開源中國社區
開源項目發現、使用和交流平台

- 項目
- 討論
- 代碼
- 資訊
- 翻譯
- 博客
- Android
- 招聘

當前訪客身份：遊客[登錄 | 加入開源中國] 你有0新留言
在 27048 款开源软件中搜
軟件
軟件
搜索


悠然紅茶 關注此人
關注(0) 粉絲(18) 積分(6)
求真求是
. 發送留言 . 請教問題

博客分類

- Android Frameworks 4.x (5)
- 日常記錄(0)
- 轉貼的文章(0)

閱讀排行

1. 1. 紅茶一杯話Binder（傳輸機制篇 上）
2. 2. 紅茶一杯話Binder（傳輸機制篇 中）
3. 3. 紅茶一杯話Binder（ServiceManager篇）
4. 4. 紅茶一杯話Binder（初始篇）
5. 5. AlarmManager研究

最新評論

- @Jessie0227：寫的太好了!!請問何時會有下一篇呢(期待中) 查看»
- @公子無憂：請教個問題,BC和BR的命令是什麼關係,分別什麼時候... 查看»
- @xkk609：分析比較深入。 查看»
- @RenKaidi：不明覺厲！ 查看»
- @enull：Mark一下，自學中，感謝。 查看»
- @xway：準備空下來的時候學習下Android開發，這樣認真的... 查看»
- @徐慶-neo：贊一個，非常好的文章 查看»
- @翠屏阿姨：我是沙發，曾經看過沒看懂，今天趁著這篇文章再看... 查看»
- @simonws：fucking source code 查看»
- @悠然紅茶：引用來自“simonws”的評論你是怎麼研究的？無他... 查看»

訪客統計

- **今日訪問： 3**
- **昨日訪問： 7**
- **本周訪問： 19**
- **本月訪問： 10**
- **所有訪問： 2285**

空間 » 博客 » Android Frameworks 4.x »博客正文

紅茶一杯話Binder（初始篇）

紅茶一杯話Binder

（初始篇）

侯亮

1 什麼是Binder？

簡單地說，Binder是Android平台上的一種跨進程交互技術。該技術最早並不是由Google公司提出的，它的前身是Be Inc公司開發的OpenBinder，而且在Palm中也有應用。後來OpenBinder的作者Dianne Hackborn加入了Google公司，並負責Android平台的開發工作，所以把這項技術也帶進了Android。

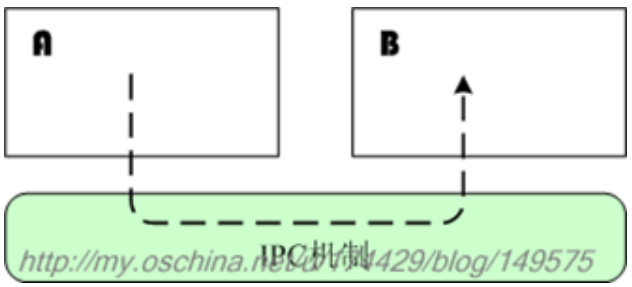
我們知道，在Android的應用層次上，基本上已經沒有過去的進程概念了。然而在實現層次，它畢竟還是要建構在一個個進程之上的。實際上，在Android內部，那些支撐應用的組件往往會身處於不同的進程，那麼應用的底層必然會牽涉大量的跨進程通信。為了保證通信的高效性，Android提供了Binder機制。

Binder機制具有兩層含義：
1）是一種跨進程通信手段（IPC，Inter-Process Communication）。
2）是一種遠程過程調用手段（RPC，Remote Procedure Call）。

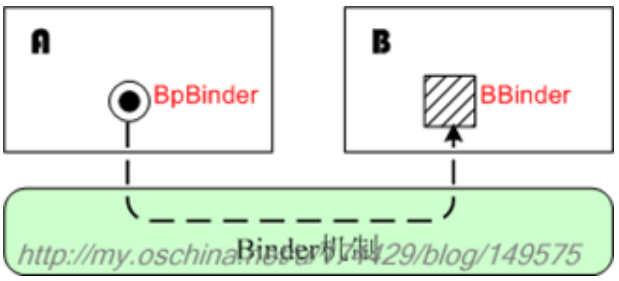
從實現的角度來說，Binder核心被實現成一個Linux驅動程序，並運行於內核態。這樣它才能具有強大的跨進程訪問能力。

