**Android Binder之应用层总结与分析**

2017年09月25日 21:48:34 [凶残的程序员](https://me.csdn.net/qian520ao) 阅读数：3190

 版权声明：本文为博主原创文章，欢迎转载，转载需标明出处。 https://blog.csdn.net/qian520ao/article/details/78089877

\*本篇文章已授权微信公众号 guolin\_blog （郭霖）独家发布

* + [概述](https://blog.csdn.net/qian520ao/article/details/78089877#%E6%A6%82%E8%BF%B0)
  + [为何选择Binder](https://blog.csdn.net/qian520ao/article/details/78089877#%E4%B8%BA%E4%BD%95%E9%80%89%E6%8B%A9binder)
    - [传输性能好](https://blog.csdn.net/qian520ao/article/details/78089877#%E4%BC%A0%E8%BE%93%E6%80%A7%E8%83%BD%E5%A5%BD)
    - [安全性高](https://blog.csdn.net/qian520ao/article/details/78089877#%E5%AE%89%E5%85%A8%E6%80%A7%E9%AB%98)
  + [Binder总体架构](https://blog.csdn.net/qian520ao/article/details/78089877#binder%E6%80%BB%E4%BD%93%E6%9E%B6%E6%9E%84)
  + [Binder原理](https://blog.csdn.net/qian520ao/article/details/78089877#binder%E5%8E%9F%E7%90%86)
  + [Binder通信模型](https://blog.csdn.net/qian520ao/article/details/78089877#binder%E9%80%9A%E4%BF%A1%E6%A8%A1%E5%9E%8B)
    - [ServiceManager 与实名Binder](https://blog.csdn.net/qian520ao/article/details/78089877#servicemanager-%E4%B8%8E%E5%AE%9E%E5%90%8Dbinder)
    - [Client 获得实名Binder的引用](https://blog.csdn.net/qian520ao/article/details/78089877#client-%E8%8E%B7%E5%BE%97%E5%AE%9E%E5%90%8Dbinder%E7%9A%84%E5%BC%95%E7%94%A8)
    - [Client 与 Server通讯](https://blog.csdn.net/qian520ao/article/details/78089877#client-%E4%B8%8E-server%E9%80%9A%E8%AE%AF)
    - [Binder的线程管理](https://blog.csdn.net/qian520ao/article/details/78089877#binder%E7%9A%84%E7%BA%BF%E7%A8%8B%E7%AE%A1%E7%90%86)
    - [整体通讯流程图](https://blog.csdn.net/qian520ao/article/details/78089877#%E6%95%B4%E4%BD%93%E9%80%9A%E8%AE%AF%E6%B5%81%E7%A8%8B%E5%9B%BE)
  + [总结](https://blog.csdn.net/qian520ao/article/details/78089877#%E6%80%BB%E7%BB%93)

**概述**

通过前两篇([Android 深入浅出AIDL（一）](http://blog.csdn.net/qian520ao/article/details/78072250)[Android 深入浅出AIDL（二）](http://blog.csdn.net/qian520ao/article/details/78074983))文章对AIDL进行运用与分析，对于Android跨进程通讯有了不少的了解，深入分析AIDL通讯过程，我们发现跨进程的主导元素是Binder，但是这个Binder是Android中很重要又很复杂的概念，本文就不深入到底层和驱动层去研究，有兴趣的童鞋可以滑动到文章底部，有详细的Binder深入探究的文章推荐哦。

**为何选择Binder**

Linux已经拥有管道，system V IPC，socket等IPC手段，却还要倚赖Binder来实现进程间通信，说明Binder具有无可比拟的优势。

**传输性能好**

Binder很重要的的优点之一就是，复杂数据类型传递可以复用内存。

* socket：是一个通用接口，导致其传输效率低，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
* 管道和消息队列：因为采用存储转发方式，所以至少需要拷贝2次数据，效率低；
* 共享内存：虽然在传输时没有拷贝数据，但其控制机制复杂。

| **IPC** | **数据拷贝次数** |
| --- | --- |
| 共享内存 | 0 |
| Binder | 1 |
| Socket/管道/消息队列 | 2 |

