图解Android - Binder 和 Service

在 Zygote启动过程 一文中我们说道，Zygote一生中最重要的一件事就是生下了 System Server 这个大儿子，System Server 担负着提供系统 Service的重任，在深入了解这些Service 之前，我们首先要了解 什么是Service？它的工作原理是什么？

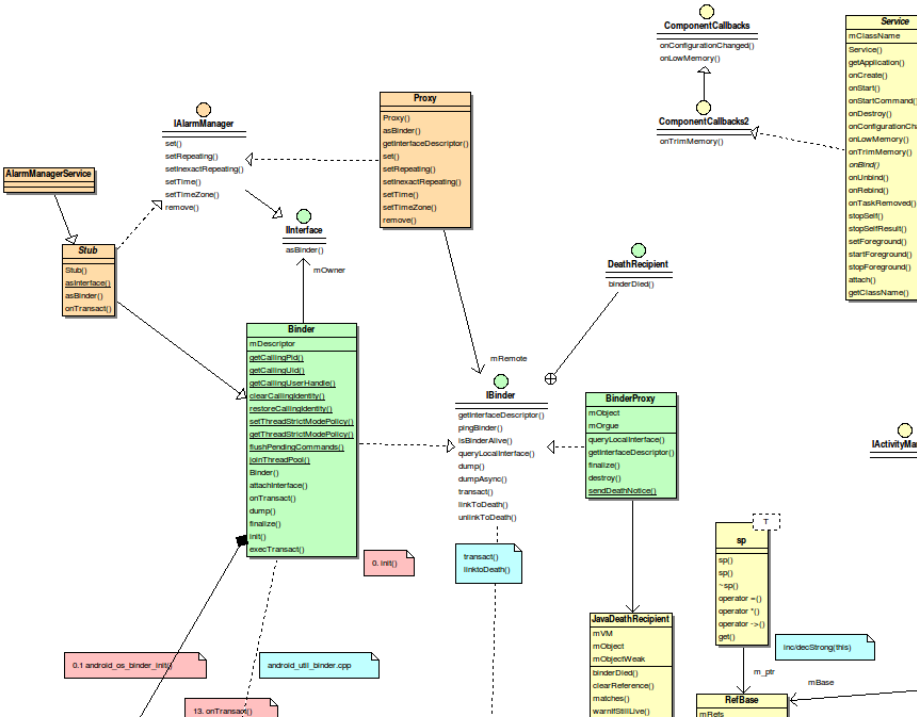
1. Service是什么？

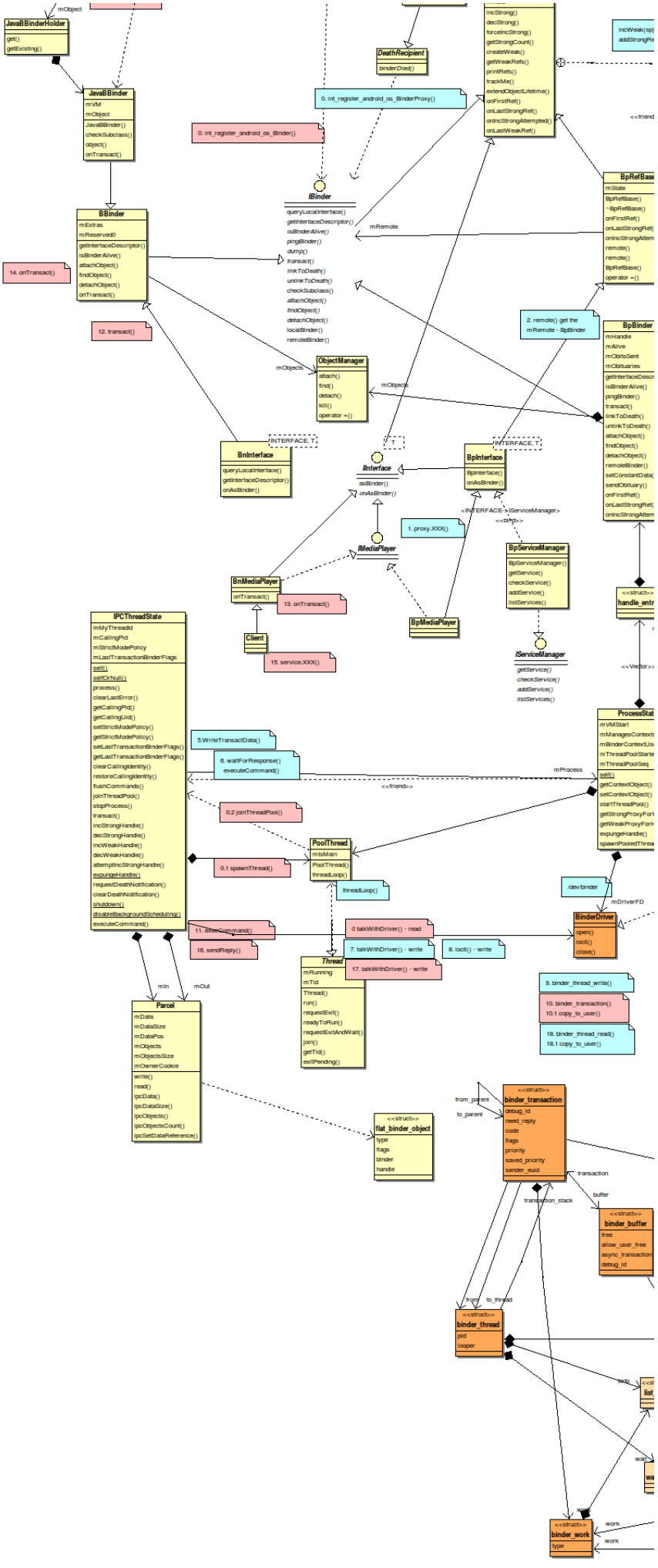
简单来说，Service就是提供服务的代码，这些代码最终体现为一个个的接口函数，所以，Service就是实现一组函数的对象，通常也称为组件。Android 的Service 有以下一些特点：

1. 请求Service服务的代码(Client) 和 Service本身(Server) 不在一个线程，很多情况下不在一个进程内。跨进程的服务称为远端(Remote)服务，跨进程的调用称为IPC。通常应用程序通过代理(Proxy)对象来访问远端的Service。
2. Service 可以运行在native 端(C/C++)，也可以运行在Java 端。同样，Proxy 可以从native 端访问Java Service, 也可以从Java端访问native service， 也就是说，service的访问与语言无关。
3. Android里大部分的跨进程的IPC都是基于Binder实现。
4. Proxy 通过 Interface 类定义的接口访问Server端代码。
5. Service可以分为匿名和具名Service. 前者没有注册到ServiceManager, 应用无法通过名字获取到访问该服务的Proxy对象。
6. Service通常在后台线程执行（相对于前台的Activity), 但Service不等同于Thread，Service可以运行在多个Thread上，一般这些Thread称为 Binder Thread.

要了解Service，我们得先从 Binder 入手。

2. Binder





先给一张Binder相关的类图一瞰Binder全貌，从上面的类图（点击看大图）跟Binder大致有这么几部

2013年11月 (2)
2013年10月 (5)
2013年9月 (2)

积分与排名

积分 - 22763
排名 - 9244

最新评论

1. Re:图解Android - Syst...
跪倒 膜拜
--Dark_Flame_Master
2. Re:图解Android - Bind...
System Server这个大儿子，笑死了，先评论再看内容。
--Bourneer
3. Re:图解Android - Andr...
nice share, thk very.
--哦哈哟嘿
4. Re:图解Android - Andr...
请问是用的什么软件画的流程图？
--juude
5. Re:图解Android - Andr...
写的已经很详细， 不知是对于opengl硬件加速部分是否有了解！
--shuangquanjun

阅读排行榜

1. 图解Android - Zygote, ...
2. 图解Android - Android ...
3. 图解Android - Android ...
4. 图解Android - Binder ...
5. 图解Android - Android ...
6. 图解Android - 如何看An...
7. 图解Android - System ...
8. 图解Android - Looper, ...
9. 推荐书单(727)

评论排行榜

1. 图解Android - Android ...
2. 图解Android - Android ...
3. 图解Android - Binder ...
4. 图解Android - Zygote, ...
5. 图解Android - Android ...
6. 图解Android - System ...
7. 图解Android - 如何看An...
8. 图解Android - Looper, ...
9. 推荐书单(1)

推荐排行榜

1. 图解Android - Binder ...
2. 图解Android - Android ...
3. 图解Android - 如何看An...
4. 图解Android - Android ...
5. 图解Android - Zygote, ...
6. 图解Android - Android ...

