端发起请求，因此它的核心也有一个transact函数，只是transact的核心却是onTransact函数，只是在onTransact没有什么实际的实现，最终会由子类去具体实现具体的功能。

3.4 JavaBBinder

IBinder是BBinder的父类，BBinder是JavaBBinder的父类。

java层直接与native层交互的对象有两个——Binder对象与BinderProxy对象。

Binder对应“Binder在本进程”的场景，BinderProxy对应“Binder在其他进程”的场景。

native层javaBBinder与java层的Binder一一对应。

native层的BinderProxyNativeData与java层的BinderProxy一一对应。

在native层，gBinderProxyOffsets(binderproxy_offsets_t)存储了java层binderProxy的对象与需要调用的方法和属性。gBinderOffsets(binderproxy_offsets_t)存储了java层binder的对象与需要调用的方法和属性。

ibinderForJavaObject负责通过java的Binder或者BinderProxy对象，找到并返回native层的IBinder对象。

javaObjectForIBinder通过native层的IBinder对象，找到或者封装成java对象返回。

七：ServiceManager解析

ServiceManager在init进程启动之后启动，用来管理系统中的service注册、查找、通讯等。

1、注册：首先会检查是否有权限注册service，如果没有权限就直接返回不能注册；然后去检查该service是否已经注册过了，如果已经注册过，那就不能再注册；再判断内存是否够用。如果都没有问题，就会注册该service，加入到svcList中来，(在servicemanager中维护service信息的地方就是svcList，里面存了service的name和handler)。

2、查找：通过name从svcList找到对应等service，返回对应的handler

3、通讯：ServiceManager以类似Loop的工作方式不断从Binder设备中读取消息，发送给对应的service；若没有消息，则会进入等待状态，等待新消息到来再被激活；由于每个App只能打开一次Binder设备，做一次内存映射，所有需要使用binder驱动的线程共享这一资源，即共享同一个ServiceManager。

ServiceManager的启动

所有的系统服务都是需要在ServiceManager中进行注册的，而ServiceManager作为一个起始的服务，是通过init.rc来启动的。

```
//system\core\rootdir\init.rc
//启动的服务，这里是用的服务名称。服务名称是在对应的rc文件中注册并启动的
start servicemanager
```

具体的服务信息是在servicemanger.rc命名并定义的

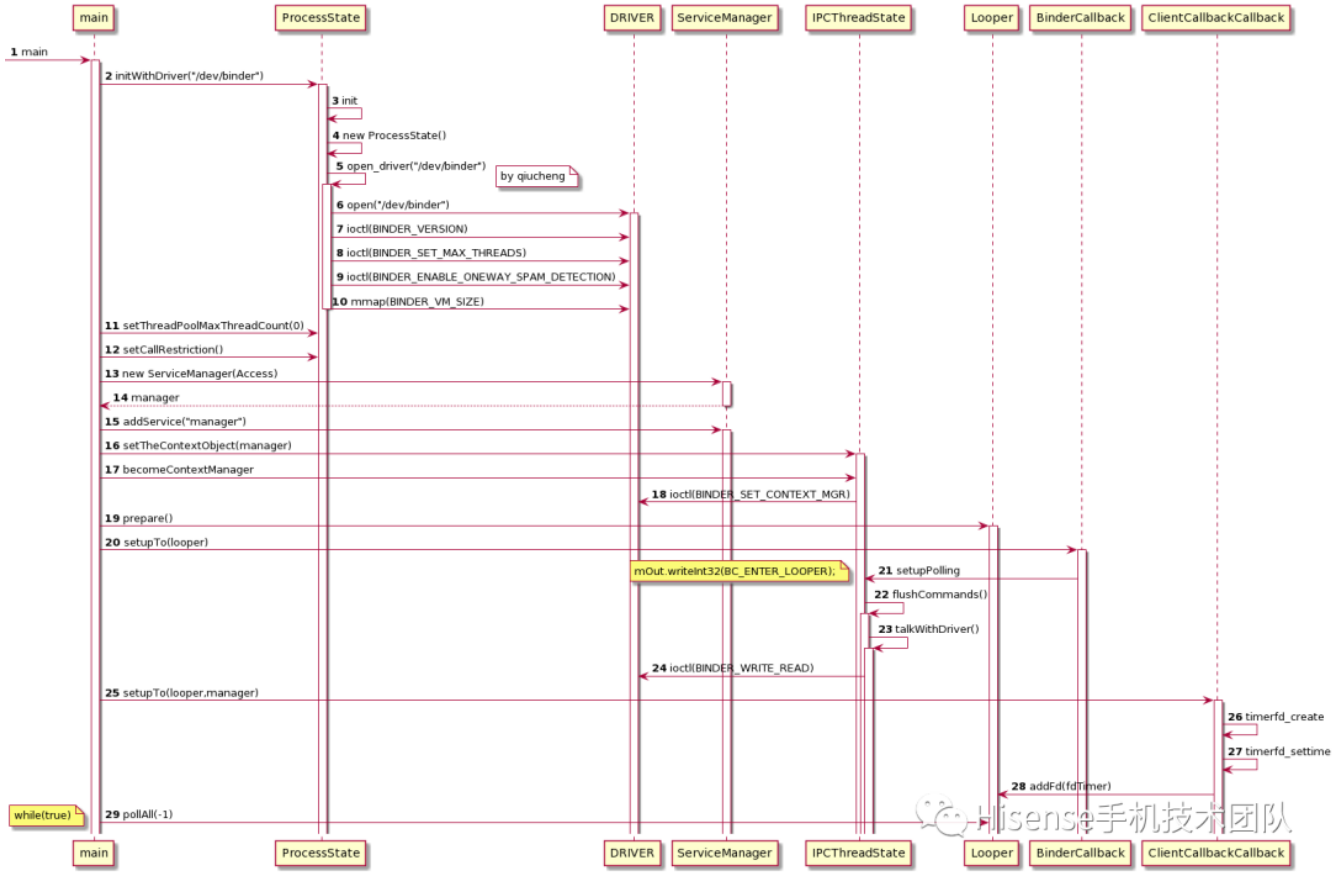
```
//\frameworks\native\cmds\servicemanager\servicemanager.rc
service servicemanager /system/bin/servicemanager
    class core animation
    user system  //说明以用户system身份运行
    group system readproc
    //说明servicemanager是系统中的关键服务，
    //关键服务是不会退出的，如果退出了，系统就会重启，当系统重启时就会启动用onrestart关键字修饰的进程，
    //比如zygote、media、surfaceflinger等等。
   critical
    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
    onrestart restart thermalservice
    ..
```

servicemanager的入口函数在frameworks\native\cmds\servicemanager\main.cpp中

代码如下：

```
int main(int argc, char** argv) {
    //根据上面的rc文件，argc == 1, argv[0] == "/system/bin/servicemanager"
    if (argc > 2) {
        LOG(FATAL) << "usage: " << argv[0] << " [binder driver]";
    }
    //此时，要使用的binder驱动为/dev/binder
    const char* driver = argc == 2 ? argv[1] : "/dev/binder";

    //初始化binder驱动
    sp<ProcessState> ps = ProcessState::initWithDriver(driver);
    ps->setThreadPoolMaxThreadCount(0);
    ps->setCallRestriction(ProcessState::CallRestriction::FATAL_IF_NOT_ONEWAY);

    //实例化ServiceManager，传入Access类用于鉴权
    sp<ServiceManager> manager = new ServiceManager(std::make_unique<Access>());
    //将自身作为服务添加
    if (!manager->addService("manager", manager, false /*allowIsolated*/,
IServiceManager::DUMP_FLAG_PRIORITY_DEFAULT).isOk()) {
        LOG(ERROR) << "Could not self register servicemanager";
    }
    //设置服务端Bbinder对象
    IPCThreadState::self()->setTheContextObject(manager);
    //设置成为binder驱动的context manager,成为上下文的管理者
    ps->becomeContextManager(nullptr, nullptr);

    //通过Looper epoll机制处理binder事务
    sp<Looper> looper = Looper::prepare(false /*allowNonCallbacks*/);
    //通知驱动BC_ENTER_LOOPER，监听驱动fd，有消息时回调到handleEvent处理binder调用
    BinderCallback::setupTo(looper);
    //服务的注册监听相关
    ClientCallbackCallback::setupTo(looper, manager);
    //无限循环等消息
    while(true) {
        looper->pollAll(-1);
    }

    // should not be reached
    return EXIT_FAILURE;
}
```

