有网友问到：“Android另起炉灶开发了Binder驱动，而没有采用已有的方案，而D-Bus这样的方案也可以实现Binder的功能，是出于什么原因和什么考虑？安全性？性能？”

在开始回答**前，先简单概括性地说说Linux现有的所有进程间IPC方式：**

1. **管道：**在创建时分配一个page大小的内存，缓存区大小比较有限；
2. **消息队列**：信息复制两次，额外的CPU消耗；不合适频繁或信息量大的通信；
3. **共享内存**：无须复制，共享缓冲区直接付附加到进程虚拟地址空间，速度快；但进程间的同步问题操作系统无法实现，必须各进程利用同步工具解决；
4. **套接字**：作为更通用的接口，传输效率低，主要用于不通机器或跨网络的通信；
5. **信号量**：常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
6. **信号**: 不适用于信息交换，更适用于进程中断控制，比如非法内存访问，杀死某个进程等；

Android的内核也是基于Linux内核，为何不直接采用Linux现有的进程IPC方案呢，难道Linux社区那么多优秀人员都没有考虑到有Binder这样一个更优秀的方案，是google太过于牛B吗？事实是真相并非如此，请细细往下看，您就明白了。

接下来正面回答这个问题，从5个角度来展开对Binder的分析：

（1）从性能的角度

数据拷贝次数：Binder数据**拷贝只需要一次**，而管道、消息队列、Socket都需要2次，但共享内存方式一次内存拷贝都不需要；从性能角度看，Binder性能仅次于共享内存。

（2）从稳定性的角度

Binder是基于C/S架构的，简单解释下C/S架构，是指客户端(Client)和服务端(Server)组成的架构，Client端有什么需求，直接发送给Server端去完成，架构清晰明朗，Server端与Client端相对独立，稳定性较好；而共享内存实现方式复杂，没有客户与服务端之别， 需要充分考虑到访问临界资源的并发同步问题，否则可能会出现死锁等问题；从这稳定性角度看，Binder架构优越于共享内存。

仅仅从以上两点，各有优劣，还不足以支撑google去采用binder的IPC机制，**那么更重要的原因是：**

（3）从安全的角度

传统Linux IPC的接收方无法获得对方进程可靠的UID/PID，从而无法鉴别对方身份；而Android作为一个开放的开源体系，拥有非常多的开发平台，App来源甚广，因此手机的安全显得额外重要；对于普通用户，绝不希望从App商店下载偷窥隐射数据、后台造成手机耗电等等问题，传统Linux IPC无任何保护措施，完全由上层协议来确保。

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**，前面提到C/S架构，**Android系统中对外只暴露Client端，Client端将任务发送给Server端，Server端会根据权限控制策略，判断UID/PID是否满足访问权限，目前权限控制很多时候是通过弹出权限询问对话框，让用户选择是否运行。**Android 6.0，也称为Android M，在6.0之前的系统是在App第一次安装时，会将整个App所涉及的所有权限一次询问，只要留意看会发现很多App根本用不上通信录和短信，但在这一次性权限权限时会包含进去，让用户拒绝不得，因为拒绝后App无法正常使用，而一旦授权后，应用便可以胡作非为。

针对这个问题，google在Android M做了调整，不再是安装时一并询问所有权限，而是在App运行过程中，需要哪个权限再弹框询问用户是否给相应的权限，对权限做了更细地控制，让用户有了更多的可控性，但**同时也带来了另一个用户诟病的地方，那也就是权限询问的弹框的次数大幅度增多。**对于Android M平台上，有些App开发者可能会写出让手机异常频繁弹框的App，企图直到用户授权为止，这对用户来说是不能忍的，用户最后吐槽的可不光是App，还有Android系统以及手机厂商，有些用户可能就跳果粉了，这还需要广大Android开发者以及手机厂商共同努力，共同打造安全与体验俱佳的Android手机。

