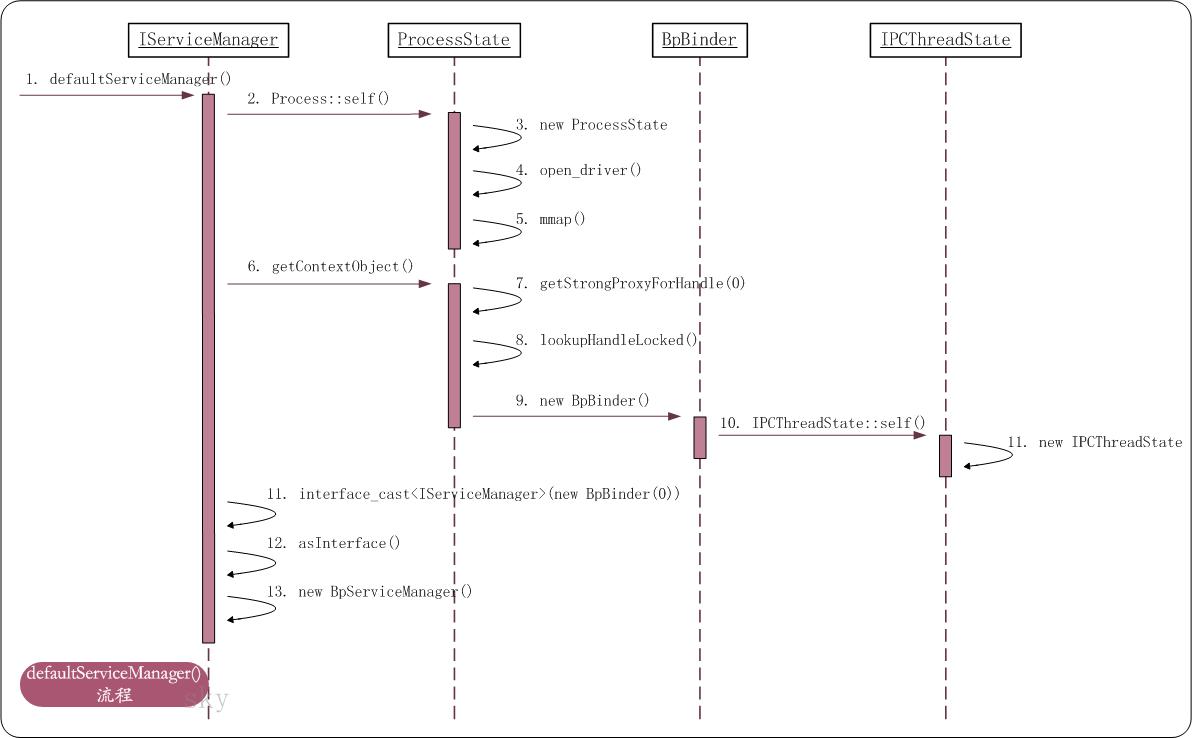
**Bp-通过ServiceManager分析binder机制**

**1、BpServiceManager与defaultServiceManager**

Code路径：frameworks/native/libs/binder/ IServiceManager.cpp

frameworks/native/include/binder/ IServiceManager.h

defaultServiceManager这个函数新建并返回BpServiceManager——ServiceManager的代理。



defaultServiceManager流程图

**2、defaultServiceManager**

defaultServiceManager()得到BpServiceManager。

**2.1、ProcessState**

Code路径：frameworks/native/libs/binder/ProcessState.cpp

**2.1.1、ProcessState::self()**

这里会新建一个全局变量gProcess（每个server进程只有一个），ProcessState是个单例模式，ProcessState类构造函数完成的工作如下：

1、mDriverFD(open\_driver())，里面是fd = open("/dev/binder", O\_RDWR)

2、mmap(0, BINDER\_VM\_SIZE, PROT\_READ, MAP\_PRIVATE | MAP\_NORESERVE, mDriverFD, 0)

#define BINDER\_VM\_SIZE ((1\*1024\*1024) - (4096 \*2))

参照《1-Bn-通过ServiceManager分析binder机制-ServiceManager服务启动》可知，这里会在binder驱动中新建binder\_proc对象，保存该进程的上下文环境，将该proc保存到"全局哈希表binder\_procs"中。并且完成用户空间和内核空间到物理内存页的映射，大小是((1\*1024\*1024) - (4096 \*2))。

注意：在open\_driver中有下面的配置，把线程数最大值设为15。到binder驱动中赋值给binder\_proc->max\_threads，进入binder ioctl，binder\_get\_thread查找当前进程在驱动中的thread信息，第一次进来查找不到，新建一个binder\_thread thread，里面会保存当前进程信息，说明此thread属于这个proc。

size\_t maxThreads = 15;

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

**2.1.2、ProcessState::self()->getContextObject(NULL)**

ProcessState::self()->getContextObject(NULL)到getStrongProxyForHandle(0)，如果是第一次进来会直接new BpBinder(0)，并把她保存在容器mHandleToObject中，下次再进来不需要新建，直接从容器mHandleToObject中获取。

**2.2、interface\_cast**

Code路径：frameworks/native/include/binder/ IInterface.h

frameworks/native/libs/binder/ IInterface.cpp

interface\_cast<IServiceManager>(ProcessState::self()->getContextObject(NULL))相当于interface\_cast<IServiceManager>(new BpBinder(0))，看到interface\_cast想到static\_cast\dynamic\_cast\const\_cast\reinpreter\_cast，介绍下这几个类型转换符：static\_cast不检查类型安全，dynamic\_cast具有类型检查的功能（前提是类里必须有虚函数），牺牲了效率，但比static\_cast安全，const\_cast可去除引用或指针的常量属性。所以interface\_cast从名字上看应该是类型转换的意思，来看看它的定义，是一个模板类，如下：