分：

Native 实现： IBinder， BBinder, BpBinder, IPCThread, ProcessState, IInterface, etc

Java 实现： IBinder, Binder, BinderProxy, Stub, Proxy.

Binder Driver: binder_proc, binder_thread, binder_node, etc.

我们将分别对这三部分进行详细的分析，首先从中间的Native实现开始。

通常来说，接口是分析代码的入口，Android中'I' 打头的类统统是接口类（C++里就是抽象类），自然，分析Binder就得先从IBinder下手。先看看他的定义。

```
class IBinder : public virtual RefBase
{
public:
    ...
    virtual sp<IInterface>  queryLocalInterface(const String16& descriptor); //返回一个
IInterface对象
    ...
    virtual const String16& getInterfaceDescriptor() const = 0;
    virtual bool            isBinderAlive() const = 0;
    virtual status_t        pingBinder() = 0;
    virtual status_t        dump(int fd, const Vector<String16>& args) = 0;
    virtual status_t        transact(   uint32_t code,
                                        const Parcel& data,
                                        Parcel* reply,
                                        uint32_t flags = 0) = 0;
    virtual status_t        linkToDeath(const sp<DeathRecipient>& recipient,
                                        void* cookie = NULL,
                                        uint32_t flags = 0) = 0;
    virtual status_t        unlinkToDeath(  const wp<DeathRecipient>& recipient,
                                            void* cookie = NULL,
                                            uint32_t flags = 0,
                                            wp<DeathRecipient>* outRecipient = NULL) = 0;
    ...
    virtual BBinder*        localBinder();  //返回一个BBinder对象
    virtual BpBinder*       remoteBinder(); //返回一个BpBinder对象
};
```

有接口必然有实现，从图中可以看出，BBinder和BpBinder都是IBinder的实现类，它们干啥用的，有啥区别？有兴趣同学可以去分别去读读他们的代码，分别在

- Bpinder: frameworks/native/lib/binder/BpBinder.cpp
- BBinder: frameworks/native/lib/binder/Binder.cpp

这里我们简单总结一下他们的区别：

接口	**BBinder**	**BpBinder**
queryLocalInterface()	没有实现，默认实现 IBinder 默认｛reutrn NULL};	没有实现 IBi
getInterfaceDescriptor（）	{return sEmptyDescriptor;}	(this)->t &reply); ... mDescrip
isBinderAlive()	{return true;}	{return mA
pingBinder()	{return NoError;}	{transact(P
linkToDeath()	{return INVALID_OPERATION;}	{self->requ
unlinkToDeath()	{return INVALID_OPERATION;}	{self->clear
localBinder()	{return this;}	没有实现，IB
remoteBinder()	没有实现，IBinder默认实现 {return NULL;}	{return this
transact()	{err = onTransact(code, data, reply, flags);}	IPCThreadS reply, flags)

onTransact()	switch (code) {
	case INTERFACE_TRANSACTION:
	reply->writeString16(getInterfaceDescriptor());
		没有实现

	return NO_ERROR; ...	

看出来了吧，它们的差异在于它们是通信两端的不同实现，BBinder是服务端，而BpBinder是客户端，为什么这么说？

1. pingBinder, BBinder直接返回OK，而BpBinder需要运行一个transact函数，这个函数具体做什么，我们后面会介绍。
2. linkToDeath()是用来在服务挂的时候通知客户端的，那服务端当然不需要自己监视自己咯，所以BBinder直接返回非法，而Bpbinder需要通过requestDeathNotification()要求某人完成这个事情，究竟是谁提供这个服务？答案后面揭晓。
3. 在Android中，remote一般代表某个远端对象的本地代理，想象一下航空公司和机票代理，BBinder是航空公司，当然没有remote的了，那BpBinder就是机票代理了，所以remote()自然返回自己了。
4. Transact的英文意思是交易，就是买卖嘛，那自然transact()就是买的操作，而onTransact()就是卖的操作，BBinder的transact()的实现就是onTransact(), 航空公司的买票当然不用通过机票代理了，直接找自己人就好了。

所以结论是，BBinder代表着服务端，而BpBinder则是它在客户端的代理，客户程序通过BpBinder的transact()发起请求，而服务器端的BBinder在onTranscat()里响应请求，并将结果返回。

可是交易肯定有目标的吧，回到航空公司和机票代理的例子，如果要订去某个地方的机票，我们怎么也得先查询一下都有那些航班，然后才能告诉机票代理订具体的航班号吧。这里的查询和预订可以看成服务的接口函数，而航班号就是我们传递给机票代理的参数。客户程序通过**queryLocalInterface**() 可以知道航空公司都提供哪些服务。

可是奇怪的是BBinder和BpBinder都没有实现这个接口啊，那肯定另有他人实现这个类了，这个人就是IInterface.h, 看看代码

```
template<typename INTERFACE>
inline sp<IInterface> BnInterface<INTERFACE>::queryLocalInterface(
        const String16& _descriptor)
{
    if (_descriptor == INTERFACE::descriptor) return this;
    return NULL;
}
```

BnInterface<INTERFACE> 对象将自己强制转换成 IInterface对象返回，看看BnInterface的定义：

```
template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder
{
public:
    virtual sp<IInterface>      queryLocalInterface(const String16& _descriptor);
    virtual const String16&     getInterfaceDescriptor() const;

protected:
    virtual IBinder*            onAsBinder();
};
```

是一个模板类，继承了BBinder, 还有模板 INTERFACE。我们刚才已经看过，BBinder没有实现queryLocalInterface(), 而BnInterface 返回自己，可以他并没有继承**IInterface**, 怎么可以强制转换呢，唯一的解释就是 **INTERFACE**模板必须继承和实现**IInterface**.

```
class IInterface : public virtual RefBase
{
public:
            IInterface();
            sp<IBinder>         asBinder();
            sp<const IBinder>   asBinder() const;
protected:
    virtual                     ~IInterface();
    virtual IBinder*            onAsBinder() = 0;
};
```

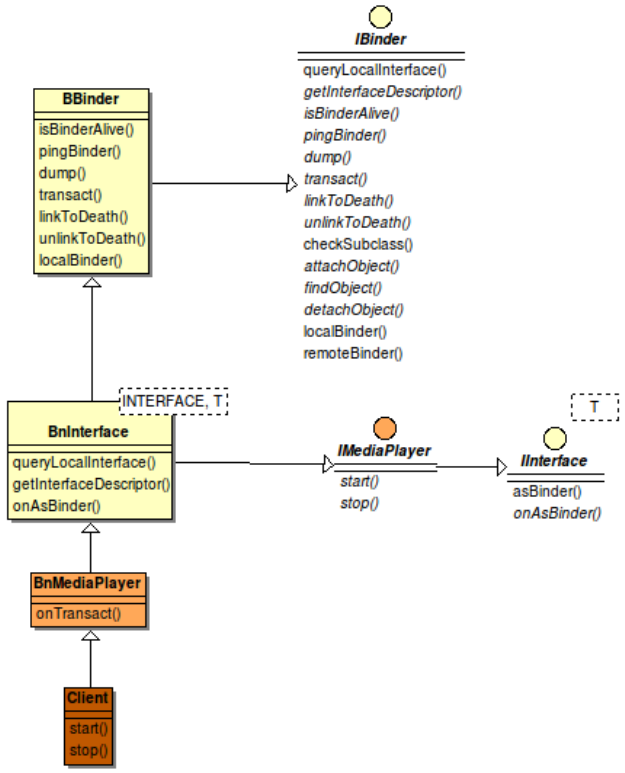
这也太简单了吧，只是定义了 从Interface 到 IBinder的转换接口 asBinder, 而刚才我们研究的queryLocalInterface() 正好反过来，说明IBinder 和 IInterface 之间是可以互转的，一个人怎么可以变成另外一个人呢？唯一的解释就是这个人有双重性格，要么他同时继承 IInterface 和 IBinder, 要么他体内有这两个对象同时存在，不卖关子了，在服务端，这个双重性格的人就是BnXXX， XXX 代表某个具体的服务，我们以图中的BnMediaPlayer为例，看看他的定义

```
class BnMediaPlayer: public BnInterface<IMediaPlayer>
{
public:
    virtual status_t    onTransact( uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);
};

class IMediaPlayer: public IInterface
{
public:
    DECLARE_META_INTERFACE(MediaPlayer);
    ...

}
```

