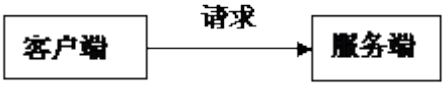
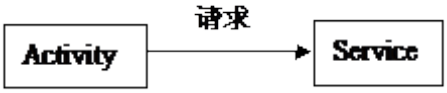
IPC 框架分析 Binder，Service，Service manager

我首先从宏观的角度观察 Binder,Service,Service Manager，并阐述各自的概念。从 Linux 的概念空间中，Android 的设计 Activity 托管在不同的的进程，Service 也都是托管在不同的进程，不同进程间的 Activity,Service 之间要交换数据属于 IPC。Binder 就是为了 Activity 通讯而设计的一个轻量级的 IPC 框架。

在代码分析中，我发现 Android 中只是把 Binder 理解成进程间通讯的实现，有点狭隘，而是应该站在公共对象请求代理这个高度来理解 Binder，Service 的概念，这样我们就会看到不一样的格局，从这个高度来理解设计意图，我们才会对 Android 中的一些天才想法感到惊奇。从 Android 的外特性概念空间中，我们看不到进程的概念，而是 Activity，Service，AIDL，INTENT。一般的如果我作为设计者，在我们的根深蒂固的想法中，这些都是如下的 C/S 架构，客户端和服务端直接通过 Binder 交互数据，打开 Binder 写入数据，通过 Binder 读取数据，通讯就可以完成了。



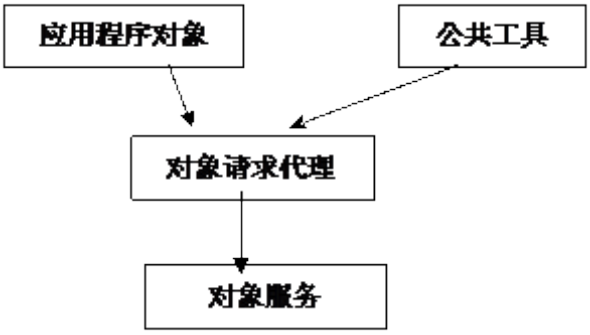


该注意到 Android 的概念中，Binder 是一个很低层的概念，上面一层根本都看不到 Binder，而是 Activity 跟一个 Service 的对象直接通过方法调用，获取服务。

