深入理解**Binder**

6.1 概述

**Binder**是**Android**系统提供的一种**IPC**（进程间通信）机制。由于**Android**是基于**Linux**内核的，因此，除了**Binder**外，还存在其他的**IPC**机制，例如管道和**socket**等。**Binder**相对于其他**IPC**机制来说，就更加灵活和方便了。对于初学**Android**的朋友而言，最难却又最想掌握的恐怕就是**Binder**机制了，因为**Android**系统基本上可以看作是一个基于**Binder**通信的**C/S**架构。**Binder**就像网络一样，把系统各个部分连接在了一起，因此它是非常重要的。

在基于**Binder**通信的**C/S**架构体系中，除了**C/S**架构所包括的**Client**端和**Server**端外，**Android**还有一个全局的**ServiceManager**端，它的作用是管理系统中的各种服务**(Service)**。**Client**、**Server**和**ServiceManager**这三者之间的交互关系，如图6-1所示

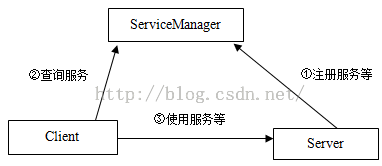


图6-1 **Client**、**Server**和**ServiceManager**三者之间的交互关系

注意：一个**Server**进程可以注册多个**Service**，就像即将讲解的**MediaServer**一样。

根据图6-1，可以得出如下结论：

· **Server**进程要先注册一些**Service**到**ServiceManager**中，所以**Server**是**ServiceManager**的客户端，而**ServiceManager**就是服务端了。

·如果某个**Client**进程要使用某个**Service**，必须先到**ServiceManager**中获取该**Service**的相关信息，所以**Client**是**ServiceManager**的客户端。

·**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**交互了，所以**Client**也是**Server**的客户端。

·最重要的一点是，三者的交互都是基于**Binder**通信的，所以通过任意两者之间的关系，都可以揭示**Binder**的奥秘。

这里，要重点强调的是**Binder**通信与**C/S**架构之间的关系。**Binder**只是为**C/S**架构提供了一种通信的方式，我们完全可以采用其他**IPC**方式进行通信，例如，系统中有很多其他的程序采用的就是**Socket**或**Pipe**的方法进行进程间通信。很多初学者可能觉得**Binder**较复杂，尤其是看到诸如**BpXXX**、**BnXXX**之类的定义便感到头晕，这很有可能是把**Binder**通信层结构和应用的业务层结构搞混了。如果能搞清楚这二者的关系，完全可以自己实现一个不使用**BpXXX**和**BnXXX**的**Service**。须知，**ServiceManager**可并没有使用它们。

6.2 庖丁解**MediaServer**

为了能像“庖丁”那样解析**Binder**，我们必须得找一头“牛”来做解剖，而**MediaServer**（简称**MS**）正是一头比较好的“牛”。它是一个可执行程序，虽然**Android**的**SDK**提供**Java**层的**API**，但**Android**系统本身还是一个完整的基于**Linux**内核的操作系统，所以不会是所有程序都用**Java**编写，这里的**MediaServer**就是一个用**C++**编写的可执行程序。

之所以选择**MediaServer**作为切入点，是因为这个**Server**是系统诸多重要**Service**的栖息地，它们包括：

· **AudioFlinger**：音频系统中的重要核心服务。

· **AudioPolicyService**：音频系统中关于音频策略的重要服务。

· **MediaPlayerService**：多媒体系统中的重要服务。

· **CameraService**：有关摄像/照相的重要服务。

可以看到，**MediaServer**除了不涉及**Surface**系统外，其他重要的服务基本上都涉及到了，它不愧是“庖丁”所要的好“牛”。

本章将以其中的**MediaPlayerService**为主切入点进行分析。先来分析**MediaServer**本身。

6.2.1 **MediaServer**的入口函数

**MediaServer**是一个可执行程序，入口函数是**main**，代码如下所示：

**[-->Main\_MediaServer.cpp]**

**int main(int argc \_\_unused, char\*\* argv)**

**{**

**signal(SIGPIPE, SIG\_IGN);**

**char value[PROPERTY\_VALUE\_MAX];**

**bool doLog = (property\_get("ro.test\_harness", value, "0") > 0) && (atoi(value) == 1);**

**pid\_t childPid;**

**if (doLog && (childPid = fork()) != 0) {**

**......**

**} else {**

**// all other services**

**if (doLog) {**

**prctl(PR\_SET\_PDEATHSIG, SIGKILL);// if parent media.log dies before me, kill me also**

**setpgid(0, 0); // but if I die first, don't kill my parent**

**}**

**//①获得一个ProcessState实例**

**sp<ProcessState> proc(ProcessState::self());**

**//②MS作为ServiceManager的客户端，需要向ServiceManger注册服务**

**//调用defaultServiceManager，得到一个IServiceManager**

**sp<IServiceManager> sm = defaultServiceManager();**

**ALOGI("ServiceManager: %p", sm.get());**

**//初始化音频系统的AudioFlinger服务**

**AudioFlinger::instantiate();**

**//③多媒体系统的MediaPlayer服务，我们将以它作为主切入点**

**MediaPlayerService::instantiate();**

**//CameraService服务**

**CameraService::instantiate();**

**//音频系统的AudioPolicy服务**

**AudioPolicyService::instantiate();**

**SoundTriggerHwService::instantiate();**

**registerExtensions();**

**......**

**//④根据名称来推断，难道是要创建一个线程池吗？**

**ProcessState::self()->startThreadPool();**

**//⑤下面的操作是要将自己加入到刚才的线程池中吗？**

**IPCThreadState::self()->joinThreadPool();**

**}**

**}**

上面的代码中，确定了5个关键点，让我们通过对这5个关键点逐一进行深入分析，来认识和理解Binder。

6.2.2 独一无二的**ProcessState**

我们在**main**函数的开始处便碰见了**ProcessState**。由于每个进程只有一个**ProcessState**，所以它是独一无二的。它的调用方式如下面的代码所示：

**[-->Main\_MediaServer.cpp]**

//①获得一个**ProcessState**实例

**sp<ProcessState> proc(ProcessState::self());**

下面，来进一步分析这个独一无二的**ProcessState**。

1. 单例的**ProcessState**

**ProcessState**的代码如下所示：

**[-->ProcessState.cpp]**

**sp<ProcessState> ProcessState::self()**

**{**

**//gProcess是在Static.cpp中定义的一个全局变量**

**//程序刚开始执行，gProcess一定为空**

**Mutex::Autolock \_l(gProcessMutex);**

**if (gProcess != NULL) {**

**return gProcess;**

**}**

**//创建一个ProcessState对象，并赋值给gProcess**

**gProcess = new ProcessState;**

**return gProcess;**

**}**

**self**函数采用了单例模式，这很明确地告诉了我们一个信息：每个进程只有一个**ProcessState**对象。这一点，从它的命名中也可看出些端倪。

2. **ProcessState**的构造

再来看**ProcessState**的构造函数。这个函数非常重要，它悄悄地打开了**Binder**设备。代码如下所示：

**[-->ProcessState.cpp]**

**ProcessState::ProcessState()**

**: mDriverFD(open\_driver()) //安卓的中有很多代码都是这么写,稍不留神就忽略这里调用了很重要的函数**

**, mVMStart(MAP\_FAILED) //映射内存的起始地址**

**, mManagesContexts(false)**

**, mBinderContextCheckFunc(NULL)**

**, mBinderContextUserData(NULL)**

**, mThreadPoolStarted(false)**

**, mThreadPoolSeq(1)**

**{**

**if (mDriverFD >= 0) {**

**// XXX Ideally, there should be a specific define for whether we**

**// have mmap (or whether we could possibly have the kernel module**

**// availabla).**

**#if !defined(HAVE\_WIN32\_IPC)**

**// mmap the binder, providing a chunk of virtual address space to receive transactions.**

**/\***

**BIDNER\_VM\_SIZE定义为(1\*1024\*1024) - (4096 \*2) = 1M-8K**

**mmap的用法希望读者man一下，不过这个函数真正的实现和驱动有关系，而Binder驱动会分配一块**

**内存用来接收数据。**

**\*/**

**mVMStart = mmap(0, BINDER\_VM\_SIZE, PROT\_READ, MAP\_PRIVATE | MAP\_NORESERVE, mDriverFD, 0);**

**if (mVMStart == MAP\_FAILED) {**

**// \*sigh\***

**ALOGE("Using /dev/binder failed: unable to mmap transaction memory.\n");**

**close(mDriverFD);**

**mDriverFD = -1;**

**}**

**#else**

**mDriverFD = -1;**

**#endif**

**}**

**LOG\_ALWAYS\_FATAL\_IF(mDriverFD < 0, "Binder driver could not be opened. Terminating.");**

**}**

3. 打开**binder**设备

**open\_driver**的作用就是打开“**/dev/binder”**这个设备，它是**android**在内核中专门用于完成进程间通信而设置的一个虚拟设备，具体实现如下所示：

**[-->ProcessState.cpp]**

**static int open\_driver()**

**{**

**int fd = open("/dev/binder", O\_RDWR); //打开/dev/binder设备**

**if (fd >= 0) {**

**fcntl(fd, F\_SETFD, FD\_CLOEXEC);**

**int vers = 0;**

**status\_t result = ioctl(fd, BINDER\_VERSION, &vers);**

**if (result == -1) {**

**ALOGE("Binder ioctl to obtain version failed: %s", strerror(errno));**

**close(fd);**

**fd = -1;**

**}**

**if (result != 0 || vers != BINDER\_CURRENT\_PROTOCOL\_VERSION) {**

**ALOGE("Binder driver protocol does not match user space protocol!");**

**close(fd);**

**fd = -1;**

**}**

**size\_t maxThreads = 15;**

**//通过ioctl方式告诉binder驱动，这个fd支持的最大线程数是15个**

**result = ioctl(fd, BINDER\_SET\_MAX\_THREADS, &maxThreads);**

**if (result == -1) {**

**ALOGE("Binder ioctl to set max threads failed: %s", strerror(errno));**

**}**

**} else {**

**ALOGW("Opening '/dev/binder' failed: %s\n", strerror(errno));**

**}**

**return fd;**

**}**

至此**Process::self**函数就分析完了。它到底干了什么呢？通过前面的分析，总结如下：

· 打开**/dev/binder**设备，这就相当于与内核的Binder驱动有了交互的通道。

· 对返回的**fd**使用**mmap**，这样**Binder**驱动就会分配一块内存来接收数据。

· 由于**ProcessState**的惟一性，因此一个进程只打开设备一次。

分析完**ProcessState**，接下来将要分析第二个关键函数**defaultServiceManager**。

6.2.3 时空穿越魔术——**defaultServiceManager**

**defaultServiceManager**函数的实现在**IServiceManager.cpp**中完成。它会返回一个**IServiceManager**对象，通过这个对象，我们可以神奇地与另一个进程**ServiceManager**进行交互。是不是有一种观看时空穿越魔术表演的感觉？

1. 魔术前的准备工作

先来看看**defaultServiceManager**都调用了哪些函数？返回的这个**IServiceManager**到底是什么？具体实现代码如下所示：

**[-->IServiceManager.cpp]**

**sp<IServiceManager> defaultServiceManager()**

**{**

**//gDefaultServiceManager定义如下：**

**//sp<IServiceManager> gDefaultServiceManager;**

**if (gDefaultServiceManager != NULL)**

**return gDefaultServiceManager;**

**{**

**AutoMutex \_l(gDefaultServiceManagerLock);**

**while (gDefaultServiceManager == NULL) {**

**//真正的gDefaultServiceManager是在这里创建的。**

**gDefaultServiceManager = interface\_cast<IServiceManager>(**

**ProcessState::self()->getContextObject(NULL));**

**if (gDefaultServiceManager == NULL)**

**sleep(1);**

**}**

**}**

**return gDefaultServiceManager;**

**}**

哦，是调用了**ProcessState**的**getContextObject**函数！注意：传给它的参数是**NULL**，即**0**。既然是“庖丁解牛”，就还要一层一层往下切。下面再看**getContextObject**函数，如下所示：

**sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& /\*caller\*/)**

**{**

**return getStrongProxyForHandle(0);**

**}**

**getStrongProxyForHandle**这个函数名怪怪的，可能会让人感到些许困惑。请注意，它的调用参数的名字叫**handle**，**Windows**编程中经常使用这个名称，它是对资源的一种标识。说白了，其实就是有一个资源项，保存在一个资源数组（也可以是别的组织结构）中，**handle**的值正是该资源项在数组中的索引。

**sp<IBinder> ProcessState::getStrongProxyForHandle(int32\_t handle)**

**{**

**sp<IBinder> result;**

**AutoMutex \_l(mLock);**

**// 根据索引查找对应资源。如果lookupHandleLocked发现没有对应的资源项，则会创建一个新的项并返回。**

**// 这个新项的内容需要填充。**

**handle\_entry\* e = lookupHandleLocked(handle);**

**if (e != NULL) {**

**// We need to create a new BpBinder if there isn't currently one, OR we**

**// are unable to acquire a weak reference on this current one. See comment**

**// in getWeakProxyForHandle() for more info about this.**

**IBinder\* b = e->binder;**

**if (b == NULL || !e->refs->attemptIncWeak(this)) {**

**if (handle == 0) {**

**// Special case for context manager...**

**// The context manager is the only object for which we create**

**// a BpBinder proxy without already holding a reference.**

**// Perform a dummy transaction to ensure the context manager**

**// is registered before we create the first local reference**

**// to it (which will occur when creating the BpBinder).**

**// If a local reference is created for the BpBinder when the**

**// context manager is not present, the driver will fail to**

**// provide a reference to the context manager, but the**

**// driver API does not return status.**

**//**

**// Note that this is not race-free if the context manager**

**// dies while this code runs.**

**//**

**// TODO: add a driver API to wait for context manager, or**

**// stop special casing handle 0 for context manager and add**

**// a driver API to get a handle to the context manager with**

**// proper reference counting.**

**Parcel data;**

**status\_t status = IPCThreadState::self()->transact(**

**0, IBinder::PING\_TRANSACTION, data, NULL, 0);**

**if (status == DEAD\_OBJECT)**

**return NULL;**

**}**

**//对于新创建的资源项，它的binder为空，所以走这个分支。注意，handle的值为0**

**b = new BpBinder(handle); //创建一个BpBinder**

**e->binder = b; //填充entry的内容**

**if (b) e->refs = b->getWeakRefs();**

**result = b;**

**} else {**

**// This little bit of nastyness is to allow us to add a primary**

**// reference to the remote proxy when this team doesn't have one**

**// but another team is sending the handle to us.**

**result.force\_set(b);**

**e->refs->decWeak(this);**

**}**

**}**

**return result; //返回BpBinder(handle)，注意，handle的值为0**

**}**

实际代码运行的如果**gDefaultServiceManager = interface\_cast<IServiceManager>(new BpBinder(0));**

2. 魔术表演的道具——**BpBinder**

众所周知，玩魔术是必须有道具的。这个穿越魔术的道具就是**BpBinder**。**BpBinder**是什么呢？有必要先来介绍它的孪生兄弟**BBinder**。

**BpBinder**和**BBinder**都是**Android**中与**Binder**通信相关的代表，它们都从**IBinder**类中派生而来，如图6-2所示：

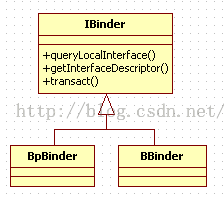


图6-2 **Binder**家族图谱

从上图中可以看出：

· **BpBinder**是客户端用来与**Server**交互的代理类，**p**即**Proxy**的意思。

· **BBinder**则是**proxy**相对的一端，它是**proxy**交互的目的端。如果说**Proxy**代表客户端，那么**BBinder**则代表服务端。这里的**BpBinder**和**BBinder**是一一对应的，即某个**BpBinder**只能和对应的**BBinder**交互。我们当然不希望通过**BpBinderA**发送的请求，却由**BBinderB**来处理。

刚才我们在**defaultServiceManager()**函数中创建了这个**BpBinder**。这里有两个问题：

· 为什么创建的不是**BBinder**？

因为我们是**ServiceManager**的客户端，当然得使用代理端以与**ServiceManager**交互了。

· 前面说了，**BpBinder**和**BBinder**是一一对应的，那么**BpBinder**如何标识它所对应的**BBinder**端呢？

答案是**Binder**系统通过**handler**来对应**BBinder**。以后我们会确认这个**Handle**值的作用。

注意：我们给**BpBinder**构造函数传的参数**handle**的值是0。这个0在整个**Binder**系统中有重要含义—因为**0**代表的就是**ServiceManager**所对应的**BBinder**。

**BpBinder**是如此重要，必须对它进行深入分析，其代码如下所示:

**[-->BpBinder.cpp]**

**BpBinder::BpBinder(int32\_t handle)**

**: mHandle(handle) //handle是0**

**, mAlive(1)**

**, mObitsSent(0)**

**, mObituaries(NULL)**

**{**

**extendObjectLifetime(OBJECT\_LIFETIME\_WEAK);**

**//另一个重要对象是IPCThreadState，我们稍后会详细讲解。**

**IPCThreadState::self()->incWeakHandle(handle);**

**}**

看上面的代码，会觉得**BpBinder**确实简单，不过再仔细查看，你或许会发现，**BpBinder**、**BBinder**这两个类没有任何地方操作**ProcessState**打开的那个**/dev/binder**设备，换言之，**BpBinder**和**BBinder**这两个**Binder**类没有和**binder**设备直接交互。那为什么说**BpBinder**会与通信相关呢？注意本小节的标题，**BpBinder**只是道具嘛！所以它后面一定还另有机关。不必急着揭秘，还是先回顾一下道具出场的历程。

我们是从下面这个函数开始分析的：

**gDefaultServiceManager =interface\_cast<IServiceManager>(**

**ProcessState::self()->getContextObject(NULL));**

现在这个函数调用将变成如下所示：

**gDefaultServiceManager =interface\_cast<IServiceManager>(new BpBinder(0));**

这里出现了一个**interface\_cast**。它是什么？其实是一个障眼法！下面就来具体分析它。

3. 障眼法——**interface\_cast**

**interface\_cast**、**dynamic\_cast**和**static\_cast**看起来是否非常眼熟？它们是指针类型转换的意思吗？如果是，那又是如何将**BpBinder\***类型强制转化成**IServiceManager\***类型的？**BpBinder**的家谱我们刚才也看了，它的“爸爸的爸爸的爸爸”这条线上没有任何一个与**IServiceManager**有任何关系。

问题谈到这里，我们得去看看**interface\_cast**的具体实现，其代码如下所示：

**[-->IInterface.h]**

**template<typename INTERFACE>**

**inline sp<INTERFACE> interface\_cast(const sp<IBinder>& obj)**

**{**

**return INTERFACE::asInterface(obj);**

**}**

哦，仅仅是一个模板函数，所以**interface\_cast<IServiceManager>()**等价于：

**inline sp<IServiceManager>interface\_cast(const sp<IBinder>& obj)**

**{**

**return IServiceManager::asInterface(obj);**

**}**

又转移到**IServiceManager**对象中去了，这难道不是障眼法吗？既然找到了“真身”，不妨就来见识见识它吧。

4. 拨开浮云见月明——**IServiceManager**

刚才提到，**IBinder**家族的**BpBinder**和**BBinder**是与通信层相关的，那么业务层的逻辑又是如何巧妙地架构在**Binder**机制上的呢？关于这些问题，可以用一个绝好的例子来解释，它就是**IServiceManager**。

（1）定义业务逻辑

先回答第一个问题：如何表述应用的业务层逻辑。可以先分析一下**IServiceManager**是怎么做的。**IServiceManager**定义了**ServiceManager**所提供的服务，看它的定义可知，其中有很多有趣的内容。**IServiceManager**定义在**IServiceManager.h**中，代码如下所示：

**[-->IServiceManager.h]**

**class IServiceManager : public IInterface**

**{**

**public:**

**DECLARE\_META\_INTERFACE(ServiceManager);//关键无比的宏！**

**virtual sp<IBinder> getService( const String16& name) const = 0;**

**virtual sp<IBinder> checkService( const String16& name) const = 0;**

**virtual status\_t addService( const String16& name,**

**const sp<IBinder>& service,**

**bool allowIsolated = false) = 0;**

**virtual Vector<String16> listServices() = 0;**

**enum {**

**GET\_SERVICE\_TRANSACTION = IBinder::FIRST\_CALL\_TRANSACTION,**

**CHECK\_SERVICE\_TRANSACTION,**

**ADD\_SERVICE\_TRANSACTION,**

**LIST\_SERVICES\_TRANSACTION,**

**};**

**};**

（2）业务与通信的挂钩

**Android**巧妙地通过**DECLARE\_META\_INTERFACE**和**IMPLENT**宏，将业务和通信牢牢地钩在了一起。**DECLARE\_META\_INTERFACE**和**IMPLEMENT\_META\_INTERFACE**这两个宏都定义在刚才的**IInterface.h**中。先看**DECLARE\_META\_INTERFACE**这个宏，如下所示：

**[-->IInterface.h::DECLARE\_META\_INTERFACE]**

**#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();**

**[--->DECLARE\_META\_INTERFACE(ServiceManager)]**

**//定义一个描述字符串**

**static const android::String16 descriptor;**

**//定义一个asInterface函数**

**static android::sp<IServiceManager> asInterface(const android::sp<android::IBinder>& obj);**

**//定义一个getInterfaceDescriptor函数，估计就是返回descriptor字符串**

**virtual const android::String16& getInterfaceDescriptor() const;**

**//定义IServiceManager的构造函数和析构函数**

**IServiceManager();**

**virtual ~IServiceManager();**

**DECLARE**宏声明了一些函数和一个变量，那么，**IMPLEMENT**宏的作用肯定就是定义它们了。**IMPLEMENT**的定义在**IInterface.h**中，**IServiceManager**是如何使用了这个宏呢？只有一行代码，在**IServiceManager.cpp**中，如下所示：

**IMPLEMENT\_META\_INTERFACE(ServiceManager,"android.os.IServiceManager");**

很简单，可直接将**IServiceManager**中的**IMPLEMENT**宏的定义展开，如下所示：

**#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() { }**

宏展开代码如下所示：

**const android::String16 IServiceManager::descriptor("android.os.IServiceManager");**

**//实现getInterfaceDescriptor函数**

**const android::String16& IServiceManager::getInterfaceDescriptor() const {**

**return IServiceManager::descriptor;**

**}**

**//实现asInterface函数**

**android::sp<IServiceManager> IServiceManager::asInterface(const android::sp<android::IBinder>& obj)**

**{**

**android::sp<IServiceManager> intr;**

**if (obj != NULL) {**

**intr = static\_cast<IServiceManager\*>(**

**obj->queryLocalInterface(**

**IServiceManager::descriptor).get());**

**if (intr == NULL) {**

**//obj是我们刚才创建的那个BpBinder(0)**

**intr = new BpServiceManager(obj);**

**}**

**}**

**return intr;**

**}**

**//实现构造函数和析构函数**

**IServiceManager::IServiceManager(){}**

**IServiceManager::~IServiceManager(){}**

我们曾提出过疑问：**interface\_cast**是如何把**BpBinder**指针转换成一个**IServiceManager**指针的呢？答案就在**asInterface**函数的一行代码中，如下所示：

**intr = new BpServiceManager(obj);**

明白了！**interface\_cast**不是指针的转换，而是利用**BpBinder**对象作为参数新建了一个**BpServiceManager**对象。我们已经知道**BpBinder**和**BBinder**与通信有关系，这里怎么突然冒出来一个**BpServiceManager**？它们之间又有什么关系呢？

