Android 面试考 Binder，看这一篇就够了

原创 2017-12-11 YogiAi 小专栏平台

> 本文作者小米 MIUI 系统工程师 **YogiAi**，作者开辟了《从源码角度看 Android》 专栏，以每周一个精品技术文更新，带你一起分析 AMS 中的 ContentProvider, Activity 的生命周期，广播，ANR 发生时的操作打印，Service, Activity 栈管理, Process 机制，所谓的保活原理等等。**今天作者用一个面试者的角度图文并茂给大家分享了 Binder 的各种姿势，以下为正文：**

简介

Android 的世界中，每个应用都运行在自己的进程中，都拥有着一个独立的Dalvik 虚拟机实例，每一个实例都作为着一个独立的进程执行，基于着这种进程"沙箱"机制，Android 得以将保证各个应用各司其职，不轻易的受到来自外部环境的安全隐患干扰（当然沙箱机制还是不够的，Anroid 还有 Cap、SELinux 等来保证系统安全）。

然而，作为一个完整的应用，又不可避免的需要同外部的应用进行数据交换。比方说应用想要在系统成功启动后做某些初始化的操作，那么应用就可以选择动态或者静态的注册一个广播，待系统成功启动后，system_server 进程会通过广播将数据发送到我们应用，最后调用到 onReceive 方法；又比方说，在从用户请求到相关的权限后，我们想要获取到手机中的联系人列表信息，那么应用就可以通过传入正确的 Uri，然后从 ContentProvider.query 方法返回的 Cursor 对象中一一读取出联系人信息。初级以上的开发者都知道，这些操作都是四大组件的基本功能，而这些功能的实现主要就是依赖了 Binder 进程间通信的能力。可以这么说，从用户最常见的Activity到开发者们常用到的AIDL，再到系统的其它生僻模块，Binder 的足迹无处不在。

Binder 的重要性我就不再多讲。只说在面试中，无论是面安卓应用开发还是安卓系统开发，倘若能够和面试官在谈论Binder时谈笑风生，至少会给面试官一个良好的技术印象。

今年年中的时候我又重读了老罗的那本经典，并将 Binder 源码和对应注释上传到了 GitHub: Binder_SourceCode ，如果大家觉着这些面试题不全面或者想要看更多细节的话，可以当参考看看~

那么接下来我们就开始从各个面试题来了解 Binder 的方方面面，这些面试题我会不断的补充，也欢迎大家对其中的错误进行指正

Android O上对 Binder 做了较大的改动，这些关于新特性的问题以后我会补上~

Binder 的概念?

- Binder 的前身是 OpenBinder，它是先后由Be和Palm公司进行开发的IPC开源系统，非常适合运行在小型手持设备上，现在由谷歌负责开发并一直被用作Android系统的主要IPC手段
- "binder" 这个单词有着粘合剂的意思，寓意为一个系统的主干，可以将系统中的不同模块粘合成一个整体
- Android 系统中每个进程都维护着一个线程池用来响应别的进程的请求；Binder 框架则负责管理对象间的引用计数、内核空间缓冲区和通信模型等
- Binder 使用 ServiceManager 来管理各个系统服务；当进程的服务被注册时， binder_node 实体节点会被插入到内核空间的红黑树中，ServiceManager 则会在进程的内核空间中添加ref引用

Linux 原有的 IPC 机制为何不用，却偏偏使用 Binder 呢?

- 传输性能：Binder 只需拷贝一次数据；虽然共享内存只需一次，但是使用起来很复杂

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

- 安全性

传统的IPC方式没有严格的安全措施；比方说Socket，IP地址可以由客户端手动填入，调用者的身份很容易进行伪造

Android会为每个已安装的应用程序添加属于自己的UID，Binder 基于此会根据请求端的UID来鉴别调用者的身份，保障了安全性

- 易用性

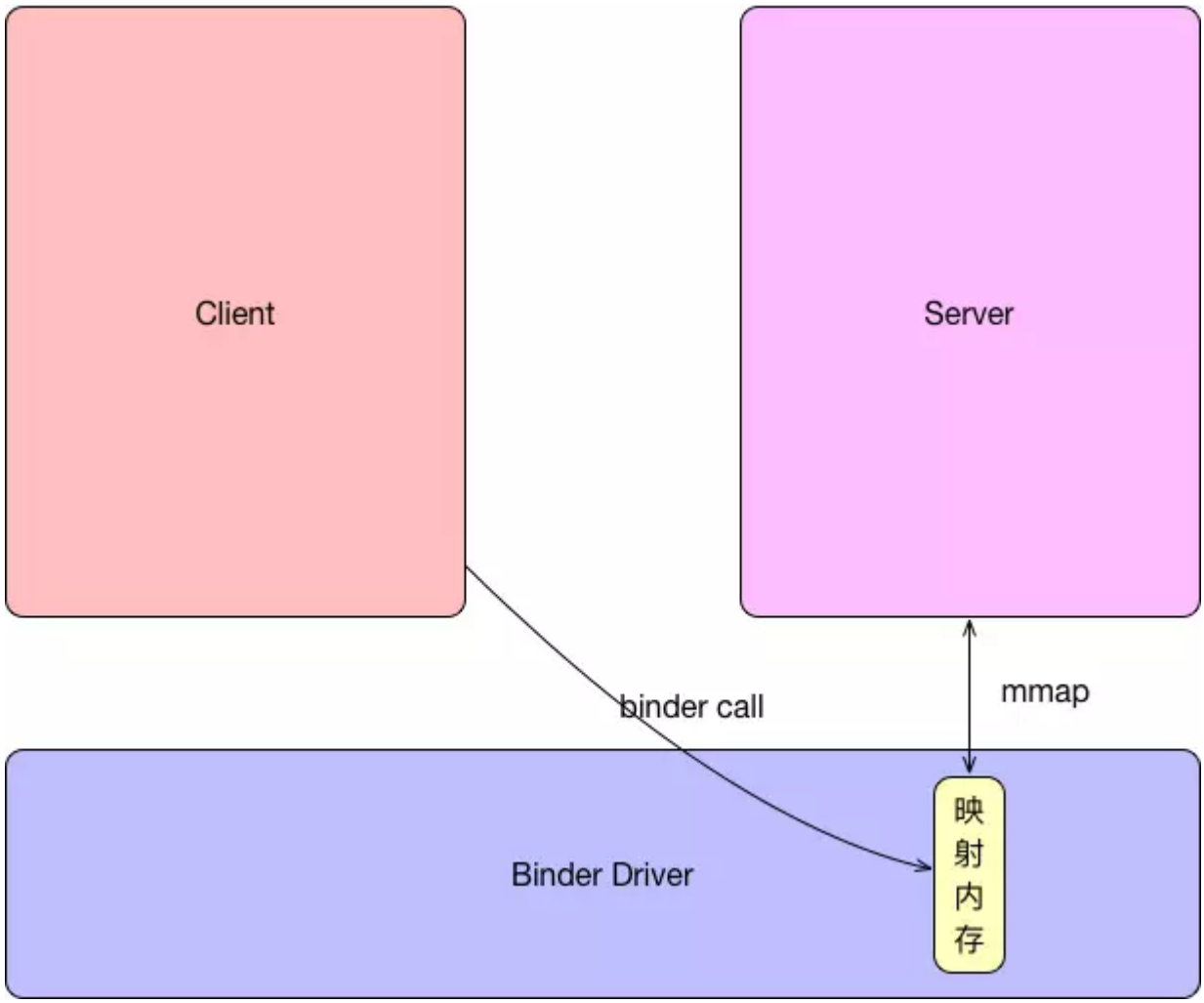
Binder 使用的是C/S通信方式，一个进程可以开启服务专门负责处理某个模块的业务，多个进程可以作为Client同时向Server发起请求

使用了面向对象的设计，发起一次binder call就像在调用本地方法一样简单

Binder Driver 如何在内核空间中做到一次拷贝的?