具体的逻辑整理成长下面的流程图：



如上图，启动流程的变动主要在进入循环的方式，Android 11 之前是通过binder_loop方法，而现在是通过looper。

和原来的 servicemanager 服务相比较，使用了 libbinder 后，代码更规范化，和其他 native 的服务风格一致了。

1）之前是直接 open、mmap 现在是借助 libbinder 2）之前是 binder_loop死 循环接收驱动的消息，现在是通过 looper 监听 fd 来handleEvent 3）之前的鉴权现在被独立到单独文件 Access.cpp **突然想起一个题目**，servicemanager 映射的虚拟内存有多大？现在的答案是和普通应用一样大：1M-2 页。

八：Framework层中的Binder 客户端通信以ServiceManager为例的源码分析

1. 获取ServiceManager对象

在前面的章节已经基本完成了对Binder的基本说明，前面讲解了binder通信的模型，Java层Binder与native层Binder的通信，如果要彻底掌握Binder的工作原理，我们还需要看Framework层以及应用层是如何具体的利用Binder机制完成通信的。因此，我们将从ServiceManager作为一个进程为例来说明如何通过Binder来具体完通信的。

在第六章中的ServcieManager场景这一小结里面我们介绍了下面的代码：

```
private static IServiceManager getIServiceManager() {
    if (sServiceManager != null) {
        return sServiceManager;
    }

    // Find the service manager
    sServiceManager = ServiceManagerNative
            .asInterface(Binder.allowBlocking(BinderInternal.getContextObject()));
    return sServiceManager;//ServiceManagerProxy
}
```

BinderInternal.getContextObject()，返回值是一个IBinder对象，它经历了几层封装，具体逻辑如下大家可以回顾一下：

首先是在Native层创建了一个BpBinder对象，然后BpBinder对象在JNI层被封装成为BinderProxy对象并返回，所以BinderInternal.getContextObject()返回的是BinderProxy对象。

然而当大家看到上面的Binder.allowBlocking的如下源码的时候，你会发现这其中并没有实质性的代码：

```
public static IBinder allowBlocking(IBinder binder) {
    try {
        if (binder instanceof BinderProxy) {
            ((BinderProxy) binder).mWarnOnBlocking = false;
        } else if (binder != null && binder.getInterfaceDescriptor() != null
                && binder.queryLocalInterface(binder.getInterfaceDescriptor()) == null) {
            Log.w(TAG, "Unable to allow blocking on interface " + binder);
        }
    } catch (RemoteException ignored) {
    }
    return binder;
}
```

接下来，我们再分析一下ServiceManagerNative.asInterface 函数

```
public static IServiceManager asInterface(IBinder obj) {
    if (obj == null) {
        return null;
    }

    // ServiceManager is never local
    return new ServiceManagerProxy(obj);
}
```

以上代码的核心逻辑在于将obj封装成为了ServiceManagerProxy对象，大家看一下它的代码

```
class ServiceManagerProxy implements IServiceManager {
    public ServiceManagerProxy(IBinder remote) {
        mRemote = remote;
        mServiceManager = IServiceManager.Stub.asInterface(remote);
    }
```

```
    public IBinder asBinder() {
        return mRemote;
    }

    @UnsupportedAppUsage
    public IBinder getService(String name) throws RemoteException {
        // Same as checkService (old versions of servicemanager had both methods).
        return mServiceManager.checkService(name);
    }

    public IBinder checkService(String name) throws RemoteException {
        return mServiceManager.checkService(name); // binder 调用
    }

    public void addService(String name, IBinder service, boolean allowIsolated, int
                           dumpPriority)throws RemoteException {
        mServiceManager.addService(name, service, allowIsolated, dumpPriority);
    }

    public String[] listServices(int dumpPriority) throws RemoteException {
        return mServiceManager.listServices(dumpPriority);
    }

    //非核心代码，一律省略
    /**
     * Same as mServiceManager but used by apps.
     *
     * Once this can be removed, ServiceManagerProxy should be removed entirely.
     */
    @UnsupportedAppUsage
    private IBinder mRemote;

    private IServiceManager mServiceManager;
}
```

以上代码其实就是ServiceManagerProxy通过手动的方案实现了IServiceManager，而IServiceManager是IServiceManager.aidl 生成的IServiceManager.java 接口，我们来看一下它生成出的`IServiceManager.Stub.asInterface`方法：

```
public static android.os.IServiceManager asInterface(android.os.IBinder obj)
{
    if ((obj == null)) {
        return null;
    }
    android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
    if (((iin != null) && (iin instanceof android.os.IServiceManager))) {
        return ((android.os.IServiceManager) iin);
    }
    return new android.os.IServiceManager.Stub.Proxy(obj);
}
```

这里我们传入的`IBinder`是`BinderProxy`，它的`queryLocalInterface`永远返回`null`，所以这里返回的是`IServiceManager.Stub.Proxy`对象。

小结

Native层的BpBinder对象被JNI层封装成为BinderProxy对象，然后BinderProxy对象又被封装成为IServiceManager.Stub.Proxy对象。

2.调用ServiceManager的getService方法

第一步：先调用IServiceManager.Stub.Proxy的getService方法

```
public android.os.IBinder getService(java.lang.String name) throws
android.os.RemoteException
{
    android.os.Parcel _data = android.os.Parcel.obtain();
    android.os.Parcel _reply = android.os.Parcel.obtain();
    android.os.IBinder _result;
    try {
        _data.writeInterfaceToken(DESCRIPTOR);
        _data.writeString(name);
        boolean _status = mRemote.transact(Stub.TRANSACTION_getService, _data, _reply,
                          0);
        _reply.readException();
        _result = _reply.readStrongBinder();
    }
    finally {
        _reply.recycle();
        _data.recycle();
    }
    return _result;
}
```

第二步：接下来会调用mRemote.transact

而mRemote则是BinderProxy对象，所以接下来执行下面的代码：

```
public boolean transact(int code, Parcel data, Parcel reply, int flags) throws
RemoteException {
    //检查Parcel大小
    Binder.checkParcel(this, code, data, "Unreasonably large binder buffer");
    ...
    //trace
    ...
    //Binder事务处理回调
    ...
    //AppOpsManager信息记录
    ...
    try {
        final boolean result = transactNative(code, data, reply, flags);

        if (reply != null && !warnOnBlocking) {
```

```
            reply.addFlags(Parcel.FLAG_IS_REPLY_FROM_BLOCKING_ALLOWED_OBJECT);
        }
        return result;
    } finally {
        ...
    }
}
```

首先，系统会通过checkParcel检测数据的格式和大小，`Android`默认设置了`Parcel`数据传输不能超过**800k**，当然，各个厂商是可以随意改动这里的代码的，如果超过了的话，便会调用`Slog.wtfStack`打印日志，需要注意的是，在当前进程不是系统进程并且系统也不是工程版本的情况下，这个方法是会结束进程的，所以在应用开发的时候，我们需要注意跨进程数据传输的大小，避免因此引发crash。