（3）**IServiceManager**家族

要搞清这个问题，必须先了解**IServiceManager**家族之间的关系，先来看图6-3，它展示了**IServiceManager**的家族图谱。

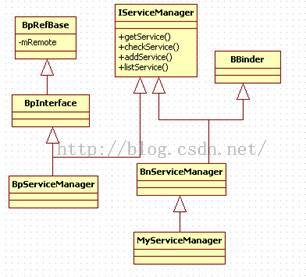


图6-3 IServiceManager的家族图谱

根据图6-3和相关的代码可知，这里有以下几个重要的点值得注意：

· **IServiceManager**、**BpServiceManager**和**BnServiceManager**都与业务逻辑相关。

· **BnServiceManager**同时从**BBinder**派生，表示它可以直接参与**Binder**通信。

· **BpServiceManager**虽然从**BpInterface**中派生，但是这条分支似乎与**BpBinder**没有关系。

· **BnServiceManager**是一个虚类，它的业务函数最终需要子类来实现。

重要说明：以上这些关系很复杂，但**ServiceManager**并没有使用错综复杂的派生关系，它直接打开**Binder**设备并与之交互。后文，还会详细分析它的实现代码。

图6-3中的**BpServiceManager**，既然不像它的兄弟**BnServiceManager**那样直接与**Binder**有血缘关系，那么它又是如何与**Binder**交互的呢？简言之，**BpRefBase**中的**mRemote**的值就是**BpBinder**。如果你不相信，仔细看**BpServiceManager**左边的派生分支树上的一系列代码，它们都在**IServiceManager.cpp**中，如下所示：

**[-->IServiceManager.cpp::BpServiceManager类]**

//通过它的参数可得知，**impl**是**IBinder**类型，看来与**Binder**有间接关系,它实际上是**BpBinder**对象

**BpServiceManager(const sp<IBinder>& impl)**

**: BpInterface<IServiceManager>(impl) //调用基类BpInterface的构造函数**

**{**

**}**

**BpInterface**的实现代码如下所示：

**[-->IInterface.h::BpInterface类]**

**template<typename INTERFACE>**

**inline BpInterface<INTERFACE>::BpInterface(const sp<IBinder>& remote)**

**:BpRefBase(remote)//基类构造函数**

**{**

**}**

**BpRefBase()**的实现代码如下所示：

**[-->Binder.cpp::BpRefBase类]**

**BpRefBase::BpRefBase(const sp<IBinder>& o)**

**: mRemote(o.get()), mRefs(NULL), mState(0) //mRemote最终等于那个new 出来的BpBinder(0)**

**{**

**extendObjectLifetime(OBJECT\_LIFETIME\_WEAK);**

**if (mRemote) {**

**mRemote->incStrong(this); // Removed on first IncStrong().**

**mRefs = mRemote->createWeak(this); // Held for our entire lifetime.**

**}**

**}**

原来，**BpServiceManager**的一个变量**mRemote**是指向了**BpBinder**。至此，我们的魔术表演完了，回想一下**defaultServiceManager**函数，可以得到以下两个关键对象：

· 有一个**BpBinder**对象，它的**handle**值是0。

· 有一个**BpServiceManager**对象，它的**mRemote**值是**BpBinder**。

**BpServiceManager**对象实现了**IServiceManager**的业务函数，现在又有**BpBinder**作为通信的代表，接下来的工作就简单了。下面，要通过分析**MediaPlayerService**的注册过程，进一步分析业务函数的内部是如何工作的。

6.2.4 注册**MediaPlayerService**

1. 业务层的工作

再回到**MediaServer**的**main**函数，下一个要分析的是**MediaPlayerService**，它的代码如下所示：

**[-->MediaPlayerService.cpp]**

**void MediaPlayerService::instantiate() {**

**defaultServiceManager()->addService(**

**String16("media.player"), new MediaPlayerService());**

**}**

根据前面的分析，**defaultServiceManager()**实际返回的对象是**BpServiceManager**，因此以上代码我们可以理解为**BpServiceManager->addService(String16(“media.player”), new MediaPlayerService())**。**BpServiceManager**它是**IServiceManager**的后代，代码如下所示：

**[-->IServiceManager.cpp::BpServiceManager的addService()函数]**

**virtual status\_t addService(const String16& name, const sp<IBinder>& service,**

**bool allowIsolated)**

**{**

**//Parcel:就把它当作是一个数据包。**

**Parcel data, reply;**

**unsigned n;**

**status\_t err;**

**//将数据打包到data中**

**data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());**

**data.writeString16(name);**

**data.writeStrongBinder(service);**

**data.writeInt32(allowIsolated ? 1 : 0);**

**for (n = 1; n <= 5; n++) {**

**//remote返回的是mRemote，也就是BpBinder对象**

**err = remote()->transact(ADD\_SERVICE\_TRANSACTION, data, &reply);**

**if (err == -EPIPE) {**

**ALOGI("%s is waiting for serviceManager... (retry %d)\n",**

**String8(name).string(), n);**

**sleep(1);**

**} else {**

**break;**

**}**

**}**

**return err == NO\_ERROR ? reply.readExceptionCode() : err;**

**}**

别急着往下走，应先思考以下两个问题：

· 调用**BpServiceManager**的**addService**是不是一个业务层的函数？

· **addService**函数中把请求数据打包成**data**后，传给了**BpBinder**的**transact**函数，这是不是把通信的工作交给了**BpBinder**？

两个问题的答案都是肯定的。至此，业务层的工作原理应该是很清晰了，它的作用就是将请求信息打包后，再交给通信层去处理。

2. 通信层的工作

下面分析**BpBinder**的**transact**函数。前面说过，在**BpBinder**中确实找不到任何与**Binder**设备交互的地方吗？那它是如何参与通信的呢？原来，秘密就在这个**transact**函数中，它的实现代码如下所示：

**[-->BpBinder.cpp]**

**status\_t BpBinder::transact(**

**uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags)**

**{**

**// Once a binder has died, it will never come back to life.**

**if (mAlive) {**

**//BpBinder果然是道具，它把transact工作交给了IPCThreadState**

**status\_t status = IPCThreadState::self()->transact(**

**mHandle, code, data, reply, flags); //mHandle也是参数**

**// give serviceManager more chances...**

**if (status == DEAD\_OBJECT && mHandle != 0) mAlive = 0;**

**return status;**

**}**

**return DEAD\_OBJECT;**

**}**

这里又遇见了**IPCThreadState**，之前也见过一次。看来，它确实与**Binder**通信有关，所以必须对其进行深入分析！

（1）“劳者一份”的**IPCThreadState**

谁是“劳者”？线程，是进程中真正干活的伙计，所以它正是劳者。而“劳者一份”，就是每个伙计一份的意思。**IPCThreadState**的实现代码在**IPCThreadState.cpp**中，如下所示：

**[-->IPCThreadState.cpp]**

**IPCThreadState\* IPCThreadState::self()**

**{**

**if (gHaveTLS) { //第一次进来为false**

**restart:**

**/\***

**TLS是Thread Local Storage（线程本地存储空间）的简称。**

**这里只需知晓：这种空间每个线程都有，而且线程间不共享这些空间。**

**通过pthread\_getspecific和pthread\_setspecific函数可以获取和设置这些空间中的内容。**

**从线程本地存储空间中获得保存在其中的IPCThreadState对象。**

**有调用pthread\_getspecific的地方，肯定也有调用pthread\_setspecific的地方**

**\*/**

**const pthread\_key\_t k = gTLS;**

**IPCThreadState\* st = (IPCThreadState\*)pthread\_getspecific(k);**

**if (st) return st;**

**// new一个对象，构造函数中会调用pthread\_setspecific**

**return new IPCThreadState;**

**}**

**if (gShutdown) return NULL;**

**pthread\_mutex\_lock(&gTLSMutex);**

**if (!gHaveTLS) {**

**if (pthread\_key\_create(&gTLS, threadDestructor) != 0) {**

**pthread\_mutex\_unlock(&gTLSMutex);**

**return NULL;**

**}**

**gHaveTLS = true;**

**}**

**pthread\_mutex\_unlock(&gTLSMutex);**

**goto restart;**

**}**

接下来，有必要转向分析它的构造函数**IPCThreadState()**，如下所示：

**[-->IPCThreadState.cpp]**

**IPCThreadState::IPCThreadState()**

**: mProcess(ProcessState::self()),**

**mMyThreadId(androidGetTid()),**

**mStrictModePolicy(0),**

**mLastTransactionBinderFlags(0)**

**{**

**//在构造函数中，把自己设置到线程本地存储中去。**

**pthread\_setspecific(gTLS, this);**

**clearCaller();**

**//mIn和mOut是两个Parcel。把它看成是发送和接收命令的缓冲区即可。**

**mIn.setDataCapacity(256);**

**mOut.setDataCapacity(256);**

**}**

每个线程都有一个**IPCThreadState**，每个**IPCThreadState**中都有一个**mIn和**一个**mOut**，其中**mIn**是用来接收来自**Binder**设备的数据的，而**mOut**则是用来存储发往**Binder**设备的数据的。

（2）勤劳的**transact**

传输工作是很辛苦的。我们刚才看到**BpBinder**的**transact**调用了**IPCThreadState**的**transact**函数，这个函数实际完成了与**Binder**通信的工作，如下面的代码所示：

**[-->IPCThreadState.cpp]**

**//注意，handle的值为0，代表了通信的目的端**

**status\_t IPCThreadState::transact(int32\_t handle,**

**uint32\_t code, const Parcel& data,**

**Parcel\* reply, uint32\_t flags)**

**{**

**status\_t err = data.errorCheck();**

**flags |= TF\_ACCEPT\_FDS;**

**......**

**if (err == NO\_ERROR) {**

**/\***

**注意这里的第一个参数BC\_TRANSACTION，它是应用程序向binder设备发送消息的消息码，**

**而binder设备向应用程序回复消息的消息码以BR\_开头。消息码的定义在binder\_module.h中，**

**请求消息码和回应消息码的对应关系，需要查看Binder驱动的实现才能将其理清楚，我们这里暂时用不上。**

**\*/**

**err = writeTransactionData(BC\_TRANSACTION, flags, handle, code, data, NULL);**

**}**

**if (err != NO\_ERROR) {**

**if (reply) reply->setError(err);**

**return (mLastError = err);**

**}**

**if ((flags & TF\_ONE\_WAY) == 0) {**

**if (reply) {**

**err = waitForResponse(reply);**

**} else {**

**Parcel fakeReply;**

**err = waitForResponse(&fakeReply);**

**}**

**......**

**} else {**

**err = waitForResponse(NULL, NULL);**

**}**

**return err;**

**}**

多熟悉的流程：先发数据，然后等结果。再简单不过了！不过，我们有必要确认一下**handle**这个参数到底起了什么作用。先来看**writeTransactionData**函数(仅把命令写到**mOut**中，并没有发送出去)，它的实现如下所示：

**status\_t IPCThreadState::writeTransactionData(int32\_t cmd, uint32\_t binderFlags,**

**int32\_t handle, uint32\_t code, const Parcel& data, status\_t\* statusBuffer)**

**{**

**//binder\_transaction\_data 是和binder设备通信的数据结构。**

**binder\_transaction\_data tr;**

**tr.target.ptr = 0; /\* Don't pass uninitialized stack data to a remote process \*/**

**//果然，handle的值传递给了target，用来标识目的端，其中0是ServiceManager的标志。**

**tr.target.handle = handle;**

**//code是消息码，用来switch/case的！**

**tr.code = code;**

**tr.flags = binderFlags;**

**tr.cookie = 0;**

**tr.sender\_pid = 0;**

**tr.sender\_euid = 0;**

**const status\_t err = data.errorCheck();**

**if (err == NO\_ERROR) {**

**tr.data\_size = data.ipcDataSize();**

**tr.data.ptr.buffer = data.ipcData();**

**tr.offsets\_size = data.ipcObjectsCount()\*sizeof(binder\_size\_t);**

**tr.data.ptr.offsets = data.ipcObjects();**

**} else if (statusBuffer) {**

**tr.flags |= TF\_STATUS\_CODE;**

**\*statusBuffer = err;**

**tr.data\_size = sizeof(status\_t);**

**tr.data.ptr.buffer = reinterpret\_cast<uintptr\_t>(statusBuffer);**

**tr.offsets\_size = 0;**

**tr.data.ptr.offsets = 0;**

**} else {**

**return (mLastError = err);**

**}**

**//把命令写到mOut中， 而不是直接发出去，可见这个函数有点名不副实。**

**mOut.writeInt32(cmd);**

**mOut.write(&tr, sizeof(tr));**

**return NO\_ERROR;**

**}**

现在，已经把**addService**的请求信息写到**mOut**中了。接下来再看发送请求和接收回复部分的实现，代码在**waitForResponse**函数中，如下所示：