1.1 簡述Binder的跨進程機制

為了理解Binder，我們可以先畫一張最簡單的跨進程通信示意圖：



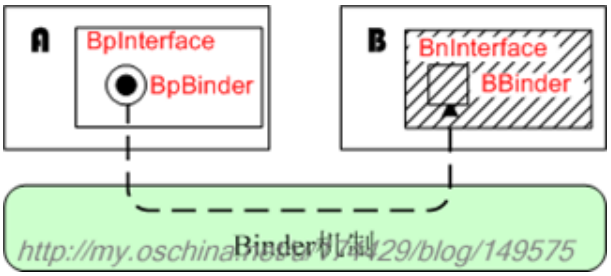
這個很容易理解，不需贅言。到了Android平台上，IPC機制就變成了Binder機制，情況類似，只不過為了便於說明問題，我們需要稍微調整一下示意圖：



圖中A側的圓形塊◉，表示“Binder代理方”，主要用於向遠方發送語義，而B側的方形塊◪則表示“Binder響應方”，主要用於響應語義。需要說明的是，這種圖形表示方法是我自己杜撰的，並沒有正規的出處。我個人覺得這種圖形非常簡便，所以在分析Android架構時，會經常使用這種表示法。

在後文中，我們可以看到，Binder代理方大概對應於C++層次的BpBinder對象，而Binder響應方則對應於BBinder對象。這兩個對像在後文會詳細闡述，此處不必太細究。

然而，上圖的Binder代理方主要只負責了“傳遞信息”的工作，並沒有起到“遠程過程調用”的作用，如果要支持遠程過程調用，我們還必須提供“接口代理方”和“接口實現體”。這樣，我們的示意圖就需要再調整一下，如下：



從圖中可以看到，A進程並不直接和BpBinder（Binder代理）打交道，而是通過調用BpInterface（接口代理）的成員函數來完成遠程調用的。此時，BpBinder已經被聚合進BpInterface了，它在BpInterface內部完成了一切跨進程的機制。另一方面，與BpInterface相對的響應端實體就是BnInterface（接口實現）了。需要注意的是，BnInterface是繼承於BBinder的，它並沒有採用聚合的方式來包含一個BBinder對象，所以上圖中B側的BnInterface塊和BBinder塊的背景圖案是相同的。

這樣看來，對於遠程調用的客戶端而言，主要搞的就是兩個東西，一個是“Binder代理”，一個是“接口代理”。而服務端主要搞的則是“接口實現體”。因為binder是一種跨進程通信機制，所以還需要一個專門的管理器來為通信兩端牽線搭橋，這個管理器就是Service Manager Service。不過目前我們可以先放下Service Manager Service，以後再詳細研究。

2 Binder相關接口和類

Android的整個跨進程通信機制都是基於Binder的，這種機制不但會在底層使用，也會在上層使用，所以必須提供Java和C++兩個層次的支持。

2.1 Java層次的binder元素

Java層次裡並沒有我們前文圖中所表示的BpBinder、BpInterface、BBinder等較低層次的概念，取而代之的是IBinder接口、IInterface等接口。Android要求所有的Binder實體都必須實現IBinder接口，該接口的定義截選如下：

