Android 进程间通信机制

----Binder 篇

一 进程间通信方式

1.1 Linux IPC

管道（pipe）：半双工通信，单向数据流，只能用于父子进程之间通信。

命名管道（named pipe）：半双工通信，单向数据流，可以全局使用。

信号量（semophore）：是一种锁机制，实质是一个计数器，可用于进程间或线程间同步访问共享资源。

消息队列（message queue）：由内核维护的一个消息链表，可以承载格式化的数据流。

信号（signal）：用于通知进程发生了什么事件，接收进程可以根据需要注册对应的事件处理函数来覆盖系统的默认行为。

共享内存（shared memory）：是效率最高的 IPC 方式，但是进程间同步比较复杂，可以配合信号量使用。

套接字（socket）：主要用于跨网络的进程间通信和本地进程间的低速通信，缺点是传输效率低，开销大，进程间同步较复杂等。

以上各种机制的使用细节请大家 google 一下。

1.2 Android Binder IPC

Binder 是 Android 特有的 IPC 方式，当然 Android 并没有抛弃 Linux 传统 IPC 方式，像管道和 socket 等也常被使用。

从 Android 源码来看，Android 普遍采用 C/S 架构来布局系统，比如像各种 sensor 的管理，多媒体的回放和录制，通知管理，窗口管理，音频子系统，SD 卡挂载（vold，MountService）等，这些模块都可以找到明确的服务端和客户端。

Android 服务绝大部分都是采用 Binder IPC 方式实现，但是也有一小部分使用 socket 实现。采用 socket 实现服务，在我看来主要是基于以下两点原因，第一是实现简单，不用写太多代码就可以实现；第二是在某些情况下不得以而为之，比如 linux 内核中各种内核通知大都采用 NetLink 进行广播，使用 socket 监听便是最简单有效的方式。

Android 广泛采用 Binder IPC，是因为 Binder 具有其无可比拟的优势。我们从三个方面进行理解。

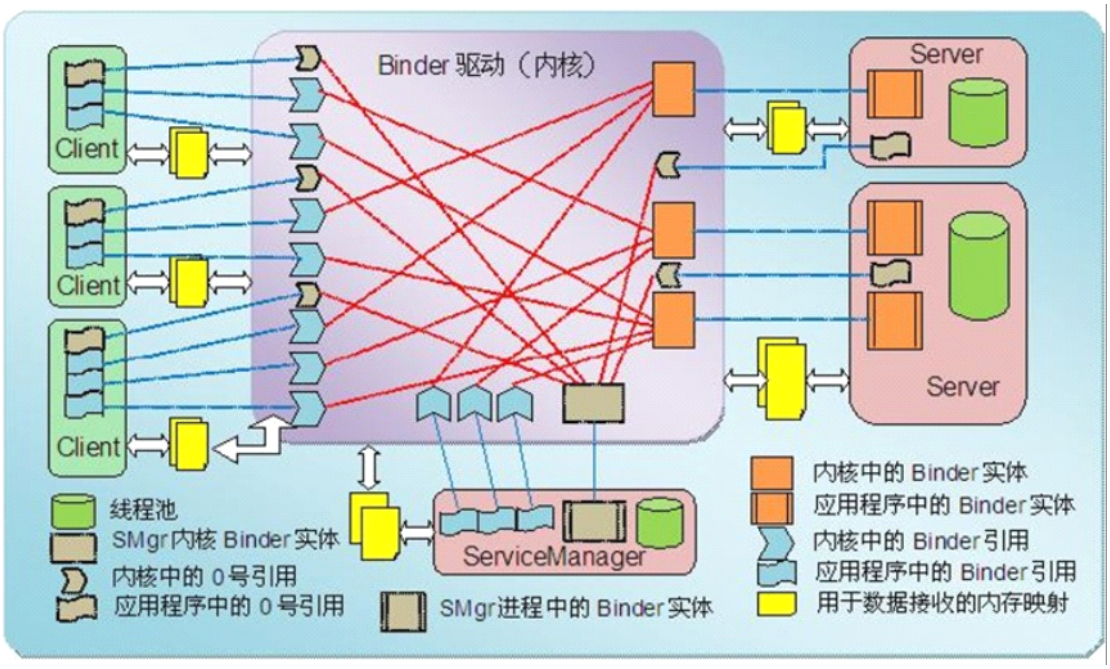
第一，传统的 IPC 方式，目前只有 socket 可以直接用于 C/S 架构。但是 socket 主要是针对跨网络进程间通信和本地进程间的低速通信，整个协议封装复杂庞大，系统开销比较大，传输效率低，使整个系统过于沉重。简单的 service 可以采用 socket 通信，目前 Android 有一些 service 的确采用 socket 方式。