**安全性高**

* 传统IPC没有任何安全措施，完全依赖上层协议来确保。首先传统IPC的接收方无法获得对方进程可靠的UID/PID（用户ID/进程ID），从而无法鉴别对方身份。
* Android为每个安装好的应用程序分配了自己的UID，故进程的UID是鉴别进程身份的重要标志。可靠的身份标记只有由IPC机制本身在内核中添加。
* 传统IPC访问接入点是开放的，无法建立私有通道。Binder可以使用匿名 Binder建立私密通道，别的进程就无法通过穷举或猜测等任何方式获得该Binder的引用，向该Binder发送请求。

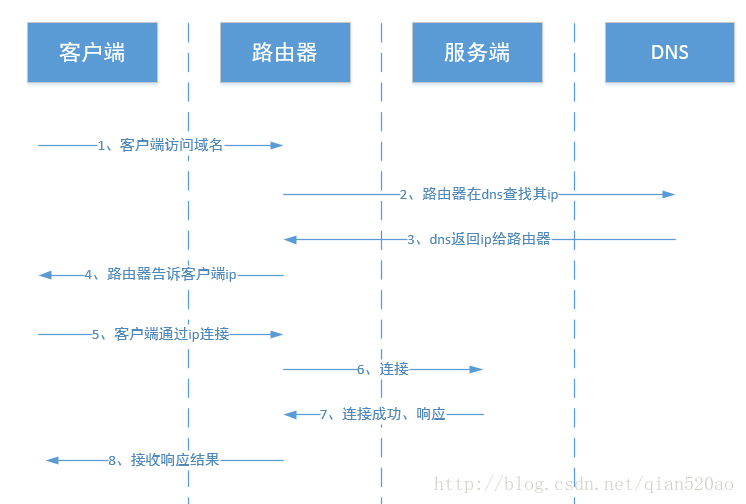
**Binder总体架构**

在Android系统中，这个运行在内核空间的，负责各个用户进程通过Binder通信的内核模块叫做Binder驱动，Binder驱动虽然默默无闻，却是通信的核心。尽管名叫‘驱动’，实际上和硬件设备没有任何关系，只是实现方式和设备驱动程序是一样的。

面向对象思想的引入将进程间通信转化为通过对某个Binder对象的引用调用该对象的方法，而其独特之处在于Binder对象是一个可以跨进程引用的对象，它的实体位于一个进程中，而它的引用却遍布于系统的各个进程之中。最诱人的是，这个引用和java里引用一样既可以是强类型，也可以是弱类型，而且可以从一个进程传给其它进程，让大家都能访问同一Server，就象将一个对象或引用赋值给另一个引用一样。Binder模糊了进程边界，淡化了进程间通信过程，整个系统仿佛运行于同一个面向对象的程序之中。形形色色的Binder对象以及星罗棋布的引用仿佛粘接各个应用程序的胶水，这也是Binder在英文里的原意。

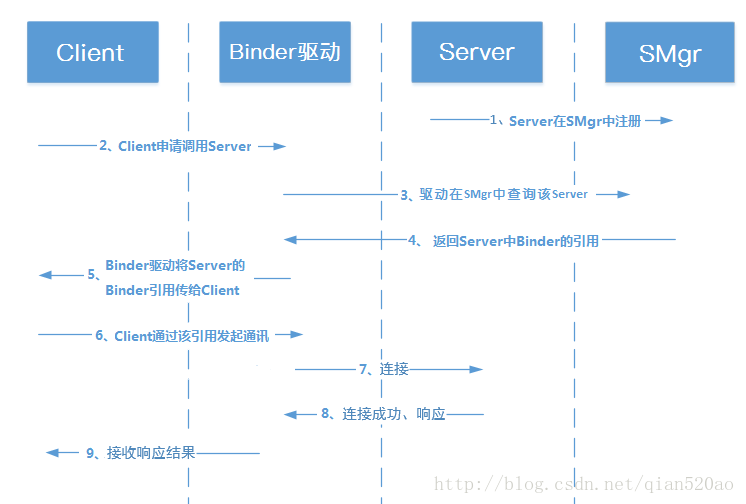
（敲黑板）首先我们要理解我们说的Binder分为Binder对象和Binder驱动，即Binder驱动就是主要的内核模块，而这个Binder对象是通讯的载体，可以自由的通过Binder驱动自由穿梭任意进程。所以客户端或者服务器就可以把数据放入Binder对象里，然后进行调用和通讯。类似于胞吞胞吐吧。