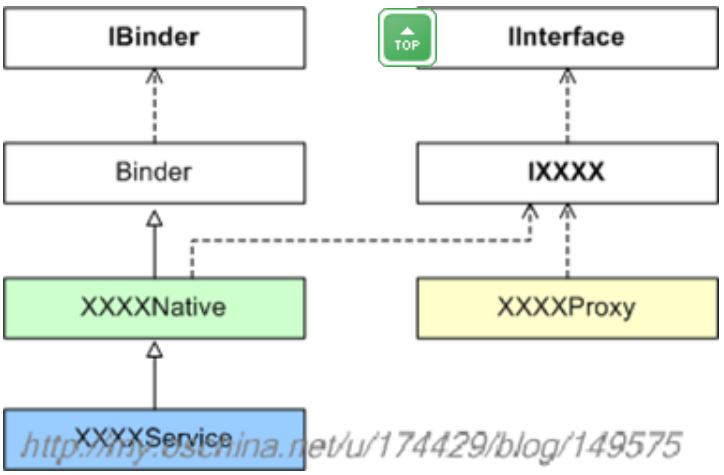
【frameworks/base/core/java/android/os/IBinder.java】

```
public  interface IBinder
{
    . . . . . .
    public String getInterfaceDescriptor() throws RemoteException;
     public boolean pingBinder();
     public boolean isBinderAlive();
     public IInterface queryLocalInterface(String descriptor);
     public  void dump(FileDescriptor fd, String[] args) throws RemoteException;
     public  void dumpAsync(FileDescriptor fd, String[] args) throws RemoteException;
     public boolean transact( int code, Parcel data, Parcel reply, int flags)
throws RemoteException;

    public  interface DeathRecipient
    {
        public  void binderDied();
    }
    public  void linkToDeath(DeathRecipient recipient, int flags)throws RemoteException;
     public boolean unlinkToDeath(DeathRecipient recipient, int flags);
}
```

另外，不管是代理方還是實體方，都必須實現IInterface接口：

```
public  interface IInterface
{
    public IBinder asBinder();
}
```

Java層次中，與Binder相關的接口或類的繼承關係如下：



在實際使用中，我們並不需要編寫上圖的XXXXNative、XXXXProxy，它們會由ADT根據我們編寫的aidl腳本自動生成。用戶只需繼承XXXXNative編寫一個具體的XXXXService即可，這個XXXXService就是遠程通信的服務實體類，而XXXXProxy則是其對應的代理類。

關於Java層次的binder組件，我們就先說這麼多，主要是先介紹一個大概。就研究跨進程通信而言，其實質內容基本上都在C++層次，Java層次只是一個殼而已。以後我會寫專文來打通Java層次和C++層次，看看它們是如何通過JNI技術關聯起來的。現在我們還是把注意力集中在C++層次吧。

2.2 C++層次的binder元素

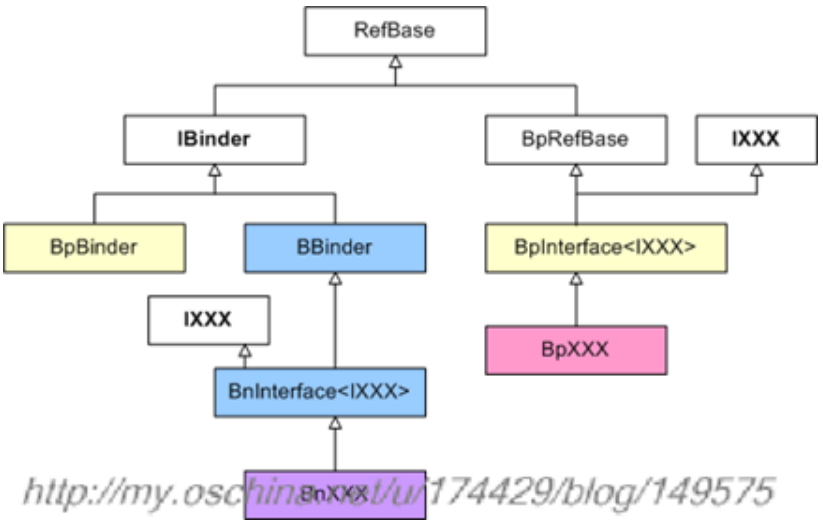
在C++層次，就能看到我們前文所說的BpBinder類和BBinder類了。這兩個類都繼承於IBinder，IBinder的定義截選如下：

【frameworks/native/include/binder/IBinder.h】

```
class IBinder : public  virtual RefBase
{
public :
    . . . . . .
    IBinder();
    virtual sp<IInterface> queryLocalInterface( const String16& descriptor);
     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;

    class DeathRecipient : public  virtual RefBase
    {
    public :
         virtual  void binderDied( const wp<IBinder>& who) = 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  bool             checkSubclass( const  void * subclassID) const ;

    typedef void (*object_cleanup_func)( const  void * id, void * obj, void * cleanupCookie);
     virtual  void             attachObject( const  void * objectID, void * object ,
                                        void * cleanupCookie, object_cleanup_func func) = 0;
     virtual  void * findObject( const  void * objectID) const = 0;
     virtual  void             detachObject( const  void * objectID) = 0;

    virtual BBinder* localBinder();
     virtual BpBinder* remoteBinder();

protected :
     virtual           ~IBinder();
 private :
};
```