这下本性都露出来了，IBinder 和 IInterface 的影子都露出来了，让我们用图梳理一下 (箭头代表继承关系)



归纳一下，

1. BBinder 实现了大部分的IBinder 接口，除了onTransact() 和 queryLocalInterface()，getInterfaceDescriptor();
2. BnInterface 实现了IBinder的queryLocalInterface()和getInterfaceDescriptor(), 但是其必须借助实际的接口类。
3. BnMediaPlayer只是定义了onTransact(), 没有实现。
4. onTransact()的具体实现在Client类。

为什么搞得那么复杂？Google 是希望通过这些封装尽可能减少开发者的工作量，开发一个native的service 开发者只需要做这么几件事（上图中深色部分）：

1. 定义一个接口文件， IXXXService, 继承IInterface
2. 定义BnXXX(), 继承 BnInterface<IXXXService)
3. 实现一个XXXService类，继承BnXXX(), 并具体实现onTransact() 函数。

那客户端呢？ 我们的目标是找到一个类，它必须同时拥有IBinder 和 IIterface的特性, 先看看BpBinder 吧

```
class BpBinder : public IBinder
```

跟IInterface 没有关系，那一定是别人，看看BpInterface 吧，

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public:
    BpInterface(const sp<IBinder>& remote);
protected:
    virtual IBinder*            onAsBinder();
};
```

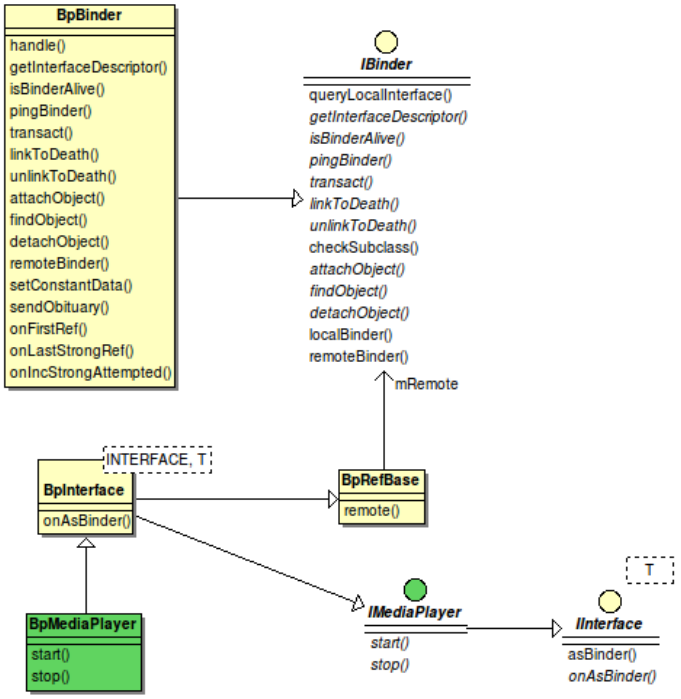
我们刚才已经知道了，INTERFACE 是 IMediaPlayer, 它继承了IInterface, IInterface 的对象找到了，但跟IBinder 没关系？只剩下BpRefBase 了，

```
class BpRefBase : public virtual RefBase
{
```

```
protected:
    ...
    inline  IBinder*        remote()                { return mRemote; }
    ...
private:
    ...
    IBinder* const          mRemote;
    RefBase::weakref_type*  mRefs;
    volatile int32_t        mState;
};
```

有了，BpRefBase 里有IBinder 成员变量，看来在客户端，没有一个类同时继承IBinder 和 IInterface， 但是有一个类继承了其一，但包含了另外一个，这种在设计模式里成为组合(Composition).

还是不太明白？还是用图解释吧，



看明白了？从BpInterface开始，通过BpRefBase 我们可以找到IBinder, 这个转换就在 asBinder() 的实现里，看看代码

```
sp<IBinder> IInterface::asBinder(){
    return this ? onAsBinder() : NULL;
}

sp<const IBinder> IInterface::asBinder() const{
    return this ? const_cast<IInterface*>(this)->onAsBinder() : NULL;
}

template<typename INTERFACE>
inline IBinder* BpInterface<INTERFACE>::onAsBinder()
{
    return remote();
}

template<typename INTERFACE>
IBinder* BnInterface<INTERFACE>::onAsBinder()
{
    return this;
}
```

这里印证我们上面两张图的正确性，onAsBinder是转换的发生的地方，服务端（BnInterface)的实现直接返回了自己，因为它继承了两者，而客户端（BpInterface)则需要通过remote()(返回mRemote 成员变量）获取，因为他自己本身不是IBinder,

那个BpRefbase的mRemote是如何被赋值的？看看以下代码

```
//frameworks/native/libs/binder/binder.cpp
BpRefBase::BpRefBase(const sp<IBinder>& o)
    : mRemote(o.get()), mRefs(NULL), mState(0)
{
   ...
}
```

```
//frameworks/native/include/binder/iinterface.h
template<typename INTERFACE>
inline BpInterface<INTERFACE>::BpInterface(const sp<IBinder>& remote)
    : BpRefBase(remote)
{
}
```

```
//frameworks/av/media/libmedia/IMediaPlayer.cpp
class BpMediaPlayer: public BpInterface<IMediaPlayer>
{
public:
    BpMediaPlayer(const sp<IBinder>& impl)
        : BpInterface<IMediaPlayer>(impl)
    {
    }
    ...
}
```

原来是从子类一级一级注入的，那唯一的问题就是在哪里完成这个注入操作，马上搜索"new BpMediaPlayer"，奇怪，竟然没有，试试搜索"IMediaPlayer"，发现了一点线索

```
   //av/media/libmedia/IMediaPlayerService.cpp

   70:     virtual sp<IMediaPlayer> create(
   71:             const sp<IMediaPlayerClient>& client, int audioSessionId) {
   72          Parcel data, reply;
   73:         ...
   77          remote()->transact(CREATE, data, &reply);
   78:         return interface_cast<IMediaPlayer>(reply.readStrongBinder()); //reply里读出
IBinder,然后转成IMediaPlayer接口对象
   79      }
```

这里通过interface_cast 直接把IBinder 转换成了 IMediaPlayer， interface_cast 到底有什么魔力？

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```

继续跟进 asInterface，结果发现里以下代码

```
#define DECLARE_META_INTERFACE(INTERFACE)                               \
    static const android::String16 descriptor;                         \
    static android::sp<I##INTERFACE> asInterface(                      \
            const android::sp<android::IBinder>& obj);                 \
    virtual const android::String16& getInterfaceDescriptor() const;   \
    I##INTERFACE();                                                    \
    virtual ~I##INTERFACE();                                           \


#define IMPLEMENT_META_INTERFACE(INTERFACE, NAME)                       \
    const android::String16 I##INTERFACE::descriptor(NAME);            \
```

```
    const android::String16&                                            \
            I##INTERFACE::getInterfaceDescriptor() const {              \
        return I##INTERFACE::descriptor;                                \
    }                                                                   \
    android::sp<I##INTERFACE> I##INTERFACE::asInterface(                \
            const android::sp<android::IBinder>& obj)                   \
    {                                                                   \
        android::sp<I##INTERFACE> intr;                                 \
        if (obj != NULL) {                                              \
            intr = static_cast<I##INTERFACE*>(                          \
                obj->queryLocalInterface(                               \
                        I##INTERFACE::descriptor).get());               \
            if (intr == NULL) {                                         \
                intr = new Bp##INTERFACE(obj);                          \
            }                                                           \
        }                                                               \
        return intr;                                                    \
    }                                                                   \
```

恍然大悟，原来在DECLARE_META_INTERFACE 这个宏里定义了asInterface, 在 IMPLEMENT_META_INTERFACE 里实现了它，这里果然有一个new BpMediaPlayer! 然后把它转换成父父类 IMediaPlayer。

一切都清楚了，用一张图来表示

客户端从远端获取一个IBinder对象，接着生成BpMediaPlayer, 将其转成 IMediaPlayer 接口对象，这是用户程序看到的对象，并通过其调用接口方法，最终调到BpBinder的transact()。

问题又来了，这个transact() 怎么传递到服务端，并最终调到 onTransact()?

回想一下，onTransact() 是IBinder的接口函数吧，而且Server的IBinder实现是BBinder, 那一定有人通过某种方式得到了BBinder对象。

这个人就是Binder Driver. 为了找到真相，必须用源头开始，那就是transact()

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
     ...
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
    ...
    return DEAD_OBJECT;
}
```