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。



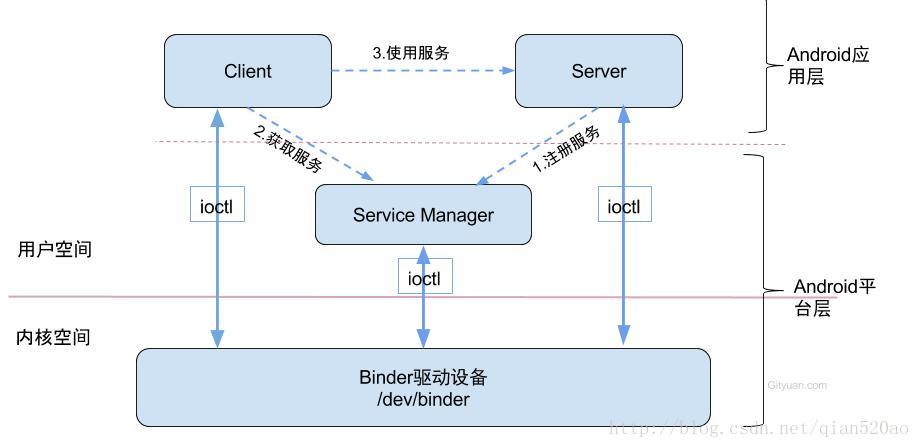
图片摘自[Binder 总体架构及相关代码浅析](https://www.qcloud.com/community/article/546217)

和DNS类似，SMgr的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名字获得对Server中Binder实体的引用。注册了名字的Binder叫实名Binder，就象每个网站除了有IP地址外还有自己的网址。Server创建了Binder实体，为其取一个字符形式，可读易记的名字，将这个Binder连同名字以数据包的形式通过Binder驱动发送给SMgr，通知SMgr注册一个名叫张三的Binder，它位于某个Server中。驱动为这个穿过进程边界的Binder创建位于内核中的实体节点以及SMgr对实体的引用，将名字及新建的引用打包传递给SMgr。SMgr收数据包后，从中取出名字和引用填入一张查找表中。Server向SMgr注册了Binder引用及其名字后，Client就可以通过名字获得该Binder的引用了。



**Binder原理**

Binder通信采用C/S架构，从组件视角来说，包含Client、Server、ServiceManager以及binder驱动，其中ServiceManager用于管理系统中的各种服务。架构图如下所示：



可以看出无论是注册服务和获取服务的过程都需要ServiceManager，需要注意的是此处的Service Manager是指Native层的ServiceManager（C++），并非指framework层的ServiceManager(Java)。ServiceManager是整个Binder通信机制的大管家，是Android进程间通信机制Binder的守护进程，要掌握Binder机制，首先需要了解系统是如何首次启动Service Manager。当Service Manager启动之后，Client端和Server端通信时都需要先获取Service Manager接口，才能开始通信服务。

图中Client/Server/ServiceManage之间的相互通信都是基于Binder机制。既然基于Binder机制通信，那么同样也是C/S架构，则图中的3大步骤都有相应的Client端与Server端。

注册服务(addService)：Server进程要先注册Service到ServiceManager。该过程：Server是客户端，ServiceManager是服务端。   
获取服务(getService)：Client进程使用某个Service前，须先向ServiceManager中获取相应的Service。该过程：Client是客户端，ServiceManager是服务端。   
使用服务：Client根据得到的Service信息建立与Service所在的Server进程通信的通路，然后就可以直接与Service交互。该过程：client是客户端，server是服务端。   
图中的Client,Server,Service Manager之间交互都是虚线表示，是由于它们彼此之间不是直接交互的，而是都通过与Binder驱动进行交互的，从而实现IPC通信方式。其中Binder驱动位于内核空间，Client,Server,Service Manager位于用户空间。Binder驱动和Service Manager可以看做是Android平台的基础架构，而Client和Server是Android的应用层，开发人员只需自定义实现client、Server端，借助Android的基本平台架构便可以直接进行IPC通信。

引自[小米系统工程师–Gityuan](http://gityuan.com/2015/10/31/binder-prepare/)