第二，嵌入式系统中资源比较紧缺，Binder 是一种由内核支持的轻量级 IPC 方式。服务端和 Binder 驱动共享内存（采用 mmap 映射），实现数据高效率传递（数据传递只拷贝一次），而且通信协议封装轻便，只有一套数量不多的通信命令和数据结构。

第三，安全性高。Binder IPC 由内核支持，可以方便有效的校验客户端身份（比如 PID/UID 等）。同时 Binder 还支持匿名 Binder（在 Android 源码中随处可见），为进程间建立一条私密信道，让外界进程无法枚举。

二 Android Binder 通信模型

下图是从网上偷来的（向高人致敬！），这张图能够完美的表达 Android Binder 通信模型的所有核心。



2.1 Binder 通信模型的组成

在 Binder 通信模型中存在四个角色，分别是 Server，Client，ServiceManager，Binder Driver。

2.1.1 Server

进程间通信的服务端，实现所有业务逻辑，监听客户端的请求，Server 要先注册到 ServiceManager 维护的服务列表中供客户端检索，否则该 Server 就成为匿名 Binder，外界无法枚举。

2.1.2 Client

进程间通信的客户端，在向某一 Server 发起请求调用之前先向 ServiceManager 查询该 Server 是否存在，如果存在，ServiceManager 会返回一个服务代理对象，后续的远程调用就靠这个服务代理对象来完成。

2.1.3 ServiceManager

关联 Server 和 Client 的中介（就像中介所，论坛中有人称 SMgr 为 DNS），所有服务端(匿名 Binder 除外)都需要向 ServiceManager 注册，ServiceManager 本身也是一个单独的服务进程，内部维护一个服务列表，建立服务名称和 Binder 实体之间的映射关系。这样客户端就可以使用可读性很强的名字字符串向 ServiceManager 查询是否存在一个和名字字符串匹配的服务正在运行。

2.1.4 Binder Driver

通讯模型的核心和基石，负责客户端和服务端数据的收发和同步，维护和调度服务端的线程池，维护服务端的 Binder 实体和客户端的 Binder 引用，并建立 Binder 实体和 Binder 引用之间的映射关系。

2.2 鸡生蛋蛋生鸡的悖论

2.2.1 客户端访问服务端流程

a. Client 向 ServiceManager 提供一个名称字符串查询服务是否存在；

b. 如果某一个服务正在运行而且名字和 Client 提供的名字字符串匹配，

那么 ServiceManager 就会返回一个服务代理对象；

c. Client 使用服务代理对象完成对远程 Server 的调用。

2.2.2 无法逃避的问题

现在的问题是 ServiceManager 本身也是一个单独的服务进程，Client 向 ServiceManager 发起的查询请求也属于标准的进程间通讯。Client 又是如何取得 ServiceManager 的代理对象的呢。Client 可以借助于 ServiceManager 来取得 Service 的代理对象，但是 ServiceManager（Service）的代理对象又该如何获得？