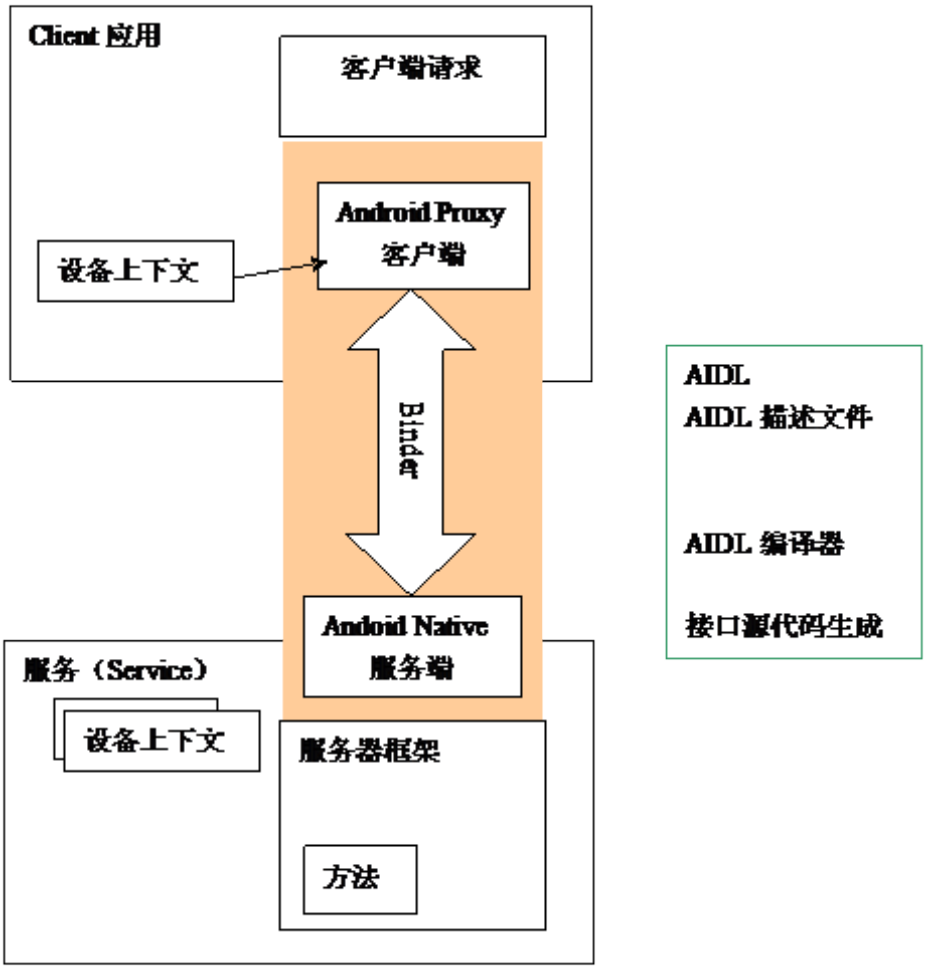
这个就是 Android 提供给我们的外特性：在 Android 中，要完成某个操作，所需要做的就是请求某个有能力的服务对象去完成动作，而无需知道这个通讯是怎样工作的，以及服务在哪里。所以 Andoid 的 IPC 在本质上属于对象请求代理架构，Android 的设计者用 CORBA 的概念将自己包装了一下，实现了一个微型的轻量级 CORBA 架构，这就是 Andoid 的 IPC 设计的意图所在，它并不是仅仅解决通讯，而是给出了一个架构，一种设计理念，这就是 Android 的闪光的地方。Android 的 Binder 更多考虑了数据交换的便捷，并且只是解决本机的进程间的通讯，所以不像 CORBA 那样复杂，所以叫做轻量级。