**Binder通信模型**

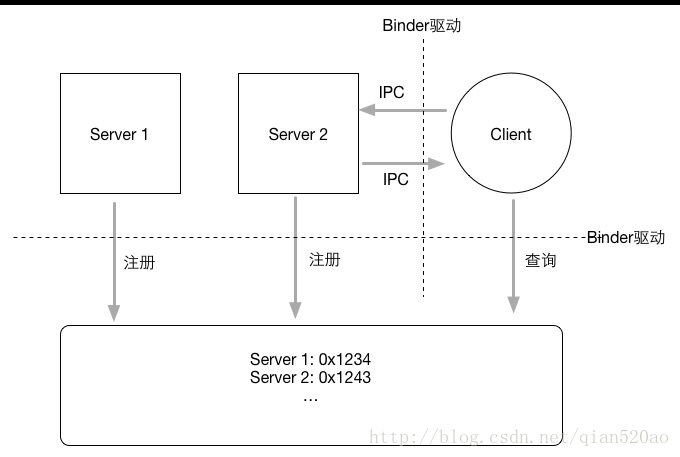
看到这里想必对Binder的一个整体构架有了大致的了解，另外通过[田维术博客](http://weishu.me/2016/01/12/binder-index-for-newer/" \o "optional title" \t "_blank)的一个栗子来描述一下整体过程。

回想一下日常生活中我们通信的过程：假设A和B要进行通信，通信的媒介是打电话（A是Client，B是Server）；A要给B打电话，必须知道B的号码，这个号码怎么获取呢？通信录.

先查阅通信录，拿到B的号码；才能进行通信；否则，怎么知道应该拨什么号码？回想一下古老的电话机，如果A要给B打电话，必须先连接通话中心，说明给我接通B的电话；这时候通话中心帮他呼叫B；连接建立，就完成了通信。

另外，光有电话和通信录是不可能完成通信的，没有基站支持；信息根本无法传达。

我们看到，一次电话通信的过程除了通信的双方还有两个隐藏角色：通信录和基站。Binder通信机制也是一样：两个运行在用户空间的进程要完成通信，必须借助内核的帮助，这个运行在内核里面的程序叫做Binder驱动，它的功能类似于基站；通信录呢，就是一个叫做ServiceManager的东西（简称SMgr）