**[-->IPCThreadState.cpp]**

**status\_t IPCThreadState::waitForResponse(Parcel \*reply, status\_t \*acquireResult)**

**{**

**int32\_t cmd;**

**int32\_t err;**

**while (1) {**

**if ((err=talkWithDriver()) < NO\_ERROR) break; //好家伙，talkWithDriver！**

**err = mIn.errorCheck();**

**if (err < NO\_ERROR) break;**

**if (mIn.dataAvail() == 0) continue;**

**cmd = mIn.readInt32();**

**switch (cmd) {**

**case BR\_TRANSACTION\_COMPLETE:**

**if (!reply && !acquireResult) goto finish;**

**break;**

**case BR\_DEAD\_REPLY:**

**err = DEAD\_OBJECT;**

**goto finish;**

**case BR\_FAILED\_REPLY:**

**err = FAILED\_TRANSACTION;**

**goto finish;**

**case BR\_ACQUIRE\_RESULT:**

**{**

**ALOG\_ASSERT(acquireResult != NULL, "Unexpected brACQUIRE\_RESULT");**

**const int32\_t result = mIn.readInt32();**

**if (!acquireResult) continue;**

**\*acquireResult = result ? NO\_ERROR : INVALID\_OPERATION;**

**}**

**goto finish;**

**case BR\_REPLY:**

**{**

**binder\_transaction\_data tr;**

**err = mIn.read(&tr, sizeof(tr));**

**ALOG\_ASSERT(err == NO\_ERROR, "Not enough command data for brREPLY");**

**if (err != NO\_ERROR) goto finish;**

**if (reply) {**

**if ((tr.flags & TF\_STATUS\_CODE) == 0) {**

**reply->ipcSetDataReference(**

**reinterpret\_cast<const uint8\_t\*>(tr.data.ptr.buffer),**

**tr.data\_size,**

**reinterpret\_cast<const binder\_size\_t\*>(tr.data.ptr.offsets),**

**tr.offsets\_size/sizeof(binder\_size\_t),**

**freeBuffer, this);**

**} else {**

**err = \*reinterpret\_cast<const status\_t\*>(tr.data.ptr.buffer);**

**freeBuffer(NULL,**

**reinterpret\_cast<const uint8\_t\*>(tr.data.ptr.buffer),**

**tr.data\_size,**

**reinterpret\_cast<const binder\_size\_t\*>(tr.data.ptr.offsets),**

**tr.offsets\_size/sizeof(binder\_size\_t), this);**

**}**

**} else {**

**freeBuffer(NULL,**

**reinterpret\_cast<const uint8\_t\*>(tr.data.ptr.buffer),**

**tr.data\_size,**

**reinterpret\_cast<const binder\_size\_t\*>(tr.data.ptr.offsets),**

**tr.offsets\_size/sizeof(binder\_size\_t), this);**

**continue;**

**}**

**}**

**goto finish;**

**default:**

**err = executeCommand(cmd);**

**if (err != NO\_ERROR) goto finish;**

**break;**

**}**

**}**

**finish:**

**if (err != NO\_ERROR) {**

**if (acquireResult) \*acquireResult = err;**

**if (reply) reply->setError(err);**

**mLastError = err;**

**}**

**return err;**

**}**

**OK**，我们已发送了请求数据，假设马上就收到了回复，后续该怎么处理呢？来看**executeCommand**函数，如下所示：

**status\_t IPCThreadState::executeCommand(int32\_t cmd)**

**{**

**BBinder\* obj;**

**RefBase::weakref\_type\* refs;**

**status\_t result = NO\_ERROR;**

**switch (cmd) {**

**case BR\_ERROR:**

**result = mIn.readInt32();**

**break;**

**case BR\_OK:**

**break;**

**case BR\_ACQUIRE:**

**refs = (RefBase::weakref\_type\*)mIn.readPointer();**

**obj = (BBinder\*)mIn.readPointer();**

**ALOG\_ASSERT(refs->refBase() == obj,**

**"BR\_ACQUIRE: object %p does not match cookie %p (expected %p)",**

**refs, obj, refs->refBase());**

**obj->incStrong(mProcess.get());**

**......**

**mOut.writeInt32(BC\_ACQUIRE\_DONE);**

**mOut.writePointer((uintptr\_t)refs);**

**mOut.writePointer((uintptr\_t)obj);**

**break;**

**case BR\_RELEASE:**

**refs = (RefBase::weakref\_type\*)mIn.readPointer();**

**obj = (BBinder\*)mIn.readPointer();**

**ALOG\_ASSERT(refs->refBase() == obj,**

**"BR\_RELEASE: object %p does not match cookie %p (expected %p)",**

**refs, obj, refs->refBase());**

**IF\_LOG\_REMOTEREFS() {**

**LOG\_REMOTEREFS("BR\_RELEASE from driver on %p", obj);**

**obj->printRefs();**

**}**

**mPendingStrongDerefs.push(obj);**

**break;**

**case BR\_INCREFS:**

**refs = (RefBase::weakref\_type\*)mIn.readPointer();**

**obj = (BBinder\*)mIn.readPointer();**

**refs->incWeak(mProcess.get());**

**mOut.writeInt32(BC\_INCREFS\_DONE);**

**mOut.writePointer((uintptr\_t)refs);**

**mOut.writePointer((uintptr\_t)obj);**

**break;**

**case BR\_DECREFS:**

**refs = (RefBase::weakref\_type\*)mIn.readPointer();**

**obj = (BBinder\*)mIn.readPointer();**

**// NOTE: This assertion is not valid, because the object may no**

**// longer exist (thus the (BBinder\*)cast above resulting in a different**

**// memory address).**

**mPendingWeakDerefs.push(refs);**

**break;**

**case BR\_ATTEMPT\_ACQUIRE:**

**refs = (RefBase::weakref\_type\*)mIn.readPointer();**

**obj = (BBinder\*)mIn.readPointer();**

**{**

**const bool success = refs->attemptIncStrong(mProcess.get());**

**ALOG\_ASSERT(success && refs->refBase() == obj,**

**"BR\_ATTEMPT\_ACQUIRE: object %p does not match cookie %p (expected %p)",**

**refs, obj, refs->refBase());**

**mOut.writeInt32(BC\_ACQUIRE\_RESULT);**

**mOut.writeInt32((int32\_t)success);**

**}**

**break;**

**case BR\_TRANSACTION:**

**{**

**binder\_transaction\_data tr;**

**result = mIn.read(&tr, sizeof(tr));**

**ALOG\_ASSERT(result == NO\_ERROR,**

**"Not enough command data for brTRANSACTION");**

**if (result != NO\_ERROR) break;**

**Parcel buffer;**

**buffer.ipcSetDataReference(**

**reinterpret\_cast<const uint8\_t\*>(tr.data.ptr.buffer),**

**tr.data\_size,**

**reinterpret\_cast<const binder\_size\_t\*>(tr.data.ptr.offsets),**

**tr.offsets\_size/sizeof(binder\_size\_t), freeBuffer, this);**

**const pid\_t origPid = mCallingPid;**

**const uid\_t origUid = mCallingUid;**

**const int32\_t origStrictModePolicy = mStrictModePolicy;**

**const int32\_t origTransactionBinderFlags = mLastTransactionBinderFlags;**

**mCallingPid = tr.sender\_pid;**

**mCallingUid = tr.sender\_euid;**

**mLastTransactionBinderFlags = tr.flags;**

**int curPrio = getpriority(PRIO\_PROCESS, mMyThreadId);**

**if (gDisableBackgroundScheduling) {**

**if (curPrio > ANDROID\_PRIORITY\_NORMAL) {**

**// We have inherited a reduced priority from the caller, but do not**

**// want to run in that state in this process. The driver set our**

**// priority already (though not our scheduling class), so bounce**

**// it back to the default before invoking the transaction.**

**setpriority(PRIO\_PROCESS, mMyThreadId, ANDROID\_PRIORITY\_NORMAL);**

**}**

**} else {**

**if (curPrio >= ANDROID\_PRIORITY\_BACKGROUND) {**

**// We want to use the inherited priority from the caller.**

**// Ensure this thread is in the background scheduling class,**

**// since the driver won't modify scheduling classes for us.**

**// The scheduling group is reset to default by the caller**

**// once this method returns after the transaction is complete.**

**set\_sched\_policy(mMyThreadId, SP\_BACKGROUND);**

**}**

**}**

**//ALOGI(">>>> TRANSACT from pid %d uid %d\n", mCallingPid, mCallingUid);**

**Parcel reply;**

**status\_t error;**

**......**

**if (tr.target.ptr) {**

**//看到了BBinder，想起图6-3了吗？BnServiceXXX从BBinder派生，**

**//这里的b实际上就是实现BnServiceXXX的那个对象，关于它的作用，我们要在6.5节中讲解。**

**sp<BBinder> b((BBinder\*)tr.cookie);**

**error = b->transact(tr.code, buffer, &reply, tr.flags);**

**} else {**

**//the\_context\_object是IPCThreadState.cpp中定义的一个全局变量，**

**//可通过setTheContextObject函数设置**

**error = the\_context\_object->transact(tr.code, buffer, &reply, tr.flags);**

**}**

**if ((tr.flags & TF\_ONE\_WAY) == 0) {**

**LOG\_ONEWAY("Sending reply to %d!", mCallingPid);**

**if (error < NO\_ERROR) reply.setError(error);**

**sendReply(reply, 0);**

**} else {**

**LOG\_ONEWAY("NOT sending reply to %d!", mCallingPid);**

**}**

**mCallingPid = origPid;**

**mCallingUid = origUid;**

**mStrictModePolicy = origStrictModePolicy;**

**mLastTransactionBinderFlags = origTransactionBinderFlags;**

**......**

**}**

**break;**

**case BR\_DEAD\_BINDER:**

**{**

**/\***

**收到binder驱动发来的service死掉的消息，看来只有Bp端能收到了，**

**后面，我们将会对此进行分析。**

**\*/**

**BpBinder \*proxy = (BpBinder\*)mIn.readPointer();**

**proxy->sendObituary();**

**mOut.writeInt32(BC\_DEAD\_BINDER\_DONE);**

**mOut.writePointer((uintptr\_t)proxy);**

**} break;**

**case BR\_CLEAR\_DEATH\_NOTIFICATION\_DONE:**

**{**

**BpBinder \*proxy = (BpBinder\*)mIn.readPointer();**

**proxy->getWeakRefs()->decWeak(proxy);**

**} break;**

**case BR\_FINISHED:**

**result = TIMED\_OUT;**

**break;**

**case BR\_NOOP:**

**break;**

**case BR\_SPAWN\_LOOPER:**

**//特别注意，这里将收到来自驱动的指示以创建一个新线程，用于和Binder通信。**

**mProcess->spawnPooledThread(false);**

**break;**

**default:**

**result = UNKNOWN\_ERROR;**

**break;**

**}**

**if (result != NO\_ERROR) {**

**mLastError = result;**

**}**

**return result;**

**}**

（3）打破砂锅问到底

你一定想知道如何和**Binder**设备交互的吧？是通过**write**和**read**函数来发送和接收请求的吗？来看**talkwithDriver**函数，如下所示：