核心函数是调用transactNative方法，这是一个native方法，在frameworks/base/core/jni/android_util_Binder.cpp中实现。

```
static jboolean android_os_BinderProxy_transact(JNIEnv* env, jobject obj,
        jint code, jobject dataObj, jobject replyObj, jint flags)
{
    if (dataObj == NULL) {
        jniThrowNullPointerException(env, NULL);
        return JNI_FALSE;
    }

    Parcel* data = parcelForJavaObject(env, dataObj);
    if (data == NULL) {
        return JNI_FALSE;
    }
    Parcel* reply = parcelForJavaObject(env, replyObj);
    if (reply == NULL && replyObj != NULL) {
        return JNI_FALSE;
    }

    IBinder* target = getBPNativeData(env, obj)->mObject.get();
    if (target == NULL) {
        jniThrowException(env, "java/lang/IllegalStateException", "Binder has been
finalized!");
        return JNI_FALSE;
    }

    //log
    ...
    status_t err = target->transact(code, *data, reply, flags);
    //log
    ...

    if (err == NO_ERROR) {
        return JNI_TRUE;
    } else if (err == UNKNOWN_TRANSACTION) {
        return JNI_FALSE;
    }

    signalExceptionForError(env, obj, err, true /*canThrowRemoteException*/,
```

```
                                data->dataSize());
    return JNI_FALSE;
}
```

这里首先是获得`native`层对应的`Parcel`并执行判断，`Parcel`实际上功能是在`native`中实现的，`java`中的`Parcel`类使用`mNativePtr`成员变量保存了其对应`native`中的`Parcel`的指针，然后调用`getBPNativeData`函数获得`BinderProxy`在`native`中对应的`BinderProxyNativeData`，再通过里面的`mObject`域成员变量得到其对应的`BpBinder`，这个函数在之前分析`javaObjectForIBinder`的时候已经出现过了。

毫无疑问，getBPNativeData(env, obj)->mObject.get();是上面代码的核心之一，调用getBPNativeData函数获得BinderProxy在native中对应的BinderProxyNativeData，target 事实上是BpBinder。

第三步，BpBinder 的transact函数

此时就顺理成章的调用到了BpBinder的`transact`函数了

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
    //判断binder服务是否存活
    if (mAlive) {
        ...
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;

        return status;
    }
    return DEAD_OBJECT;
}
```

这里有一个`Alive`判断，可以避免对一个已经死亡的`binder`服务再发起事务，浪费资源，除此之外便是调用`IPCThreadState`的`transact`函数了

第四步：IPCThreadState的transact函数调用

路径：frameworks/native/libs/binder/IPCThreadState.cpp

还记得我们之前提到的`ProcessState`吗？`IPCThreadState`和它很像，`ProcessState`负责打开`binder`驱动并进行`mmap`映射，而`IPCThreadState`则是负责与`binder`驱动进行具体的交互

`IPCThreadState`也有一个`self`函数，与`ProcessState`的`self`不同的是，`ProcessState`是进程单例，而`IPCThreadState`是线程单例，感兴趣的可以看看它是怎么实现。

我们接着看它的ProcessState的`transact`函数:

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    LOG_ALWAYS_FATAL_IF(data.isForRpc(), "Parcel constructed for RPC, but being used with
```

```
binder.");

    status_t err;
    flags |= TF_ACCEPT_FDS;
    //log
    ...
    err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, nullptr);
    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }
    if ((flags & TF_ONE_WAY) == 0) {    //binder事务不为TF_ONE_WAY
        //当线程限制binder事务不为TF_ONE_WAY时
        if (UNLIKELY(mCallRestriction != ProcessState::CallRestriction::NONE)) {
            if (mCallRestriction == ProcessState::CallRestriction::ERROR_IF_NOT_ONEWAY) {
                //这个限制只是log记录
                ALOGE("Process making non-oneway call (code: %u) but is restricted.", code);
                CallStack::logStack("non-oneway call", CallStack::getCurrent(10).get(),
                    ANDROID_LOG_ERROR);
            } else /* FATAL_IF_NOT_ONEWAY */ {
                //这个限制会终止线程
                LOG_ALWAYS_FATAL("Process may not make non-oneway calls (code: %u).", code);
            }
        }
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        //log
        ...
    } else {  //binder事务为TF_ONE_WAY
        err = waitForResponse(nullptr, nullptr);
    }

    return err;
}
```

这个函数的重点在于`writeTransactionData`和`waitForResponse`，我们依次分析:

首先看writeTransactionData:

```
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;

    tr.target.ptr = 0; /* Don't pass uninitialized stack data to a remote process */
    //目标binder句柄值，ServiceManager为0
    tr.target.handle = handle;
    tr.code = code;
    tr.flags = binderFlags;
```

```
    tr.cookie = 0;
    tr.sender_pid = 0;
    tr.sender_euid = 0;

    const status_t err = data.errorCheck();
    if (err == NO_ERROR) {
        //数据大小
        tr.data_size = data.ipcDataSize();
        //数据区起始地址
        tr.data.ptr.buffer = data.ipcData();
        //传递的偏移数组大小
        tr.offsets_size = data.ipcObjectsCount()*sizeof(binder_size_t);
        //偏移数组的起始地址
        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);
    }
    //核心代码所在
    //这里为BC_TRANSACTION
    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof(tr));

    return NO_ERROR;
}
```

binder_transaction_data结构体(tr结构体）是向Binder驱动通信的数据结构，上面函数中，我们将`binder`请求码（这里为`BC_TRANSACTION`）和`binder_transaction_data`结构体依次写入到`mOut`中，为之后`binder_tansaction`做准备。

再看`waitForResponse`函数

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    uint32_t cmd;
    int32_t err;

    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;
        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: "
```

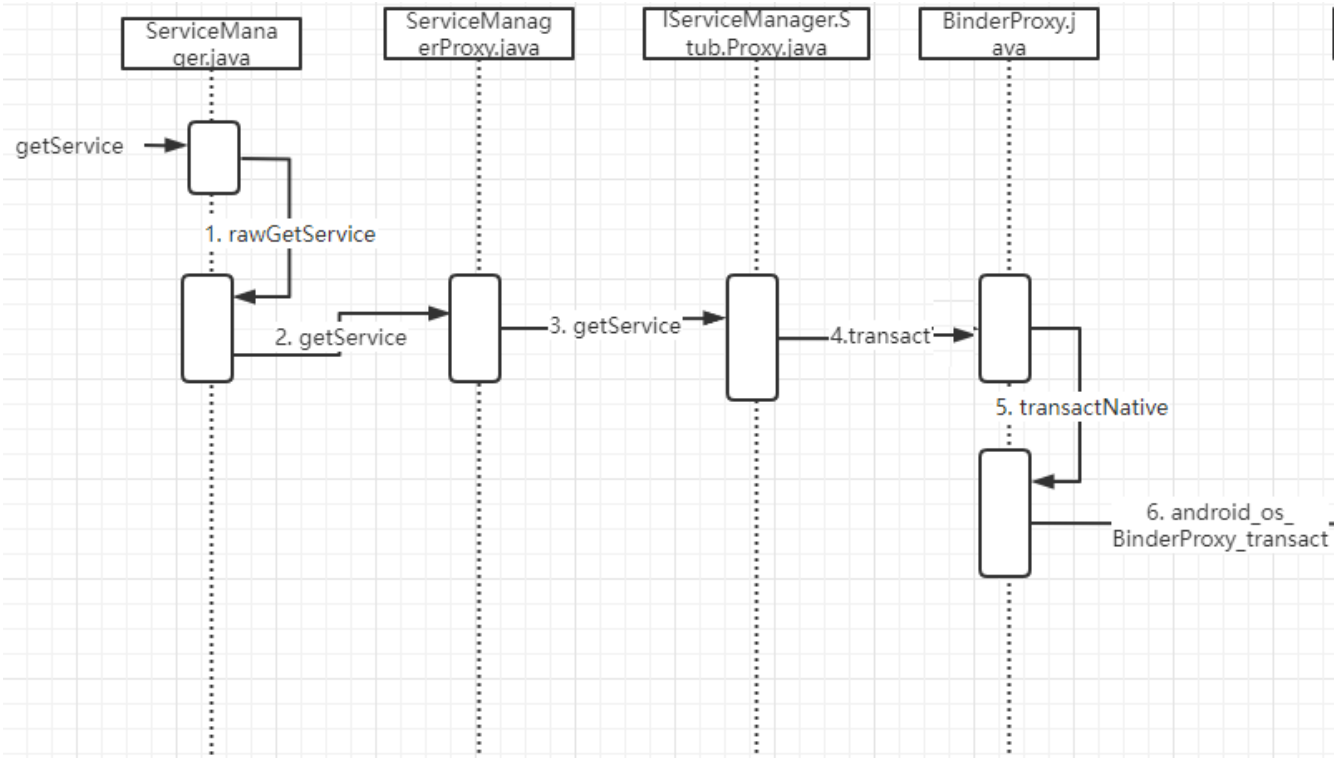
```
            << getReturnString(cmd) << endl;
    }