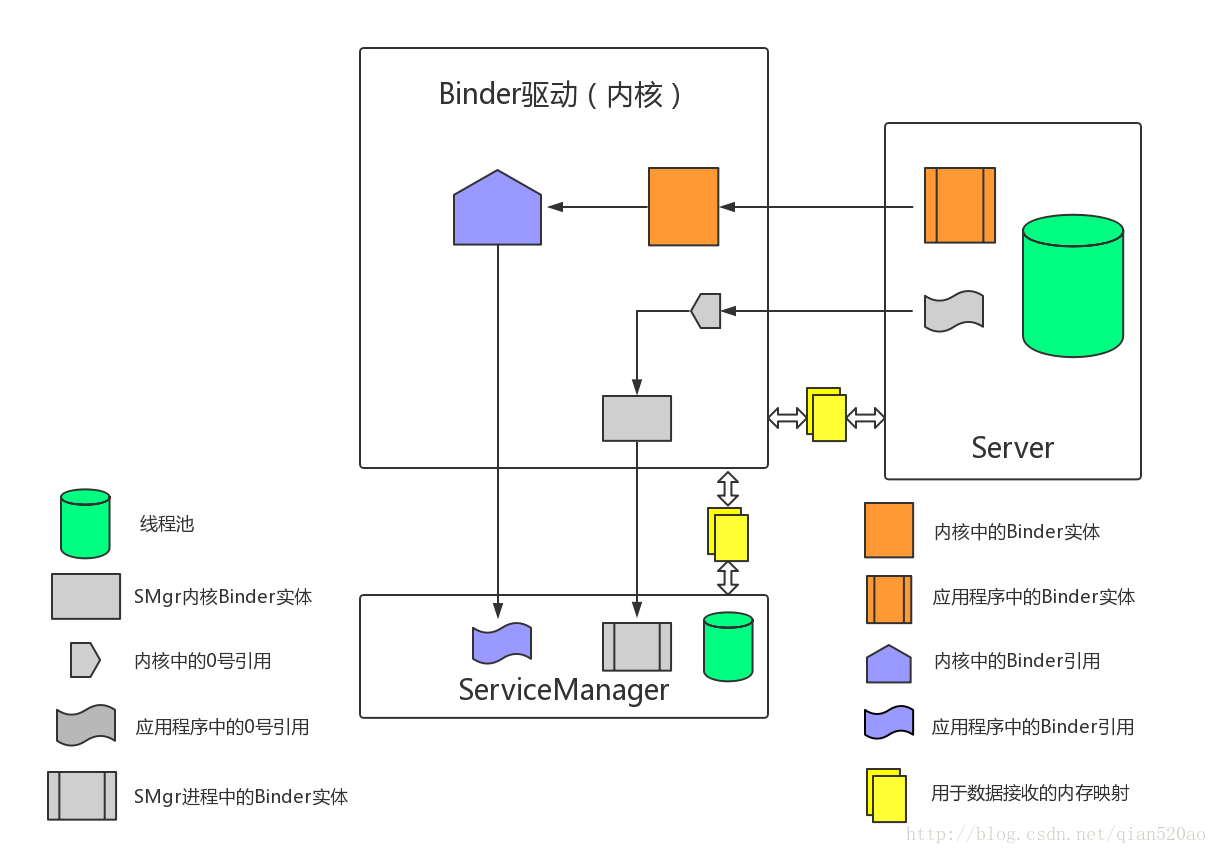
整个通信步骤如下：

1. SM建立(建立通信录)；首先有一个进程向驱动提出申请为SM；驱动同意之后，SM进程负责管理Service（注意这里是Service而不是Server，因为如果通信过程反过来的话，那么原来的客户端Client也会成为服务端Server）不过这时候通信录还是空的，一个号码都没有。
2. 各个Server向SM注册(完善通信录)；每个Server端进程启动之后，向SM报告，我是zhangsan, 要找我请返回0x1234(这个地址没有实际意义，类比)；其他Server进程依次如此；这样SM就建立了一张表，对应着各个Server的名字和地址；就好比B与A见面了，说存个我的号码吧，以后找我拨打10086；
3. Client想要与Server通信，首先询问SM；请告诉我如何联系zhangsan，SM收到后给他一个号码0x1234；Client收到之后，开心滴用这个号码拨通了Server的电话，于是就开始通信了。

Server进程里面的Binder对象指的是Binder本地对象，Client里面的对象指的是Binder代理对象；在Binder对象进行跨进程传递的时候，Binder驱动会自动完成