C++層次的繼承關係圖如下：



其中有以下幾個很關鍵的類：

- BpBinder
- BpInterface
- BBinder
- BnInterface

它們扮演著很重要的角色。

2.2.1 BpBinder

BpBinder的定義截選如下：

```
class BpBinder : public IBinder
{
public :
BpBinder(int32_t handle);
    inline int32_t handle() const { return mHandle; }

    virtual  const String16& getInterfaceDescriptor() const ;
     virtual  bool         isBinderAlive() const ;
     virtual status_t pingBinder();
     virtual status_t dump( int fd, const Vector<String16>& args);

    virtual status_t transact(uint32_t code, const Parcel& data,
                              Parcel* reply, uint32_t flags = 0);
```

```
    virtual status_t linkToDeath( const sp<DeathRecipient>& recipient,
                                  void * cookie = NULL, uint32_t flags = 0);
     virtual status_t unlinkToDeath( const wp<DeathRecipient>& recipient,
                                      void * cookie = NULL, uint32_t flags = 0,
                                     wp<DeathRecipient>* outRecipient = NULL);
. . . . . .
. . . . . .
```

作為代理端的核心，BpBinder最重要的職責就是實現跨進程傳輸的傳輸機制，至於具體傳輸的是什麼語義，它並不關心。我們觀察它的transact()函數的參數，可以看到所有的語義都被打包成Parcel了。其他的成員函數，我們先不深究，待我們儲備了足夠的基礎知識後，再回過頭研究它們不遲。

2.2.2 BpInterface

另一個重要的類是BpInterface，它的定義如下：

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

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

其基類BpRefBase的定義如下：

```
class BpRefBase : public  virtual RefBase
{
protected :
                          BpRefBase( const sp<IBinder>& o);
     virtual                ~BpRefBase();
     virtual  void           onFirstRef();
     virtual  void           onLastStrongRef( const  void * id);
     virtual  bool           onIncStrongAttempted(uint32_t flags, const  void * id);
    inline IBinder* remote() { return mRemote; }
    inline IBinder* remote() const          { return mRemote; }

private :
BpRefBase( const BpRefBase& o);
    BpRefBase&              operator =( const BpRefBase& o);
    IBinder* const           mRemote;
    RefBase::weakref_type* mRefs;
    volatile int32_t mState;
};
```

BpInterface使用了模板技術，而且因為它繼承了BpRefBase，所以先天上就聚合了一個mRemote成員，這個成員記錄的就是前面所說的BpBinder對象啦。以後，我們還需要繼承BpInterface<>實現我們自己的代理類。

在實際的代碼中，我們完全可以創建多個聚合同一BpBinder對象的代理對象，這些代理對象就本質而言，對應著同一個遠端binder實體。在Android框架中，常常把指向同一binder實體的多個代理稱為token，這樣即便這些代理分別處於不同的進程中，它們也具有了某種內在聯繫。這個知識點需要大家關注。

2.2.3 BBinder

Binder遠程通信的目標端實體必須繼承於BBinder類，該類和BpBinder相對，主要關心的只是傳輸方面的東西，不太關心所傳輸的語義。

```
class BBinder : public IBinder
{
public :
BBinder();
    virtual  const String16& getInterfaceDescriptor() const ;
     virtual  bool         isBinderAlive() const ;
     virtual status_t pingBinder();
     virtual status_t dump( int fd, const Vector<String16>& args);

    virtual status_t transact(uint32_t code, const Parcel& data,
                                    Parcel* reply, uint32_t flags = 0);

    virtual status_t linkToDeath( const sp<DeathRecipient>& recipient,
                                   void * cookie = NULL, uint32_t flags = 0);

    virtual status_t unlinkToDeath( const wp<DeathRecipient>& recipient,
                                         void * cookie = NULL, uint32_t flags = 0,
                                        wp<DeathRecipient>* outRecipient = NULL);

    virtual  void         attachObject( const  void * objectID, void * object ,
                                        void * cleanupCookie, object_cleanup_func func);
     virtual  void * findObject( const  void * objectID) const ;
     virtual  void         detachObject( const  void * objectID);

    virtual BBinder* localBinder();

protected :
     virtual             ~BBinder();

    virtual status_t onTransact(uint32_t code, const Parcel& data,
                                     Parcel* reply, uint32_t flags = 0);
private :
    BBinder( const BBinder& o);
    BBinder&     operator =( const BBinder& o);

    class Extras;
    Extras* mExtras;
    void * mReserved0;
};
```