**status\_t IPCThreadState::talkWithDriver(bool doReceive)**

**{**

**if (mProcess->mDriverFD <= 0) {**

**return -EBADF;**

**}**

**binder\_write\_read bwr;**

**// Is the read buffer empty?**

**const bool needRead = mIn.dataPosition() >= mIn.dataSize();**

**// We don't want to write anything if we are still reading**

**// from data left in the input buffer and the caller**

**// has requested to read the next data.**

**const size\_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;**

**bwr.write\_size = outAvail;**

**bwr.write\_buffer = (uintptr\_t)mOut.data();**

**// This is what we'll read.**

**if (doReceive && needRead) {**

**//接收数据缓冲区信息的填充。如果以后收到数据，就直接填在mIn中了。**

**bwr.read\_size = mIn.dataCapacity();**

**bwr.read\_buffer = (uintptr\_t)mIn.data();**

**} else {**

**bwr.read\_size = 0;**

**bwr.read\_buffer = 0;**

**}**

**......**

**// Return immediately if there is nothing to do.**

**if ((bwr.write\_size == 0) && (bwr.read\_size == 0)) return NO\_ERROR;**

**bwr.write\_consumed = 0;**

**bwr.read\_consumed = 0;**

**status\_t err;**

**do {**

**......**

**#if defined(HAVE\_ANDROID\_OS)**

**if (ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr) >= 0)**

**err = NO\_ERROR;**

**else {**

**err = -errno;**

**}**

**#else**

**err = INVALID\_OPERATION;**

**#endif**

**if (mProcess->mDriverFD <= 0) {**

**err = -EBADF;**

**}**

**......**

**} while (err == -EINTR);**

**......**

**if (err >= NO\_ERROR) {**

**if (bwr.write\_consumed > 0) {**

**if (bwr.write\_consumed < mOut.dataSize())**

**mOut.remove(0, bwr.write\_consumed);**

**else**

**mOut.setDataSize(0);**

**}**

**if (bwr.read\_consumed > 0) {**

**mIn.setDataSize(bwr.read\_consumed);**

**mIn.setDataPosition(0);**

**}**

**......**

**return NO\_ERROR;**

**}**

**return err;**

**}**

较为深入地分析了**MediaPlayerService**的注册过程后，下面还剩最后两个函数了，就让我们向它们发起进攻吧！

6.2.5 秋风扫落叶

重要的内容都已讲过了，现在就剩下最后两个函数**startThreadPool()**和**joinThreadPool**没有分析了。它们就太简单了不是？

1. 创造劳动力——**startThreadPool()**

**startThreadPool()的实现，如下面的代码所示：**

**[-->ProcessState.cpp]//太简单，没什么好说的**

**void ProcessState::startThreadPool()**

**{**

**AutoMutex \_l(mLock);**

**//如果要是已经startThreadPool的话，这个函数就没有什么实质作用了**

**if (!mThreadPoolStarted) {**

**mThreadPoolStarted = true;**

**spawnPooledThread(true); //注意，传进去的参数是true**

**}**

**}**

**[-->ProcessState.cpp]**

**void ProcessState::spawnPooledThread(bool isMain)**

**{**

**//注意，isMain参数是true。**

**if (mThreadPoolStarted) {**

**String8 name = makeBinderThreadName();**

**ALOGV("Spawning new pooled thread, name=%s\n", name.string());**

**sp<Thread> t = new PoolThread(isMain);**

**t->run(name.string());**

**}**

**}**

**PoolThread是在IPCThreadState中定义的一个Thread子类，它的实现，如下所示：**

**[-->IPCThreadState.h::PoolThread类]**

**class PoolThread : public Thread**

**{**

**public:**

**PoolThread(bool isMain)**

**: mIsMain(isMain)**

**{**

**}**

**protected:**

**virtual bool threadLoop()**

**{**

**//线程函数如此简单，不过是在这个新线程中又创建了一个IPCThreadState。**

**// 你还记得它是每个伙计都有一个的吗？**

**IPCThreadState::self()->joinThreadPool(mIsMain);**

**return false;**

**}**

**const bool mIsMain;**

**};**

2. 万众归一——**joinThreadPool**

还需要看看**IPCThreadState**的**joinThreadPool**的实现，因为新创建的线程也会调用这个函数，具体代码如下所示：

**[-->IPCThreadState.cpp]**

**void IPCThreadState::joinThreadPool(bool isMain)**

**{**

**//注意，如果isMain为true，我们需要循环处理。把请求信息写到mOut中，待会儿一起发出去**

**mOut.writeInt32(isMain ? BC\_ENTER\_LOOPER : BC\_REGISTER\_LOOPER);**

**// This thread may have been spawned by a thread that was in the background**

**// scheduling group, so first we will make sure it is in the foreground**

**// one to avoid performing an initial transaction in the background.**

**set\_sched\_policy(mMyThreadId, SP\_FOREGROUND);**

**status\_t result;**

**do {**

**processPendingDerefs();**

**// now get the next command to be processed, waiting if necessary**

**result = getAndExecuteCommand();**

**if (result < NO\_ERROR && result != TIMED\_OUT && result != -ECONNREFUSED && result != -EBADF)**

**{**

**ALOGE("getAndExecuteCommand(fd=%d) returned unexpected error %d, aborting",**

**mProcess->mDriverFD, result);**

**abort();**

**}**

**// Let this thread exit the thread pool if it is no longer**

**// needed and it is not the main process thread.**

**if(result == TIMED\_OUT && !isMain) {**

**break;**

**}**

**} while (result != -ECONNREFUSED && result != -EBADF);**

**mOut.writeInt32(BC\_EXIT\_LOOPER);**

**talkWithDriver(false);**

**}**

原来，我们的两个伙计在**talkWithDriver**，它们希望能从**Binder**设备那里找到点可做的事情。

3. 有几个线程在服务

到底有多少个线程在为**Service**服务呢？目前看来是两个：

· **startThreadPool**中新启动的线程通过**joinThreadPool**读取**Binder**设备，查看是否有请求。

· 主线程也调用**joinThreadPool**读取**Binder**设备，查看是否有请求。看来，**binder**设备是支持多线程操作的，其中一定是做了同步方面的工作。

**mediaserver**这个进程一共注册了4个服务，繁忙的时候，两个线程会不会显得有点少呢？另外，如果实现的服务负担不是很重，完全可以不调用**startThreadPool**创建新的线程，使用主线程即可胜任。

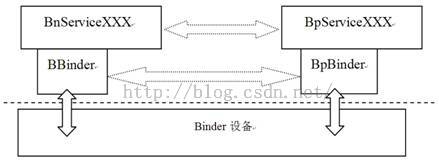
6.2.6 你彻底明白了吗？

我们以**MediaServer**为例，分析了**Binder**的机制，这里还是有必要再次强调一下**Binder**通信和基于**Binder**通信的业务之间的关系。

· **Binder**是通信机制。

· 业务可以基于**Binder**通信，当然也可以使用别的**IPC**方式通信。

**Binder**之所以复杂，重要原因之一在于**Android**通过层层封装，巧妙地把通信和业务融合在了一起。如果透彻地理解了这一点，**Binder**对我们来说就较为简单了。它们之间的交互关系可通过图6-4来表示：



6.3 服务总管**ServiceManager**

6.3.1 **ServiceManager**的原理

前面说过，**defaultServiceManager**返回的是一个**BpServiceManager**，通过它可以把命令请求发送给**handle**值为**0**的目的端。按照图6-3所示的**IServiceManager**“家谱”，无论如何也应该有一个类从**BnServiceManager**派生出来并处理这些来自远方的请求吧？

很可惜，源码中竟然没有这样的一个类存在！但确实又有这么一个程序完成了**BnServiceManager**未尽的工作，这个程序就是**servicemanager**，它的代码在**service\_manager.c**中，如下所示：