这两种类型的转换；因此Binder驱动必然保存了每一个跨越进程的Binder对象的相关信息；在驱动中，Binder本地对象的代表是一个叫做binder\_node的数据结构，Binder代理对象是用binder\_ref代表的；有的地方把Binder本地对象直接称作Binder实体，把Binder代理对象直接称作Binder引用（句柄）

**ServiceManager 与实名Binder**

细心的读者可能会发现其中的蹊跷：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。



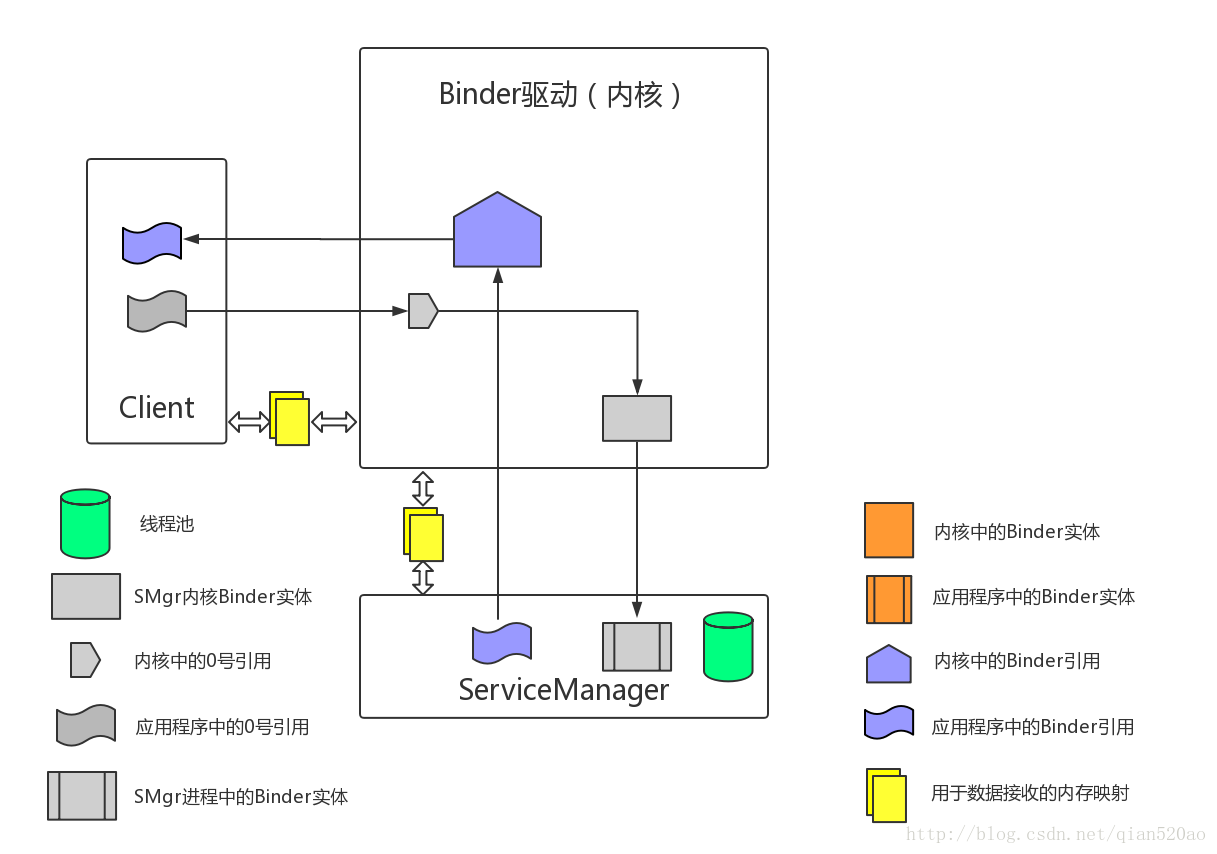
* 首先，Server在自己的进程中向Binder驱动申请创建一个Server的Binder的实体。
* Binder驱动为这个Server创建位于内核中的Binder实体节点以及Binder的引用。(在Binder驱动中创建一块内存)
* 然后Server通过0这个引用号和SMgr的Binder通信将名字和新建的引用打包传递给SM（实体没有传给SM），通知SM注册一个名叫XXX的Server。
* SM收到数据包后，从中取出Server名字和引用，填入一张查找表中。

