Binder系列3—获取Service Manager

Nov 8, 2015

- 源码分析
  - [1] defaultServiceManager
  - [2] ProcessState::self()
  - [3] new ProcessState
  - [4] open_driver()
  - [8] getContextObject
  - [9] getContextObject
  - [10]. lookupHandleLocked
  - [11]. new BpBinder()
  - [13]. interface_cast<IServiceManager>()
  - [14]. IServiceManager::asInterface()
  - [15]. new BpServiceManager
- 小结

> 基于Android 6.0的源码剖析， 本文详细地讲解了如何获取Service Manager(defaultServiceManager)

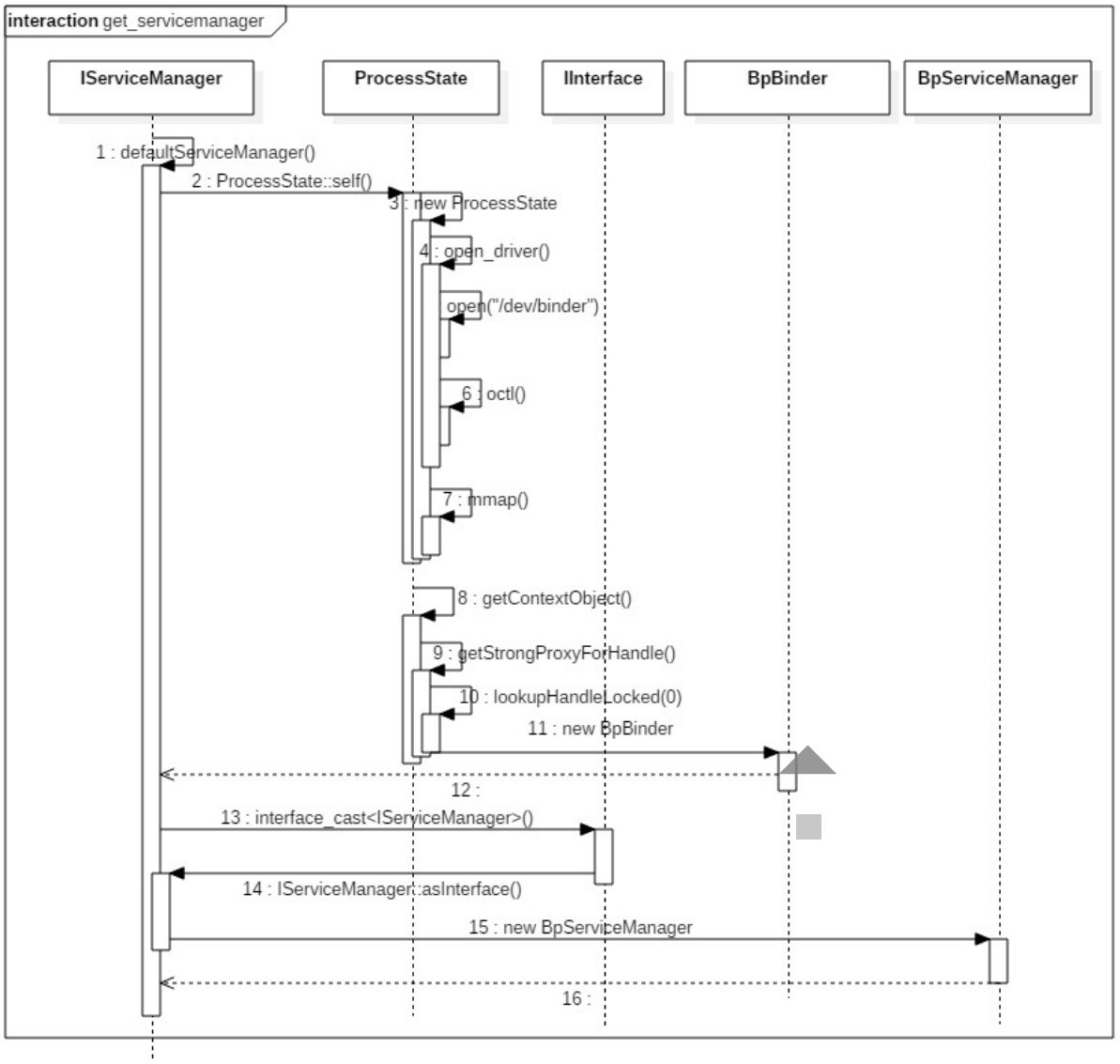
源码分析

相关源码

```
/framework/native/libs/binder/IServiceManager.cpp
/framework/native/libs/binder/ProcessState.cpp
/framework/native/libs/binder/BpBinder.cpp
/framework/native/libs/binder/Binder.cpp

/framework/native/include/binder/IServiceManager.h
/framework/native/include/binder/IInterface.h
```