所以要理解 Android 的 IPC 架构，就需要了解 CORBA 的架构。而 CORBA 的架构在本质上可以使用下面图来表示：

在服务端，多了一个代理器，更为抽象一点我们可以下图来表示。



分析和 CORBA 的大体理论架构，我给出下面的 Android 的对象代理结构。



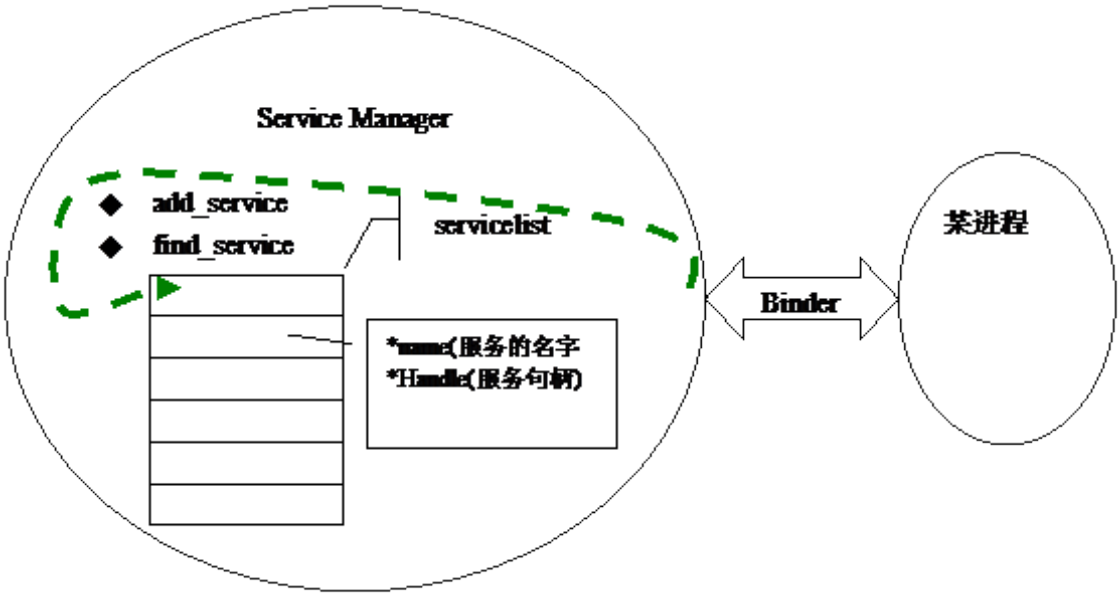
在结构图中，我们可以较为清楚的把握 Android 的ＩＰＣ包含了如下的概念：

设备上下文什（ContextObject）

设备上下文包含关于客服端，环境或者请求中没有作为参数传递个操作的上下文信息，应用程序开发者用 ContextObject 接口上定义的操作来创建和操作上下文。

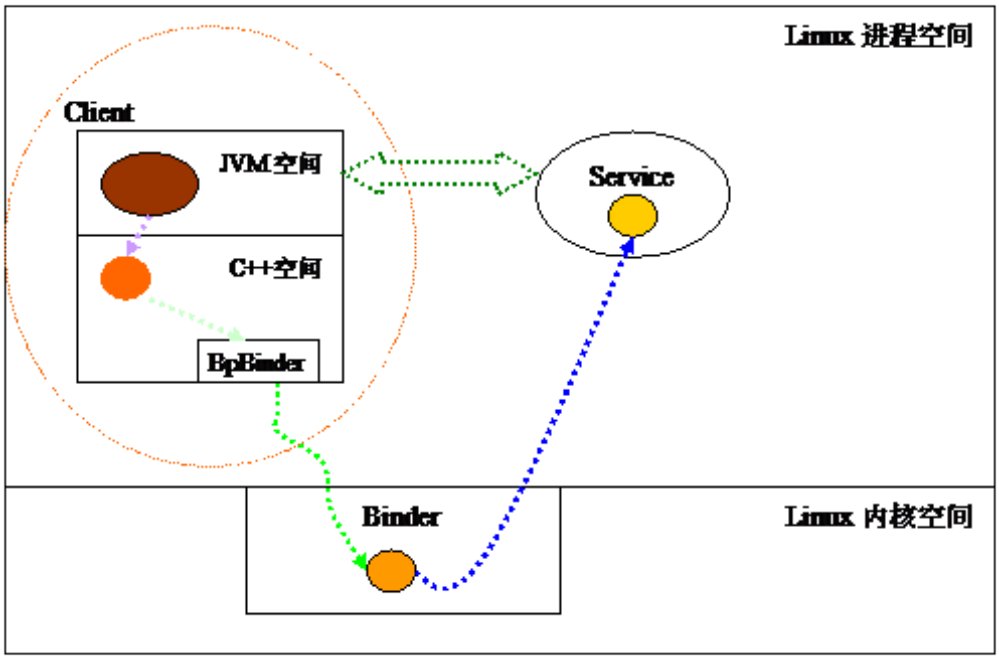
Android 代理：这个是指代理对象
Binder Linux 内核提供的 Binder 通讯机制

Android 的外特性空间是不需要知道服务在那里，只要通过代理对象完成请求，但是我们要探究 Android 是如何实现这个架构，首先要问的是在 Client 端要完成云服务端的通讯，首先应该知道服务在哪里？我们首先来看看 Service Manger 管理了那些数据。Service Manager 提供了 add service,check service 两个重要的方法，并且维护了一个服务列表记录登记的服务名称和句柄。



Service manager service 使用0来标识自己。并且在初始化的时候，通过 binder 设备使用 BINDER_SET_CONTEXT_MGR ioctl 将自己变成了 CONTEXT_MGR。Svclist 中存储了服务的名字和 Handle，这个 Handle 作为 Client 端发起请求时的目标地址。服务通过 add_service 方法将自己的名字和 Binder 标识 handle 登记在 svclist 中。而服务请求者，通过 check_service 方法，通过服务名字在 service list 中获取到 service 相关联的 Binder 的标识 handle,通过这个 Handle 作为请求包的目标地址发起请求。