当Client向Server发送数据时，Client会先从自己的进程空间把通信数据拷贝到内核空间，因为Server和内核共享数据，所以不再需要重新拷贝数据，而是直接通过内存地址的偏移量直接获取到数据地址。总体来说只拷贝了一次数据。

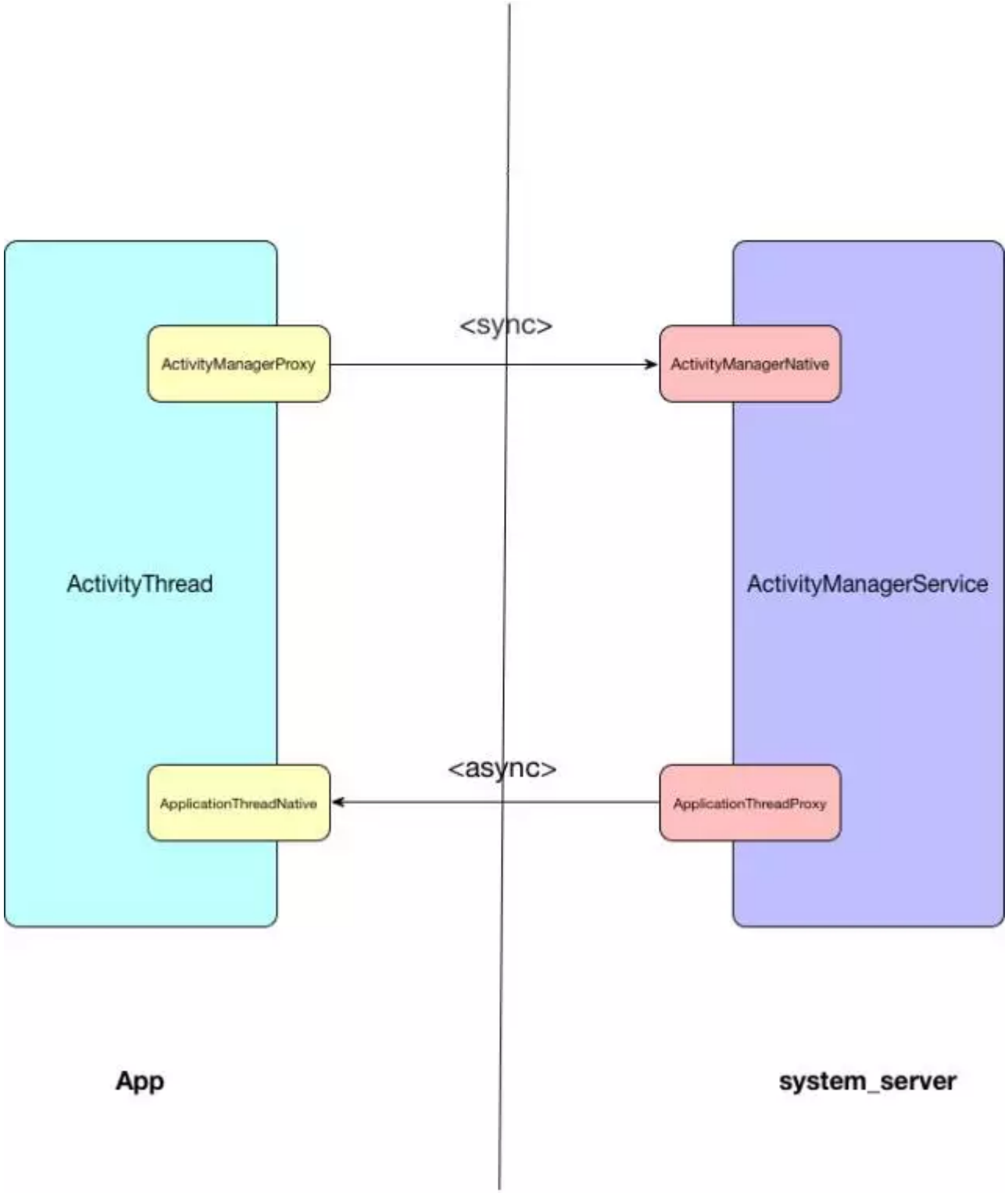
Server和内核空间之所以能够共享一块空间数据主要是通过binder_mmap来实现的。它的主要功能是在内核的虚拟地址空间申请一块和用户虚拟内存相同大小的内存，然后再申请一个page大小的内存，将它映射到内核虚拟地址空间和用户虚拟内存空间，从而实现了用户空间缓冲和内核空间缓冲同步的功能。



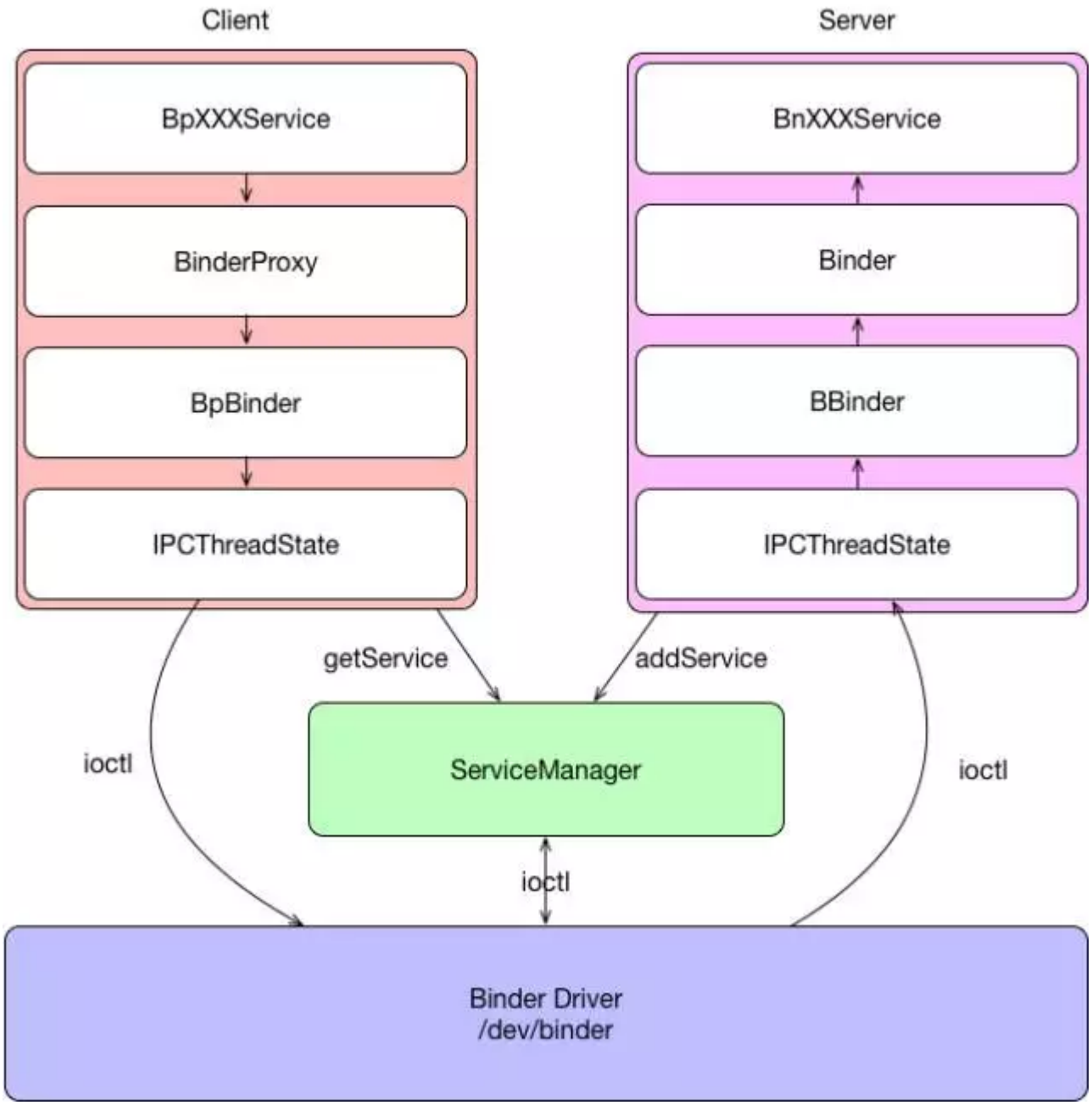
四大组件中常见的 2 个 Binder 服务是?

