**一、总体概述**

Binder采用的是CS服务的机制，提供服务的成为Server进程而访问服务的进程成为Client进程。同一个Server进程可以同时运行多个组件来向Client进程提供服务，这些组件**称为Service组件**；同时一个Client进程也可以同时向多个Service组件请求服务，每次请求都对应一个**Cilent组件**。Server进程和Client进程之间的通信要依靠运行在内核控件的Binder驱动程序来进行，Binder驱动程序向用户控件暴露了一个设备文件/dev/binder，使得应用程序进程可以间接的通过它来建立通信信道。Service组件在启动时，会将自己注册到一个ServiceManager中，以便Client组件通过ServiceManager来找到他，可以说ServiceManager是一个上下文的管理者。

**二、基础数据结构**

**binder\_work**：表示一个工作项

**binder\_node**：表示一个**Binder实体对象**，每一个**Service组件**在Binder驱动程序中都对应一个Binder实体对象，用来描述他在内核中的状态，Binder驱动程序通过强引计数和弱引用计数来维护它的生命周期。它是存在一个宿主进程中的，宿主进程中用一个红黑树来维护它内部所有的Binder实体对象，它可能会被多个Client组件所引用，所以它内部使用一个hash列表来保存引用了它的Binder引用对象。它内部的成员变量ptr和cookie用来**描述一个用户空间的Service组件。**

**binder\_ref**：表示一个**Binder引用对象**，每一个Client组件在Binder驱动程序中都对应一个Binder引用对象，用来描述它在内核中的状态，Binder驱动程序通过强引计数和弱引用计数来维护它的生命周期。**成员变量node是用来描述一个Binder引用对象所引用的Binder实体对象的。**binder\_ref内部有一个desc变量是一个描述符，在Client进程的用户空间中，一个Binder引用对象是使用一个句柄值来描述的。当Client进程的用户空间通过Binder驱动来访问一个Service组件时，只需要指定一个句柄值，通过它来找到Binder引用对象，在从上面提到的node中就可以找到对应的实体对象，然后就可以通过实体对象找到要访问的Service组件。

**binder\_buffer**：用来描述一个内核缓冲区，它是用来在进程间传递数据的。每一个使用Binder进程间通讯机制的进程在Binder驱动程序中都有一个内核缓冲区列表，用来保存Binder驱动程序为它所分配的内核缓冲区，而这个成员变量entry正是这个内核缓冲区列表中的一个节点。同时进程有使用了两个红黑树分别保存正在使用的内核缓冲区和空闲的内核缓冲区。

transaction和target\_node事用来描述一个内核缓冲区正在交给哪个事物以及哪一个Binder实体来使用。

**binder\_proc：**用来描述一个正在使用Binder进程间通信机制的进程。当一个进程调用函数open来打开设备文件/dev/binder是，Binder驱动程序就会为它创建一个binder\_proc结构体，并且将它保存在一个全局的hash列表中，而成员变量proc\_node就正好是该hash列表中的一个节点。进程打开了设备文件/dev/binder后，还必须调用mmap将它映射到进程的地址空间中来，实际上是请求Binder驱动程序为它分配一块内核缓冲区，以便可以用来进程间传输数据。这些内核缓冲区实际上有两个地址，其中一个是内核空间地址，另一个是用户空间地址，内核控件地址是在Binder驱动程序内部使用的，保存在buffer中而用户空间是在应用程序进程中使用的，保存在成员变量vma中，这两个地址相差一个固定的保存下来的值，给定一个就可以计算出另一个的大小。他们实际上都是虚拟内存，对应的真正的物理内存保存在成员变量pages中，只有分配一个物理页面不够时才分配新的。