注意：通过这件事情是否能感悟到什么？嗯，我们确实可以抛开前面所有的那些封装，直接与**Binder**设备打交道。

下面来看**ServiceManager**是怎么放弃华丽的封装去做**Manager**的。

1. **ServiceManager**的入口函数

**ServiceManager**的入口函数如下所示。

**[-->frameworks/native/cmds/servicemanager/service\_manager.c]**

**int main(int argc, char \*\*argv)**

**{**

**struct binder\_state \*bs;**

**//①应该是打开binder设备吧？**

**bs = binder\_open(128\*1024);**

**if (!bs) {**

**ALOGE("failed to open binder driver\n");**

**return -1;**

**}**

**//②成为manager，是不是把自己的handle置为0？**

**if (binder\_become\_context\_manager(bs)) {**

**ALOGE("cannot become context manager (%s)\n", strerror(errno));**

**return -1;**

**}**

**selinux\_enabled = is\_selinux\_enabled();**

**sehandle = selinux\_android\_service\_context\_handle();**

**if (selinux\_enabled > 0) {**

**if (sehandle == NULL) {**

**ALOGE("SELinux: Failed to acquire sehandle. Aborting.\n");**

**abort();**

**}**

**if (getcon(&service\_manager\_context) != 0) {**

**ALOGE("SELinux: Failed to acquire service\_manager context. Aborting.\n");**

**abort();**

**}**

**}**

**union selinux\_callback cb;**

**cb.func\_audit = audit\_callback;**

**selinux\_set\_callback(SELINUX\_CB\_AUDIT, cb);**

**cb.func\_log = selinux\_log\_callback;**

**selinux\_set\_callback(SELINUX\_CB\_LOG, cb);**

**svcmgr\_handle = BINDER\_SERVICE\_MANAGER;**

**//③处理客户端发过来的请求。**

**binder\_loop(bs, svcmgr\_handler);**

**return 0;**

**}**

这里，一共有三个重要关键点。必须对其逐一地进行分析。

注意：有一些函数是在**Binder.c**中实现的，此**Binder.c**不是前面碰到的那个**Binder.cpp**。

2. 打开**Binder**设备

**binder\_open**函数用于打开**Binder**设备，它的实现如下所示：

**[-->Binder.c]**

**/\***

**这里的binder\_open应该与我们之前在ProcessState中看到的一样：**

**1）打开Binder设备**

**2）内存映射**

**\*/**

**struct binder\_state \*binder\_open(size\_t mapsize)**

**{**

**struct binder\_state \*bs;**

**struct binder\_version vers;**

**bs = malloc(sizeof(\*bs));**

**if (!bs) {**

**errno = ENOMEM;**

**return NULL;**

**}**

**bs->fd = open("/dev/binder", O\_RDWR);**

**if (bs->fd < 0) {**

**fprintf(stderr,"binder: cannot open device (%s)\n",**

**strerror(errno));**

**goto fail\_open;**

**}**

**if ((ioctl(bs->fd, BINDER\_VERSION, &vers) == -1) ||**

**(vers.protocol\_version != BINDER\_CURRENT\_PROTOCOL\_VERSION)) {**

**fprintf(stderr, "binder: driver version differs from user space\n");**

**goto fail\_open;**

**}**

**bs->mapsize = mapsize;**

**bs->mapped = mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0);**

**if (bs->mapped == MAP\_FAILED) {**

**fprintf(stderr,"binder: cannot map device (%s)\n",**

**strerror(errno));**

**goto fail\_map;**

**}**

**return bs;**

**fail\_map:**

**close(bs->fd);**

**fail\_open:**

**free(bs);**

**return NULL;**

**}**

果然如此，有了之前所学习掌握的知识，这里真的就不难理解了。

3. 成为老大

怎么才成为系统中独一无二的**manager**了呢？**manger**的实现，如下面的代码所示：

**[-->Binder.c]**

**int binder\_become\_context\_manager(struct binder\_state \*bs)**

**{**

**//实现太简单了！这个0是否就是设置自己的handle呢？**

**return ioctl(bs->fd, BINDER\_SET\_CONTEXT\_MGR, 0);**

**}**

4. 死磕**Binder**

**binder\_loop**是一个很尽责的函数。为什么这么说呢？因为它老是围绕着**”/dev/binder”**设备转悠，实现代码如下所示：

**[-->Binder.c]**

**//注意binder\_handler参数，它是一个函数指针，binder\_loop读取请求后将解析**

**//这些请求，最后调用binder\_handler完成最终的处理。**

**void binder\_loop(struct binder\_state \*bs, binder\_handler func)**

**{**

**int res;**

**struct binder\_write\_read bwr;**

**uint32\_t readbuf[32];**

**bwr.write\_size = 0;**

**bwr.write\_consumed = 0;**

**bwr.write\_buffer = 0;**

**readbuf[0] = BC\_ENTER\_LOOPER;**

**binder\_write(bs, readbuf, sizeof(uint32\_t));**

**for (;;) {**

**bwr.read\_size = sizeof(readbuf);**

**bwr.read\_consumed = 0;**

**bwr.read\_buffer = (uintptr\_t) readbuf;**

**res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);**

**if (res < 0) {**

**ALOGE("binder\_loop: ioctl failed (%s)\n", strerror(errno));**

**break;**

**}**

**//接收到请求，交给binder\_parse，最终会调用func来处理这些请求。**

**res = binder\_parse(bs, 0, (uintptr\_t) readbuf, bwr.read\_consumed, func);**

**if (res == 0) {**

**ALOGE("binder\_loop: unexpected reply?!\n");**

**break;**

**}**

**if (res < 0) {**

**ALOGE("binder\_loop: io error %d %s\n", res, strerror(errno));**

**break;**

**}**

**}**

**}**

5. 集中处理

往**binder\_loop**中传的那个函数指针是**svcmgr\_handler**，它的代码如下所示：

**[-->Service\_manager.c]**

**int svcmgr\_handler(struct binder\_state \*bs,**

**struct binder\_transaction\_data \*txn,**

**struct binder\_io \*msg,**

**struct binder\_io \*reply)**

**{**

**struct svcinfo \*si;**

**uint16\_t \*s;**

**size\_t len;**

**uint32\_t handle;**

**uint32\_t strict\_policy;**

**int allow\_isolated;**

**if (txn->target.handle != svcmgr\_handle)**

**return -1;**

**if (txn->code == PING\_TRANSACTION)**

**return 0;**

**// Equivalent to Parcel::enforceInterface(), reading the RPC**

**// header with the strict mode policy mask and the interface name.**

**// Note that we ignore the strict\_policy and don't propagate it**

**// further (since we do no outbound RPCs anyway).**

**strict\_policy = bio\_get\_uint32(msg);**

**s = bio\_get\_string16(msg, &len);**

**if (s == NULL) {**

**return -1;**

**}**

**if ((len != (sizeof(svcmgr\_id) / 2)) ||**

**memcmp(svcmgr\_id, s, sizeof(svcmgr\_id))) {**

**fprintf(stderr,"invalid id %s\n", str8(s, len));**

**return -1;**

**}**

**if (sehandle && selinux\_status\_updated() > 0) {**

**struct selabel\_handle \*tmp\_sehandle = selinux\_android\_service\_context\_handle();**

**if (tmp\_sehandle) {**

**selabel\_close(sehandle);**

**sehandle = tmp\_sehandle;**

**}**

**}**

**switch(txn->code) {**

**case SVC\_MGR\_GET\_SERVICE: //得到某个service的信息，service用字符串表示。**

**case SVC\_MGR\_CHECK\_SERVICE:**

**s = bio\_get\_string16(msg, &len); //s是字符串表示的service名称。**

**if (s == NULL) {**

**return -1;**

**}**

**handle = do\_find\_service(bs, s, len, txn->sender\_euid, txn->sender\_pid);**

**if (!handle)**

**break;**

**bio\_put\_ref(reply, handle);**

**return 0;**

**case SVC\_MGR\_ADD\_SERVICE: //对应addService请求**

**s = bio\_get\_string16(msg, &len);**

**if (s == NULL) {**

**return -1;**

**}**

**handle = bio\_get\_ref(msg);**

**allow\_isolated = bio\_get\_uint32(msg) ? 1 : 0;**

**if (do\_add\_service(bs, s, len, handle, txn->sender\_euid,**

**allow\_isolated, txn->sender\_pid))**

**return -1;**

**break;**

**case SVC\_MGR\_LIST\_SERVICES: { //得到当前系统已经注册的所有service的名字。**

**uint32\_t n = bio\_get\_uint32(msg);**

**if (!svc\_can\_list(txn->sender\_pid)) {**

**ALOGE("list\_service() uid=%d - PERMISSION DENIED\n",**

**txn->sender\_euid);**

**return -1;**

**}**

**si = svclist;**

**while ((n-- > 0) && si)**

**si = si->next;**

**if (si) {**

**bio\_put\_string16(reply, si->name);**

**return 0;**

**}**

**return -1;**

**}**

**default:**

**ALOGE("unknown code %d\n", txn->code);**

**return -1;**

**}**

**bio\_put\_uint32(reply, 0);**

**return 0;**

**}**

6.3.2 服务的注册

上面提到的switch/case语句，将实现IServiceManager中定义的各个业务函数，我们重点看do\_add\_service这个函数，它最终完成了对addService请求的处理实现，代码如下所示：

**[-->Service\_manager.c]**

**int do\_add\_service(struct binder\_state \*bs,**

**const uint16\_t \*s, size\_t len,**

**uint32\_t handle, uid\_t uid, int allow\_isolated,**

**pid\_t spid)**

**{**

**struct svcinfo \*si;**

**if (!handle || (len == 0) || (len > 127))**

**return -1;**

**//1. 不是什么都可以注册的，svc\_can\_register见下**

**if (!svc\_can\_register(s, len, spid)) {**

**ALOGE("add\_service('%s',%x) uid=%d - PERMISSION DENIED\n",**

**str8(s, len), handle, uid);**

**return -1;**

**}**

**si = find\_svc(s, len);**

**if (si) {**

**if (si->handle) {**

**ALOGE("add\_service('%s',%x) uid=%d - ALREADY REGISTERED, OVERRIDE\n",**

**str8(s, len), handle, uid);**

**svcinfo\_death(bs, si);**

**}**

**si->handle = handle;**

**} else {**

**si = malloc(sizeof(\*si) + (len + 1) \* sizeof(uint16\_t));**

**if (!si) {**

**ALOGE("add\_service('%s',%x) uid=%d - OUT OF MEMORY\n",**

**str8(s, len), handle, uid);**

**return -1;**

**}**

**si->handle = handle;**

**si->len = len;**

**memcpy(si->name, s, (len + 1) \* sizeof(uint16\_t));**

**si->name[len] = '\0';**

**si->death.func = (void\*) svcinfo\_death; //service退出的通知函数**

**si->death.ptr = si;**

**si->allow\_isolated = allow\_isolated;**

**//这个svclist是一个list，保存了当前注册到ServiceManager中的信息**

**si->next = svclist;**

**svclist = si;**

**}**

**binder\_acquire(bs, handle);**

**// 我们希望当服务进程退出后，ServiceManager能有机会做一些清理工作，例如释放前面malloc出来的si。**

**// binder\_link\_to\_death完成这项工作，每当有服务进程退出时，ServiceManager都会得到来自**

**// Binder设备的通知。**

**binder\_link\_to\_death(bs, handle, &si->death);**

**return 0;**

**}**

至此，服务注册分析完毕。可以知道，**ServiceManager**不过就是保存了一些服务的信息。那么，这样做又有什么意义呢？

6.3.3 **ServiceManager**存在的意义

为何需要一个**ServiceManager**，其重要作用何在？

· **ServiceManger**能集中管理系统内的所有服务，它能施加权限控制，并不是任何进程都能注册服务。

· **ServiceManager**支持通过字符串名称来查找对应的**Service**。这个功能很像**DNS**。

· 由于各种原因，**Server**进程可能生死无常。如果让每个**Client**都去检测，压力实在太大。现在有了统一的管理机构，**Client**只需要查询**ServiceManager**，就能把握动向，得到最新信息。这可能正是**ServiceManager**存在的最大意义吧。

6.4 **MediaPlayerService**和它的**Client**

前面，一直在讨论**ServiceManager**和它的**Client**，现在我们以**MediaPlayerService**的**Client**换换口味吧。由于**ServiceManager**不是从**BnServiceManager**中派生的，所以之前没有讲述请求数据是如何从通讯层传递到业务层来处理的过程。本节，我们以**MediaPlayerService**和它的**Client**做为分析对象，试解决这些遗留问题。

6.4.1 查询**ServiceManager**

前文曾分析过**ServiceManager**的作用，一个**Client**想要得到某个**Service**的信息，就必须先和**ServiceManager**打交道，通过调用**getService**函数来获取对应**Service**的信息。请看来源于**IMediaDeathNotifier.cpp**中的例子**getMediaPlayerService()**，它的代码如下所示：

**[-->IMediaDeathNotifier.cpp]**

**// 这个函数通过与ServiceManager通信，获得一个能够与MediaPlayerService通信的BpBinder，**

**// 然后再通过障眼法interface\_cast，转换成一个BpMediaPlayerService。**

**const sp<IMediaPlayerService>&**

**IMediaDeathNotifier::getMediaPlayerService()**

**{**

**Mutex::Autolock \_l(sServiceLock);**

**if (sMediaPlayerService == 0) {**

**sp<IServiceManager> sm = defaultServiceManager();**

**sp<IBinder> binder;**

**do {**

**//向ServiceManager查询对应服务的信息，返回BpBinder。**

**binder = sm->getService(String16("media.player"));**

**if (binder != 0) {**

**break;**

**}**

**//如果ServiceManager上还没有注册对应的服务，则需要等待，直到对应服务注册**

**//到ServiceManager中为止。**

**usleep(500000); // 0.5 s**

**} while (true);**

**if (sDeathNotifier == NULL) {**

**sDeathNotifier = new DeathNotifier();**

**}**

**binder->linkToDeath(sDeathNotifier);**

**//通过interface\_cast，将这个binder转化成BpMediaPlayerService，**

**//binder中的handle标识的一定是目的端MediaPlayerService。**

**//sp<IMediaPlayerService> IMediaDeathNotifier::sMediaPlayerService;**

**sMediaPlayerService = interface\_cast<IMediaPlayerService>(binder);**

**}**

**return sMediaPlayerService;**

**}**

有了**BpMediaPlayerService**，就能够使用任何**IMediaPlayerService**提供的业务逻辑函数了。例如它提供的**createMediaRecorder**和**createMetadataRetriever**等。

显而易见的是，调用的这些函数都将把请求数据打包发送给Binder驱动，由BpBinder中的handle值找到对应端的处理者来处理。这中间经历过如下的过程：

（1）通讯层接收到请求。

（2）递交给业务层处理。

想了解这中间的过程吗？下面就对此做详细分析。

6.4.2 子承父业

根据前面的分析可知，**MediaPlayerService**驻留在**MediaServer**进程中，这个进程有两个线程在**talkWithDriver**。假设其中有一个线程收到了请求，它最终会通过**executeCommand**调用来处理这个请求，实现代码如下所示：