一个是实现了IActivityManager接口的ActivityManagerNative，ActivityManagerService是它的子类，提供了AMS中的所有服务，从APP调用binder call到AMS都是同步的，APP需要阻塞等待AMS执行完毕

一个是实现了IApplicationThread接口的ApplicationThreadNative，ActivityThread中的内部类ApplicationThread是它的子类，AMS可以发起异步请求到APP，不需要等待APP执行完成

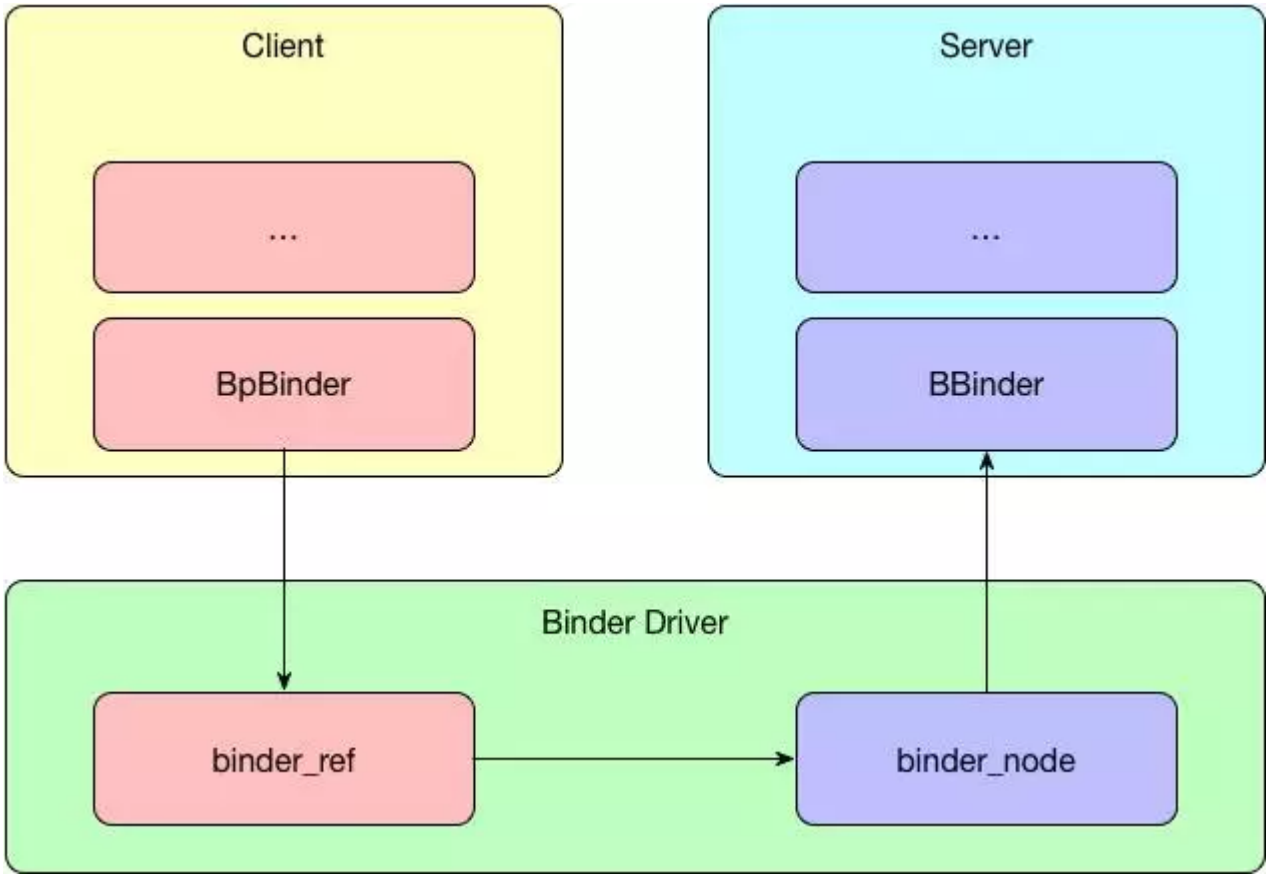


BpBinder 与 BBinder 有什么区别?



- BpBinder: Binder的代理对象，内部有一个成员变量mHandle，记录着袁侧滑盖服务对象的handle
- BBinder: Binder 本地服务对象

binder_ref 与 binder_node 有什么区别?



- binder_ref与binder_node都存在于内核空间
- binder_node是实体对象、binder_ref是引用对象
- binder_node被binder_ref引用，binder_ref被BpBinder引用

应用该如何获取和添加 Binder 服务?

获取与添加 Binder 服务的操作是交给"大管家"ServiceManager去做的；ServiceManager进程其启动后会通过 binder_loop 睡眠等待客户端的请求，如果进程被客户端唤醒就会调用 svcmgr_handler 来处理获取或者添加服务的请求

ServiceManager 也是一种 Binder 服务，应用该如何获取它的服务呢?