成员变量buffer指向的是一个大的内核缓冲区，Binder驱动为了方便对它的管理，把它划分为若干小块，就是使用binder\_buffer结构来描述的，而buffers指向的是该列表的头部，列表中有正在使用的，即已经分配了物理页面也有未使用的，没有分配物理页面，分别组织在两个红黑树中。每一个使用了Binder进程间通信机制的进程都有一个线程池，用来处理进程间通信请求，binder\_proc成员变量threads指向了一个红黑树的根节点，以线程ID为关键字来组织一个进程的Binder线程池。

一个进程内部包含了一系列的Binder实体对象和Binder引用对象，进程使用三个红黑树来组织他们。

Binder驱动程序为进程分配内核缓冲区时，会为这个内核缓冲区创建一个文件描述符，进程可以通过这个文件描述符将内核缓冲区映射到自己的地址空间。.

**binder\_thread**：用来描述一个Binder线程池中的线程，进程可以通过调用ioctl函数将一个线程注册到Binder驱动程序中，当Binder驱动程序决定讲一个事物交给某一个Binder线程处理时，它会将该事物封装成一个binder\_transaction结构体，并且添加到线程结构体的成员变量transaction\_stack所描述的一个事物堆栈中。

**binder\_transaction**：用来描述一个进程间通信过程，这个过程又成为一个事务，Binder对事物的分发使用了栈遍历机制来进行任务的最优分发，从而达到**提高并发性的**效果。

**binder\_write\_read**：当应用程序进程打开驱动之后，需要通过IO控制函数ioctl来进一步与Binder驱动程序进行交互，因此驱动提供了一套控制命令来和应用程序通讯。此结构题就是用于描述进程间通讯过程中所发送的数据，这些数据包括输入数据和输出数据，write\_buffer指向一个用户空间缓冲区的地址，里面保存的是要传输到Binder驱动程序中的数据，read\_buffer也指向一个用户空间缓冲区的地址，里面保存的内容为Binder驱动程序返回给用户空间的进程间通信结果数据。

（3）Binder设备文件的打开过程——binder\_open函数

一个进程在使用Binder进行进程间通信之前，首先调用函数open打开设备文件来获取一个文件描述符，之后才能通过这个文件描述符来和Binder驱动程序交互。当使用open时会调用驱动中的binder\_open函数。

binder\_open函数主要为当前进程创建了一个binder\_proc结构体proc并且进行初始化，然后返回一个文件描述符给进程，后面通过这个文件描述符来执行mmap或者ioctl函数进行与驱动的交互是，驱动通过此描述符的private\_data来获取进程对应的binder\_proc结构体。

（4）Binder设备文件的内存映射过程——binder\_mmap函数

一个进程打开设备驱动后，还需要调用mmap函数将这个设备文件映射到进程的地址空间中，实际上调用了binder\_mmap函数。它的一个参数是结构体vm\_area\_struct，用来表示需要进行映射的用户地址空间，我们用vma来表示，mmap函数主要做下面几步：

1、检查需要映射的用户空间地址

首先如果它大于4M，就截取为4M，因为Binder为每一个进程规定最多4M的内核缓冲区来传输数据；并且用户地址空间只能读不能写，也不可以拷贝；

2、为进程分配内核缓冲区

使用get\_vm\_area在进程的内核地址空间分配一段大小和用户地址空间相同的空间，当然也是虚拟地址，并且记录了内核地址空间和用户地址空间的差值。用户进程通过用户地址空间来访问这块内核缓冲区的内容，而Binder驱动程序使用内核空间地址来访问这块缓冲区的内容，这是Binder进程间通讯的关键所在。它们都是连续的，只需要知道其中一个地址就可以计算出另一个的值。