    switch (cmd) {
    case BR_ONEWAY_SPAM_SUSPECT:
        ...
    case BR_TRANSACTION_COMPLETE:
        //当TF_ONE_WAY模式下收到BR_TRANSACTION_COMPLETE直接返回，本次binder通信结束
        if (!reply && !acquireResult) goto finish;
        break;
    case BR_DEAD_REPLY:
        ...
    case BR_FAILED_REPLY:
        ...
    case BR_FROZEN_REPLY:
        ...
    case BR_ACQUIRE_RESULT:
        ...
    case BR_REPLY:
        {
            binder_transaction_data tr;
            err = mIn.read(&tr, sizeof(tr));
            ALOG_ASSERT(err == NO_ERROR, "Not enough command data for brREPLY");
            //失败直接返回
            if (err != NO_ERROR) goto finish;

            if (reply) {    //客户端需要接收replay
                if ((tr.flags & TF_STATUS_CODE) == 0) {    //正常reply内容
                    reply->ipcSetDataReference(
                        reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                        tr.data_size,
                        reinterpret_cast<const binder_size_t*>(tr.data.ptr.offsets),
                        tr.offsets_size/sizeof(binder_size_t),
                        freeBuffer /*释放缓冲区*/);
                } else {    //内容只是一个32位的状态码
                    //接收状态码
                    err = *reinterpret_cast<const status_t*>(tr.data.ptr.buffer);
                    //释放缓冲区
                    freeBuffer(nullptr,
                        reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                        tr.data_size,
                        reinterpret_cast<const binder_size_t*>(tr.data.ptr.offsets),
                        tr.offsets_size/sizeof(binder_size_t));
                }
            } else {    //客户端不需要接收replay
                //释放缓冲区
                freeBuffer(nullptr,
                    reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                    tr.data_size,
                    reinterpret_cast<const binder_size_t*>(tr.data.ptr.offsets),
                    tr.offsets_size/sizeof(binder_size_t));
                continue;
            }
```

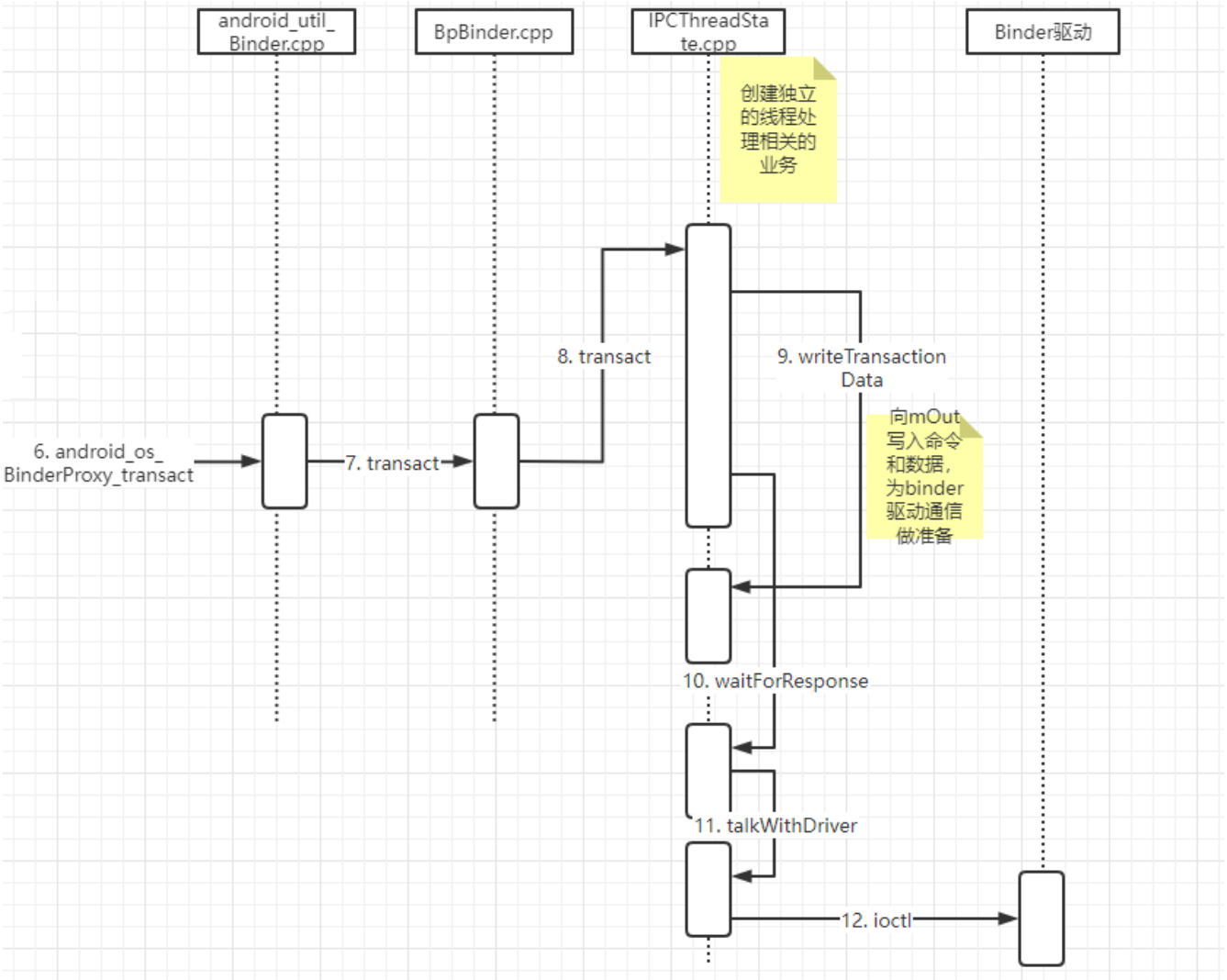
```
            }
            goto finish;
        default:
            //这里是binder服务端部分的处理，现在不需要关注
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
            break;
        }
    }

finish:
    if (err != NO_ERROR) {
        if (acquireResult) *acquireResult = err;
        if (reply) reply->setError(err);
        mLastError = err;
        logExtendedError();
    }

    return err;
}
```

这里有一个循环，正如函数名所描述，会一直等待到一整条 `binder` 事务链结束返回后才会退出这个循环，在这个循环的开头，便是 `talkWithDriver` 方法，在talkWithDriver 函数里面主要是调用Binder驱动的ioctl方法完成数据的传输，再次我们就不再深入分析了。

总结

本章总结了Client端通过ServiceManager的getService函数去获取对应服务的对象的过程，整体展示了如何将getService的命令从IServiceManager接口发送到Binder驱动的过程，下面我们用一张图来全面的解释一下整个流程。





上面流程最终是讲getService的请求通过层层调用转交给了Binder驱动，Binder驱动不是我们的研究重点，因此我们无需过多的关注。

总的来说，我们已经从Client的角度，通过ServiceManager提供的Binder去完成跨进程通信获取Service的过程，整个过程是围绕着getService函数的运行而执行，而且我们只是分析到了将getService请求发送给Binder驱动，至于驱动如何处理，目前我们无需关注。下一节我们将开始分析服务端收到getService的请求后如何处理的总流程。

九：Framework层中的Binder 服务端通信以ServiceManager为例的源码分析

在分析这一节之前，请大家先认真回顾阅读 ServiceManager解析章节。

1.ServiceManager进程 死循环

servicemanager进程的入口函数在frameworks\native\cmds\servicemanager\main.cpp中

代码如下：