2.2.3 解决方案

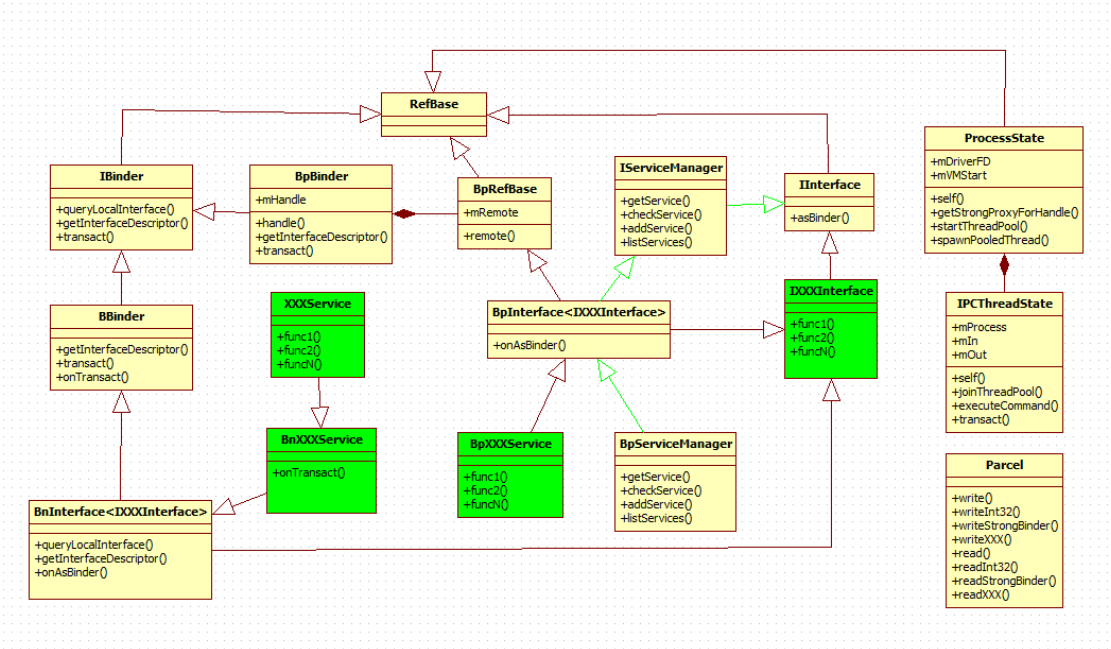
针对 ServiceManager 这个特殊的服务，Android 的方案是另外制定规则让 ServiceManager 逃出《**2.2.1 客户端访问服务端流程**》。脱离 libbinder 那套框架手工打造一个服务端，并把 ServiceManager 的引用号定为 0，这样其他进程需要访问 ServiceManager 时直接使用 0 号服务。

三 Android Binder 实现

由于 Android Framework 的实现是跨语言环境的，Android Binder 的实现分为 3 部分，Native Binder 实现，Java Binder 实现，以及这两部分之间一层很薄的 JNI（glue class）。