传统IPC只能由用户在数据包里填入UID/PID；另外，可靠的身份标记只有由IPC机制本身在内核中添加。其次传统IPC访问接入点是开放的，无法建立私有通道。从安全角度，Binder的安全性更高。

说到这，可能有人要反驳，Android就算用了Binder架构，而现如今Android手机的各种流氓软件，不就是干着这种偷窥隐射，后台偷偷跑流量的事吗？没错，确实存在，但这不能说Binder的安全性不好，因为Android系统仍然是掌握主控权，可以控制这类App的流氓行为，只是对于该采用何种策略来控制，在这方面android的确存在很多有待进步的空间，这也是google以及各大手机厂商一直努力改善的地方之一。在Android 6.0，google对于app的权限问题作为较多的努力，大大收紧的应用权限；另外，在**Google举办的Android Bootcamp 2016**大会中，google也表示在Android 7.0 （也叫Android N）的权限隐私方面会进一步加强加固，比如SELinux，Memory safe language(还在research中)等等，在今年，google推出了Android N。

话题扯远了，继续说Binder。

（4）从语言层面的角度

大家多知道Linux是基于C语言(面向过程的语言)，而Android是基于Java语言(面向对象的语句)，而对于Binder恰恰也符合面向对象的思想，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。可以从一个进程传给其它进程，让大家都能访问同一Server，就像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。**从语言层面，Binder更适合基于面向对象语言的Android系统，对于Linux系统可能会有点“水土不服”。

另外，Binder是为Android这类系统而生，而并非Linux社区没有想到Binder IPC机制的存在，对于Linux社区的广大开发人员，我还是表示深深佩服，让世界有了如此精湛而美妙的开源系统。也并非Linux现有的IPC机制不够好，相反地，经过这么多优秀工程师的不断打磨，依然非常优秀，每种Linux的IPC机制都有存在的价值，同时在Android系统中也依然采用了大量Linux现有的IPC机制，根据每类IPC的原理特性，因时制宜，不同场景特性往往会采用其下最适宜的。比如在**Android OS中的Zygote进程的IPC采用的是Socket（套接字）机制**，Android中的**Kill Process采用的signal（信号）机制**等等。而**Binder更多则用在system_server进程与上层App层的IPC交互。**

(5) 从公司战略的角度

总所周知，Linux内核是开源的系统，所开放源代码许可协议GPL保护，该协议具有“病毒式感染”的能力，怎么理解这句话呢？受GPL保护的Linux Kernel是运行在内核空间，对于上层的任何类库、服务、应用等运行在用户空间，一旦进行SysCall（系统调用），调用到底层Kernel，那么也必须遵循GPL协议。

而Android 之父 Andy Rubin对于GPL显然是不能接受的，为此，Google巧妙地将GPL协议控制在内核空间，将用户空间的协议采用Apache-2.0协议（允许基于Android的开发商不向社区反馈源码），同时在GPL协议与Apache-2.0之间的Lib库中采用BSD证授权方法，有效隔断了GPL的传染性，仍有较大争议，但至少目前缓解Android，让GPL止步于内核空间，这是Google在GPL Linux下 开源与商业化共存的一个成功典范。

有了这些铺垫，我们再说说Binder的今世前缘

Binder是基于开源的 OpenBinder实现的，OpenBinder是一个开源的系统IPC机制,最初是由 Be Inc. 开发，接着由Palm, Inc.公司负责开发，现在OpenBinder的作者在Google工作，既然作者在Google公司，在用户空间采用Binder 作为核心的IPC机制，再用Apache-2.0协议保护，自然而然是没什么问题，减少法律风险，以及对开发成本也大有裨益的，那么从公司战略角度，Binder也是不错的选择。

另外，再说一点关于OpenBinder，在**2015年OpenBinder以及合入到Linux Kernel主线 3.19版本**，这也算是Google对Linux的一点回馈吧。