```
int main(int argc, char** argv) {
    //根据上面的rc文件, argc == 1, argv[0] == "/system/bin/servicemanager"
    if (argc > 2) {
```

```
        LOG(FATAL) << "usage: " << argv[0] << " [binder driver]";
    }
    //此时，要使用的binder驱动为/dev/binder
    const char* driver = argc == 2 ? argv[1] : "/dev/binder";

    //初始化binder驱动
    sp<ProcessState> ps = ProcessState::initWithDriver(driver);
    ps->setThreadPoolMaxThreadCount(0);
    ps->setCallRestriction(ProcessState::CallRestriction::FATAL_IF_NOT_ONEWAY);

    //实例化ServiceManager，传入Access类用于鉴权
    sp<ServiceManager> manager = new ServiceManager(std::make_unique<Access>());
    //将自身作为服务添加
    if (!manager->addService("manager", manager, false /*allowIsolated*/,
IServiceManager::DUMP_FLAG_PRIORITY_DEFAULT).isOk()) {
        LOG(ERROR) << "Could not self register servicemanager";
    }
    //设置服务端Bbinder对象
    IPCThreadState::self()->setTheContextObject(manager);
    //设置成为binder驱动的context manager,成为上下文的管理者
    ps->becomeContextManager(nullptr, nullptr);

    //通过Looper epoll机制处理binder事务
    sp<Looper> looper = Looper::prepare(false /*allowNonCallbacks*/);
    //通知驱动BC_ENTER_LOOPER，监听驱动fd，有消息时回调到handleEvent处理binder调用
    BinderCallback::setupTo(looper);
    //服务的注册监听相关
    ClientCallbackCallback::setupTo(looper, manager);
    //无限循环等消息
    while(true) {
        looper->pollAll(-1);
    }

    // should not be reached
    return EXIT_FAILURE;
}
```

这里的`Looper`和我们平常应用开发所说的`Looper`是一个东西，本篇就不做过多详解了，只需要知道，可以通过`Looper::addFd`函数监听文件描述符，通过`Looper::pollAll`或`Looper::pollOnce`函数接收消息，消息抵达后会回调`LooperCallback::handleEvent`函数

```
class BinderCallback : public LooperCallback {
public:
    static sp<BinderCallback> setupTo(const sp<Looper>& looper) {
        sp<BinderCallback> cb = new BinderCallback;

        int binder_fd = -1;
        //向binder驱动发送BC_ENTER_LOOPER事务请求，并获得binder设备的文件描述符
        IPCThreadState::self()->setupPolling(&binder_fd);
        LOG_ALWAYS_FATAL_IF(binder_fd < 0, "Failed to setupPolling: %d", binder_fd);

        // Flush after setupPolling(), to make sure the binder driver
```

```
        // knows about this thread handling commands.
        IPCThreadState::self()->flushCommands();

        //监听binder文件描述符
        int ret = looper->addFd(binder_fd,
                                Looper::POLL_CALLBACK,
                                Looper::EVENT_INPUT,
                                cb,
                                nullptr /*data*/);
        LOG_ALWAYS_FATAL_IF(ret != 1, "Failed to add binder FD to Looper");

        return cb;
    }

    int handleEvent(int /* fd */, int /* events */, void* /* data */) override {
        //从binder驱动接收到消息并处理
        IPCThreadState::self()->handlePolledCommands();
        return 1;  // Continue receiving callbacks.
    }
};
```

在 `servicemanager` 进程启动的过程中调用了 `BinderCallback::setupTo` 函数，这个函数首先想 `binder` 驱动发起了一个 `BC_ENTER_LOOPER` 事务请求，获得 `binder` 设备的文件描述符，然后调用 `Looper::addFd` 函数监听 `binder` 设备文件描述符，这样当 `binder` 驱动发来消息后，就可以通过 `Looper::handleEvent` 函数接收并处理了

```
status_t IPCThreadState::setupPolling(int* fd)
{
    if (mProcess->mDriverFD < 0) {
        return -EBADF;
    }

    //设置binder请求码
    mOut.writeInt32(BC_ENTER_LOOPER);
    //检查写缓存是否有可写数据，有的话发送给binder驱动
    flushCommands();
    //赋值binder驱动的文件描述符
    *fd = mProcess->mDriverFD;
    pthread_mutex_lock(&mProcess->mThreadCountLock);
    mProcess->mCurrentThreads++;
    pthread_mutex_unlock(&mProcess->mThreadCountLock);
    return 0;
}
```

基于以上的分析，一旦Client端发送IPC请求，就会通过Binder驱动发送消息给服务端，而服务端则通过BinderCallback来接收消息，并做下一步的处理。

2.Binder驱动事务处理

`BinderCallback` 类重写了 `handleEvent` 函数，里面调用了 `IPCThreadState::handlePolledCommands` 函数来接收处理 `binder` 事务

```
status_t IPCThreadState::handlePolledCommands()
{
    status_t result;

    //当读缓存中数据未消费完时，持续循环
    do {
        result = getAndExecuteCommand();
    } while (mIn.dataPosition() < mIn.dataSize());

    //当我们清空执行完所有的命令后，最后处理BR_DECREFS和BR_RELEASE
    processPendingDerefs();
    flushCommands();
    return result;
}
```

读取并处理响应

这个函数的重点在`getAndExecuteCommand`，首先无论如何从binder驱动那里读取并处理一次响应，如果处理完后发现读缓存中还有数据尚未消费完，继续循环这个处理过程（理论来说此时不会再从`binder`驱动那里读写数据，只会处理剩余读缓存）

```
status_t IPCThreadState::getAndExecuteCommand()
{
    status_t result;
    int32_t cmd;

    //从binder驱动中读写数据（理论来说此时写缓存dataSize为0，也就是只读数据）
    result = talkWithDriver(/* true */);
    if (result >= NO_ERROR) {
        size_t IN = mIn.dataAvail();
        if (IN < sizeof(int32_t)) return result;
        //读取BR响应码
        cmd = mIn.readInt32();
        ...
        result = executeCommand(cmd);
        ...
    }