Server初始化的时候，SMgr做了一下操作：

1. 为binder分配128k的内存
2. 通知binder驱动，使自身成为binder驱动的“DNS”
3. 维护一个监听Server的死循环，并且维护持有所有Server句柄的svclist
4. 添加Server的时候，进行权限，内存（充足）进行判断，如果没有添加过则将Server添加至svclist。

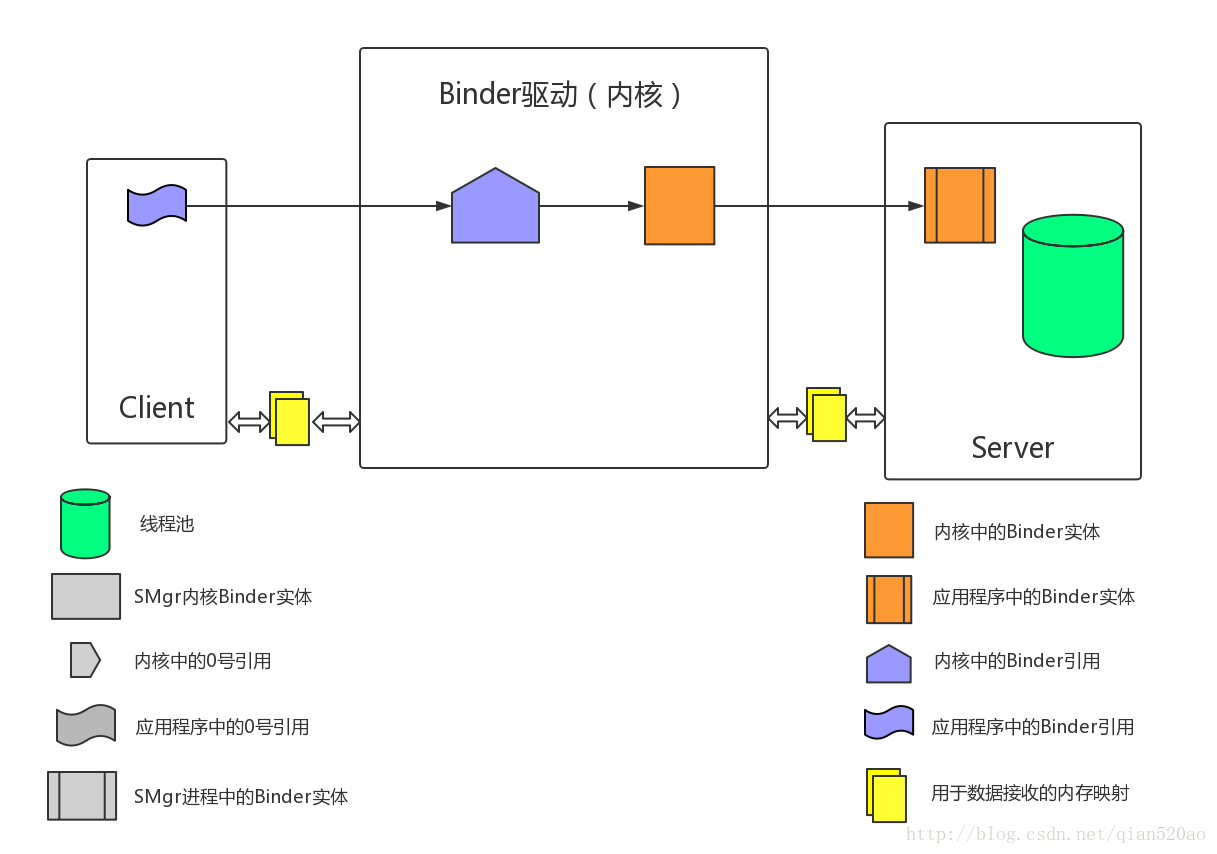
**Client 获得实名Binder的引用**

Server向SMgr注册了Binder引用及其名字后，Client就可以通过名字获得该Binder的引用了。Client也利用保留的0号引用向SMgr请求访问某个Binder：我申请获得名字叫张三的Binder的引用。SMgr收到这个连接请求，从请求数据包里获得Binder的名字，在查找表里找到该名字对应的条目，从条目中取出Binder的引用，将该引用作为回复发送给发起请求的Client。从面向对象的角度，这个Binder对象现在有了两个引用：一个位于SMgr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，就象java里一个对象存在多个引用一样。而且类似的这些指向Binder的引用是强类型，从而确保只要有引用Binder实体就不会被释放掉。通过以上过程可以看出，SMgr象个火车票代售点，收集了所有火车的车票，可以通过它购买到乘坐各趟火车的票-得到某个Binder的引用。



**Client 与 Server通讯**

Client向SM发送申请服务Server的请求，那么SM就可以在查找表中找到该Service的Binder引用，并把Binder引用(BpBinder)返回给Client，此时Client便可以通过这个引用向Server（间接）发起调用，Binder引用将参数包装然后交给驱动并获取Server的调用结果。