IPCThreadState的transact()函数相比IBinder 多了一个mHandle, 啥来历？

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle)
```

构造带进来的，赶紧找"new BpBinder", 结果在ProcessState.cpp 看到了

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    ...
        IBinder* b = e->binder;
        if (b == NULL || !e->refs->attemptIncWeak(this)) {
            b = new BpBinder(handle);
```

找谁call了getStrongProxyForHandle？为了快速找到调用栈，我们在BpBinder的构造函数里加了这么几句话：

```
#include <utils/CallStack.h>
...
CallStack cs;
cs.update();
cs.dump("BpBinder")
```

然后得到了下面的打印

```
09-29 07:11:14.363  1624  1700 D BpBinder: #00  pc 0001eb34  /system/lib/libbinder.so
(android::BpBinder::BpBinder(int)+260)
09-29 07:11:14.363  1624  1700 D BpBinder: #01  pc 0003b9a2  /system/lib/libbinder.so
(android::ProcessState::getStrongProxyForHandle(int)+226)
09-29 07:11:14.363  1624  1700 D BpBinder: #02  pc 00032b8c  /system/lib/libbinder.so
(android::Parcel::readStrongBinder() const+316)
//frameworks/native/libs/binder/Parcel.cpp:247
09-29 07:11:14.363  1624  1700 D BpBinder: #03  pc 000ad9d2
/system/lib/libandroid_runtime.so //frameworks/base/core/jni/android_os_Parcel.cpp:355
09-29 07:11:14.363  1624  1700 D BpBinder: #04  pc 00029c5b  /system/lib/libdvm.so
(dvmPlatformInvoke+79) //dalvik/vm/arch/x86/Call386ABI.S:128
```

#04 dvmPlatformInvork 说明这是一个Jni调用，#03 对应的代码是

```
return javaObjectForIBinder(env, parcel->readStrongBinder());
```

应该是Java传下来一个Parcel对象，然后由本地代码进行解析，从中读出IBinder对象，并最终返回。也就是说，远端有人将这个IBinder对象封在Parcel里。还是没有头绪？继续顺着调用栈往前看，

#02 对应于下面的代码

```
status_t unflatten_binder(const sp<ProcessState>& proc,
    const Parcel& in, sp<IBinder>* out)
{
    const flat_binder_object* flat = in.readObject(false);
    ...case BINDER_TYPE_HANDLE:
                *out = proc->getStrongProxyForHandle(flat->handle);
                return finish_unflatten_binder(
                    static_cast<BpBinder*>(out->get()), *flat, in);
        }
    }
    return BAD_TYPE;
}
```

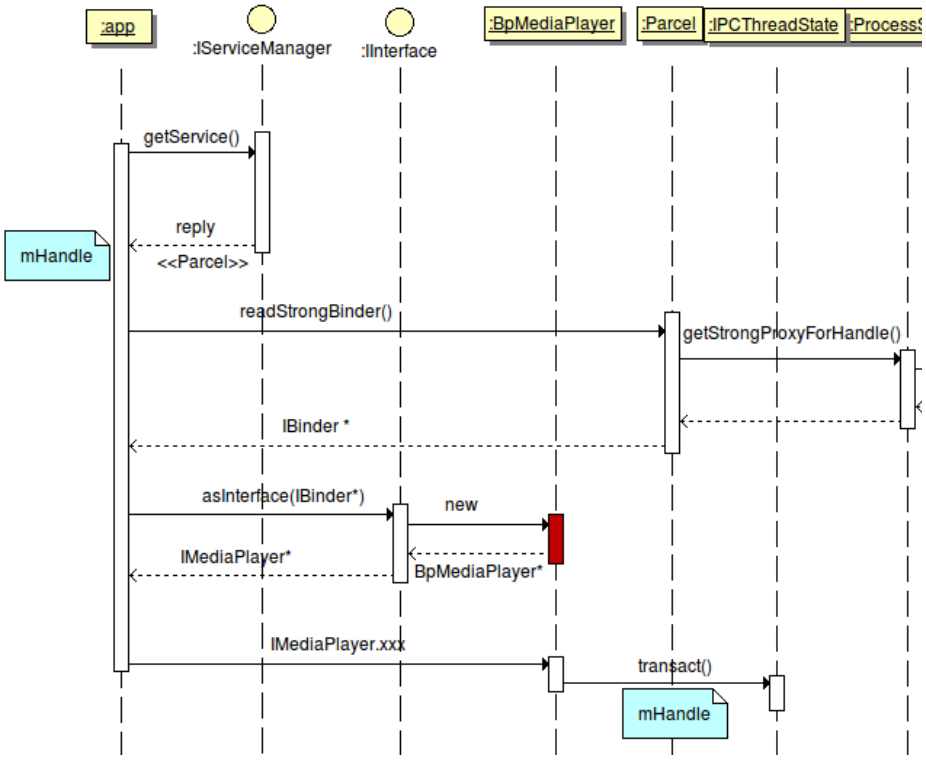
```
#bionic/libc/kernel/common/linux/binder.h
struct flat_binder_object {
 unsigned long type;
 unsigned long flags;
 union {
    void *binder;
    signed long handle;
 };
 void *cookie;
};
```

原来mHandle就是flat_binder_object里面的handle, 它只是一个数字！这个数据结构定义在Kernel里，是经过Kernel转手的。越来越乱了，赶紧整理一下思路：

1. Kernel 封装了一个数据结构(flat_binder_object)，里面带有一个数字(mHandle)。
2. 客户端获取这个数字后，生成一个BpBinder的对象。
3. 然后当客户端需要访问远端服务的时候，将这个数字附上。

回到现实生活，机票代理需要向航空公司查询或订票的话，一定要知道是哪个航空公司，莫非这个号就是航空公司的编号？

恭喜你，就是那么简单，这个号就对应了服务器端的提供的某一个服务，Android 中每个Service都有一个号码（根据创建时间递增，0号Service 是ServiceManager，让我用下面的图来描述整个过程吧，



1. 在已知服务名的情况下，App 通过getService() 从ServiceManager 获取该服务的信息，该信息封装在**Parcel**里。

2. 应用程序收到返回的这个Parcel对象（通过Kernel), 从中读取出flat_binder_object 对象，最终从对象中得到服务对应的服务号，**mHandle**.

3. 以该号码作为参数输入生成一个IBinder对象(实际是**BpBinder**）。

4. 应用获取该对象后，通过asInterface(IBinder*) 生成服务对应的Proxy对象（BpXXX),并将其强转为接口对象（**IXXX**），然后直接调用接口函数。

5. 所有的接口对象调用最终会走到**BpBinder->transact**()函数，这个函数调用IPCThreadState->transact()并以Service号作为参数之一。

6. 最终通过系统调用ioctrl() 进入内核空间，Binder驱动根据传进来的Service 号寻找该Service正处于等待状态的Binder Thread, 唤醒它并在该线程内执行相应的函数，并返回结果给APP。

强调一下：

1. 从应用程序的角度来看，他只认识IBinder 和 IMediaPlayer 这两个类，但真正的实现在BpBinder 和 BpMediaPlayer, 这正是设计模式所推崇的" Programs to interface, not implementations", 可以说 Android 是一个严格遵循设计模式思想精心设计的系统，我们将来会就这个话题进行深入的探讨。

2. 客户端应该层层的封装，最终的目的就是获取和传递这个mHandle 值，从图中，我们看到，这个mHandle至来自与IServiceManager, 他是一个管理其他服务的服务，通过服务的名字我们可以拿到这个服务对应的Handle号，类似网络域名服务系统。但是我们说了，IServiceManager也是服务啊，要访问他我们也需要一个Handle号啊，对了，就如同你必须为你的机器设置DNS 服务器地址，你才能获得 DNS 服务。在Android系统里, 默认的将ServiceManger的Handler号设为0，0就是DNS服务器的地址，这样，我们通过调用 getStrongProxyForHandle(0) 就可以拿到ServiceManager 的IBinder 对象，当然，系统提供一个 getService(char *）函数来帮助完成这个过程。