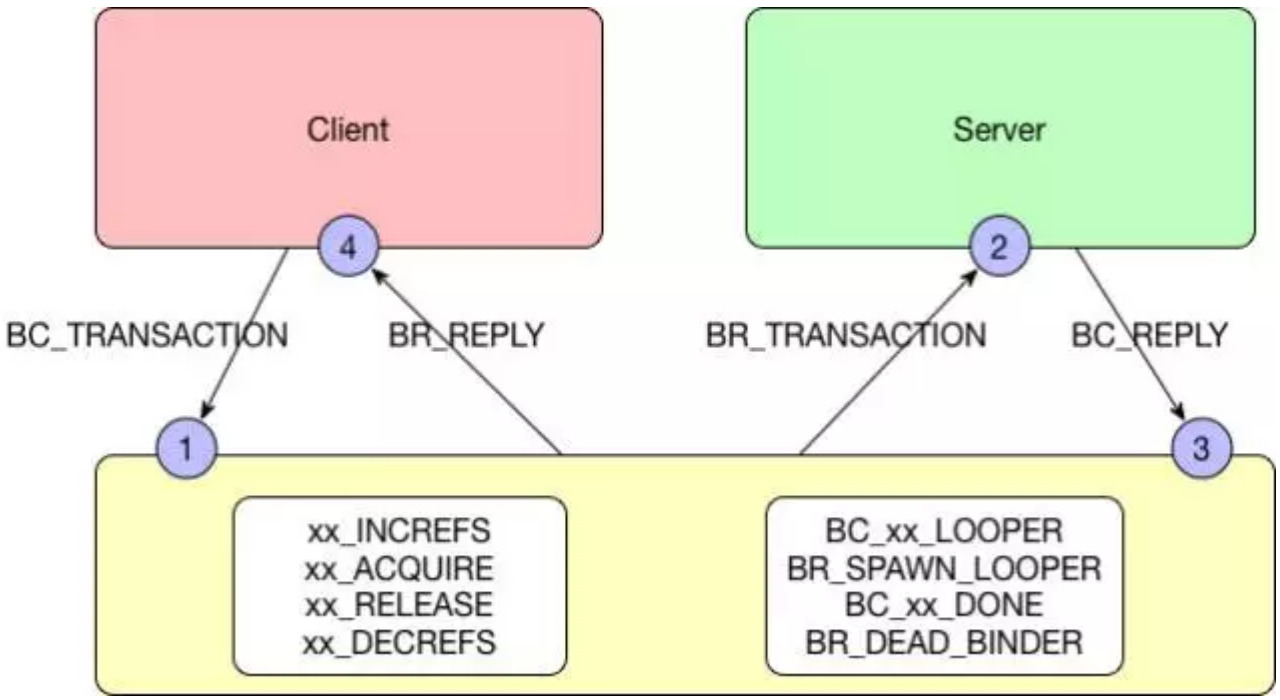
典型的一个鸡生蛋还是蛋生鸡问题

当应用获取ServiceeManager服务的代理时，它的handle句柄固定为0，所以才不需要去查找，具体见以下代码

```
private static IServiceManager getIServiceManager() {
    if (sServiceManager != null) {
        return sServiceManager;
    }
    // Find the service manager
```

```
    sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());
    return sServiceManager;
}
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& /*caller*/)
{
 return getStrongProxyForHandle(0);
}
```

Binder 都有什么主要的协议?



协议	意义
xx_TRANSACTION	传输请求数据
xx_REPLY	回复数据
xx_INCREFS	弱引用计数+1
xx_DECREFS	弱引用计数-1
xx_ACQUIRE	强引用计数+1
xx_RELEASE	强引用计数-1
BR_SPAWN_LOOPER	Binder驱动通知接收方进程创建一个新线程
BR_TRANSACTION_COMPLETE	Binder驱动通知发送方发送成功
BR_DEAD_REPLY	告知发送方服务进程已经死亡
BR_DEAD_BINDER	向发送方发送死亡通知

BC_ENTER_LOOPER	通知Binder驱动主线程已经进入循环
BC_EXIT_LOOPER	通知Binder驱动主线程已经退出循环
BC_REGISTER_LOOPER	通知Binder驱动新线程已经创建
BC_FREE_BUFFER	通知Binder驱动释放内存

Binder 协议中BC_与BR_开头的协议都有什么区别?

- BC_: 全称Binder Command, 进程发送给Binder驱动数据时携带的协议
- BR_: 全称Binder Return，Binder驱动发送给进程数据时携带的协议

Binder 服务在调用期间抛出了RuntimeException异常，服务端会Crash么?

服务端并不会Crash，RuntimeException被Binder服务端线程捕捉，随后将异常信息写入到reply中，发回Binder客户端进程，最后会客户端binder线程抛出这个异常，如果没有捕捉这个RuntimeException，那么Binder客户端进程会Crash

以下为AMS.startService出现异常为例进行讲解

```
// ---frameworks/base/core/java/android/app/ContextImpl.java
@Override
public ComponentName startService(Intent service) {
    return startServiceCommon(service, mUser);
}
private ComponentName startServiceCommon(Intent service, UserHandle user) {
    ComponentName cn = ActivityManagerNative.getDefault().startService(
        mMainThread.getApplicationThread(), service, service.resolveTypeIfNeeded(
                getContentResolver()), getOpPackageName(), user.getIdentifier());
}
// ---frameworks/base/core/java/android/app/ActivityManagerNative.java
public ComponentName startService(IApplicationThread caller, Intent service,
        String resolvedType, String callingPackage, int userId) throws RemoteException
{
    Parcel data = Parcel.obtain();
    // 此时还在Binder客户端进程，发送binder call到AMS
    mRemote.transact(START_SERVICE_TRANSACTION, data, reply, 0);
    reply.readException();
    return res;
}
// ---frameworks/base/core/java/android/os/Binder.java
```

```
private boolean execTransact(int code, long dataObj, long replyObj,
        int flags) {
    Parcel data = Parcel.obtain(dataObj);
    try {
        res = onTransact(code, data, reply, flags);
    } catch (RemoteException e) {
    } catch (RuntimeException e) {
        if ((flags & FLAG_ONEWAY) != 0) {
            Log.w(TAG, "Caught a RuntimeException from the binder stub implementation.", e);
        } else {
            reply.setDataPosition(0);
            reply.writeException(e);
        }
        res = true;
    } catch (OutOfMemoryError e) {
    }
    return res;
}
/* ---此时已经在AMS binder 服务端，onTransact会调用AMS.startService方法，如果发生了异常将会被捕
捉---*/
// ---frameworks/base/core/java/android/os/Parcel.java
public final void writeException(Exception e) {
    int code = 0;
    // 异常转换成code
    writeInt(code);
}
// ---frameworks/base/core/java/android/app/ActivityManagerNative.java
public ComponentName startService(IApplicationThread caller, Intent service,
        String resolvedType, String callingPackage, int userId) throws RemoteException
{
    Parcel data = Parcel.obtain();
    mRemote.transact(START_SERVICE_TRANSACTION, data, reply, 0);
    // 返回到Binder客户端，首先去查看是否有异常
    reply.readException();
    return res;
}
// ---frameworks/base/core/java/android/os/Parcel.java
public final void readException(int code, String msg) {
    switch (code) {
        // 此处省略了各种判断，先进行检查随后会抛出异常
        case EX_SECURITY:
            throw new SecurityException(msg);
        ...
    }
}
```

客户端调用 Binder 接口后抛出的DeadObjectException是什么意思?