    return result;
}
```

处理响应

这里有很多线程等其他操作，我们不需要关心，我在这里把他们简化掉了，剩余的代码很清晰，首先从binder驱动中读取数据，然后从数据中读取出`BR`响应码，接着调用`executeCommand`函数继续往下处理

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    BBinder* obj;
    RefBase::weakref_type* refs;
    status_t result = NO_ERROR;
```

```
    switch ((uint32_t)cmd) {
    ...
    case BR_TRANSACTION_SEC_CTX:
    case BR_TRANSACTION:
        {
            binder_transaction_data_secctx tr_secctx;
            binder_transaction_data& tr = tr_secctx.transaction_data;

            if (cmd == (int) BR_TRANSACTION_SEC_CTX) {
                result = mIn.read(&tr_secctx, sizeof(tr_secctx));
            } else {
                result = mIn.read(&tr, sizeof(tr));
                tr_secctx.secctx = 0;
            }

            ALOG_ASSERT(result == NO_ERROR,
                "Not enough command data for brTRANSACTION");
            if (result != NO_ERROR) break;

            //读取数据到缓冲区
            Parcel buffer;
            buffer.ipcSetDataReference(
                reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                tr.data_size,
                reinterpret_cast<const binder_size_t*>(tr.data.ptr.offsets),
                tr.offsets_size/sizeof(binder_size_t), freeBuffer, this);

            ...

            Parcel reply;
            status_t error;
            //对于ServiceManager的binder节点来说，是没有ptr的
            if (tr.target.ptr) {
                // We only have a weak reference on the target object, so we must first try
to
                // safely acquire a strong reference before doing anything else with it.
                //对于其他binder服务端来说，tr.cookie为本地BBinder对象指针
                if (reinterpret_cast<RefBase::weakref_type*>(
                        tr.target.ptr)->attemptIncStrong(this)) {
                    error = reinterpret_cast<BBinder*>(tr.cookie)->transact(tr.code, buffer,
                            &reply, tr.flags);
                    reinterpret_cast<BBinder*>(tr.cookie)->decStrong(this);
                } else {
                    error = UNKNOWN_TRANSACTION;
                }

            } else {
                //对于ServiceManager来说，使用the_context_object这个BBinder对象
                error = the_context_object->transact(tr.code, buffer, &reply, tr.flags);
            }

            if ((tr.flags & TF_ONE_WAY) == 0) {
                LOG_ONEWAY("Sending reply to %d!", mCallingPid);
```

```
                if (error < NO_ERROR) reply.setError(error);
                //非TF_ONE_WAY模式下需要Reply
                sendReply(reply, 0);
            } else {
                ... //TF_ONE_WAY模式下不需要Reply，这里只打了些日志
            }
            ...
        }
        break;
        ...
    }

    if (result != NO_ERROR) {
        mLastError = result;
    }

    return result;
}
```

我们重点分析这个函数在`BR_TRANSACTION`下的case，其余的都删减掉

首先，这个函数从读缓存中读取了`binder_transaction_data`，我们知道这个结构体记录了实际数据的地址、大小等信息，然后实例化了一个`Parcel`对象作为缓冲区，从`binder_transaction_data`中将实际数据读取出来。

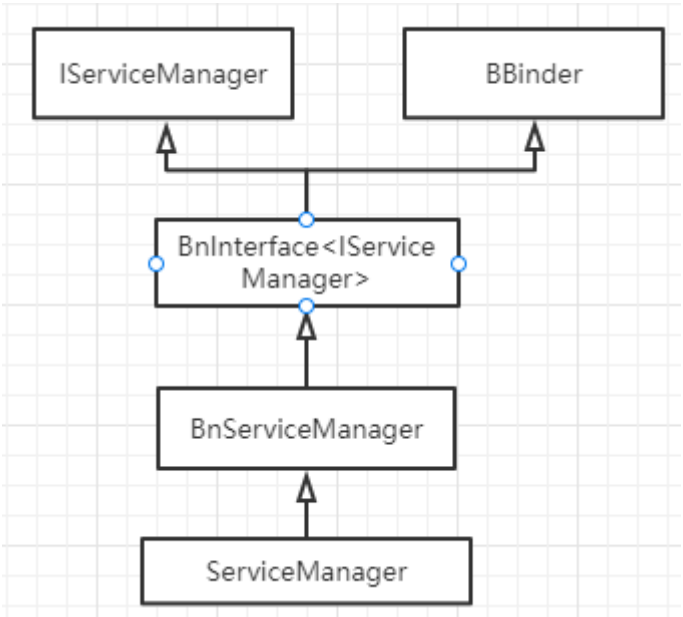
接着找到本地`BBinder`对象，对于`ServiceManager`来说就是之前在`main`函数中`setTheContextObject`的`ServiceManager`对象，而对于其他`binder`服务端来说，则是通过`tr.cookie`获取，然后调用`BBinder`的`transact`函数。理论上来说，the_context_object->transact(tr.code, buffer, &reply, tr.flags)，应该是执行ServiceManager中的transact函数，但是在ServiceManager中没有实现改函数，因此只能去父类BnServiceManager中去找transact函数，但是很不巧BnServiceManager中也不存在于是再找父类，只有在BBinder中存在transact函数，因此会执行到BBinder中的transact函数。

```
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    //确保从头开始读取数据
    data.setDataPosition(0);

    if (reply != nullptr && (flags & FLAG_CLEAR_BUF)) {
        //标记这个Parcel在释放时需要将内存中数据用0覆盖（涉及安全）
        reply->markSensitive();
    }

    status_t err = NO_ERROR;
    //这里的code是由binder客户端请求传递过来的
    //是客户端与服务端的一个约定
    //它标识了客户端像服务端发起的是哪种请求
    switch (code) {
        ...
        default:
            err = onTransact(code, data, reply, flags);
            break;
    }
```

```
    // In case this is being transacted on in the same process.
    if (reply != nullptr) {
        //设置数据指针偏移为0，这样后续读取数据便会从头开始
        reply->setDataPosition(0);
        if (reply->dataSize() > LOG_REPLIES_OVER_SIZE) {
            ALOGW("Large reply transaction of %zu bytes, interface descriptor %s, code %d",
                  reply->dataSize(), String8(getInterfaceDescriptor()).c_str(), code);
        }
    }

    return err;
}
```

为什么此处是BBinder的transact？而不是ServiceManager的transact函数呢？

我们看一下类的继承关系图：



上面的类，我们要说明一下：

1）frameworks\native\cmds\servicemanager\ServiceManager.cpp

2）IServiceManager.aidl ->从android29开始可以编译出对于的C++的接口代码，BnServiceManager类就是通过编译产生，代码结构如下：

```
class BnServiceManager : public ::android::BnInterface<IServiceManager> {
public:
  explicit BnServiceManager();
  ::android::status_t onTransact(uint32_t _aidl_code, const ::android::Parcel& _aidl_data,
::android::Parcel* _aidl_reply, uint32_t _aidl_flags) override;
};  // class BnServiceManager
```

3）frameworks\native\libs\binder\include\binder\IInterface.h中定义了BnInterface

```
template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder
{
public:
    virtual sp<IInterface>      queryLocalInterface(const String16& _descriptor);
    virtual const String16&     getInterfaceDescriptor() const;

protected:
    typedef INTERFACE           BaseInterface;
    virtual IBinder*            onAsBinder();
};
```