**[-->IPCThreadState.cpp]**

**status\_t IPCThreadState::executeCommand(int32\_tcmd)**

**{**

**BBinder\*obj;**

**RefBase::weakref\_type\* refs;**

**status\_tresult = NO\_ERROR;**

**switch(cmd) {**

**case BR\_ERROR:**

**result = mIn.readInt32();**

**break;**

**......**

**case BR\_TRANSACTION:**

**{**

**binder\_transaction\_data tr;**

**result = mIn.read(&tr, sizeof(tr));**

**if (result != NO\_ERROR) break;**

**Parcel buffer;**

**Parcel reply;**

**if (tr.target.ptr) {**

**/\***

**看到BBinder，想起图6-3了吗？ BnServiceXXX从BBinder派生，**

**这里的b实际就是实现BnServiceXXX的那个对象，这样就直接定位到了业务层的对象。**

**\*/**

**sp<BBinder> b((BBinder\*)tr.cookie);**

**const status\_t error = b->transact(tr.code, buffer, &reply, 0);**

**if (error < NO\_ERROR) reply.setError(error);**

**} else {**

**/\***

**the\_context\_object是IPCThreadState.cpp中定义的一个全局变量。可通过**

**setTheContextObject函数设置。**

**\*/**

**const status\_t error =**

**the\_context\_object->transact(tr.code,buffer, &reply, 0);**

**if (error < NO\_ERROR) reply.setError(error);**

**}**

**break;**

**......**

**BBinder**和业务层有什么关系？还记得图6-3吗？我们以**MediaPlayerService**为例，来梳理一下其派生关系，如图6-5所示：

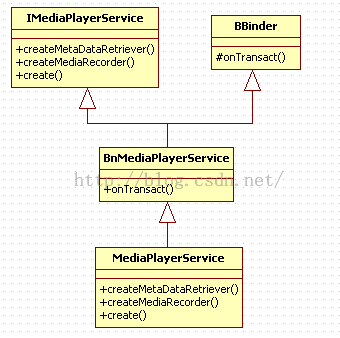


图6-5 MediaPlayerService家谱

**BnMediaPlayerService**实现了**onTransact**函数，它将根据消息码调用对应的业务逻辑函数，这些业务逻辑函数由**MediaPlayerService**来实现。这一路的历程，如下面的代码所示：

**[-->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:**

**//调用子类的onTransact，这是一个虚函数。**

**err = onTransact(code, data, reply, flags);**

**break;**

**}**

**if (reply != NULL) {**

**reply->setDataPosition(0);**

**}**

**return err;**

**}**

**status\_t BnMediaPlayerService::onTransact(**

**uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags)**

**{**

**switch (code) {**

**......**

**case CREATE\_MEDIA\_RECORDER: {**

**CHECK\_INTERFACE(IMediaPlayerService, data, reply);**

**//子类要实现createMediaRecorder函数**

**sp<IMediaRecorder> recorder = createMediaRecorder();**

**reply->writeStrongBinder(recorder->asBinder());**

**return NO\_ERROR;**

**} break;**

**case CREATE\_METADATA\_RETRIEVER: {**

**CHECK\_INTERFACE(IMediaPlayerService, data, reply);**

**//子类要实现createMetadataRetriever函数**

**sp<IMediaMetadataRetriever> retriever = createMetadataRetriever();**

**reply->writeStrongBinder(retriever->asBinder());**

**return NO\_ERROR;**

**} break;**

**case GET\_OMX: {**

**CHECK\_INTERFACE(IMediaPlayerService, data, reply);**

**sp<IOMX> omx = getOMX();**

**reply->writeStrongBinder(omx->asBinder());**

**return NO\_ERROR;**

**} break;**

**case MAKE\_CRYPTO: {**

**CHECK\_INTERFACE(IMediaPlayerService, data, reply);**

**sp<ICrypto> crypto = makeCrypto();**

**reply->writeStrongBinder(crypto->asBinder());**

**return NO\_ERROR;**

**} break;**

**case MAKE\_DRM: {**

**CHECK\_INTERFACE(IMediaPlayerService, data, reply);**

**sp<IDrm> drm = makeDrm();**

**reply->writeStrongBinder(drm->asBinder());**

**return NO\_ERROR;**

**} break;**

**case MAKE\_HDCP: {**

**CHECK\_INTERFACE(IMediaPlayerService, data, reply);**

**bool createEncryptionModule = data.readInt32();**

**sp<IHDCP> hdcp = makeHDCP(createEncryptionModule);**

**reply->writeStrongBinder(hdcp->asBinder());**

**return NO\_ERROR;**

**} break;**

**case ADD\_BATTERY\_DATA: {**

**CHECK\_INTERFACE(IMediaPlayerService, data, reply);**

**uint32\_t params = data.readInt32();**

**addBatteryData(params);**

**return NO\_ERROR;**

**} break;**

**case PULL\_BATTERY\_DATA: {**

**CHECK\_INTERFACE(IMediaPlayerService, data, reply);**

**pullBatteryData(reply);**

**return NO\_ERROR;**

**} break;**

**case LISTEN\_FOR\_REMOTE\_DISPLAY: {**

**CHECK\_INTERFACE(IMediaPlayerService, data, reply);**

**sp<IRemoteDisplayClient> client(**

**interface\_cast<IRemoteDisplayClient>(data.readStrongBinder()));**

**String8 iface(data.readString8());**

**sp<IRemoteDisplay> display(listenForRemoteDisplay(client, iface));**

**reply->writeStrongBinder(display->asBinder());**

**return NO\_ERROR;**

**} break;**

**case GET\_CODEC\_LIST: {**

**CHECK\_INTERFACE(IMediaPlayerService, data, reply);**

**sp<IMediaCodecList> mcl = getCodecList();**

**reply->writeStrongBinder(mcl->asBinder());**

**return NO\_ERROR;**

**} break;**

**default:**

**return BBinder::onTransact(code, data, reply, flags);**

**}**

**}**

通过上面的分析，我们是否能更清晰地理解了图6-3、图6-4和图6-5所表达的意义了呢？

6.5 拓展思考

这一节，让我们进一步讨论和思考几个与**Binder**有关的问题。这几个问题和**Binder**的实现有关系。先简单介绍一下**Binder**驱动相关的内容：

· **Binder**的驱动代码在kernel-3.10/drivers/android/binder.c中，另外该目录下还有一个**binder.h**头文件。**Binder**是一个虚拟设备，所以它的代码相比而言还算简单，读者只要有基本的Linux驱动开发方面的知识就能读懂它。

· /proc/binder目录下的内容可用来查看Binder设备的运行状况。

6.5.1 **Binder**和线程的关系

以**MediaServer**为例，现在程序运行正常。此时**MediaServer**：

（1）通过**startThreadPool**启动了一个线程，这个线程在**talkWithDriver**。

（2）主线程通过**joinThreadPool**，也在**talkWithDriver**。

至此我们已知道，有两个线程在和**Binder**设备打交道。这时在业务逻辑上需要与**ServiceManager**交互，比如要调用**listServices**打印所有服务的名字，假设这是**MediaServer**中的第三个线程。按照之前的分析，它最终会调用**IPCThreadState**的**transact**函数，这个函数会**talkWithDriver**并把请求发到**ServiceManager**进程，然后等待来自**Binder**设备的回复。那么现在一共有三个线程（不论是在等待来自其他**Client**的请求，还是在等待**listService**的回复）都在**talkWithDriver**。

**ServiceManager**处理完了**listServices**，把回复结果写回**Binder**驱动，那么，**MediaServer**中哪个线程会收到回复呢？此问题如图6-6表示：

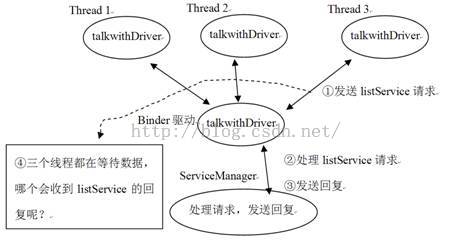


图6-6 本问题的示意图

显而易见，当然是调用**listServices**的那个线程会得到结果。为什么？因为如果不这么做，则会导致下面情况的发生：

· 如果是没有调用**listServices**的线程1或者线程2得到回复，那么它们应该唤醒调用**listServices**的线程3。因为这时已经有了结果，线程3应该从**listServices**函数调用中返回。

· 这其中的线程等待、唤醒、切换会浪费不少宝贵的时间片，而且代码逻辑会极其复杂。

看来，**Binder**设备把发起请求的线程牢牢地拴住了，必须收到回复才能放它离开。这种一一对应的方式极大简化了代码层的处理逻辑。

说明：想想**socket**编程吧，同一时刻不能有多个线程操作**socket**的读/写，否则数据会乱套。

另外，我们再分析**executeCommand**的时候，特别提到**BR\_SPAWN\_LOOPER**这个**case**的处理，它用于建立新的线程来和**Binder**通信。什么会收到这个消息呢？请读者研究**binder**的驱动。如果有新发现，请告诉我，大家再一起学习。

6.5.2 有人情味的讣告

在socket编程中，当一个socket 关闭后，我们无比希望另一端的select/poll/epoll/WaitForXXX有相应返回，以示通知。

说明：在广域网中，我们常常会因为收不到或者延时收到socket的close消息而烦恼。

在Binder系统中，要是明确表示对BnXXX的生死非常关心，那么在它离世后你会收到一份讣告。你可以嚎啕大哭，或者什么也不做。

关于这个问题，请直接看源码中的例子吧。

1. 表达你的关心

要想收到讣告，必须先要表达你的关心：做下面两件事：

· 从IBinder::DeathRecipient派生一个类，并实现其中的通知函数binderDied。这个函数一旦被调用，就相当于你收到了讣告。

· 把这个类注册到系统，告诉你关心哪一个BnXXX的生死。

看示例代码，它在mediametadataretriever.cpp中，如下所示：

[-->frameworks/av/media/libmedia/mediametadataretriever.cpp]

const sp<IMediaPlayerService>& MediaMetadataRetriever::getService()

{

Mutex::Autolock lock(sServiceLock);

if (sService == 0) {

sp<IServiceManager> sm = defaultServiceManager();

sp<IBinder> binder;

do {

binder = sm->getService(String16("media.player"));

if (binder != 0) {

break;

}

ALOGW("MediaPlayerService not published, waiting...");

usleep(500000); // 0.5 s

} while (true);

if (sDeathNotifier == NULL) {

sDeathNotifier = new DeathNotifier();

}

//调用下面这个函数，告诉系统我们对这个binder的生死有兴趣

//这个binder是一个BpBinder，它关心的是对端BBinder，也即是BnXXX的父类。

binder->linkToDeath(sDeathNotifier);

sService = interface\_cast<IMediaPlayerService>(binder);

}

ALOGE\_IF(sService == 0, "no MediaPlayerService!?");

return sService;

}

2. 讣告是怎么收到的？

那么，这份讣告是怎么收到的呢？答案也在executeCommand中，代码如下所示：

[-->IPCThreadState.cpp]

status\_t IPCThreadState::executeCommand(int32\_t cmd)

{

BBinder\*obj;

RefBase::weakref\_type\* refs;

status\_t result = NO\_ERROR;

switch(cmd) {

case BR\_ERROR:

result = mIn.readInt32();

break;

......

case BR\_DEAD\_BINDER:

{

//Binder驱动会通知死亡消息。下面的proxy对应着已经死亡的远端BBinder。

BpBinder \*proxy =(BpBinder\*)mIn.readInt32();

//发送讣告，Obituary是讣告的意思。最终会传递到你的DeathNotifier中。

proxy->sendObituary();

mOut.writeInt32(BC\_DEAD\_BINDER\_DONE);

mOut.writeInt32((int32\_t)proxy);

}

break;

default:

result = UNKNOWN\_ERROR;

break;

}

3. 你死了，我怎么办？

收到讣告后该怎么办呢？有一些代码看起来非常薄情寡义，如下所示：

[-->frameworks/av/media/libmedia/mediametadataretriever.cpp]

void MediaMetadataRetriever::DeathNotifier::binderDied(const wp<IBinder>& who \_\_unused) {

Mutex::Autolock lock(MediaMetadataRetriever::sServiceLock);

//把自己保存的BpMediaPlayerService对象干掉！

MediaMetadataRetriever::sService.clear();

ALOGW("MediaMetadataRetriever server died!"); //打印一下LOG，这样就完事大吉了

}

4. 承受不住的诺言

我答应收到讣告后给你送终，可是如果我要是死在你前面或者中途我不想接收讣告，又该怎么办呢？先来看下面的代码：

MediaMetadataRetriever::DeathNotifier::~DeathNotifier()

{

Mutex::Autolock lock(sServiceLock);

// DeathNotifier对象不想活了，但是BnMediaPlayerService还活着，

// 或者DeathNotifier中途变卦。怎么办？

//unlinkToDeath调用可以取消对BnMediaPlayerService的关心。

if (sService != 0) {

sService->asBinder()->unlinkToDeath(this);

}

}

Binder的这个讣告是不是很有人情味呢？想知道它是怎么做到的吗？还是先去看看驱动的实现吧。

6.5.3 匿名Service

匿名Service就是没有注册的Service，这句话是什么意思？

· 没有注册意味着这个Service没有在ServiceManager上注册。

· 它是一个Service又表示它确实是一个基于Binder通讯的C/S结构。

再看下面的代码，或许就会明白是什么意思了。

[-->IMediaPlayerService.cpp]

status\_t BnMediaPlayerService::onTransact(uint32\_tcode, const Parcel& data,

Parcel\* reply, uint32\_t flags)

{

switch(code) {

case CREATE\_URL: {

CHECK\_INTERFACE(IMediaPlayerService, data, reply);

...