3.1 Native Binder 实现

3.1.1 Native Binder 框架类图



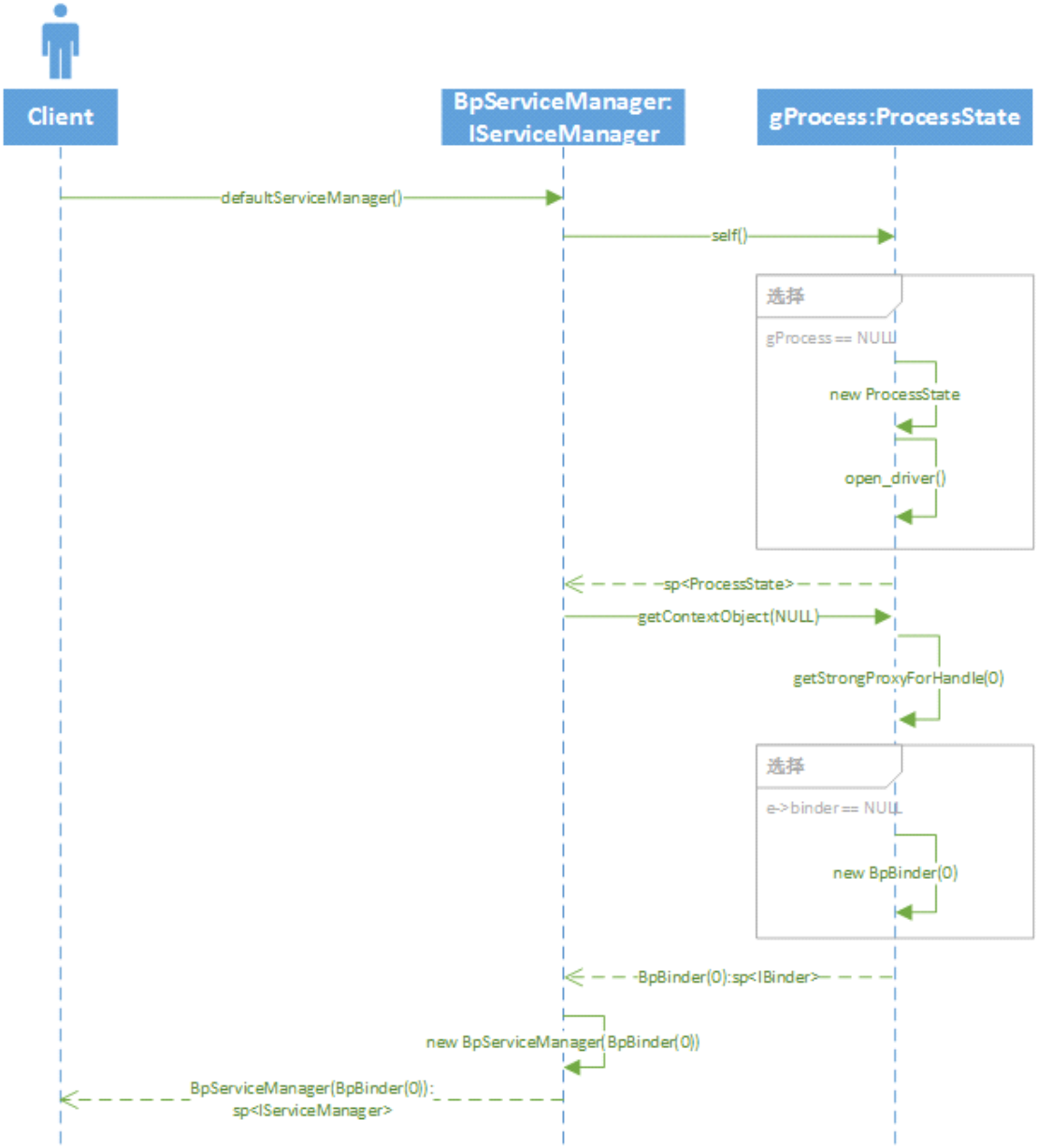
图中黄色部分是 Binder IPC 的基础框架，最终会被编译成 libbinder.so 库文件， 是 Binder IPC 的通讯层实现；绿色部分为项目实践中要具体实现的部分，这一部分是业务层实现，依附于 Binder IPC 基础框架之上；绿色箭头串起来的那条继承链是辅助类，为所有应用进程（client，service）提供访问 ServiceManager 服务的接口。

3.1.2 ProcessState 对象和 IPCThreadState 对象

ProcessState 是 Binder IPC 的 Stepstone，负责打开 Binder 驱动，设置线程池，映射内存等，这些都与通信紧密相关。ProcessState 采用单例模式（singleton），每个进程只有一个对象，通过它的静态方法 self（）获取。一个进程中的所有线程共享一个 ProcessState 对象完成进程间通信。

IPCThreadState 对象是线程私有对象，保存到每个线程的 TLS（线程本地存储）中。这个对象在创建时保存一个指向 ProcessState 对象的指针，发送数据到特定的远端 Binder 实体，接收并解析来自另一个进程发来的数据，最终把远端访问路由到指定的实现函数中去执行。