我們目前只需關心上面的transact()成員函數，其他函數留待以後再分析。transact函數的代碼如下：

【frameworks/native/libs/binder/Binder.cpp】

```
status_t BBinder::transact(uint32_t code, const Parcel& data,
                           Parcel* reply, uint32_t flags)
{
    data.setDataPosition(0);
    status_t err = NO_ERROR;
    switch (code)
    {
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break ;
         default :
            err = onTransact(code, data, reply, flags);
            break ;
    }

    if (reply != NULL)
    {
        reply->setDataPosition(0);
    }
    return err;
}
```

看到了嗎，transact()內部會調用onTransact()，從而走到用戶所定義的子類的onTransact()裡。這個onTransact()的一大作用就是解析經由Binder機制傳過來的語義了。

2.2.4 BnInterface

遠程通信目標端的另一個重要類是BnInterface<>，它是與BpInterface<>相對應的模板類，比較關心傳輸的語義。一般情況下，服務端並不直接使用BnInterface<>，而是使用它的某個子類。為此，我們需要編寫一個新的BnXXX子類，並重載它的onTransact()成員函數。

BnInterface<>的定義如下：

```
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();
};
```

如上所示，BnInterface<>繼承於BBinder，但它並沒有實現一個默認的onTransact()成員函數，所以在遠程通信時，前文所說的BBinder::transact()調用的onTransact()應該就是BnInterface< >的某個子類的onTransact()成員函數。

2.3 幾個重要的C++宏或模板

為了便於編寫新的接口和類，Android在C++層次提供了幾個重要的宏和模板，比如我們在IInterface.h文件中，可以看到DECLARE_META_INTERFACE、IMPLEMENT_META_INTERFACE的定義。

2.3.1 DECLARE_META_INTERFACE()

DECLARE_META_INTERFACE()的定義如下：

```
#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(); \
```

我們舉個實際的例子，來說明如何使用這個宏：

```
class ICamera: public IInterface
{
public:
    DECLARE_META_INTERFACE(Camera);

    virtual void            disconnect() = 0;

    // connect new client with existing camera remote
    virtual status_t        connect(const sp<ICameraClient>& client) = 0;
```

上例中ICamera內部使用了DECLARE_META_INTERFACE(Camera)，我們把宏展開後，可以看到ICamera類的定義相當於：

```
class ICamera: public IInterface
{
public :
```

```
static  const android::String16 descriptor;
 static android::sp<ICamera> asInterface( const android::sp<android::IBinder>& obj);
 virtual  const android::String16& getInterfaceDescriptor() const ;
ICamera();
virtual ~ICamera();

virtual  void disconnect() = 0;
. . . . . .
```

宏展開的部分就是中間那5行代碼，其中最關鍵的就是asInterface()函數了，這個函數將承擔把BpBinder打包成BpInterface的職責。

2.3.2 IMPLEMENT_META_INTERFACE()

與DECLARE_META_INTERFACE相對的就是IMPLEMENT_META_INTERFACE宏。它的定義如下：

```
#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; \
    } \
    I##INTERFACE::I##INTERFACE() { } \
    I##INTERFACE::~I##INTERFACE() { } \
```

其中，實現了關鍵的asInterface()函數。

實際使用IMPLEMENT_META_INTERFACE時，我們只需把它簡單地寫在binder實體所處的cpp文件中即可，舉例如下：

```
IMPLEMENT_META_INTERFACE(Camera, "android.hardware.ICamera");

// ----------------------------------------------------------------------

status_t BnCamera::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case DISCONNECT: {
            LOGV("DISCONNECT");
            CHECK_INTERFACE(ICamera, data, reply);
            disconnect();
            return NO_ERROR;
        } break;
```