流程图



下面开始讲解每一个流程：

[1] defaultServiceManager

==> /framework/native/libs/binder/IServiceManager.cpp

获取默认ServiceManager对象。

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
    {
        AutoMutex _l(gDefaultServiceManagerLock); //加锁
        while (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));  //【见流程2和13】
            if (gDefaultServiceManager == NULL)
                sleep(1);   //休眠1秒
        }
    }

    return gDefaultServiceManager;
}
```

这是**单例模式**，我们发现与一般的单例模式不太一样，里面多了一层while循环，这是google在2013年1月Todd Poynor提交的修改。defaultServiceManager需要等待service manager就绪。当我们尝试创建一个本地的代理时，如果service manager没有准备好，那么就会失败，这时sleep 1秒后会重新尝试获取，直到成功。

[2] ProcessState::self()

==> /framework/native/libs/binder/ProcessState.cpp

获得ProcessState对象

```
sp<ProcessState> ProcessState::self()
{
    Mutex::Autolock _l(gProcessMutex);
    if (gProcess != NULL) {
        return gProcess;
    }
    gProcess = new ProcessState;  //实例化ProcessState 【见流程3】
    return gProcess;
}
```

这也是**单例模式**，，从而保证每一个进程只有一个 ProcessState 对象。其中 gProcess 和 gProcessMutex 是保存在 Static.cpp 类的全局变量。

[3] new ProcessState

==> /framework/native/libs/binder/ProcessState.cpp

初始化ProcessState对象

```
ProcessState::ProcessState()
    : mDriverFD(open_driver()) // 打开Binder驱动【见流程4】
    , mVMStart(MAP_FAILED)
    , mThreadCountLock(PTHREAD_MUTEX_INITIALIZER)     // [Android 6.0新增]
    , mThreadCountDecrement(PTHREAD_COND_INITIALIZER) // [Android 6.0新增]
    , mExecutingThreadsCount(0)                       // [Android 6.0新增]
    , mMaxThreads(DEFAULT_MAX_BINDER_THREADS)         // [Android 6.0新增]
    , mManagesContexts(false)
    , mBinderContextCheckFunc(NULL)
    , mBinderContextUserData(NULL)
    , mThreadPoolStarted(false)
    , mThreadPoolSeq(1)
{
    if (mDriverFD >= 0) {
        //采用内存映射函数mmap，给binder分配一块虚拟地址空间,用来接收事务
        mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NO
RESERVE, mDriverFD, 0);
        if (mVMStart == MAP_FAILED) {
            close(mDriverFD); //没有足够空间分配给/dev/binder, 关闭设备
            mDriverFD = -1;
        }
    }
}
```

- `ProcessState` 的单例模式的惟一性，因此一个进程只打开binder设备一次,其中ProcessState的成员变量 `mDriverFD` 记录binder驱动的fd，用于访问binder设备。
- `BIDNER_VM_SIZE = (1*1024*1024) - (4096 *2)`，binder分配的默认内存大小为1M-8k。
- `DEFAULT_MAX_BINDER_THREADS = 15` ，binder默认的最大可并发访问的线程数为15。

[4] open_driver()

`==> /framework/native/libs/binder/ProcessState.cpp`

打开Binder驱动设备

```
static int open_driver()
{
    int fd = open("/dev/binder", O_RDWR); //打开/dev/binder设备，建立与内核的Binder驱动的交互通道
    if (fd >= 0) {
        fcntl(fd, F_SETFD, FD_CLOEXEC);
        int vers = 0;
        status_t result = ioctl(fd, BINDER_VERSION, &vers);
        if (result == -1) {
            close(fd);
            fd = -1;
        }
        if (result != 0 || vers != BINDER_CURRENT_PROTOCOL_VERSION) {
            close(fd);
            fd = -1;
        }
        size_t maxThreads = DEFAULT_MAX_BINDER_THREADS;
        //通过ioctl设置binder驱动，能支持的最大线程数
        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;
}
```

open_driver作用是打开/dev/binder设备，binder支持的最大线程数默认是15。关于binder驱动操作，详细见Binder系列1 —— Binder驱动 (http://www.yuanhh.com/2015/11/01/binder-driver/)

[8] getContextObject

==> /framework/native/libs/binder/ProcessState.cpp

获取handle=0的IBinder

```
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& /*caller*/)
{
    return getStrongProxyForHandle(0);  //【见流程9】
}
```

[9] getContextObject

==> /framework/native/libs/binder/ProcessState.cpp

获取IBinder

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;

    AutoMutex _l(mLock);

    handle_entry* e = lookupHandleLocked(handle); //查找handle对应的资源项【见流程10】

    if (e != NULL) {
        IBinder* b = e->binder;
        if (b == NULL || !e->refs->attemptIncWeak(this)) {
            if (handle == 0) {
                Parcel data;
                status_t status = IPCThreadState::self()->transact(
                        0, IBinder::PING_TRANSACTION, data, NULL, 0); //通过ping操作测试binder是否准备就绪
                if (status == DEAD_OBJECT)
                   return NULL;
            }
                        //当handle值所对应的IBinder不存在或弱引用无效时，则新建BpBinder【见流程11】
            b = new BpBinder(handle);
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
            result = b;
        } else {
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }
    return result;
}
```