3.1.3 defaultServiceManager()的流程分析

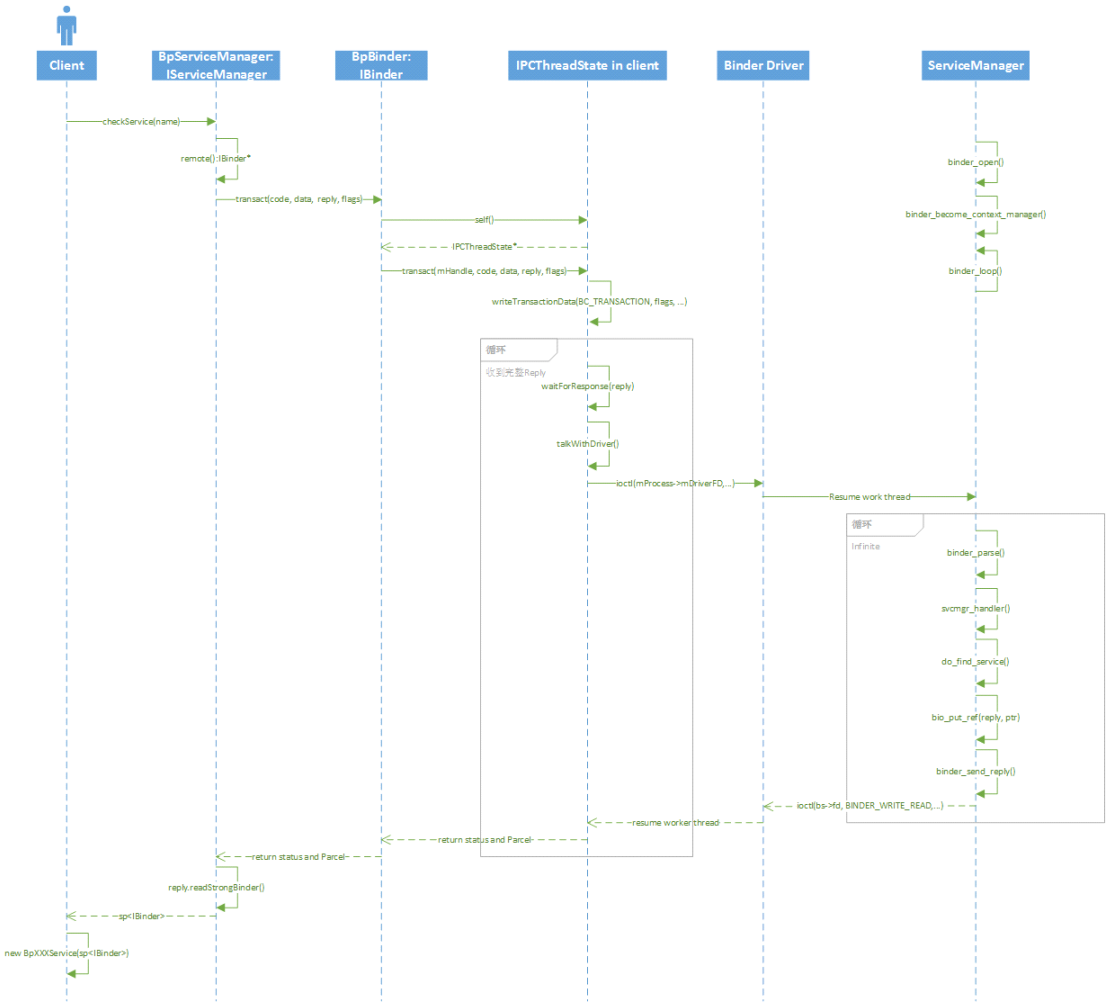


参与进程间通讯的每个进程都要创建一个单例的 ProcessState 对象，在构造 ProcessState 对象时，会打开 Binder 驱动，映射内存，设置线程池限制等；