其中的IMPLEMENT_META_INTERFACE(Camera, “android.hardware.ICamera”);一句相當於以下這段代碼：

```
const android::String16 ICamera::descriptor(“android.hardware.ICamera”);
```

```
 const android::String16& ICamera::getInterfaceDescriptor() const
{
return ICamera::descriptor;
}

android::sp<ICamera> ICamera::asInterface( const android::sp<android::IBinder>& obj)
{
android::sp<ICamera > intr;
if (obj != NULL)
{
intr = static_cast<ICamera*>(obj->queryLocalInterface(
ICamera::descriptor).get());
        if (intr == NULL)
{
            intr = new BpCamera(obj);
        }
}
    return intr;
}

ICamera::ICamera() { }
ICamera::~ICamera () { }
```

看來，其中重點實現了asInterface()成員函數。請注意，asInterface()函數中會先嘗試調用queryLocalInterface()來獲取intr。此時，如果asInterface()的obj參數是個代理對象（BpBinder），那麼intr = static_cast<ICamera*>(obj->queryLocalInterface(...)一句得到的intr基本上就是NULL啦。這是因為除非用戶編寫的代理類重載queryLocalInterface()函數，否則只會以默認函數為準。而IBinder類中的默認queryLocalInterface()函數如下：

【frameworks/native/libs/binder/Binder.cpp】

```
sp<IInterface> IBinder::queryLocalInterface( const String16& descriptor)
{
    return NULL;
}
```

另一方面，如果obj參數是個實現體對象（BnInterface對象）的話，那麼queryLocalInterface()函數的默認返回值就是實體對象的this指針了，代碼如下：

【frameworks/native/include/binder/IInterface.h】

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

在我們所舉的Camera例子中，我們要研究的是如何將BpBinder轉成BpInterface，所以現在我們只闡述obj參數為BpBinder的情況。此時asInterface()函數中obj->queryLocalInterface()的返回值為NULL，於是asInterface()會走到new BpCamera(obj)一句，這一句是最關鍵的一句。我們知道，BpCamera繼承於BpInterface<ICamera>，所以此時所創建的BpCamera對象正是可被App使用的BpInterface代理對象。

BpCamera的定義如下：

```
class BpCamera: public BpInterface<ICamera>
{
public :
    BpCamera( const sp<IBinder>& impl)
        : BpInterface<ICamera>(impl)
    {
    }

    // disconnect from camera service
```

```
    void disconnect()
    {
        LOGV( "disconnect" );
        Parcel data, reply;
        data.writeInterfaceToken(ICamera::getInterfaceDescriptor());
        remote()->transact(DISCONNECT, data, &reply);
    }
. . . . . .
```

至此，IMPLEMENT_META_INTERFACE宏和asInterface()函數的關係就分析完畢了。

2.3.3 interface_cast

不過，我們經常使用的其實並不是asInterface()函數，而是interface_cast()，它簡單包裝了asInterface()：

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

以上就是關於C++層次中一些binder元素的介紹，下面我們再進一步分析其他細節。

3 ProcessState

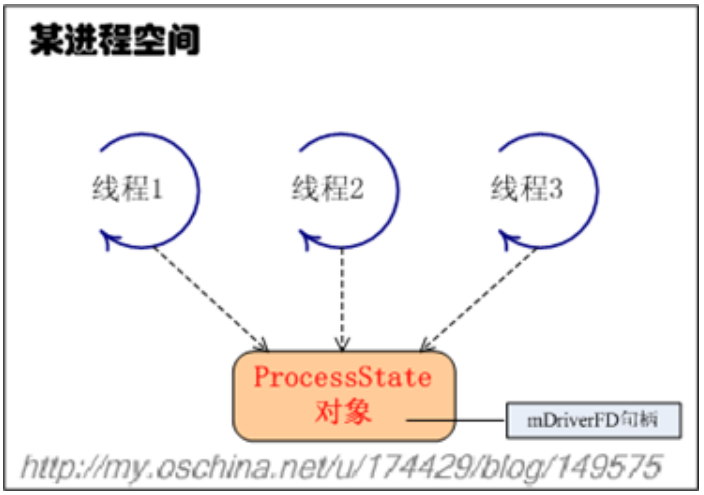
前文我們已經提到過，在Android的上層架構中，已經大幅度地弱化了進程的概念。應用程序員能看到的主要是activity、service、content provider等概念，再也找不到以前熟悉的main()函數了。然而，底層程序（C++層次）畢竟還是得跑在一個個進程之上，現在我們就來看底層進程是如何運用Binder機制來完成跨進程通信的。