INTERFACE其实就是IServiceManager。

大家可以发现BBinder::transact中核心代码是onTransact(code, data, reply, flags)函数。

这个时候会执行到BnServiceManager中的onTransact中。

3.BnServiceManager中 onTransact

```
android::status_t BnServiceManager::onTransact(uint32_t _aidl_code, const ::android::Parcel&
_aidl_data, ::android::Parcel* _aidl_reply, uint32_t _aidl_flags) {
  ::android::status_t _aidl_ret_status = ::android::OK;
  switch (_aidl_code) {
  // 省略代码
  case ::android::IBinder::FIRST_CALL_TRANSACTION + 2 /* addService */:
  {
    ::std::string in_name;
    ::android::sp<::android::IBinder> in_service;
    bool in_allowIsolated;
    int32_t in_dumpPriority;
    if (!(_aidl_data.checkInterface(this))) {
      _aidl_ret_status = ::android::BAD_TYPE;
      break;
    }
    _aidl_ret_status = _aidl_data.readUtf8FromUtf16(&in_name);
    if (((_aidl_ret_status) != (::android::OK))) {
      break;
    }
    _aidl_ret_status = _aidl_data.readStrongBinder(&in_service);
    if (((_aidl_ret_status) != (::android::OK))) {
      break;
    }
    _aidl_ret_status = _aidl_data.readBool(&in_allowIsolated);
    if (((_aidl_ret_status) != (::android::OK))) {
      break;
    }
    _aidl_ret_status = _aidl_data.readInt32(&in_dumpPriority);
    if (((_aidl_ret_status) != (::android::OK))) {
      break;
    }
    ::android::binder::Status _aidl_status(addService(in_name, in_service,
in_allowIsolated, in_dumpPriority));  //addService 是核心
```

```
    _aidl_ret_status = _aidl_status.writeToParcel(_aidl_reply);
    if (((_aidl_ret_status) != (::android::OK))) {
      break;
    }
    if (!_aidl_status.isOk()) {
      break;
    }
  }
  break;
  //省略代码
  default:
  {
    _aidl_ret_status = ::android::BBinder::onTransact(_aidl_code, _aidl_data, _aidl_reply,
_aidl_flags);
  }
  break;
  }
  if (_aidl_ret_status == ::android::UNEXPECTED_NULL) {
    _aidl_ret_status =
::android::binder::Status::fromExceptionCode(::android::binder::Status::EX_NULL_POINTER).wri
teToParcel(_aidl_reply);
  }
  return _aidl_ret_status;
}
```

上面的代码很乱，但是没有关系，我们只需要看核心代码。此时，代码的核心是addService，而addService的实现类就是ServiceManager里面的addService函数。

4. ServiceManager.cpp 中addService

具体代码如下：

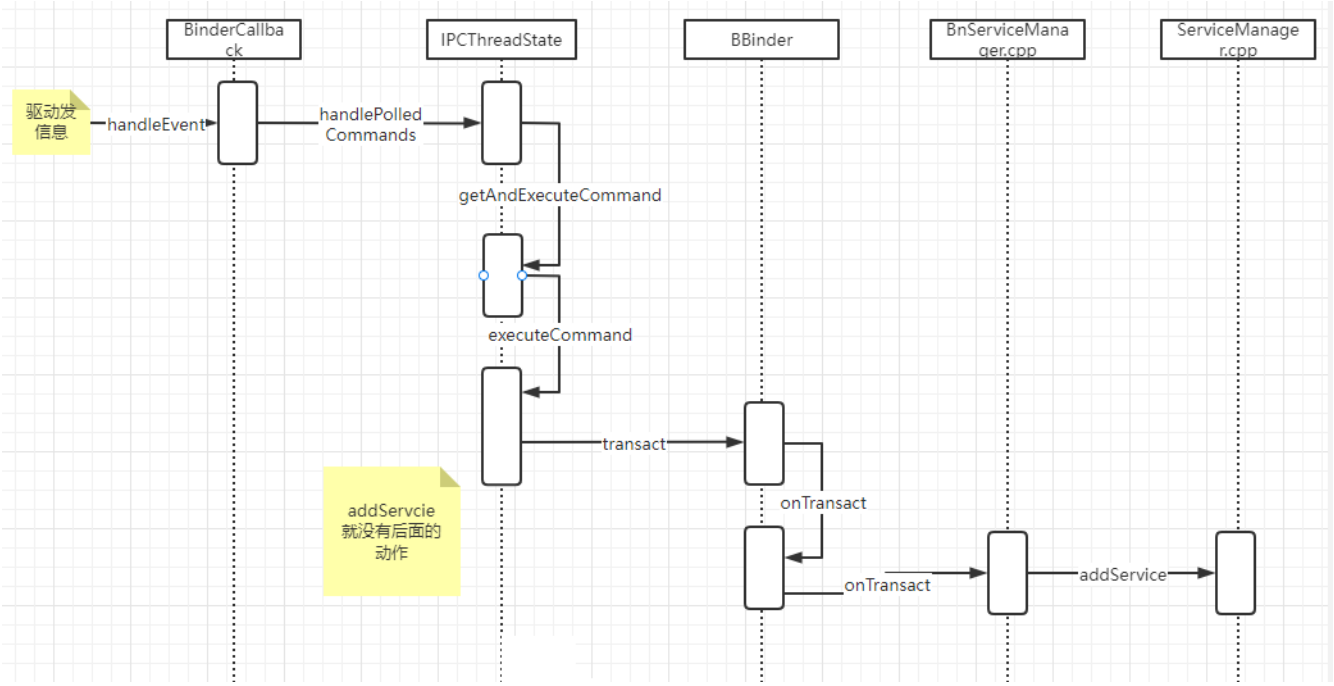
```
Status ServiceManager::addService(const std::string& name, const sp<IBinder>& binder, bool
allowIsolated, int32_t dumpPriority) {
    auto ctx = mAccess->getCallingContext();

    // apps cannot add services
    if (multiuser_get_app_id(ctx.uid) >= AID_APP) {
        return Status::fromExceptionCode(Status::EX_SECURITY);
    }

    if (!mAccess->canAdd(ctx, name)) {
        return Status::fromExceptionCode(Status::EX_SECURITY);
    }

    if (binder == nullptr) {
        return Status::fromExceptionCode(Status::EX_ILLEGAL_ARGUMENT);
    }

    if (!isValidServiceName(name)) {
        LOG(ERROR) << "Invalid service name: " << name;
        return Status::fromExceptionCode(Status::EX_ILLEGAL_ARGUMENT);
    }
```

```
#ifndef VENDORSERVICEMANAGER
    if (!meetsDeclarationRequirements(binder, name)) {
        // already logged
        return Status::fromExceptionCode(Status::EX_ILLEGAL_ARGUMENT);
    }
#endif  // !VENDORSERVICEMANAGER

    // implicitly unlinked when the binder is removed
    if (binder->remoteBinder() != nullptr && binder->linkToDeath(this) != OK) {
        LOG(ERROR) << "Could not linkToDeath when adding " << name;
        return Status::fromExceptionCode(Status::EX_ILLEGAL_STATE);
    }

    // Overwrite the old service if it exists
    mNameToService[name] = Service {
        .binder = binder,
        .allowIsolated = allowIsolated,
        .dumpPriority = dumpPriority,
        .debugPid = ctx.debugPid,
    };

    auto it = mNameToRegistrationCallback.find(name);
    if (it != mNameToRegistrationCallback.end()) {
        for (const sp<IServiceCallback>& cb : it->second) {
            mNameToService[name].guaranteeClient = true;
            // permission checked in registerForNotifications
            cb->onRegistration(name, binder);
        }
    }

    return Status::ok();
}
```

最终，服务的binder被封装成为了一个Service，并复制给了mNameToService数组进行存储。

5. 总结

我们将总的调度流程绘制成下面的流程图，请大家认真阅读



十：IPCThreadState 解析

1. IPCThreadState

在Android中，每个参与Binder通信的线程都会有一个IPCThreadState实例与之关联。我最开始接触到这个类是在BpBinder::transact方法中。

transact

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
    //判断binder服务是否存活
    if (mAlive) {
       //非核心代码