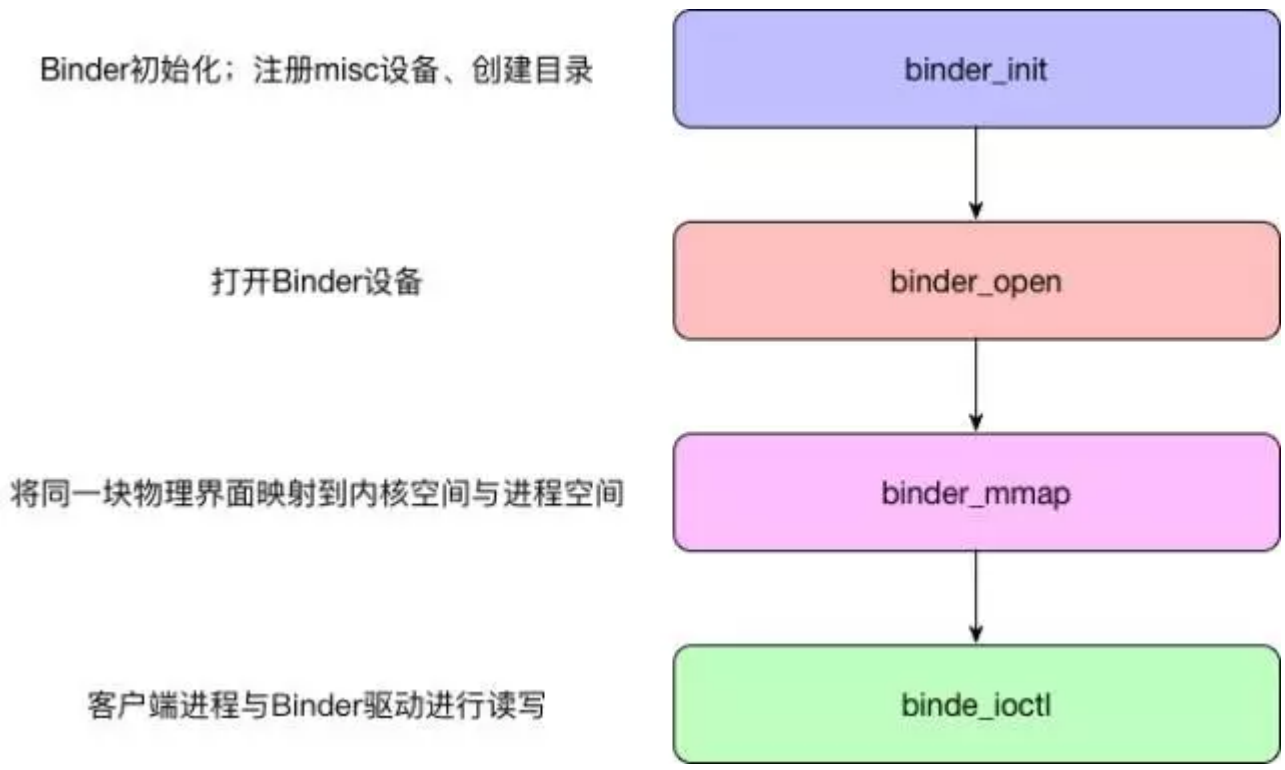
以下代码是一般binder调用函数的常见写法，一般都需要捕捉到DeadObjectException与RemoteException

```
private void binderCallSample() {
    try {
      // xxx
    } catch (DeadObjectException e) {
      // xxx
    } catch (RemoteException e) {
       // xxx
    }
}
```

之所以抛出DeadObjectException最常见的原因就是Binder服务端进程已经死亡。客户端进行binder调用时，Binder驱动发现服务端进程不能回应请求，那么就会抛出异常给客户端；

还有一些比较冷僻的原因，比如说服务端此时的缓存内存空间(1016K)已经被占满了，Binder驱动就认为服务端此时并不能处理这个调用，那么就会在C++层抛出DeadObjectException到Java层

Binder 驱动加载过程中有哪些重要的步骤?



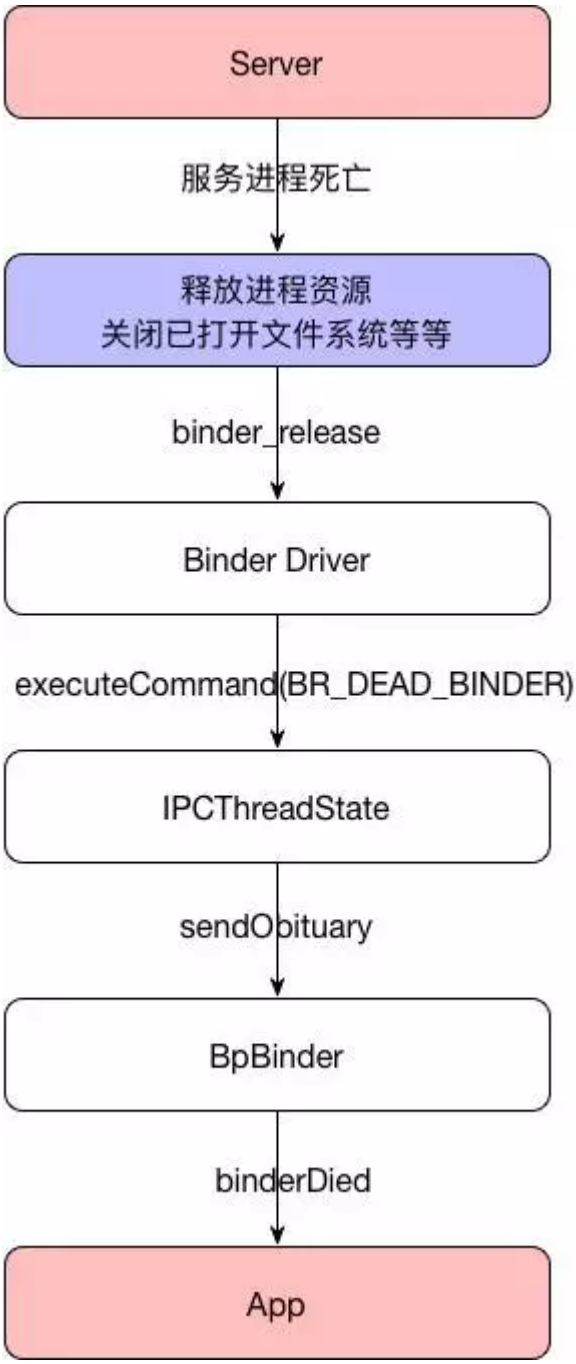
- binder_init: 初始化Binder驱动环境，内核工作队列、文件系统节点、misc设备等

- binder_open: 打开Binder设备，获取Binder驱动的文件描述符 (fd)
- binder_mmap: 将用户进程地址空间映射到Binder驱动设备内存。这也是Binder能够实现内存一次拷贝依赖的根本
- binder_ioctl: Binder驱动的核心功能，用来进行数据读写操作

Binder 的死亡通知机制的作用是什么，它是如何实现的?

Binder服务端进程死亡后，依赖着Binder实体对象的客户端代理对象也会失效。当Binder服务无效时，驱动程序会发送死亡通知给给各个已注册服务的客户端进程，以方便客户端做些销毁之类的操作

死亡通知机制最常用在APP与AMS服务，是典型的C/S架构，当APP端的进程死亡后，其ApplicationThreadNative会被销毁，随后Binder驱动会发出死亡讣告给AMS，方便清理已经失效了的四大组件及应用进程信息



Binder 驱动的加载阶段会打开 /dev/driver 文件，当服务端的进程死亡后，系统会进入清理阶段，关闭所有与进程相关联的资源，这其中就包含了关闭 /dev/driver 文件描述符的操作，随后就会调用到 Binder 驱动的binder_release方法。经过层层调用最后会调用到 binderDied 这个回调方法。如果App进程之前注册并实现过 DeathRecipient 这个接口，此时App进程就能够对Server进程的死亡作出处理。