在每個進程中，會有一個全局的ProcessState對象。這個很容易理解，ProcessState的字面意思不就是“進程狀態”嗎，當然應該是每個進程一個ProcessState。ProcessState的定義位於frameworks/native/include/binder/ProcessState.h中，我們只截選其中的一部分：

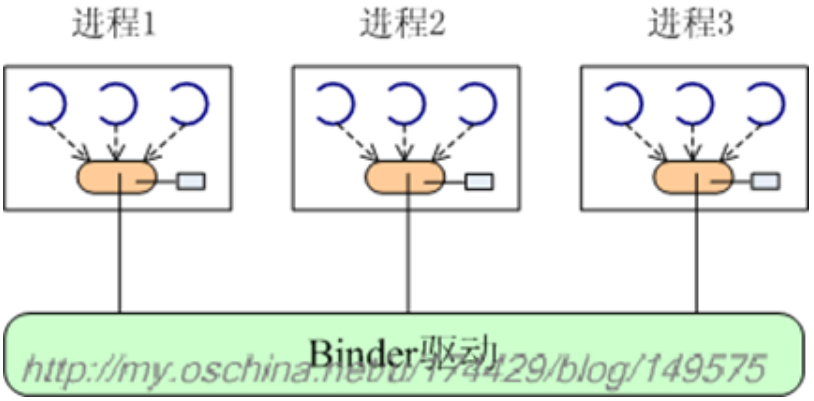
```
class ProcessState : public  virtual RefBase
{
public :
     static   sp<ProcessState> self();
    . . . . . .
    void                startThreadPool();
    . . . . . .
    void                spawnPooledThread( bool isMain);
    status_t setThreadPoolMaxThreadCount(size_t maxThreads);

private :
    friend class IPCThreadState;
    . . . . . .

    struct handle_entry
    {
        IBinder* binder;
        RefBase::weakref_type* refs;
    };
    handle_entry* lookupHandleLocked(int32_t handle);
    int                 mDriverFD;
     void * mVMStart;
    mutable Mutex mLock;   // protects everything below.

    Vector<handle_entry> mHandleToObject;
    . . . . . .
```

```
    KeyedVector<String16, sp<IBinder> > mContexts;
    . . . . . .
};
```

我們知道，Binder內核被設計成一個驅動程序，所以ProcessState裡專門搞了個mDriverFD域，來記錄binder驅動對應的句柄值，以便隨時和binder驅動通信。ProcessState對象採用了典型的單例模式，在一個應用進程中，只會有唯一的一個ProcessState對象，它將被進程中的多個線程共用，因此每個進程裡的線程其實是共用所打開的那個驅動句柄（mDriverFD）的，示意圖如下：



每個進程基本上都是這樣的結構，組合起來的示意圖就是：



我們常見的使用ProcessState的代碼如下：

```
int main( int argc, char ** argv)
{
    sp<ProcessState> proc(ProcessState::self());
    . . . . . .
    . . . . . .
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

因為ProcessState採用的是單例模式，所以它的構造函數是private的，我們只能通過調用ProcessState::self()來獲取進程中唯一的一個ProcessState對象。self()函數的代碼如下：

```
sp<ProcessState> ProcessState::self()
{
    Mutex::Autolock _l(gProcessMutex);
    if (gProcess != NULL) {
         return gProcess;
    }
    gProcess = new ProcessState;
     return gProcess;
}
```

ProcessState對象構造之時，就會打開binder驅動：

```
ProcessState::ProcessState()
    : mDriverFD(open_driver())      //打開binder驅動。
    , mVMStart(MAP_FAILED)
    , mManagesContexts( false )
    , mBinderContextCheckFunc(NULL)
    , mBinderContextUserData(NULL)
    , mThreadPoolStarted( false )
    , mThreadPoolSeq(1)
{
    . . . . . .
    mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE, mDriverFD, 0);
    . . . . . .
}
```

注意上面那句mDriverFD(open_driver())，其中的open_driver()就負責打開“/dev/binder”驅動：