综合上述5点，可知Binder是Android系统上层进程间通信的不二选择。

接着，讨论下网友提到的D-Bus

也采用C/S架构的IPC机制，**D-Bus**是在用户空间实现的方法，效率低，消息拷贝次数和上下文切换次数都明显多过于Binder。针对D-Bus这些缺陷，于是就产生了**kdbus**，这是D-Bus在内核实现版，效率得到提升，与Binder一样在内核作为字符设计，通过open()打开设备，mmap()映射内存。

（1）kdbus在进程间通信过程，Client端将消息在内存的消息队列，可以存储大量的消息，Server端不断从消息队里中取消息，大小只受限内存；

（2）Binder的机制是每次通信，会通信的进程或线程中的todo队里中增加binder事务，并且每个进程所允许Binder线程数，google提供的默认最大线程数为16个，受限于CPU，由于线程数太多，增加系统负载，并且每个进程默认分配内存。

而kdbus对于内存消耗较大，同时也适合传输大量数据和大量消息的系统。Binder对CPU和内存的需求比较低，效率比较高，从而进一步说明Binder适合于移动系统Android，但是，也有一定缺点，就是不同利用Binder输出大数据，比如利用Binder传输几M大小的图片，便会出现异常，虽然有厂商会增加Binder内存，但是也不可能比系统默认内存大很多，否则整个系统的可用内存大幅度降低。

最后，简单讲讲Android Binder架构

Binder在Android系统中江湖地位非常之高。**在Zygote孵化出system_server进程后，在system_server进程中出初始化支持整个Android framework的各种各样的Service，而这些Service从大的方向来划分，分为Java层Framework和Native Framework层(C++)的Service，几乎都是基于BInder IPC机制。**

1. **Java framework：作为Server端继承(或间接继承)于Binder类，Client端继承(或间接继承)于BinderProxy类。**例如 ActivityManagerService(用于控制Activity、Service、进程等) **这个服务作为Server端，间接继承Binder类，而相应的ActivityManager作为Client端，间接继承于BinderProxy类。** 当然还有PackageManagerService、WindowManagerService等等很多系统服务都是采用C/S架构；

2. **Native Framework层：这是C++层，作为Server端继承(或间接继承)于BBinder类，Client端继承(或间接继承)于Bp****Binder。**例如MediaPlayService(用于多媒体相关)作为Server端，继承于BBinder类，而相应的MediaPlay作为Client端，间接继承于BpBinder类。

总之，一句话"无Binder不Android"。

总结来说：

1、首先澄清一点，Android没有另起炉灶，Binder机制源于OpenBinder.

2、Binder与传统IPC机制

那么，与Linux上传统的IPC机制，比如System+V，Socket相比，Binder好在哪呢？

1. 性能；Binder传输只需要一次copy；socket两次，别小看这一倍带来的差距。对于移动设备，性能一直是个大问题；想一下Android绘制界面的时候都需要与WindowManager进行跨进程通信，如果这里效率不高，那岂不是卡死？
2. 安全性；Binder机制对于通信双方的身份是内核进行校检支持的；socket方式只需要知道地址都可以连接；安全机制需要上层协议架设。
3. 易用性；共享内存不需要copy性能够高，可是使用复杂；B%2FS模式的通信，如果管道%2F消息队列还得进行包装；另外，Binder使用面向对象的方式设计，进行一次远程过程调用就好像直接调用本地对象一样，异常方便。

另外，引用+Brian+Swetland+大神的回答：

大意就是：

1. 避免内核空间到数据接受端的直接的数据拷贝；数据接受端接收数据的时候，由于数据大小不确定，要么分配一个很大的空间装数据，要么动态扩容；两种方式都有问题；Binder使用mmap直接

把接受端的内存映射到内存空间，避免了数据的直接拷贝；另外通过data_buffer等方式让数据仅包含定长的消息头，解决了接受端内存分配的问题。

2. 需要管理跨进程传递的代理对象的生命周期；这一点其他机制无法完成；Binder驱动通过引用计数技术解决这个问题。