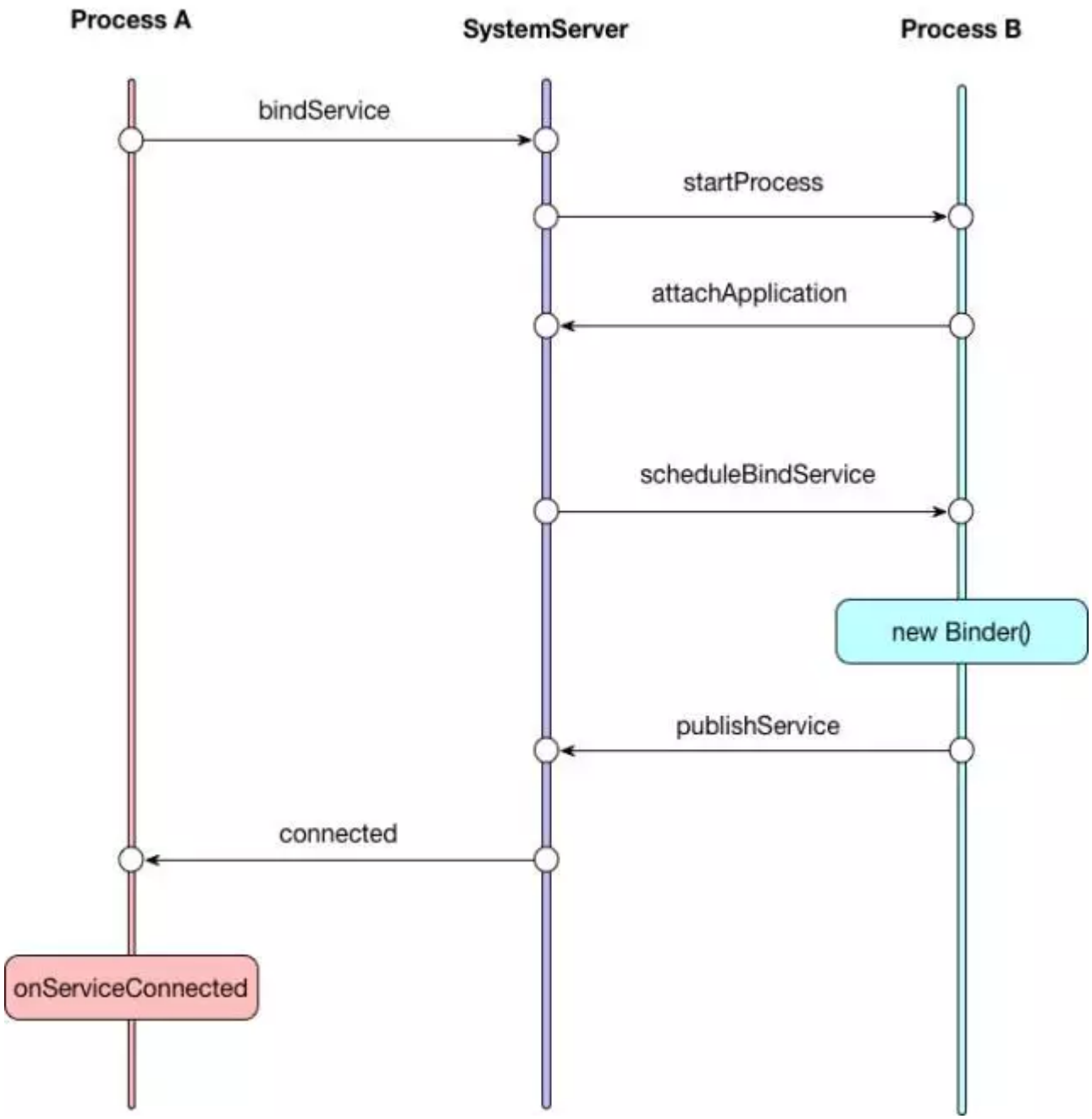
bindService所绑定的"服务概念和Binder中的服务Server有什么区别"?

bindService绑定的服务所指的是四大组件中的Service; 调用context.bindService方法可以让Activity与Service形成一种"绑定"的概念，这个概念是在Android Framework定义，形成"绑定"的关系之后，Activity除了能够和Service进行通信之外，它们所在的进程存活状态也会相关联。注意，这个Service不确定是运行在本地还是远端的，大部分情况下我们调用startService只是用来启动一

个在本地定义的Service组件。当使用bindService时，一般是需要调用Service通过onBind返回的Binder服务接口，以此实现Activity与Service之间的通信

Binder 的服务则是一个真正的C/S架构中的服务端角色。这个服务需要继承Binder类并实现一套服务接口才能生效。我们常见的ActivityManagerService正是一个继承了Binder类并运行在system_server的服务，用来提供四大组件以及进程方面的服务

注意，调用bindService之后，Service.onBind方法会返回Binder服务Stub对象，然后APP会在AMS进行登记，随后经过层层调用才会在Binder客户端调用onConnected方法，如果Binder客户端和其服务端在同一个进程中，客户端拿到的应该还是Stub对象，如果不再同一进程中，客户端拿到的应该是Binder服务的Proxy对象。



本质上，bindService所绑定的服务和Binder的服务属于同一类，他们实现进程间通信的原理都是借助了Binder这一套机制。

writeStrongBinder与readStrongBinder的作用和原理?

主要作用是为了实现两个进程之间的双工通信；还是以App客户端的ApplicationThread和system_server服务端的ActivityManagerService这两个Binder服务来举例子，这两个服务的接口的大致模式都相同，选用startActivity进行讲解。

```
// --frameworks/base/core/java/android/app/Instrumentation.java
public ActivityResult execStartActivity(
        Context who, IBinder contextThread, IBinder token, Activity target,
        Intent intent, int requestCode, Bundle options) {
    // ApplicationThread Stub对象
    IApplicationThread whoThread = (IApplicationThread) contextThread;
...
    try {
...
        int result = ActivityManagerNative.getDefault()
            .startActivity(whoThread, who.getBasePackageName(), intent,
                    intent.resolveTypeIfNeeded(who.getContentResolver()),
                    token, target != null ? target.mEmbeddedID : null,
                    requestCode, 0, null, options);
    } catch (RemoteException e) {
        throw new RuntimeException("Failure from system", e);
    }
    return null;
}
// --frameworks/base/core/java/android/app/ActivityManagerNative.java
public int startActivity(IApplicationThread caller, String callingPackage, Intent intent,
        String resolvedType, IBinder resultTo, String resultWho, int requestCode,
        int startFlags, ProfilerInfo profilerInfo, Bundle options) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
...
    // 将ApplicationThread Stub对象传到对端，方便AMS发送消息到客户端
    data.writeStrongBinder(caller != null ? caller.asBinder() : null);
...
    return result;
}
// --frameworks/base/services/core/java/com/android/server/am/ActivityStackSupervisor.java
final boolean realStartActivityLocked(ActivityRecord r,
        ProcessRecord app, boolean andResume, boolean checkConfig)
        throws RemoteException {
```

```
        ...
        // 这个app.thread对象正是从客户端writeStrongBinder传输到AMS中的Binder代理对象，使用它AMS
    可以 binder call到客户端
        app.thread.scheduleLaunchActivity(new Intent(r.intent), r.appToken,
            System.identityHashCode(r), r.info, new Configuration(mService.mConfiguration),
            new Configuration(stack.mOverrideConfig), r.compat, r.launchedFromPackage,
            task.voiceInteractor, app.repProcState, r.icicle, r.persistentState, results,
            newIntents, !andResume, mService.isNextTransitionForward(), profilerInfo);
        ...
    }
```

以上代码只是选用APP进行启动Activity操作期间与system_server进行双工通信的方式。值得注意的是，APP与AMS双工通信的方式都和启动Activity随后调用APP端的模式一致。