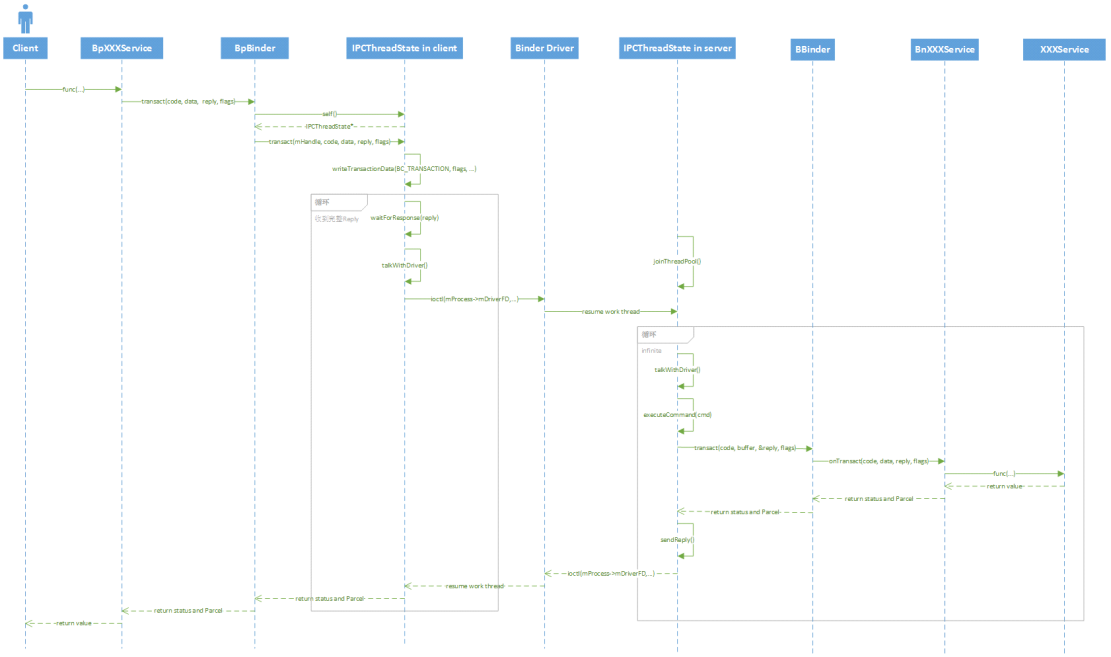
在需要使用 ServiceManager 的地方调用 defaultServiceManager()函数得到一个代理对象 IServiceManager。后续调用代理对象的 getService，addService，listService 来查询和注册服务等操作。

结论：客户端实际上是持有一个 BpBinder 对象，BpBinder 对象中含有服务端 Binder 实体的引用 Handle，Binder 驱动最终通过引用 Handle 找到对应的服务端 Binder 实体。