        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;

        return status;
    }

    return DEAD_OBJECT;
}
```

其就是调用的IPCThreadState::transact来完成的数据传输工作，其工作可以分为两步：

2. 发送数据

实际上，writeTransactionData只是将数据转换成binder_transaction_data结构并重新写入到IPCThreadState::mOut中。

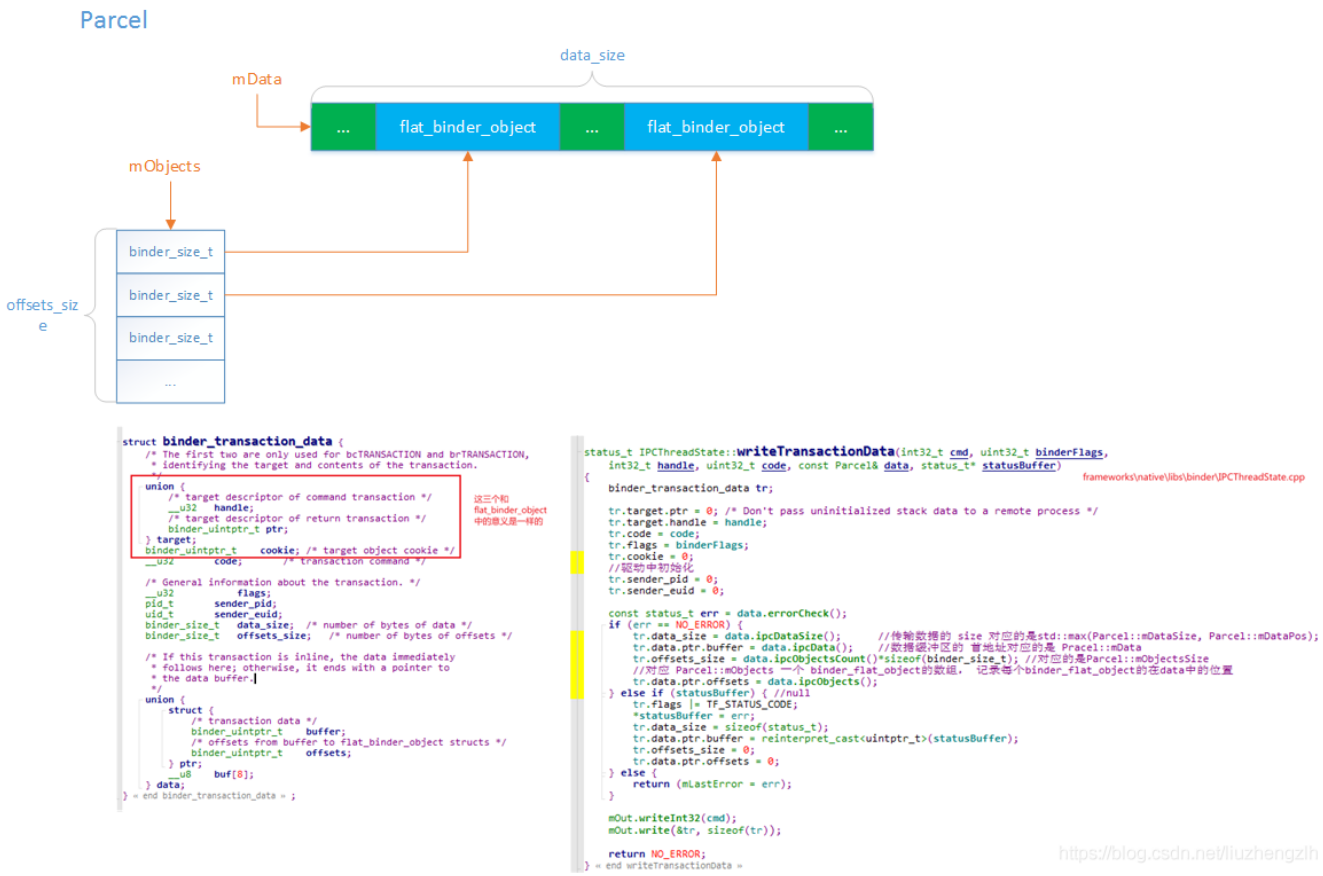
并没有真正的将数据发送出去。实际的发送操作是在waitForResponse中完成的。

3. 接收数据

TF_ONE_WAY表示的是单向通信，不需要对端回复。所以这里接收数据就多了几个判断分支。区别就是参数不一样。该函数必定需要被执行的，因为数据要发出去啊。。。

```
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;

    //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
    err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, nullptr);

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

    if ((flags & TF_ONE_WAY) == 0) {  //binder事务不为TF_ONE_WAY
        //省略非核心代码

        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }

    } else {
        err = waitForResponse(nullptr, nullptr);
    }

    return err;
}
```

4. writeTransactionData



我们创建一个Service，通过addService注册到ServiceManager时，就涉及到了传输一个IBinder对象的问题。libbinder将IBinder对象转换成一个flat_binder_object结构，虽然将该结构写入到Parcel中，最终在转换成binder_transaction_data。

5. waitForResponse

waitForResponse实际上又是通过调用IPCThreadState::talkWithDriver方法来完成数据接发工作。随后再处理我们接收到的数据。

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    uint32_t cmd;
    int32_t err;

    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;
        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) {
```

```
        //处理命令

        default:
            //这里是binder服务端部分的处理，现在不需要关注
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
            break;
        }
    }

finish:
    if (err != NO_ERROR) {
        if (acquireResult) *acquireResult = err;
        if (reply) reply->setError(err);
        mLastError = err;
    }

    return err;
}
```

我们需要处理如下这些cmd：

```
static const char *kReturnStrings[] = {
    "BR_ERROR",
    "BR_OK",
    "BR_TRANSACTION",
    "BR_REPLY",
    "BR_ACQUIRE_RESULT",
    "BR_DEAD_REPLY",
    "BR_TRANSACTION_COMPLETE",
    "BR_INCREFS",
    "BR_ACQUIRE",
    "BR_RELEASE",
    "BR_DECREFS",
    "BR_ATTEMPT_ACQUIRE",
    "BR_NOOP",
    "BR_SPAWN_LOOPER",
    "BR_FINISHED",
    "BR_DEAD_BINDER",
    "BR_CLEAR_DEATH_NOTIFICATION_DONE",
    "BR_FAILED_REPLY",
    "BR_TRANSACTION_SEC_CTX",
};
```

处理CMD的函数就是waitForResponse和executeCommand。这里就不贴代码了。

6. talkWithDriver

很容易理解，我们前面通过writeTransactionData,已经把数据写入到了binder_transaction_data中。
talkWithDriver就是调用ioctl(BINDER_WRITE_READ)完成真正的数据接发。

其他

```
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    //检查打开的binder设备的fd
    if (mProcess->mDriverFD < 0) {
        return -EBADF;
    }

    binder_write_read bwr;

    // Is the read buffer empty?
    const bool needRead = mIn.dataPosition() >= mIn.dataSize();

    //需要写的数据大小，这里的doReceive默认为true，如果上一次的数据还没读完，则不会写入任何内容
    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;

    bwr.write_size = outAvail;
    bwr.write_buffer = (uintptr_t)mOut.data();

    // This is what we'll read.
    if (doReceive && needRead) {
        //将read_size设置为读缓存可用容量
        bwr.read_size = mIn.dataCapacity();
        //设置读缓存起始地址
        bwr.read_buffer = (uintptr_t)mIn.data();
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }

    //非核心代码

    // Return immediately if there is nothing to do.
     //没有要读写的数据就直接返回
    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;

    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
    status_t err;
    //省略异常处理log

    if (err >= NO_ERROR) {
        //写数据被消费了
        if (bwr.write_consumed > 0) {
            //写数据没有被消费完
            if (bwr.write_consumed < mOut.dataSize())
                LOG_ALWAYS_FATAL("Driver did not consume write buffer. "
                                 "err: %s consumed: %zu of %zu",
                                 statusToString(err).c_str(),
                                 (size_t)bwr.write_consumed,
                                 mOut.dataSize());
            else {
                 //写数据消费完了，将数据大小设置为0，这样下次就不会再写数据了
                mOut.setDataSize(0);
```

```
                processPostWriteDerefs();
            }
        }
        //读到了数据
        if (bwr.read_consumed > 0) {
            //设置数据大小及数据指针偏移，这样后面就可以从中读取出来数据了
            mIn.setDataSize(bwr.read_consumed);
            mIn.setDataPosition(0);
        }
        return NO_ERROR;
    }

    return err;
}
```