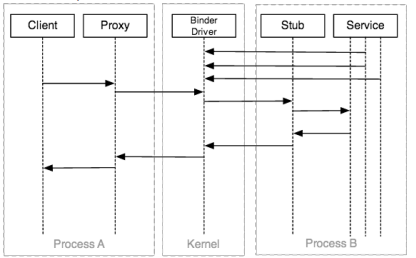
**三、Binder原理简述**

**（1）如何跨进程调用**

那么如何使得调用者能像上述一样简单地调用远程方法？毕竟两者存在于不同的进程空间里面。那么可以引入一个黑盒模块，用这个黑盒模块来帮我们完成其中的细节，这个模块也被称为Binder Driver。方法的跨进程调用受到了 Linux 进程隔离的限制，而解决方案就是将其置于所有进程都能共享的区域 -- Kernel，而 Binder Driver 提供的功能也就是让各进程使用内核空间，将进程中的地址和Kernel中的地址映射起来，其中Linux ioctl 函数实现了从用户空间转移到内核空间的功能。在 Binder Driver 的支持下，就能实现跨进程调用。

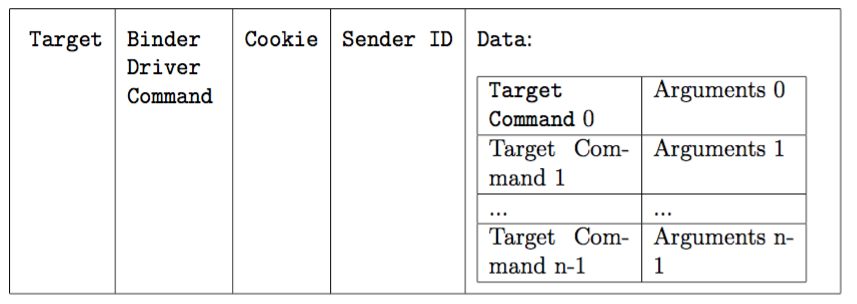
**（2）Client / Server 架构**

在设计的时候，Binder Framework 交互模型采用的是客户端/服务器模型。客户端需要调用远程服务的内容时， 会初始化一个连接，并等待服务器的返回，同时会block住自己。Binder Framework在客户端这边实现了一个代理，而在服务端，通过线程池的方式来响应请求。在如下图所示，A进程就是客户端，并且通过Proxy来完成对Binder Driver内核的交互。Process B是系统服务进程，在这个进程里面维护这多个Binder Thread，直到达到设置的线程上限。Proxy对象通过和Binder Driver进行交互，从而使得Binder Driver将信息传递到目标对象。从Android开发者的角度出发，Binder Framework提供的最方便的改进就是能像调用本地方法一样调用远程方法或对象。客户端的进程调用会在Server进程返回之前一直处于block的状况。在这种机制下，客户端就不必提供一个单独的线程模型和回调机制。（同步转异步简单，而异步变同步则很困难）



（3）**传递的数据格式**

在实现跨进程调用的时候，涉及到参数和命令的传递，得有一个合适的数据结构来表达需要远程执行的东西。



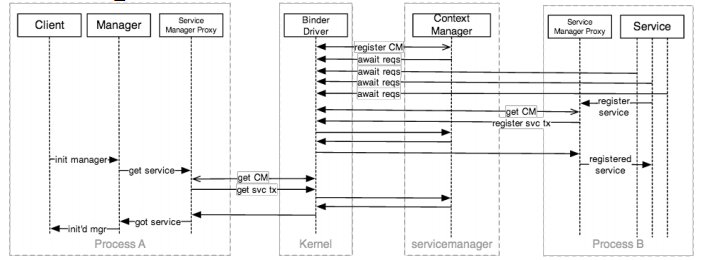
Target是指目标binder，Cookie这涵盖着一些内部信息，sender Id则包含了安全相关的信息，data则包含着一些数据的数组。每个数组的Entry是由相关的命令和参数组成的，这部分参数将传递给目标binder。

而这里面的Sender Id 则非常的重要，不仅可以起到唯一标示Binder的作用，还可以在跨进程的地方作为标记的作用，在接下来的文章里再详细说明。

**（4）Server Manager**

我们接触的服务很多，从Display到Location，从Audio到Wifi，如果我们和每一个服务都通过前面描述的方式进行交互，即便通过 Proxy 的方式也是非常的繁琐。而且在调用每个系统服务的时候，必须知道对应的系统服务的地址，而系统服务的地址出于安全性的考虑也不应该暴露出来。那么如何方便我们进行系统服务调用了？