3.1.4 Service 的注册和获取流程分析



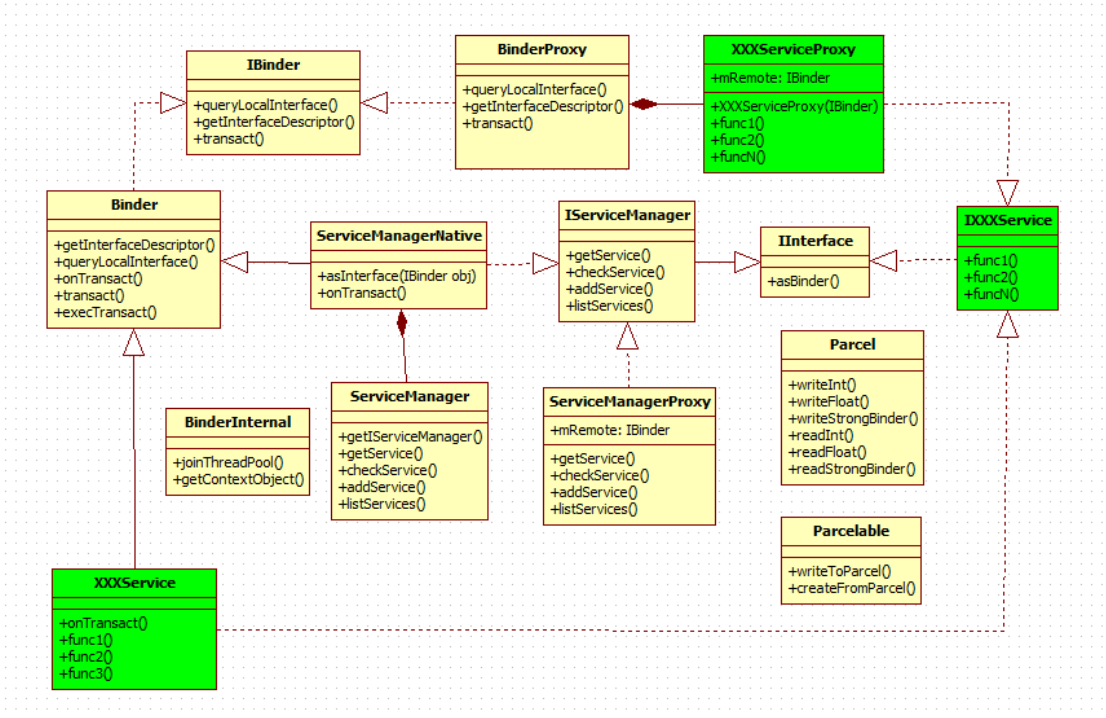
3.1.5 Client 和 Service 的交互流程分析



3.1.6 服务进程的消息循环

3.2 JAVA Binder 实现

3.2.1 JAVA Binder“全家福”



黄色部分是 JAVA Binder 的基础框架部分，封装了所有共性的东西；绿色部分是项目实践中需要实现的部分，直接从基础框架继承，依附于 JAVA Binder 基础框架之上。

上图是在纯手工创建 JAVA Service 情形下的类图关系。如果采用 AIDL 工具自动创建，图中绿色部分需要增加 stub 类和 stub.proxy 类，具体细节让我们一起跟踪源码。

3.2.2 JAVA Binder 和 Native Binder 的对应关系

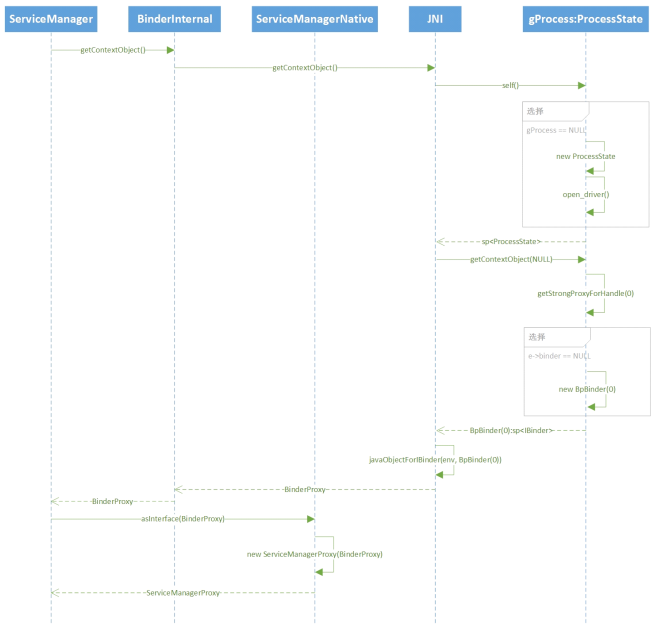
我们先讲结论：JAVA Binder 和 Native Binder 是分别在两种语言环境中实现的对等物，等价物。当创建 JAVA Binder 的各个类对象时，Native Binder 中对应的类对象也同时被创建了，二者之间通过 JNI 粘合在一起。