3. Android Binder 的设计目标就是让访问远端服务就像调用本地函数一样简单，但是远端的对象不在本地控制之内，我们必须保证调用过程中远端的对象不能被析构，否则本地应用程序将很有可能崩溃。同时，万一远端服务异常退出，如Crash, 本地对象必须知晓从而避免后续的错误。Android 通过 智能指针 和 DeathNotification 来支持这两个要求，我们会有专门的章节介绍智能指针，这里我们会在后面简单介绍 DeathNotifycation的实现原理。

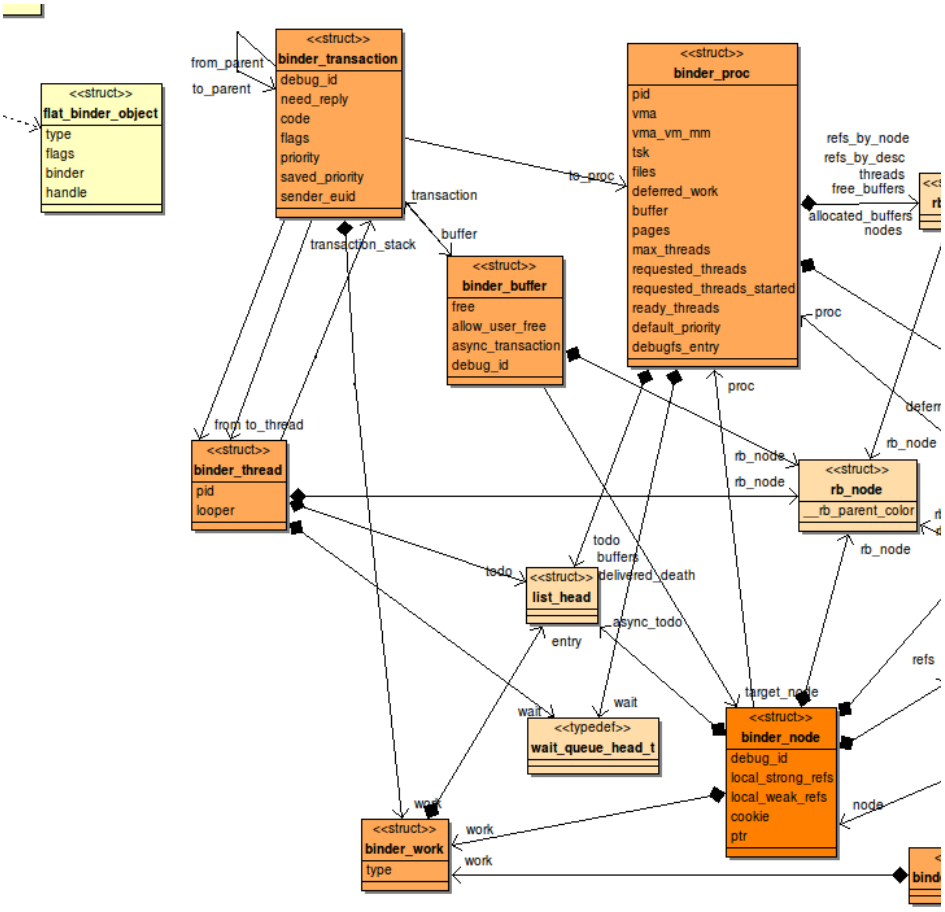
Binder的上层设计逻辑简单介绍完毕。我们接下来看看Binder的底层设计。

3. Binder Driver

我们知道，Linux的进程空间相互独立，两个进程只能通过Kernel space 进行互访，所有的IPC 机制，最底层的实现都是在Kernel space. Binder 也是如此，通过系统调用切入内核态，内核寻找到提供服务的进程，唤醒他并进入用户空间，然后在某个线程里调用onTransact(), 完成特定操作，并将结果返回

到应用程序。那Binder Driver是如何搭起连接服务端和客户端的这座桥梁呢？

先看看binder driver 内部的数据结构吧：



下面一一进行解释：

1. Binder node:

我们前面说过Service 其实是一个存在于某个进程里的对象，因此，进程PID 和 对象地址可以唯一的标识一个Service 对象，除此之外，因为这个对象可能被很多应用所使用，必须有引用计数来管理他的生命周期。这些工作都必须在内核里完成，Binder node 就是这样一个结构体来管理每个Service 对象。

```
struct binder_node {
    int debug_id;               //kernel内部标识node的id
    struct binder_work work;
    union {
        struct rb_node rb_node;
        struct hlist_node dead_node;
    };
    struct binder_proc *proc;   //Service所在进程的结构体
    struct hlist_head refs;     //双向链表头，链表里存放一系列指针，指向引用该Service的binder_ref
对象，
    int internal_strong_refs;   //内部强引用计数
    int local_weak_refs;        //弱引用计数
    int local_strong_refs;      //强引用计数
    binder_ptr __user ptr;      //Service对象地址
    binder_ptr __user cookie;
    unsigned has_strong_ref:1;
    unsigned pending_strong_ref:1;
    unsigned has_weak_ref:1;
    unsigned pending_weak_ref:1;
    unsigned has_async_transaction:1;
    unsigned accept_fds:1;
    unsigned min_priority:8;
    struct list_head async_todo;
};
```

2. binder_ref

binder_ref 描述了每个对服务对象的引用，对应与Client端。如上图所示，每个Ref通过node指向binder_node. 一个进程所有的binder_ref通过两个红黑树（RbTree)进行管理，通过binder_get_ref() 和 binder_get_ref_for_node快速查找。

```
struct binder_ref {
    /* Lookups needed: */
    /*   node + proc => ref (transaction) */
    /*   desc + proc => ref (transaction, inc/dec ref) */
    /*   node => refs + procs (proc exit) */
    int debug_id;
    struct rb_node rb_node_desc;
    struct rb_node rb_node_node;
    struct hlist_node node_entry;
    struct binder_proc *proc;           //应用进程
    struct binder_node *node;
    uint32_t desc;
    int strong;
    int weak;
    struct binder_ref_death *death;  //如果不为空，则client想获知binder的死亡
};
```

3. binder_proc

一个进程既包含的Service对象，也可能包含对其他Service对象的引用. 如果作为Service对象进程，它可能会存在多个Binder_Thread。这些信息都在binder_proc结构体进行管理。

```
struct binder_proc {
    struct hlist_node proc_node; //全局链表 binder_procs 的node之一
    struct rb_root threads; //binder_thread红黑树，存放指针，指向进程所有的binder_thread，用于Server端
    struct rb_root nodes;   //binder_node红黑树，存放指针，指向进程所有的binder 对象
    struct rb_root refs_by_desc; //binder_ref 红黑树，根据desc(service No) 查找对应的引用
    struct rb_root refs_by_node; //binder_ref 红黑树，根据binder_node 指针查找对应的引用
    int pid;
    struct vm_area_struct *vma;
    struct mm_struct *vma_vm_mm;
    struct task_struct *tsk;
    struct files_struct *files;
    struct hlist_node deferred_work_node;
    int deferred_work;
    void *buffer;
    ptrdiff_t user_buffer_offset;

    struct list_head buffers;
    struct rb_root free_buffers;
    struct rb_root allocated_buffers;
    size_t free_async_space;

    struct page **pages;
    size_t buffer_size;
    uint32_t buffer_free;
    struct list_head todo; //task_list, binder_work链表，存放指针最终指向某个binder_transaction对象
    wait_queue_head_t wait;
    struct binder_stats stats;
    struct list_head delivered_death;
    int max_threads;
    int requested_threads;
    int requested_threads_started;
    int ready_threads;
    long default_priority;
    struct dentry *debugfs_entry;
};
```