关于writeStrongBinder与readStrongBinder都是直接调用native方法进行实现的，看代码

```
// -------writeStrongBinder----------
// --frameworks/base/core/jni/android_os_Parcel.cpp
static void android_os_Parcel_writeStrongBinder(JNIEnv* env, jclass clazz, jlong nativePtr, jobject
object)
{
    Parcel* parcel = reinterpret_cast<Parcel*>(nativePtr);
    if (parcel != NULL) {
         // 主要的操作交给了ibindForJavaObject方法去做
        const status_t err = parcel->writeStrongBinder(ibinderForJavaObject(env, object));
        if (err != NO_ERROR) {
            signalExceptionForError(env, clazz, err);
        }
    }
}
// --frameworks/base/core/jni/android_util_Binder.cpp
sp<IBinder> ibinderForJavaObject(JNIEnv* env, jobject obj)
{
    if (obj == NULL) return NULL;
    // 如果写入的binder对象是实体，那么返回BBinder对象
    if (env->IsInstanceOf(obj, gBinderOffsets.mClass)) {
        JavaBBinderHolder* jbh = (JavaBBinderHolder*)
            env->GetLongField(obj, gBinderOffsets.mObject);
        return jbh != NULL ? jbh->get(env, obj) : NULL;
    }
    // 如果写入的binder对象是代理，那么返回BpBinder对象
    if (env->IsInstanceOf(obj, gBinderProxyOffsets.mClass)) {
```

```
        return (IBinder*)
            env->GetLongField(obj, gBinderProxyOffsets.mObject);
    }
    ALOGW("ibinderForJavaObject: %p is not a Binder object", obj);
    return NULL;
}
// --frameworks/native/libs/binder/Parcel.cpp
// 将Binder对象写入Parcel
status_t Parcel::writeStrongBinder(const sp<IBinder>& val)
{
    return flatten_binder(ProcessState::self(), val, this);
}
// --frameworks/native/libs/binder/Parcel.cpp
status_t flatten_binder(const sp<ProcessState>& /*proc*/,
    const sp<IBinder>& binder, Parcel* out)
{
    flat_binder_object obj;
     // 将binder对象转换成flat_binder_object对象，随后写入到parcel中
    return finish_flatten_binder(binder, obj, out);
}
// -------
// -------readStrongBinder
// -------
// --frameworks/base/core/jni/android_os_Parcel.cpp
static jobject android_os_Parcel_readStrongBinder(JNIEnv* env, jclass clazz, jlong nativePtr)
{
    Parcel* parcel = reinterpret_cast<Parcel*>(nativePtr);
    if (parcel != NULL) {
        return javaObjectForIBinder(env, parcel->readStrongBinder());
    }
    return NULL;
}
// --frameworks/native/libs/binder/Parcel.cpp
sp<IBinder> Parcel::readStrongBinder() const
{
    sp<IBinder> val;
    unflatten_binder(ProcessState::self(), *this, &val);
    return val;
}
// --frameworks/native/libs/binder/Parcel.cpp
status_t unflatten_binder(const sp<ProcessState>& proc,
    const Parcel& in, sp<IBinder>* out)
{
    const flat_binder_object* flat = in.readObject(false);
    // 将parcel中的数据转换为flat_binder_object对象
    return finish_unflatten_binder(
                static_cast<BpBinder*>(out->get()), *flat, in);
```

```
}
```

Parcel在Binder IPC担任着信息的载体的角色，StrongBinder正是通过它进行传输的，AMS中的publishProvider与publishService正是依靠此来实现的，有了它，App之间可以互相声明与调用跨进程的服务

每个进程最多存在多少个 Binder 线程，这些线程都被占满后会导致什么问题?

```
// --frameworks/native/libs/binder/ProcessState.cpp
#define DEFAULT_MAX_BINDER_THREADS 15
static int open_driver()
{
    int fd = open("/dev/binder", O_RDWR);
    if (fd >= 0) {
...
        size_t maxThreads = DEFAULT_MAX_BINDER_THREADS;
        result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
    }
    return fd;
}
```

Binder线程池中的线程数量是在Binder驱动初始化时被定义的; 进程池中的线程个数上限为15个，加上主Binder线程，一共最大能存在16个binder线程；

当Binder线程都在执行工作时，也就是当出现线程饥饿的时候，从别的进程调用的binder请求如果是同步的话，会在todo队列中阻塞等待，直到线程池中有空闲的binder进程来处理请求

使用 Binder 传输数据的最大限制是多少，被占满后会导致什么问题?

```
// --frameworks/native/libs/binder/ProcessState.cpp
#define BINDER_VM_SIZE ((1*1024*1024) - (4096 *2))
ProcessState::ProcessState()
    : mDriverFD(open_driver())
    , mVMStart(MAP_FAILED)
    , mThreadCountLock(PTHREAD_MUTEX_INITIALIZER)
    , mThreadCountDecrement(PTHREAD_COND_INITIALIZER)
    , mExecutingThreadsCount(0)
```

```
    , mMaxThreads(DEFAULT_MAX_BINDER_THREADS)
    , mManagesContexts(false)
    , mBinderContextCheckFunc(NULL)
    , mBinderContextUserData(NULL)
    , mThreadPoolStarted(false)
    , mThreadPoolSeq(1)
{
    // mmap the binder, providing a chunk of virtual address space to receive transactions.
    mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE,
mDriverFD, 0);
}
```

在调用mmap时会指定Binder内存缓冲区的大小为1016K；当服务端的内存缓冲区被Binder进程占用满后，Binder驱动不会再处理binder调用并在c++层抛出DeadObjectException到binder客户端

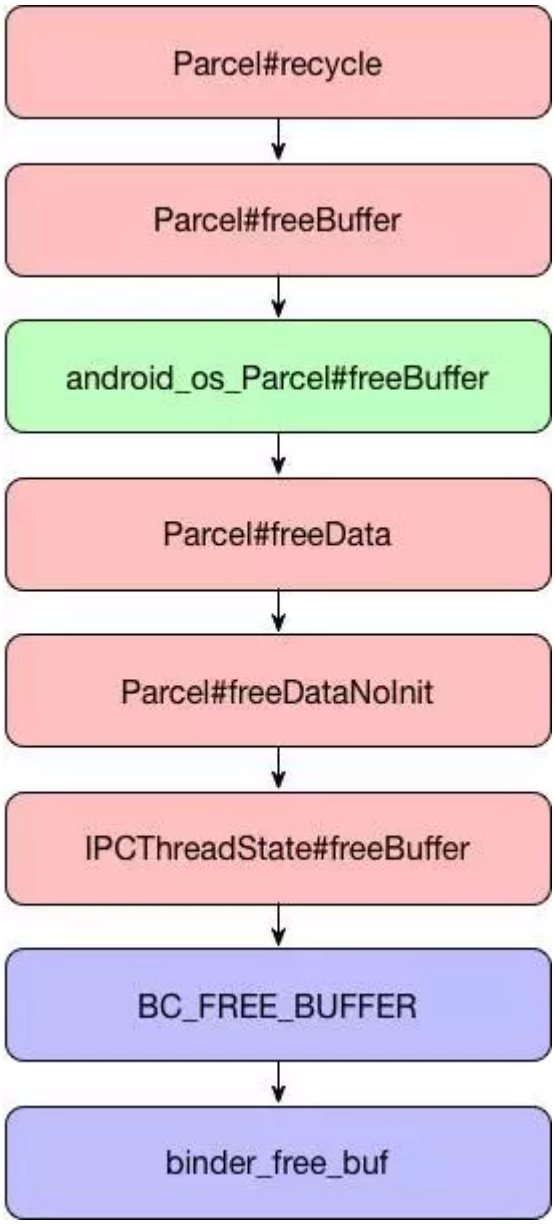
有一点注意的是，同步空间是1016K，异步空间只有它的一半，也就是508K

```
// --kernel/msm-3.18/drivers/staging/android/binder_alloc.c
struct binder_buffer *binder_alloc_new_buf(struct binder_alloc *alloc,
                    size_t data_size,
                    size_t offsets_size,
                    size_t extra_buffers_size,
                    int is_async)
{
...
if (is_async &&
    // 当Binder缓存空间不足时将会出现异常，Binder驱动不再会派发这个binder请求
    alloc->free_async_space < size + sizeof(struct binder_buffer)) {
    binder_alloc_debug(BINDER_DEBUG_BUFFER_ALLOC,
            "%d: binder_alloc_buf size %zd failed, no async space left\n",
             alloc->pid, size);
    eret = ERR_PTR(-ENOSPC);
    goto error_unlock;
}
error_unlock:
mutex_unlock(&alloc->mutex);
return eret;
}
```

Binder 驱动什么时候释放缓冲区的内存?