下面我们一起跟踪一遍相关代码。

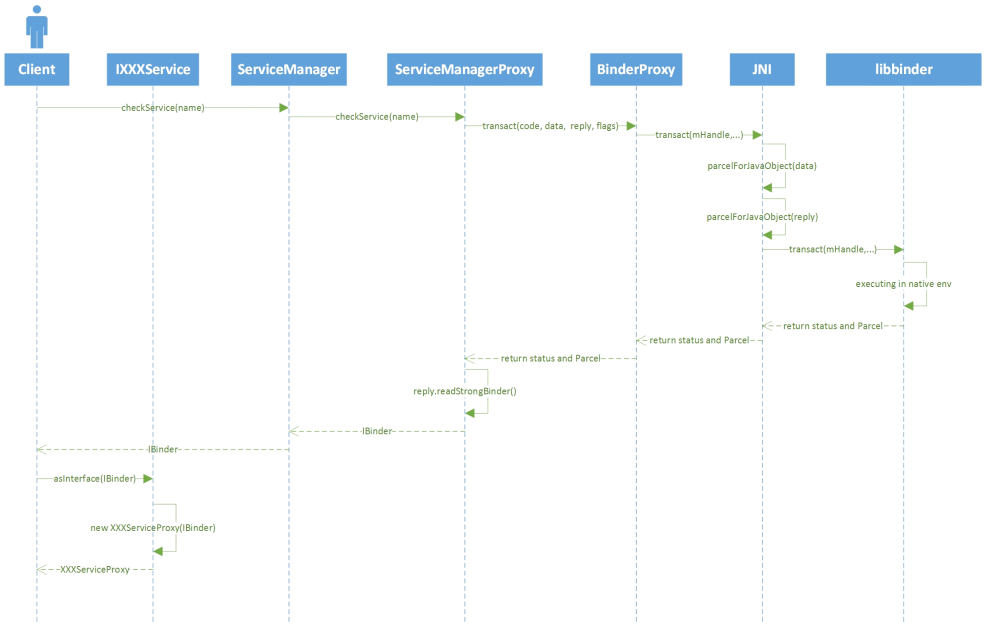
3.3 JAVA Binder 实现

下面给出各个流程的序列图，我们一起结合代码分析。如果图片看不清，请查看原图。

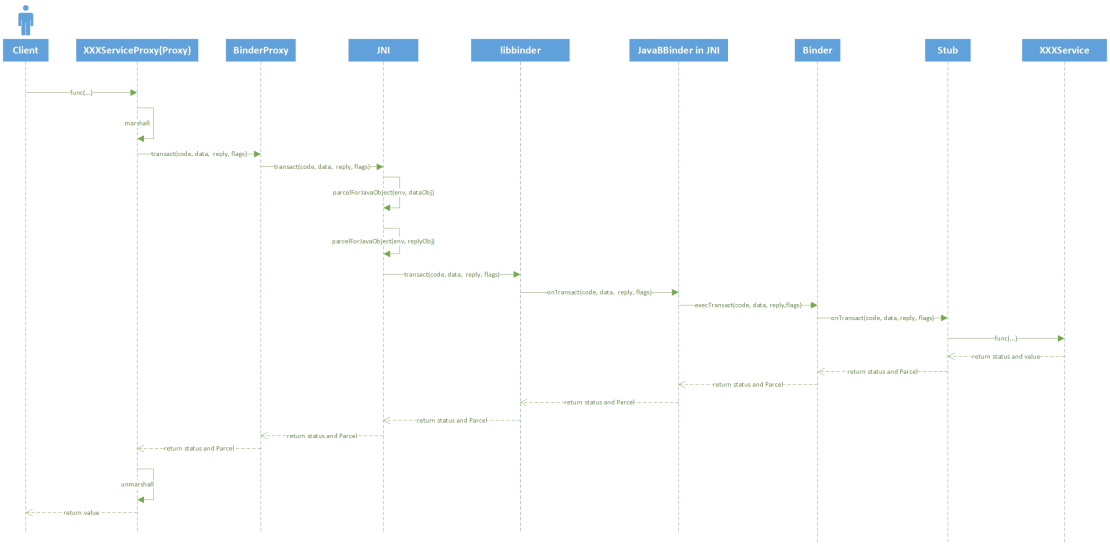
3.3.1 ServiceManager 的获取流程分析



3.3.2 Service 的注册和获取流程分析



3.3.3 Client 和 Service 的交互流程分析



3.4 曲径通幽，柳暗花明

两种 Binder 本质上是统一的，JAVA Binder 通过 JNI 最终汇入 Native Binder，Native Binder 通过 JNI 也可以回调到 JAVA Binder，因此客户端和服务端用哪种语言实现不是关键，关键点是客户端的请求总能被路由到正确的服务端。

四 实现服务程序的步骤

4.1 Native Service 实现步骤
4.2 Java Service 实现步骤

五 Android Binder 的应用实例

4.1 一个实际开发案例

4.2 BatteryService 分析

4.3 Binder 与其他技术结合使用的案例