为了实现快速的查找，binder_proc内部维护了若干个数据结构，如图中黄色高亮所示，

4. binder_transaction

每个transact() 调用在内核里都会生产一个binder_transaction 对象，这个对象会最终送到Service进程或线程的todo队列里，然后唤醒他们来最终完成onTransact()调用。

```
struct binder_transaction {
    int debug_id;              //一个全局唯一的ID
    struct binder_work work; // 用于存放在todo链表里
    struct binder_thread *from; //transaction 发起的线程。如果BC_TRANSACTION, 则为客户端线程，如果是BC_REPLY, 则为服务端线程。
    struct binder_transaction *from_parent; //上一个binder_transaction. 用于client端
    struct binder_proc *to_proc; //目标进程
    struct binder_thread *to_thread; //目标线程
    struct binder_transaction *to_parent; //上一个binder_transaction, 用于server端
    unsigned need_reply:1;
    /* unsigned is_dead:1; */    /* not used at the moment */

    struct binder_buffer *buffer;
    unsigned int    code;
    unsigned int    flags;
    long    priority;
    long    saved_priority;
    kuid_t    sender_euid;
};
```

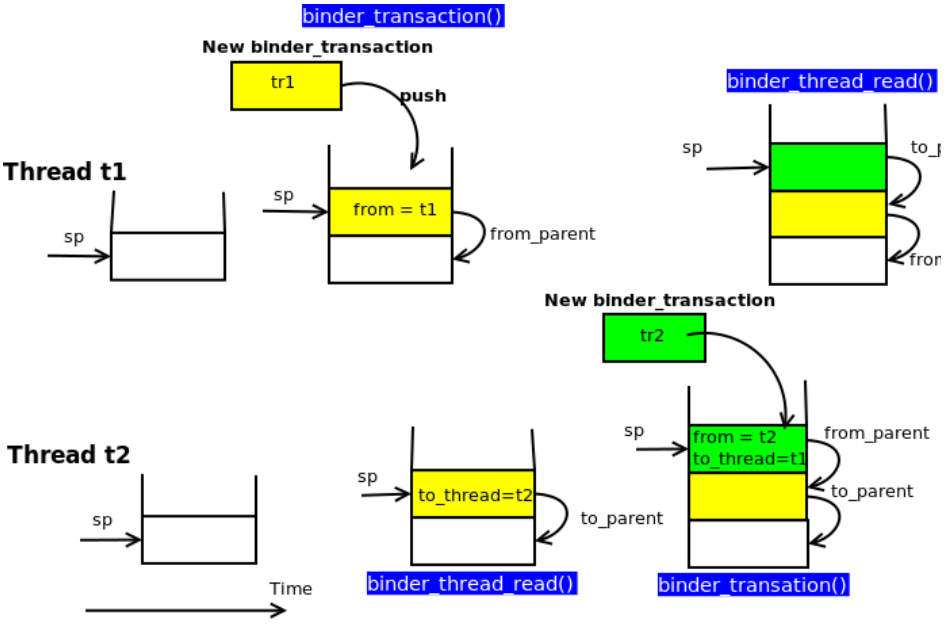
5. binder_thread

binder_proc里的threads 红黑树存放着指向binder_thread对象的指针。这里的binder_thread 不仅仅包括service的binder thread, 也包括访问其他service的调用thread. 也就是说所有与binder相关的线程都会在binder_proc的threads红黑树里留下记录。binder_thread里最重要的两个成员变量是transaction_stack 和 wait.

```
struct binder_thread {
    struct binder_proc *proc;
    struct rb_node rb_node; //红黑树节点
    int pid;
    int looper;  //
    struct binder_transaction *transaction_stack; //transaction栈
    struct list_head todo;
    uint32_t return_error;
    uint32_t return_error2;
    wait_queue_head_t wait; //等待队列，用于阻塞等待
    struct binder_stats stats;
};
```

在binder_proc里面我们也能看到一个wait 队列，是不是意味着线程既可以在proc->wait上等待，也可以在thread->wait上等待？binder driver 对此有明确的用法，所有的binder threads (server 端）都等待在proc->wait上。因为对于服务端来说，用哪个thread来响应远程调用请求都是一样的。然而所有的ref thread(client端）的返回等待都发生在调用thread的wait 队列，因为，当某个binder thread 完成服务请求后，他必须唤醒特定的等待返回的线程。但是有一个例外，在双向调用的情况下，某个Server端的thread将会挂在thread->wait上等待，而不是proc->wait. 举个例子，假设两个进程P1 和P2，各自运行了一个Service， S1，S2， P1 在 thread T1 里调用S2提供的服务，然后在T1->wait里等待返回。S2的服务在P2的binder thread(T2)里执行，执行过程中，S2又调到S1里的某个接口，按理S1 将在P1的binder thread T3里执行， 如果P1接下来又调到了P2，那又会产生新的进程 T4，如果这个反复调用栈很深，需要耗费大量的线程，显然这是非常不高效的设计。所以，binder driver 里做了特殊的处理。当T2 调用 S1的接口函数时，binder driver 会遍历T2的transaction_stack, 如果发现这是一个双向调用（binder_transaction->from->proc 等于P1), 便会唤醒正在等待reply的T1，T1 完成这个请求后，继续等待S2的回复。这样，只需要最多两个Thread就可以完成多层的双向调用。

binder_thread里的transaction_stack 是用链表实现的堆栈， 调用线程和服务线程的transaction有着不同的堆栈。下图是上面这个例子的堆栈情形：

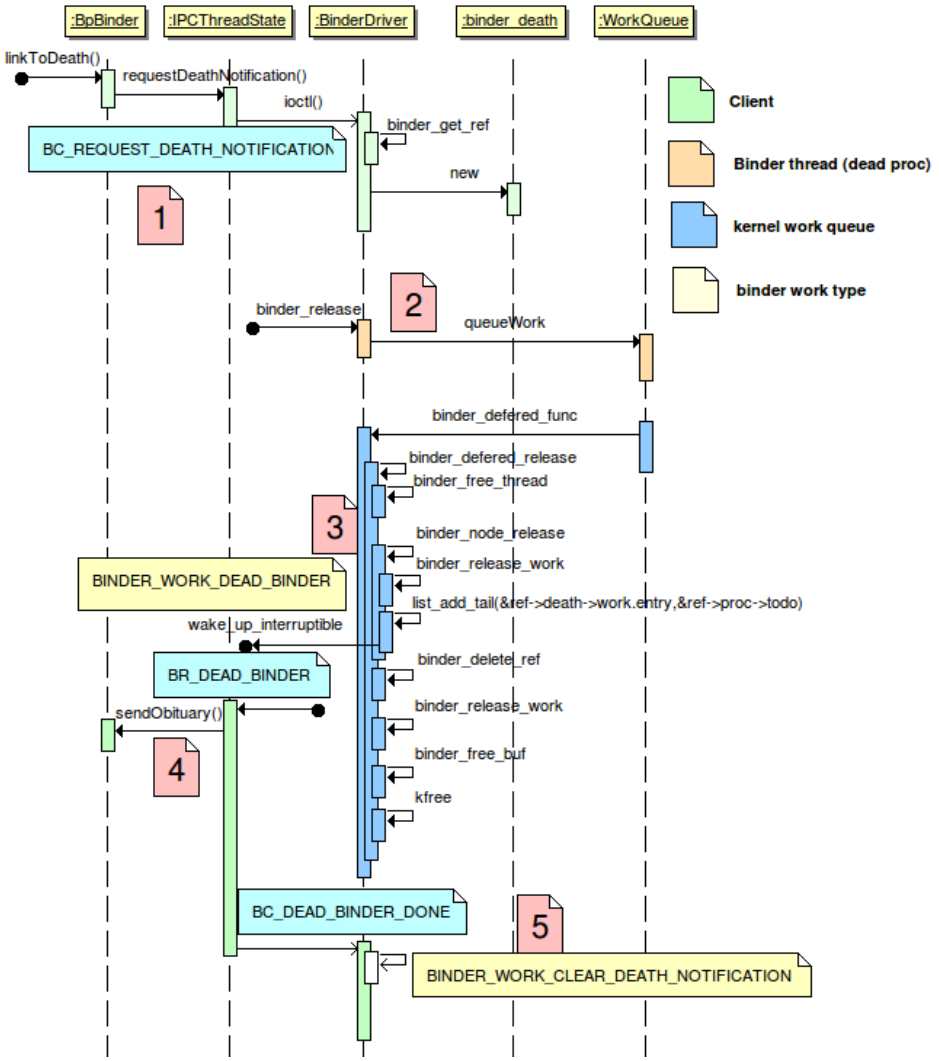


6. binder_ref_death

binder_ref 记录了从client进程到server进程某个service的引用，binder_ref_death 是binder_ref的一个成员变量，它的不为空说明了client进程想得到这个service的死亡通知（严格意义上讲，是service所在进程的死亡通知，因为一个进程一个/dev/binder的fd，只有进程死亡了，driver才会知晓，通过file_operations->release 接口）。

```
struct binder_ref_death {
    struct binder_work work;
    binder_ptr __user cookie;
};
```