我们理解了 Service Manager 的工作就是登记功能，现在再回到ＩＰＣ上，客服端如何建立连接的？我们首先回到通讯的本质：IPC。从一般的概念来讲，Android 设计者在 Linux 内核中设计了一个叫做 Binder 的设备文件，专门用来进行 Android 的数据交换。所有从数据流来看 Java 对象从 Java 的 VM 空间进入到 C++空间进行了一次转换，并利用 C++空间的函数将转换过的对象通过 driver/binder 设备传递到服务进程，从而完成进程间的 IPC。这个过程可以用下图来表示。



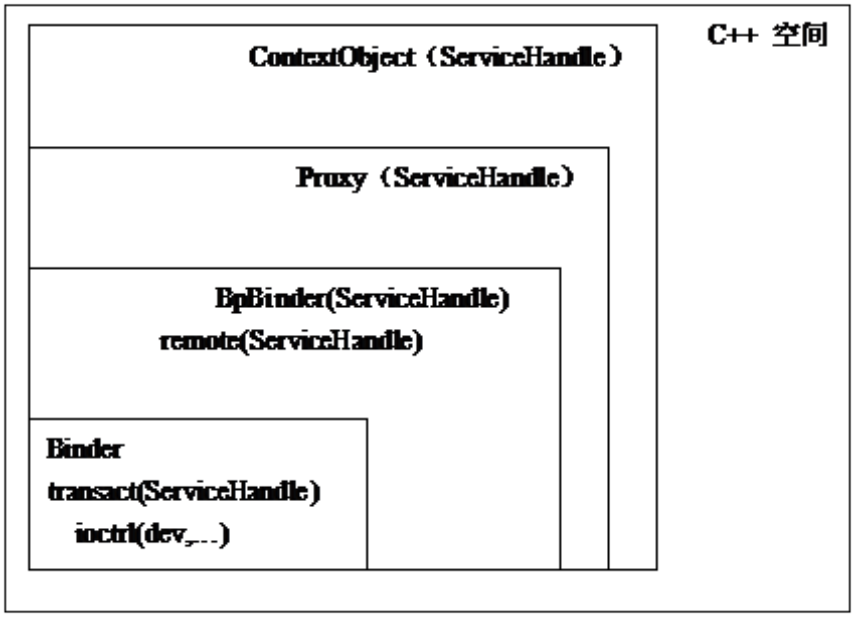
这里数据流有几层转换过程。

（1） 从 JVM 空间传到 c++空间，这个是靠 JNI 使用 ENV 来完成对象的映射过程。

（2） 从 c++空间传入内核 Binder 设备，使用 ProcessState 类完成工作。

（3） Service 从内核中 Binder 设备读取数据。

Ａｎｄｒｏｉｄ设计者需要利用面向对象的技术设计一个框架来屏蔽掉这个过程。要让上层概念空间中没有这些细节。Android 设计者是怎样做的呢？我们通过 c++空间代码分析，看到有如下空间概念包装(ProcessState@(ProcessState.cpp)



在 ProcessState 类中包含了通讯细节，利用 open_binder 打开 Linux 设备 dev/binder,通过 ioctrl 建立的基本的通讯框架。利用上层传递下来的 servicehandle 来确定请求发送到那个 Service。通过分析我终于明白了 Bnbinder，BpBinder 的命名含义，Bn-代表 Native，而 Bp 代表 Proxy。一旦理解到这个层次，ProcessState 就容易弄明白了。

下面我们看 JVM 概念空间中对这些概念的包装。为了通篇理解设备上下文，我们需要将 Android VM 概念空间中的设备上下文和 C++空间总的设备上下文连接起来进行研究。