//player是一个IMediaPlayer类型的对象

sp<IMediaPlayer> player =create(

pid, client, url, numHeaders > 0 ? &headers : NULL);

//下面这句话也很重要

reply->writeStrongBinder(player->asBinder());

return NO\_ERROR;

}break;

当MediaPlayerClient调用create函数时，MediaPlayerService会返回一个IMediaPlayer对象，此后，MediaPlayerClient即可直接使用这个IMediaPlayer来进行跨进程的函数调用了。

请看，这里确实也存在C/S的两端：

· BpMediaPlayer，由MediaPlayerClient使用，它用来调用IMediaPlayer提供的业务服务

· BnMediaPlayer，由MediaPlayerService使用，用来处理来自Client端的业务请求。

上面明显是一个C/S结构，但在ServiceManager中，肯定没有IMediaPlayer的信息，那么BpMediaPlayer是如何得到BnMediaPlayer的handle值的呢？

注意：handle事关通信的目的端，因此它非常重要。

答案可能就在下面这句话中：

reply->writeStrongBinder(player->asBinder());//将Binder类型作为一种特殊数据类型处理

当这个reply写到Binder驱动中时，驱动可能会特殊处理这种IBinder类型的数据，例如为这个BBinder建立一个独一无二的handle，这其实相当于在Binder驱动中注册了一项服务。

通过这种方式，MediaServer输出了大量的Service，例如IMediaPlayer和IMediaRecorder等。

说明：关于这个问题，也可以查看驱动的实现来验证这一想法。

6.6 学以致用

全书可能惟有Binder系统有如此大的魅力，让我们用单独一节来介绍如何使用它。

6.6.1 纯Native的Service

纯Native的Service表示我们的代码都在Native层。Native层有很多service，前面的MediaServer不就是一个重量级的吗？

假设service叫Test，我们该如何实现呢？完全可以模仿MS！具体实现过程，如代码所示：

[-->Test.cpp::范例]

int main()

{

sp<ProcessState> proc(ProcessState::self());

sp<IServiceManager> sm =defaultServiceManager();

//记住注册你的服务，否则谁也找不着你！

sm->addService(“service.name”,newTest());

//如果压力不大，可以不用单独搞一个线程。

ProcessState::self()->startThreadPool();

//这个是必需的，否则主线程退出了，你也完了。

IPCThreadState::self()->joinThreadPool();

}

Test是怎么定义的呢？我们是跨进程的C/S，所以本地需要一个BnTest，对端需要提供一个代理BpTest。为了不暴露Bp的身份，Bp的定义和实现都放在BnTest.cpp中好了。

注意：你虽可以暴露Bp的身份（输出它的头文件），但也没有必要，因为客户端用的是基类ITest指针。

1. 我能干什么

ITest接口表明它所提供的服务，例如getTest和setTest等，这个与业务逻辑相关，代码如下所示：

说明：getTest也可以返回一个ITestService类型的Service！

[-->ITest.h::声明ITest]

//需要从IInterface派生

class ITest: public IInterface

{

public:

//神奇的宏 DECLARE\_META\_INTERFACE

DECLARE\_META\_INTERFACE(Test);

virtualvoid getTest() = 0;

virtual voidsetTest() = 0;

}//ITest是一个接口类。

2. 定义BnTest和BpTest

为了把ITest融入到Binder系统，需要定义BnTest和对客户端透明的BpTest。BnTest定义既可以与上面的Test定义放在一块，也可以分开，如下所示：

[-->ITest.h::声明BnTest]

class BnTest: public BnInterface<ITest>

{

public:

//由于ITest是个纯虚类，而BnTest只实现了onTransact函数，所以BnTest依然是一个纯虚类

virtual status\_t onTransact(uint32\_t code,

const Parcel& data,

Parcel \*reply,

uint32\_t flags = 0);

};

另外，我们还要使用IMPLEMNT宏。参考BnMediaPlayerService的方法，把BnTest和BpTest的实现都放在ITest.cpp中，如下所示：

[-->ITest.cpp::BnTest的实现]

IMPLEMENT\_META\_INTERFACE(Test, "android.Test.ITest");//IMPLEMENT宏

status\_t BnTest::onTransact(uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags)

{

switch(code) {

case GET\_Test:{

CHECK\_INTERFACE(ITest, data, reply);

getTest();//子承父业，由Test完成。

return NO\_ERROR;

}break; //SET\_XXX类似

BpTest也在这里实现吧，如下所示：

[-->ITest.cpp::BpTest的实现]

class BpTest: public BpInterface<ITest>

{

public:

BpXXX(const sp<IBinder>& impl):BpInterface< ITest >(impl)

{

}

vitural getTest()

{

Parcel data, reply;

data.writeInterfaceToken(ITest::getInterfaceDescriptor());

data.writeInt32(pid);

//打包请求数据，然后交给BpBinder通讯层处理。

remote()->transact(GET\_Test, data, &reply);

return;

}

//setTest类似

纯Native的Service写起来量大一些，上面的代码还只是把C/S的框架写好了，真实的业务处理尚未开始，不过感觉却很踏实，很厚重。那么，Java层的Service该怎么写呢？

6.6.2 扶得起的阿斗

阿斗（aidl谐音）本来是扶不起的，可是我们有了AIDL工具，就有可能将他扶起！

1. 我能干什么

在Java层中，如果想要利用Binder进行跨进程的通信，也得定义一个类似ITest的接口，不过这是一个aidl文件。现假设，服务端程序都在com.test.service包中。

ITest.aidl文件内容如下：

[-->ITest.aidl]

package com.test.service;

import com.test.complicatedDataStructure

interface ITest{

// complicatedDataStructure类是自己定义的复杂数据结构,in表示输入参数，out表示输出参数

//in和out的表示一定要准确。切记！

int getTest(outcomplicatedDataStructure);

int setTest(inString name,in boolean reStartServer);

}

定义完后，如果用Eclipse进行编译，会在gen目录下生成一个com.test.ITest.java文件（也会生成对应包结构的目录）。内容就不具体罗列了，只关注其是如何实现服务端的。

说明：Eclipse用的也是aidl工具，可以手动使用这个工具编译aidl文件。

2. 实现服务端

com.test.ITest.java只是实现了一个类似BnTest的东西，具体的业务实现还需从ITest.Stub派生，实现代码如下所示：

[-->ITestImpl.java]

/\*

ITest.Stub是在aidl生成的那个java文件中定义的，非常类似Native层的BnTest,。

ITestImpl必须从ITest.Stub中派生，用来实现具体的业务函数。

\*/

package com.test.service

class ITestImpl extends ITest.Stub{

public void getTest(complicatedDataStructurecds) throws RemoteException {

//在这里实现具体的getTest

}

public voidsetTest(in String name,in boolean reStartServer)

throwsRemoteException

{

//在这里实现具体的setTest

}

}

这时，你的Eclipse下会有如下两个目录：

· src下有一个com.test.service包结构目录。

· gen下也有一个com.test.service包结构目录，其中的内容是由aidl工具生成的。

3. 实现代理端

代理端往往在另外一个程序中使用。假设是com.test.client包，把刚才com.test.service工程的gen下com.test.service目录全部复制到com.test.client中。这样，client工程也有两个包结构目录：

· com.test.client。

· com.test.service。不过这个目录仅仅只有aidl生成的Java文件。

服务端一般驻留于Service进程，所以可以在Client端的onServiceConnected函数中获得代理对象，实现代码如下所示：

[-->Client端示例代码]

//不一定是在onServiceConnected中，但它是比较合适的。

private ServiceConnection serviceConnection = newServiceConnection() {

//@Override

public void onServiceConnected(ComponentName name, IBinder service) {

if(ITestProxy== null)

ITestProxy = ITest.Stub.asInterface(service);//这样你就得到BpTest了

}

4. 传递复杂的数据结构

AIDL支持简单数据结构与Java 中String类型的数据进行跨进程传递，如果想做到跨进程传递复杂的数据结构，还须另做一些工作。

以ITest.aidl文件中使用的complicatedDataStructure为例：

· 它必须实现implements Parcelable接口。

· 内部必须有一个静态的CREATOR类。

· 定义一个complicatedDataStructure.aidl文件

说明：可参考Android API文档的parcelable类，里边有一个很简单的例子。

来看这个Java文件的实现：

[-->complicatedDataStructure.java]

package com.test.service;

import android.os.Parcel;

import android.os.Parcelable;

public class complicatedDataStructure implementsParcelable {

public static final int foo1 = 0;

public static final int foo2 = 1;

public String fooString1 = null;

public String fooString2 = null;

//静态Creator类

public static final Parcelable.Creator< complicatedDataStructure> CREATOR =

new Parcelable.Creator< complicatedDataStructure >() {

public complicatedDataStructure createFromParcel(Parcel in)

{

return new complicatedDataStructure (in);

}

public complicatedDataStructure [] newArray(int size)

{

return new complicatedDataStructure [size];//用于传递数组

}

};

public complicatedDataStructure(complicatedDataStructureother) {

fooString1= other. fooString1;

fooString2= other. fooString2;

foo1= other. foo1;

foo2 =other. foo2;

}

private complicatedDataStructure(Parcel in) {

readFromParcel(in);

}

/\*@Override \*/

publicint describeContents() {

return 0;

}

public void readFromParcel(Parcel in) {

foo1= in. readInt ();

foo2= in. readInt ();

fooString1= in.readString();

fooString2= in.readString();

}

/\*@Override \*/

publicvoid writeToParcel(Parcel dest, int flags) {

dest.writeInt(foo1);

dest.writeInt(foo2);

dest.writeString(fooString1);

dest.writeString(fooString2);

}

}

complicatedDataStructure.aidl该怎么写呢？如下所示：

[-->complicatedDataStructure.aidl]

package com.test.service;

parcelable complicatedDataStructure;

然后，在使用它的aidl文件中添加下行代码即可：

import com.test.complicatedDataStructure

有了AIDL，再看我们的阿斗是不是能扶得起了呢？当然，想让上面的程序正确工作，还得再努把力，把未尽的业务层事业完成。另外，还要经得起残酷环境的考验（即通过测试来检验自己的程序）。

6.7 本章小结

本章对Android平台中最为重要的IPC通信机制Binder进行了较为全面、详细的分析。我们首先以MediaServer为切入点，讲述了Android是如何通过层层封装将Binder机制集成到应用程序中的，然后对服务总管ServiceManager进行了介绍，最后分析了MediaPlayerService是如何通过Binder处理来自客户端的请求的。

在拓展思考部分，我们重点讨论了和Binder有关的三个问题。要弄清这三个问题，都需要从Binder驱动的实现中寻找答案。

由于Binder的重要性，本章还专门增加了“学以致用”一节，旨在研讨怎么写纯Native的Service，如何通过编写AIDL文件来实现Java中的Service。