我们可以下面一张时序图来了解binder death notifycation 的全过程。



7. binder_work

从应用程序角度来看，所有的binder调用都是同步的。但在binder driver 内部，两个进程间的交互都是异步的，一个进程产生的请求会变成一个binder_work, 并送入目标进程或线程的todo 队列里，然后唤醒目标进程和线程来完成这个请求，并阻塞等待结果。binder_work的定义如下：

```
struct binder_work {
    struct list_head entry;
    enum {
        BINDER_WORK_TRANSACTION = 1,
        BINDER_WORK_TRANSACTION_COMPLETE,
        BINDER_WORK_NODE,
        BINDER_WORK_DEAD_BINDER,
        BINDER_WORK_DEAD_BINDER_AND_CLEAR,
        BINDER_WORK_CLEAR_DEATH_NOTIFICATION,
    } type;
};
```

很简单，其实只定义了一个链表的节点和work的类型。

8. binder_buffer

进程间通信除了命令，还有参数和返回值的交换，要将数据从一个进程的地址空间，传到另外一个进程的地址空间，通常需要两次拷贝，进程A -> 内核 -> 进程B。binder_buffer 就是内核里存放交换数据的空间（这些数据是以Parcel的形式存在）。为了提高效率，Android 的 binder 只需要一次拷贝，因为 binder 进程通过mmap将内核空间地址映射到用户空间，从而可以直接访问binder_buffer的内容而无需一次额外拷贝。binder_buffer由内核在每次发起的binder调用创建，并赋给binder_transaction->buffer. binder driver 根据binder_transaction 生产 transaction_data（包含buffer的指针而非内容），并将其复制到用户空间。

9. flat_binder_obj

前面我们说过，<proc, handle> 可以标识一个BpBinder 对象，而<proc, ptr> 可以标识一个BBinder对象。Binder Driver 会收到来自与BpBinder 和 BBinder的系统调用，它是如何判别它们的身份呢？答案就在flat_binder_obj里，先看看它的定义，

```
struct flat_binder_object {
 unsigned long type;  //见下面定义
 unsigned long flags;
 union {
 void *binder;            //BBinder，通过它driver可以找到对应的node
 signed long handle; //BpBinder，根据它driver可以找到对应的ref
 };
 void *cookie;
};

enum {
 BINDER_TYPE_BINDER = B_PACK_CHARS('s', 'b', '*', B_TYPE_LARGE),
 BINDER_TYPE_WEAK_BINDER = B_PACK_CHARS('w', 'b', '*', B_TYPE_LARGE),
 BINDER_TYPE_HANDLE = B_PACK_CHARS('s', 'h', '*', B_TYPE_LARGE),
 BINDER_TYPE_WEAK_HANDLE = B_PACK_CHARS('w', 'h', '*', B_TYPE_LARGE),
 BINDER_TYPE_FD = B_PACK_CHARS('f', 'd', '*', B_TYPE_LARGE),
};
```

union表明了在Server端和Client端它有着不同的解读。type则表明了它的身份。binder driver 根据它可以找到BpBinder 和 BBinder 在内核中相对应的对象 (ref 或 node). flat_binder_obj 封装在parcel里，详见Parcel.cpp.

至此，binder driver里面重要的数据结构都介绍完了，大家对binder driver的工作原理也有了大致的了解，这里再稍作总结：

1. 当一个service向binder driver 注册时（通过flat_binder_object), driver 会创建一个**binder_node**, 并挂载到该service所在进程的nodes红黑树。

2. 这个service的binder线程在**proc->wait** 队列上进入睡眠等待。等待一个**binder_work**的到来。

3. 客户端的BpBinder 创建的时候，它在driver内部也产生了一个**binder_ref**对象，并指向某个binder_node, 在driver内部，将client和server关联起来。如果它需要或者Service的死亡状态，则会生成相应的**binfer_ref_death**.

4. 客户端通过transact() (对应内核命令BC_TRANSACTION)请求远端服务，driver通过**ref->node**的映射，找到service所在进程，生产一个**binder_buffer**, **binder_transaction** 和 **binder_work** 并插入proc->todo队列，接着唤醒某个睡在proc->wait队列上的**Binder_thread**. 与此同时，该客户端线程在其线程的wait队列上进入睡眠，等待返回值。

5. 这个binder thread 从**proc->todo** 队列中读出一个binder_transaction, 封装成**transaction_data** (命令为 BR_TRANSACTION) 并送到用户空间。Binder用户线程唤醒并最终执行对应的on_transact() 函数。

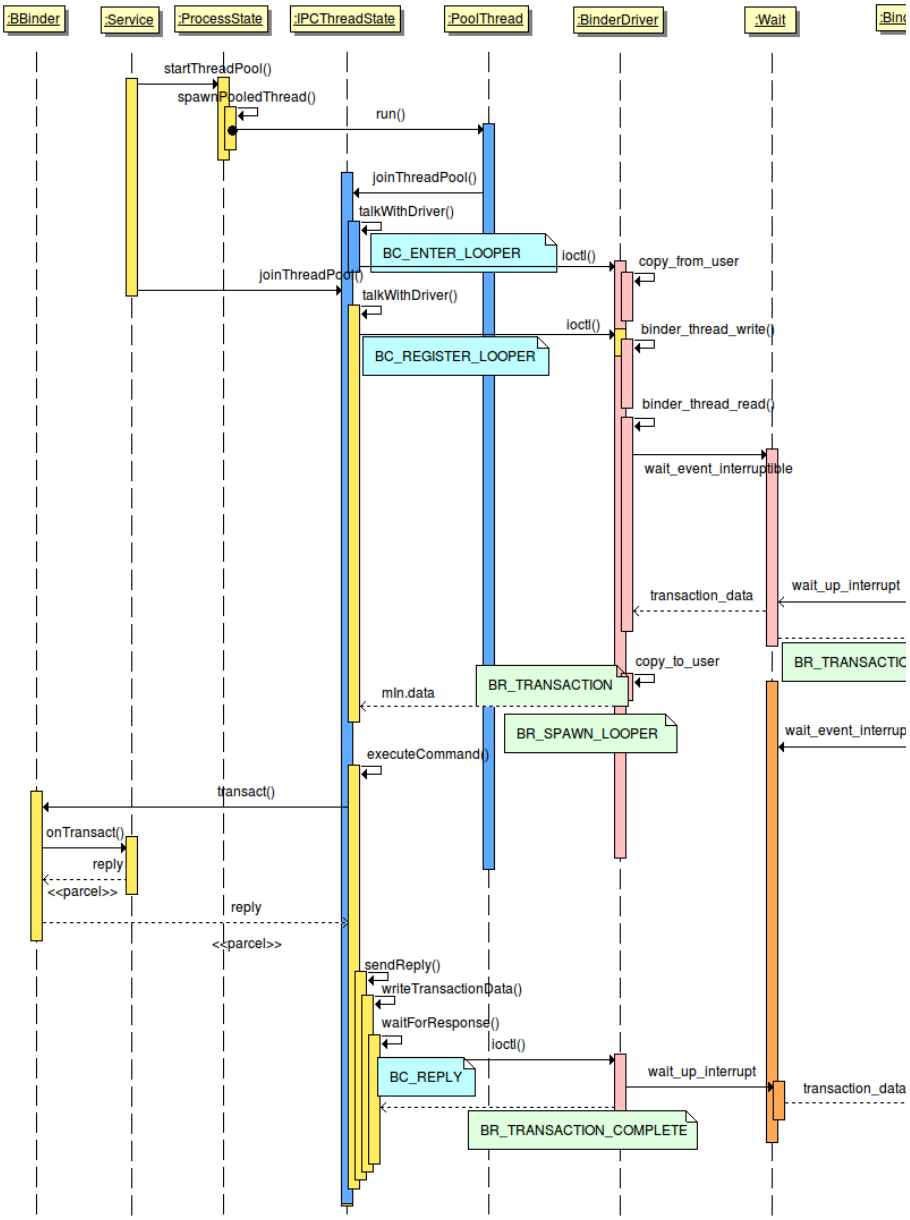
6. Binder用户线程通过transact() 向内核发送 BC_REPLY命令，driver收到后从其**thread->transaction_stack**中找到对应的binder_transaction, 从而知道是哪个客户端线程正在等待这个返回。