把IBinder转为IInterface接口，类型转换过程：

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

{

return INTERFACE::asInterface(obj);

}

相当于调用IServiceManager::asInterface(new BpBinder(0))

**2.3、class IServiceManager : public IInterface**

通过宏DECLARE\_META\_INTERFACE定义asInterface接口，用宏IMPLEMENT\_META\_INTERFACE实现

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;

}

BpBinder没有覆盖IBinder中方法queryLocalInterface，基类IBinder queryLocalInterface返回的是NULL，所以asInterface走new Bp##INTERFACE(obj)，生成BpServiceManager(new BpBinder(0))。用IServiceManager的asInterface方法创建的BpServiceManager类又继承了IServiceManager，在基类方法中构建自己的派生类并返回，最终defaultServiceManager()得到的是BpServiceManager(new BpBinder(0))。

* 1. **BpServiceManager**

**2.4.1、类BpServiceManager : public BpInterface<IServiceManager>**

构造函数BpServiceManager(const sp<IBinder>& impl): BpInterface<IServiceManager>(impl)，传入IBinder参数（new BpBinder(0)）给基类

**2.4.2、BpInterface是一个模板类，与BnInterface对应**

template<typename INTERFACE> class BpInterface : public INTERFACE, public BpRefBase，这里的INTERFACE为IServiceManager，BpInterface这个类继承了IInterface和BpRefBase，并没有继承IBinder，只是在BpRefBase的数据成员中保存了一个引用IBinder（BpBinder(0)）。

a、方法onAsBinder覆盖IInterface中的onAsBinder，BpInterface::onAsBinder返回的是remote()，是一个BpBinder。而IInterface的asBinder方法直接调用的是onAsBinder，所以如果BpInterface及其派生类通过asBinder得到ibinder是remote()

b、构造函数BpInterface(const sp<IBinder>& remote)，传一个IBinder的参数给基类BpRefBase，保存在BpRefBase的私有成员mRemote中，remote()返回的就是mRemote，这里这个值是new BpBinder(0)

**2.4.3、BpRefBase**

功能：方法remote()。返回mRemote，这里为上面保存的new BpBinder(0)。

**2.4.4、class IServiceManager : public IInterface**

DECLARE\_META\_INTERFACE和IMPLEMENT\_META\_INTERFACE，用宏的形式定义和实现接口asInterface，重要方法getService和addService，都是通过remote()->transact（其实是BpBinder->transact）把命令发送出去，这里是同步的，进程会陷入内核态，等待调用返回，即，Bn端内核binder驱动的返回；**注意：BnServiceManager继承了IServiceManager，但是并没有实现IServiceManager的纯虚函数，所以BnServiceManager也是一个纯虚类，无法被实例化，BnServiceManager::onTransact收到命令后把数据写在reply中，然后呢？BnServiceManager并没有被用到？真实的BnServiceManager其实是service\_manager.c。**从transact发出的命令经过binder驱动最终到达ServiceManager进程。其它章节会详细分析BpBinder、BBinder、IBinder以及binder驱动。

**3、BpBinder**

**3.1、构造函数**

frameworks/native/libs/binder/ BpBinder.cpp

1、保存handle值到mHandle

2、extendObjectLifetime(OBJECT\_LIFETIME\_WEAK);对象生命周期设置，弱引用为0后释放。

3、新建对象IPCThreadState

**3.2、IPCThreadState**

IPCThreadState::self，这里涉及到一个线程私有数据的知识，线程局部存储。简单分析下：

pthread\_key\_create(&gTLS, threadDestructor) != 0；函数pthread\_key\_create()为线程局部数据创建一个新键缓冲区，并通过key：gTLS指向新创建的键缓冲区；destructor所指向的是一个自定义的函数，只要线程终止时与key关联的值不为NULL，则destructor所指的函数将会自动被调用。

pthread\_getspecific，获取gTLS指向的键缓冲区值

IPCThreadState::self从gTLS中去获取存起来的IPCThreadState，如果没有会新建一个IPCThreadState。

构造函数：

1、保存gProcess（ProcessState）到mProcess

2、pthread\_setspecific(gTLS, this)，保存自己到线程的键缓冲区。

3、mIn.setDataCapacity(256)、mOut.setDataCapacity(256)

设置mIn和mOut的容量为256字节。IPCThreadState是和Binder驱动交互的类，mOut是用来保存"IPCThreadState需要发送给Binder驱动的内容的"，而mIn则是用来保存"Binder驱动反馈给IPCThreadState的内容的"。

**4、延伸**

**BpBinder(0)的作用？以及怎样关联到内核空间的Binder引用？参考《3-通过MediaPlayerService分析binder机制-addService》**

**其实内核Binder驱动只根据BpBinder中的mHandle（0）去寻找Binder实体，0对应的刚好是全局变量binder\_context\_mgr\_node，这个全局变量就是ServiceManager对应的内核中的Binder实体binder node对象了。**

**调用defaultServiceManager函数的线程，就会有一个自己的ProcessState和IPCThreadState对象，并且驱动中会新建一个binder proc结构体对象，但是并没有binder实体和引用被创建，所以proc中的红黑树都是空的，实体和引用都是在addService、getService时被创建的。**

**疑点：BnServiceManager到底有什么用？然并卵。真实的“BnServiceManager”是service\_manager.c。**