Service Manager 就是来帮助我们解决这个问题。这是Binder Framework的一个特殊节点，也是第一个起点。其作为一个命令服务，起到了DNS的作用，使得可以通过名字的方式来查找相应的Binder接口。这很重要，因为客户端不应该知道远程服务的调用地址，如果知道了这势必会很不安全。每一个Binder需要将自己的名字和Binder Token交给Service Manager，客户端只需要知道服务的名字就可以。



**四、Binderc++实现与Binder的Stub/Proxy以及AIDL的关系**

我们创建的远程调用必须使用IBinder接口，它定义了跨进程通信的方式，为什么有这么多接口呢？实际上是分层的调用的缘故。

**第一层（最底层C++实现）：**C++层提供了进程间的通信，进程之间通过ServiceManager来通信，而通信的过程就是利用注册Binder给Manager，Client利用Binder名来进行调用，这是最底层的C++实现机制；

**第二层（Java层调用C++）：**Java层使用的也就是利用了C++服务的接口，也就是我们的IBinder接口，一个在同进程的对象的抽象是 Object，但这个对象是不能被跨进程使用的，要想跨进程使用，在 Android 中就必须依附于 Binder Framework。基于抽象设计的原理，Android系统将一个可远程操作的应用定义为 [IBinder](https://developer.android.com/reference/android/os/IBinder.html)，在这个接口中定义了一个可远程调用对象应该具有的属性和方法，在代码中实际使用的Binder 也都是继承自 IBinder 对象。onTransact就是利用了c++层的实现来进行跨进程通讯的实际过程。

**第三层（Java层使用Binder）：**可是Binder的过程实际上需要被客户端和服务端利用，他们两个如果完全分开更加复杂，那么就使用**Stub/Proxy模式，**都继承自相应的接口，分别负责服务端和客户端，服务端负责实现，客户端负责调用Java层的跨进程接口，这样两边相互分开，Client只需要使用Proxy，Service只需要实现Stub，这样相互之间不需要知道实现细节。

**第四层（AIDL，真正使用）：**我们的Proxy和Stub说到底是实现了一个相同的方法，只不过Client调用并通过跨进程真正在Service执行，而大量的跨进程过程都是相同的，为了简化使用的成本Android使用了AIDL，只需要定义一个接口，自动生成客户端和服务器的跨进程实现代码，这至于跨进程有关，节省了事件，而如果想实现**类似验证之类的操作**，只需要在Stub的onTransact中进行相应的实现就可。

**五、Binder的数据拷贝**

Linux内核实际上没有从一个用户空间到另一个用户空间直接拷贝的函数，需要先用copy\_from\_user()拷贝到内核空间，再用copy\_to\_user()拷贝到另一个用户空间。**为了实现用户空间到用户空间的拷贝，mmap()分配的内存除了映射进了接收方进程里，还映射进了内核空间**。所以调用copy\_from\_user()将数据拷贝进内核空间也相当于拷贝进了接收方的用户空间，这就是Binder只需一次拷贝的"秘密"。

最底层的是Android的ashmen(Anonymous shared memory)机制，它负责辅助实现内存的分配，以及跨进程所需要的内存共享。AIDL(android interface definition language)对Binder的使用进行了封装，可以让开发者方便的进行方法的远程调用，后面会详细介绍。Intent是最高一层的抽象，方便开发者进行常用的跨进程调用。

使用共享内存通信的一个显而易见的好处是效率高，因为 **进程可以直接读写内存，而不需要任何数据的拷贝**。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次内存数据：一次从输入文件到共享内存区，另一次从共享内存到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域，而是保持共享区域，直到通信完成为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除内存映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。