7. Driver 生产新的binder_transaction (命令 BR_REPLY), binder_buffer, binder_work, 将其插入应用线程的todo对立，并将该线程唤醒。

8. 客户端的用户线程收到回复数据，该Transaction完成。

9. 当service所在进程发生异常退出，driver 的 **release**函数被调到，在某位内核work_queue 线程里完成该service在内核态的清理工作（thread，buffer，node，work...), 并找到所有引用它的binder_ref, 如果某个binder_ref 有不为空的binder_ref_death, 生成新的binder_work, 送人其线程的todo 对立，唤醒它来执行剩余工作，用户端的DeathRecipient 会最终被调用来完成client端的清理工作。

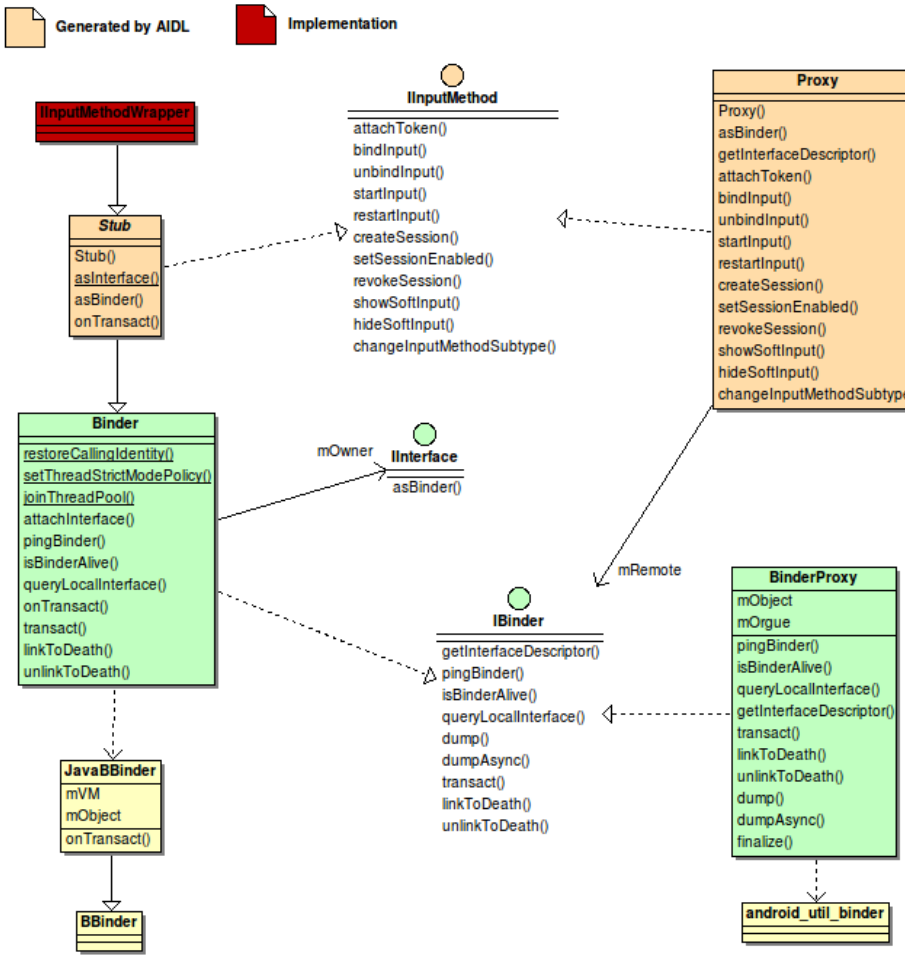
下面这张时序图描述了上述一个transaction完成的过程。不同的颜色代表不同的线程。注意的是，虽然Kernel和User space 线程的颜色是不一样的，但所有的系统调用都发生在用户进程的上下文里（所谓上下文，就是Kernel能通过某种方式找到关联的进程（通过Kernel的current 宏），并完成进程相关的操作，比如说唤醒某个睡眠的线程，或跟用户空间交换数据，copy_from, copy_to, 与之相对应的是中断上下文，其完全异步触发，因此无法做任何与进程相关的操作，比如说睡眠，锁等）。



4. Java Binder

Binder 的学习已经接近尾声了，我们已经研究了Binder Driver, C/C++的实现，就差最后一个部分了，Binder在Java端的实现了。Java端的实现与Native端类似，我们用下面的表格和类图概括他们的关系

Native	Java	Note
IBinder	IBinder	
IInterface	IInterface	
IXXX	IXXX	aidl文件定义
BBinder	Binder	通过JavaBBinder类作为桥梁
BpBinder	BinderProxy	通过JNI访问Native的实现
BnInterface	N/A	
BpInterface	N/A	
BnXXX	Stub	aidl工具自动生成
BpXXX	Proxy	aidl工具自动生成



可见，Java较Native端实现简单很多，通过Aidl工具来实现类似功能。所以，要实现一个Java端的service，只需要做以下几件事情：

1. 写一个.aidl文件，里面用AIDL语言定义一个接口类IXXX。

2.在Android.mk里加入该文件，这样编译系统会自动生成一个IXXX.java, 放在out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/src/core 下面。

3. 在服务端，写一个类，扩展IXXX.Stub，具体实现IXXX的接口函数。

分类: 图解Android 系列

标签: Android , Framework , Binder , Service

好文要顶 关注我 收藏该文

漫天尘沙
关注 - 0
粉丝 - 66
+加关注

4 0

(请您对文章做出评价)

« 上一篇: 图解Android - Android GUI 系统 (1) - 概论

» 下一篇: 图解Android - Zygote, System Server 启动分析

posted @ 2013-10-25 00:08 漫天尘沙 阅读(7572) 评论(8) 编辑 收藏

评论列表

#1楼 2013-10-29 09:36 zhiqiang.tan

看不太懂啊

支持(0) 反对(0)

#2楼 [楼主] 2013-10-29 10:11 漫天尘沙

@zhiqiang.tan
是图还是文字，还是其他方面？具体一点？看看如何改进。

支持(0) 反对(0)

#3楼 2014-01-16 14:28 mahahadm

看见类图就注册了个账号，回复一把，结合老罗的书看，更加清晰，牛人

支持(0) 反对(0)

#4楼 2014-02-07 14:28 suchangyu

感谢楼主，看后很受教。
但能否再多讲一下Binder中回调的实现。
例如：MediaPlayer作为MediaPlayerService的Client，在调用其IPC接口create时，会将this作为IMediaPlayerClient的引用传递给Server。当MediaPlayerService有通知时，会根据此引用找到对应的MediaPlayer进程，调用其提供的notify接口，此时应该MediaPlayerService是Client，MediaPlayer是Server。那么，在MediaPlayer进程中是否会像MediaPlayerService一样，调用IPCThreadState::talkWithDriver()，去监听/dev/binder的消息。

支持(0) 反对(0)

#5楼 2014-02-24 23:53 cason_ai

好清晰，拜服楼主哇。

只是binder_buffer那段有个东西没点到，就是发起者的data是直接拷贝到接受者的mmap区域的，由于那块内存区域是驱动管理，所以接受者不能通过mmap返回的addr直接访问。

不妨授人以渔吧：您的研究方法是什么？本科能达到您的层次吗？

支持(0) 反对(0)

#6楼 2014-08-11 17:49 ilotuo

博主把类图的源文件也共享一下吧~
279952003@qq.com 或者发我一份.谢谢啦

支持(0) 反对(0)

#7楼 2014-12-10 10:16 Divan_Chen

讲的挺细致的

支持(0) 反对(0)

#8楼 2015-11-19 19:20 Bourneer

System Server这个大儿子，笑死了，先评论再看内容。

支持(0) 反对(0)

最新**IT**新闻:
· 彗星无法解释恒星的神秘变暗
· 盘点雅虎美女CEO的精彩起伏人生
· 谈谈Model S的设计失误与Model X的车门及Autopilot
· 快速将C#类型转成TypeScript介面定义
· 你的代码活着吗?
» 更多新闻...

最新知识库文章:
· Docker简介
· Docker简明教程
· Git协作流程
· 企业计算的终结
· 软件开发的核心
» 更多知识库文章...