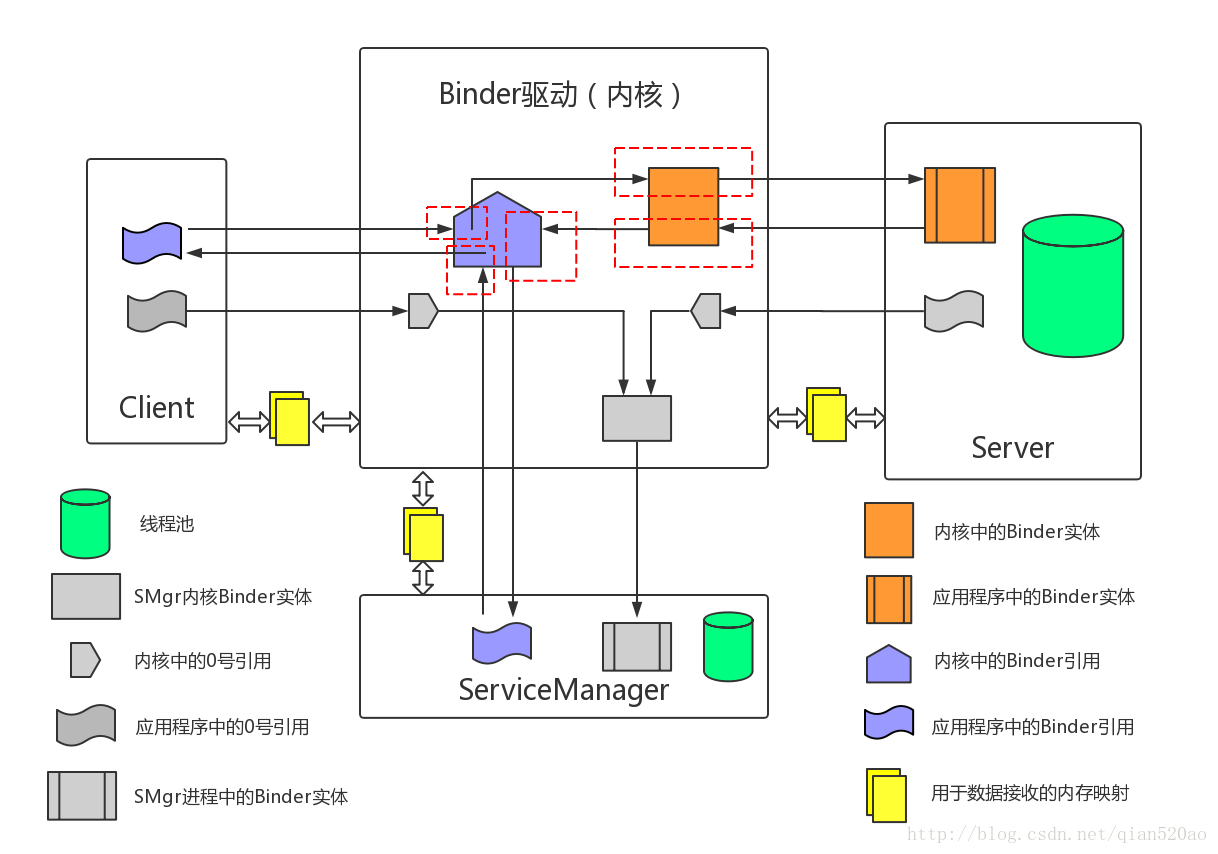
**Binder的线程管理**

每个Binder的Server进程会创建很多线程来处理Binder请求，可以简单的理解为创建了一个Binder的线程池吧（虽然实际上并不完全是这样简单的线程管理方式），而真正管理这些线程并不是由这个Server端来管理的，而是由Binder驱动进行管理的。

一个进程的Binder线程数默认最大是16，超过的请求会被阻塞等待空闲的Binder线程。理解这一点的话，你做进程间通信时处理并发问题就会有一个底，比如使用ContentProvider时（又一个使用Binder机制的组件），你就很清楚它的CRUD（创建、检索、更新和删除）方法只能同时有16个线程在跑。(应用与ContentProvider为不同进程时)

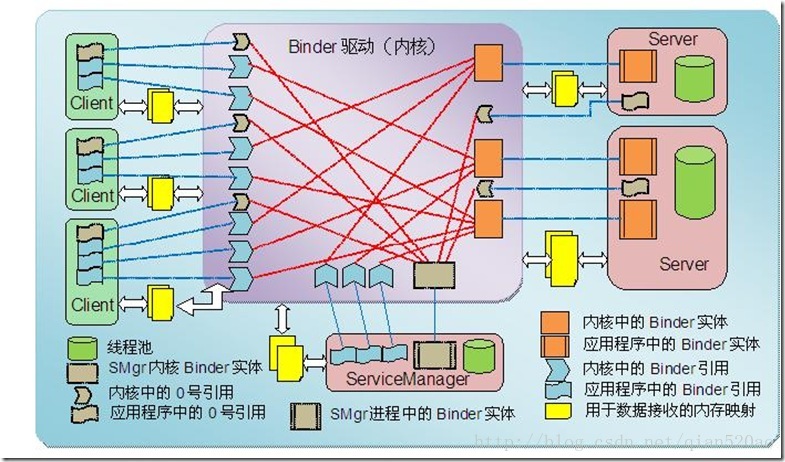
摘自goeasyway【[Android面试一天一题–Binder](http://www.jianshu.com/p/c7bcb4c96b38)】

**整体通讯流程图**



参考超强的女博主[Android Bander设计与实现]   
(<http://blog.csdn.net/universus/article/details/6211589> “optional title”)

最后可以通过该博主写的博文进一步对Binder进行探索和了解，比如Binder 内存映射和接收缓存区管理，Binder在驱动/传输中的表述等，因为该篇文章主要是在应用层做的一个分析和总结。下面来看一下原图（上面3张图借鉴此图）。



**总结**

本篇博客图文并茂的总结了Binder应用层的总体流程，总的来说算是对大佬们的文章做的一个总结笔记，Binder非三日之功可破冰，个人觉得在it上的学习也是一样，可能短期的学习并不能展现出有多大改变，但是循序渐进，量变引起质变。骐骥一跃,不能十步;驽马十驾,功在不舍。