为了在上层使用统一的接口，在 JVM 层面有两个东西。在 Android 中，为了简化管理框架,引入了 ServiceManger 这个服务。所有的服务都是从 ServiceManager 开始的，只用通过 Service Manager 获取到某个特定的服务标识构建代理 IBinder。在 Android 的设计中利用 Service Manager 是默认的 Handle 为0，只要设置请求包的目标句柄为0，就是发给 Service Manager 这个 Service 的。在做服务请求时，Android 建立一个新的 Service Manager Proxy。Service Manager Proxy 使用 ContexObject 作为 Binder 和 Service Manager Service（服务端）进行通讯。

我们看到 Android 代码一般的获取 Service 建立本地代理的用法如下：

```
IXXX  mIxxx=IXXXInterface.Stub.asInterface(ServiceManager.getService("xxx"));
```

例如：使用输入法服务：

```
IInputMethodManager mImm=

IInputMethodManager.Stub.asInterface(ServiceManager.getService("input_method"))
;
```

这些服务代理获取过程分解如下：

（1） 通过调用 GetContextObject 调用获取设备上下对象。注意在 AndroidJVM 概念空间的 ContextObject 只是与 Service Manger Service 通讯的代理 Binder 有对应关系。这个跟 c++ 概念空间的 GetContextObject 意义是不一样的。

注意看看关键的代码

BinderInternal.getContextObject（） @BinderInteral.java

NATIVE JNI:getContextObject() @android_util_Binder.cpp

Android_util_getConextObject @android_util_Binder.cpp

ProcessState::self()->getCotextObject(0) @processState.cpp

getStrongProxyForHandle(0) @

NEW BpBinder(0)

注意 ProcessState::self()->getCotextObject(0) @processstate.cpp，就是该函数在进程空间建立了 ProcessState 对象，打开了 Binder 设备 dev/binder,并且传递了参数0，这个0代表了与 Service Manager 这个服务绑定。

（2） 通过调用 ServiceManager.asInterface（ContextObject）建立一个代理 ServiceManger。

mRemote= ContextObject(Binder)

这样就建立起来 ServiceManagerProxy 通讯框架。

(3)客户端通过调用 ServiceManager 的 getService 的方法建立一个相关的代理 Binder。

ServiceMangerProxy.remote.transact(GET_SERVICE)