```
static  int open_driver()
{
    int fd = open( "/dev/binder" , O_RDWR);
    . . . . . .
    status_t result = ioctl(fd, BINDER_VERSION, &vers);
    . . . . . .
    size_t maxThreads = 15;
    result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
    . . . . . .
    return fd;
}
```

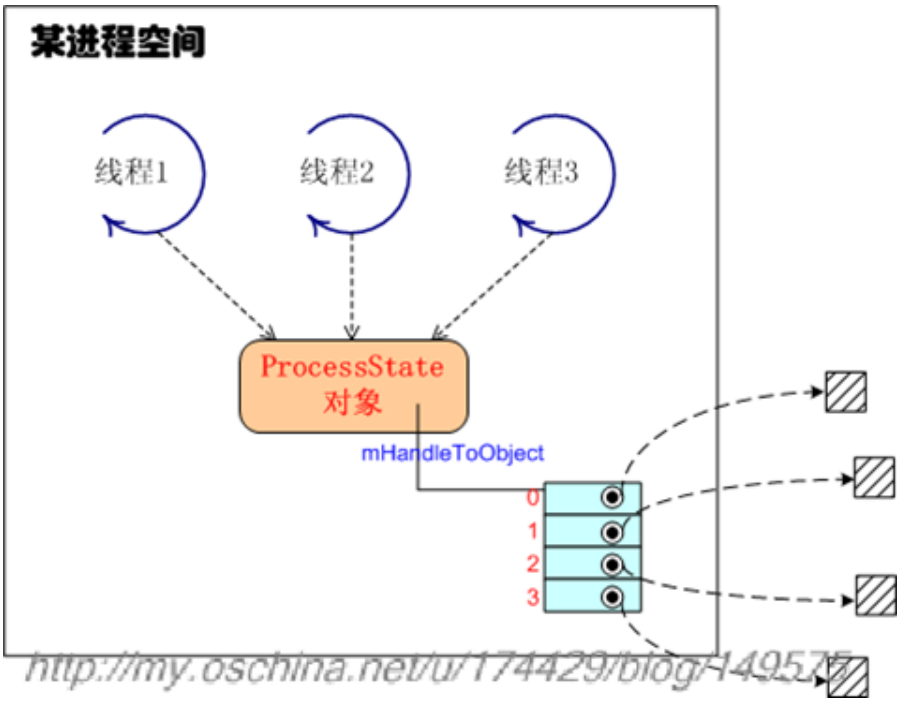
ProcessState中另一個比較有意思的域是mHandleToObject：

```
Vector<handle_entry> mHandleToObject;
```

它是本進程中記錄所有BpBinder的向量表噢，非常重要。我們前文已經說過，BpBinder是代理端的核心，現在終於看到它的藏身之處了。在Binder架構中，應用進程是通過“binder句柄”來找到對應的BpBinder的。從這張向量表中我們可以看到，那個句柄值其實對應著這個向量表的下標。這張表的子項類型為handle_entry，定義如下：

```
struct handle_entry
{
    IBinder* binder;
    RefBase::weakref_type* refs;
};
```

其中的binder域，記錄的就是BpBinder對象。



Ok，有關Binder的初步知識，我們就先說這麼多。我也不想一下子把所有的信息都塞到一篇文章中，所以打算把更多技術細節安排到其他文章中闡述，呵呵，這需要一點兒時間。

如需轉載本文內容，請註明出處。

謝謝

- « AlarmManager研究
- 紅茶一杯話Binder（ServiceManager篇） »

共有2 條網友評論

- 
  1樓：徐慶-neo發表於2013-08-13 09:27 回复此評論
  贊一個，非常好的文章
- 
  2樓：xway發表於2013-08-13 16:18 回复此評論
  準備空下來的時候學習下Android開發，這樣認真的文章值得先收藏下

發表評論

文明上網，理性發言

回到頁首 | 回到評論列表

關閉相關文章閱讀

- 2013/08/02 紅茶一杯話Binder（ServiceManager篇...
- 2013/08/12 紅茶一杯話Binder（傳輸機制篇_上）...
- 2013/08/15 紅茶一杯話Binder（傳輸機制篇_中）...
- 2013/08/04 Android Binder的使用和設計[androi...
- 2012/06/02 Android Binder IPC分析...