还是以ActivityManagerProxy的startActivity来举例子；是在binder call完成之后，调用Parcel.recycle来完成释放内存的。

```
public int startActivity(IApplicationThread caller, String callingPackage, Intent intent,
        String resolvedType, IBinder resultTo, String resultWho, int requestCode,
        int startFlags, ProfilerInfo profilerInfo, Bundle options) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    mRemote.transact(START_ACTIVITY_TRANSACTION, data, reply, 0);
    reply.recycle();
    data.recycle();
    return result;
}
```



为什么使用广播传输 2MB的Bitmap会抛异常，而使用AIDL生成的Binder接口传输Bitmap就不会抛异常呢?

一言以蔽之，通过Binder直接传输Bitmap，如果Bitmap的大于128K，那么传输Bitmap内容的方式就会使用ashmem，Binder只需要负责传输ashmem的fd到服务端即可。这种Binder+ashmem的方式在Android中其实很常见，比如四大组件中的ContentProvider.query方法，从Provider中查找的数据通常会超过1016K这个限制，使用ashmem不仅能突破这个限制，还有提高大量数据传输的效率

使用广播来传输跨进程传输数据的话则不一样，Bitmap是被填入到Bundle中，随后以Parcelable的序列化方式传输到AMS的，如果Bundle的数据量大于800K，就会抛出TransactionTooLargeException的异常

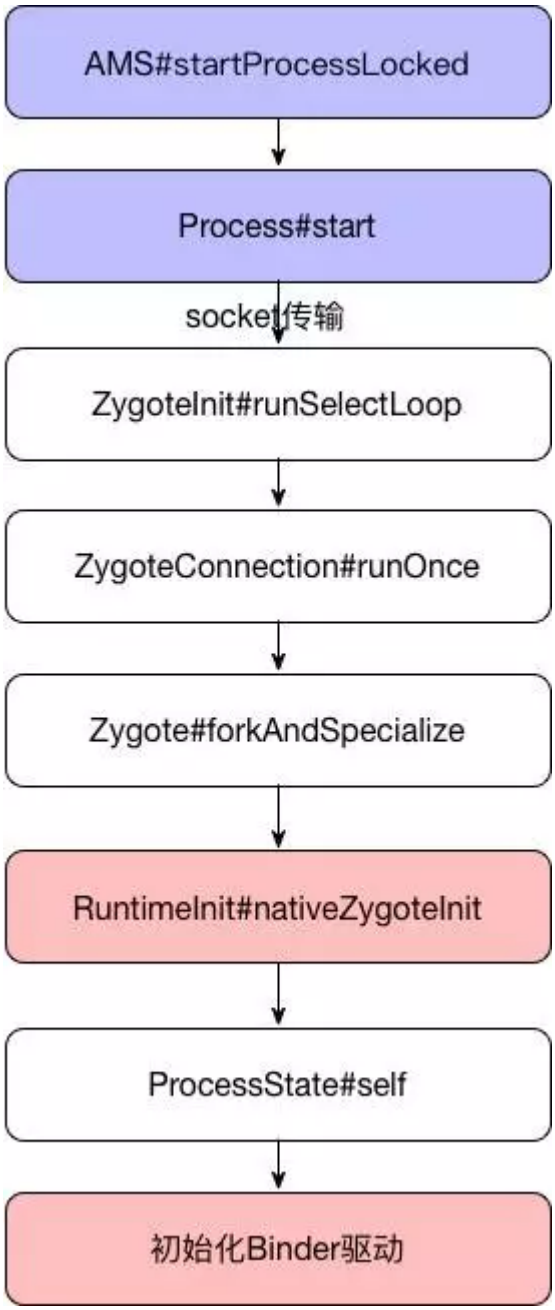
依据Bitmap是否过大来使用ashmem还是Binder的方式传输内容的逻辑在native层的Bitmap_createFromParcel，有兴趣的话大家去看看细节。

应用进程为什么支持 Binder 通信，直接可以使用四大组件呢?

首先一点，所有应用的进程都是通过调用AMS.startProcessLocked方法来fork Zygote进程创建的；Zygote在启动时会在preloadClasses中预先加载上千个类，而在fork子进程时，这些操作就不需要再做，大大节约了子进程的启动时间

应用进程的Binder驱动初始化的操作正是在zygote fork自身之后做的，注意一点的是，system_server与zygote的通信使用的是socket，之所以不使用binder来通信的原因很简单，socket的使用对于写这个功能的工程师相对来说更简单 :-)

在zygote进程的初始化操作完成后，zygote会通过socket返回给system_server pid, 随后AMS会将pid和应用的Application进行绑定。也就是说在应用调用Application.onCreate之前，Binder驱动的初始化就已经完成了，所以直接就可以使用Binder来通信。



本文是付费专栏《Android 面试指南》的一篇技术知识点剖析，在这里免费分享出来给大家。如果还想看到其他更多经常的技术文章，大家可以订阅。

《Android 面试指南》

由来自微信的张绍文、欢聚时代的Dclock、阿里巴巴的 jack、wingjay、悦跑圈的键盘男、Powerinfo 许建林(Piasy)、腾讯的宅男潇涧、映客 Android 架构师 Jacky 王世昌、GcsSloop、聚美优品的孤独狂饮、刚刚经历校招的拿到美团华为offer 的LTNS，即将入职的 MIUI 的Jucongyuan 一起创建小额付费专栏。

专栏除了对 Android 面试心得面试题的解析之外，还包含了对大公司小公司，职业发展等等各种探讨。订阅本专栏还将可以进入《Android面试指南》作者微信群，和腾讯、微信、阿里巴巴等等技术大牛一起交流，同

时在职业成长、求职过程中遇到的问题也可以直接发布到群里获取作者们的答疑，另外我们还会邀请小米、360、百度等其他技术大牛进群给大家答疑以及内推。

目前已经发布的文章：

- 《谈谈我对大公司、小公司的理解 》
- 《我们需要怎样的工程师，我们需要成为怎样的工程师？》
- 《Android 开发面试 “108” 问》
- 《应届生应该如何准备校招 Android 面试》
- 《微信资深工程师张绍文答读者问：T 型工程师更受大公司欢迎》
- 《答读者问：准备面试需要做哪些技术储备，面试官更加关心什么方面的技术点？》
- 《拿到了新浪微博、小米和京东的 Android offer，应该如何做选择？》
- 《当你求职跳槽时，需要注意的一些细节》
- 《2017 阿里、百度、美团、京东、今日头条、华为等秋季校招Android面经》
- 《Android面试指南 — 算法面试心得》

等等18个文章。

推荐阅读：

iOS 和 Android 开发是否要采用 React Native?
微信资深工程师张绍文答读者问：T 型工程师更受大公司欢迎
我们需要怎样的工程师，我们需要成为怎样的工程师?
Android 和 iOS 开发者如何转型做机器学习?
原来 Google 还可以这么用，每个程序员都应该学会这些技巧
腾讯架构师给应届生的一封信：谈谈开发入行选择、技术准备和技术信仰
一些优秀的 iOS 开发专栏推荐
一些优秀的 Android 开发专栏推荐
2018 年将至，iOS 工程师如何自我提高
亚马逊高级工程师：平凡是程序员唯一的答案吗?

最后欢迎大家关注小专栏公众号，**小专栏：一个技术氛围更浓厚、阅读体验和写作体验更棒、内容更深度的的专业技术写作社区**。小专栏服务号每周会推送一次精品的技术文章，绝对不容错过哟。