IBinder=ret.ReadStrongBinder()-》这个就是 JVM 空间的代理 Binder

JNI Navite: android_os_Parcel_readStrongBinder() @android_util_binder.cpp

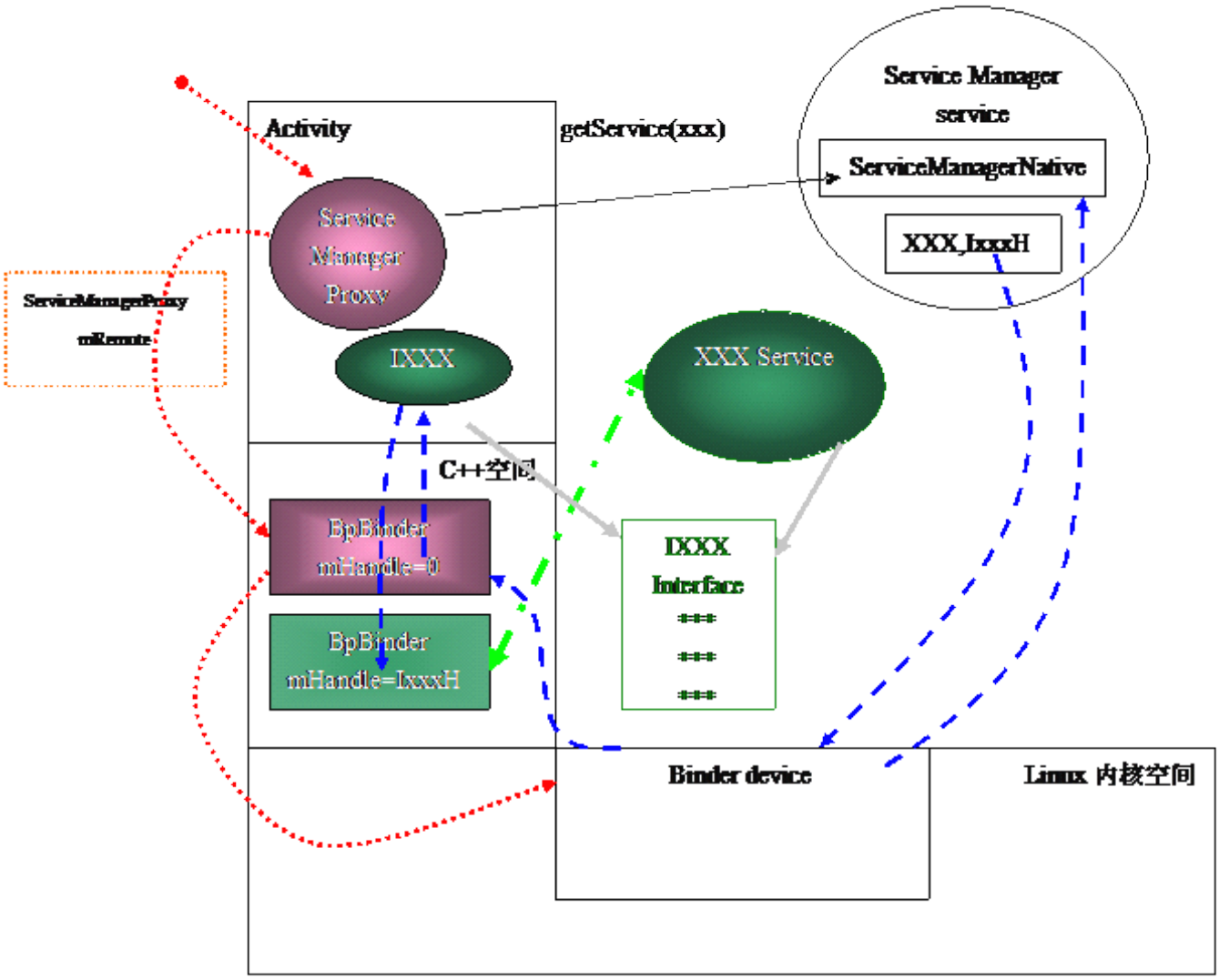
Parcel->readStrongBinder() @pacel.cpp

unflatten_binder @pacel.cpp

getStrongProxyForHandle(flat_handle)

NEW BpBinder(flat_handle)-》这个就是底层 c++空间新建的代理 Binder。

整个建立过程可以使用如下的示意图来表示：



Activity 为了建立一个 IPC，需要建立两个连接：访问 Servicemanager Service 的连接，IXXX 具体 XXX Service 的代理对象与 XXXService 的连接。这两个连接对应 c++空间 ProcessState 中 BpBinder。对 IXXX 的操作最后就是对 BpBinder 的操作。由于我们在写一个 Service 时，在一个 Package 中写了 Service Native 部分和 Service Proxy 部分，而 Native 和 Proxy 都实现相同的接口：IXXX Interface，但是一个在服务端，一个在客服端。客户端调用的方式是使用 remote->transact 方法向 Service 发出请求，而在服务端的 OnTransact 中则是处理这些请求。所以在 Android Client 空间就看到这个效果：只需要调用代理对象方法就达到了对远程服务的调用目的，实际上这个调用路径好长好长。

我们其实还一部分没有研究，就是同一个进程之间的对象传递与远程传递是区别的。同一个进程间专递服务地和对象，就没有代理 BpBinder 产生，而只是对象的直接应用了。应用程序并不知道数据是在同一进程间传递还是不同进程间传递，这个只有内核中的 Binder 知道，所以内核 Binder 驱动可以将 Binder 对象数据类型从 BINDER_TYPE_BINDER 修改为 BINDER_TYPE_HANDLE 或者 BINDER_TYPE_WEAK_HANDLE 作为引用传递。