当handle值所对应的IBinder不存在或弱引用无效时会创建BpBinder，否则直接获取。针对handle==0的特殊情况，通过PING_TRANSACTION来判断是否准备就绪。如果在context manager还未生效前，一个BpBinder的本地引用就已经被创建，那么驱动将无法提供context manager的引用。

[10]. lookupHandleLocked

==> /framework/native/libs/binder/ProcessState.cpp

根据IBinder来查找对应的IBinder

```
ProcessState::handle_entry* ProcessState::lookupHandleLocked(int32_t handl
e)
{
    const size_t N=mHandleToObject.size();
    if (N <= (size_t)handle) {
        handle_entry e;
        e.binder = NULL;
        e.refs = NULL;
        status_t err = mHandleToObject.insertAt(e, N, handle+1-N);
        if (err < NO_ERROR) return NULL;
    }
    return &mHandleToObject.editItemAt(handle);
}
```

根据handle值来查找对应的 handle_entry , handle_entry 是一个结构体，里面记录IBinder和weakref_type两个指针。当在 hanlde_entry 没有找到跟handle值相对应的IBinder，或存在的弱引用无法获取时，需要创建一个新的 BpBinder 。

[11]. new BpBinder()

==> /framework/native/libs/binder/BpBinder.cpp

创建BpBinder对象

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle)
    , mAlive(1)
    , mObitsSent(0)
    , mObituaries(NULL)
{
    extendObjectLifetime(OBJECT_LIFETIME_WEAK); //延长对象的生命时间
    IPCThreadState::self()->incWeakHandle(handle); //handle所对应的bindle弱
引用 + 1
}
```

创建BpBinder对象中，会将handle相对应Binder的弱引用增加1.

[13]. interface_cast<IServiceManager>()

==> /framework/native/include/binder/IInterface.h

模板函数

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj); 【见流程14】
}
```

故 interface_cast<IServiceManager>() 等价于 IServiceManager::asInterface() .

[14]. IServiceManager::asInterface()

1. DECLARE_META_INTERFACE(IServiceManager)

==> /framework/native/include/binder/IServiceManager.h

根据 IInterface.h 中的模板函数，展开即可得：

```
static const android::String16 descriptor;

static android::sp< IServiceManager > asInterface(const android::sp<androi
d::IBinder>& obj)

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

IServiceManager ();
virtual ~IServiceManager();
```

2. IMPLEMENT_META_INTERFACE(ServiceManager,” android.os.IServiceManager”)

==> /framework/native/libs/binder/IServiceManager.cpp

根据 IInterface.h 中的模板函数，展开即可得：

```
const android::String16 IServiceManager::descriptor(“android.os.IServiceMa
nager”);

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

 android::sp<IServiceManager> IServiceManager::asInterface(const androi
d::sp<android::IBinder>& obj)
{
       android::sp<IServiceManager> intr;
        if(obj != NULL) {
           intr = static_cast<IServiceManager *>(
               obj->queryLocalInterface(IServiceManager::descriptor).ge
t());
           if (intr == NULL) {
               intr = new BpServiceManager(obj);
            }
        }
       return intr;
}

IServiceManager::IServiceManager () { }
IServiceManager::~ IServiceManager() { }
```

故 `IServiceManager::asInterface()` 等价于 `new BpServiceManager()`。括号内的参数是IBinder，准确说，应该是BpBinder。

[15]. new BpServiceManager

1. 初始化BpServiceManager

`==> /framework/native/libs/binder/IServiceManager.cpp`

```
BpServiceManager(const sp<IBinder>& impl)
    : BpInterface<IServiceManager>(impl)
{
}
```

2. 初始化父类BpInterface

`==> /framework/native/include/binder/IInterface.h`

```
inline BpInterface<INTERFACE>::BpInterface(const sp<IBinder>& remote)
    :BpRefBase(remote)
{
}
```

（3）初始化父类BpRefBase

`==> /framework/native/libs/binder/Binder.cpp`

```
BpRefBase::BpRefBase(const sp<IBinder>& o)
    : mRemote(o.get()), mRefs(NULL), mState(0)
{
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);

    if (mRemote) {
        mRemote->incStrong(this);
        mRefs = mRemote->createWeak(this);
    }
}
```

`new BpServiceManager()`，在初始化过程中，比较重要工作的是类BpRefBase的mRemote指向new BpBinder(0)，从而BpServiceManager能够利用Binder进行通过通信。

小结

1. `defaultServiceManager()` 单例模式:
   - 当gDefaultServiceManager存在，直接返回，否则继续；
   - defaultServiceManager 等价于： `sp<IServiceManager> sm = new BpServiceManager(new BpBinder(0));`
2. `ProcessState::self()` 单例模式：
   - 当ProcessState对象存在，则直接返回，否则依次进行下面步骤;

- 打开内核的/dev/binder设备，建立与内核的Binder驱动的交互通道;
- 利用 `mmap` 为Binder驱动映射内存空间;
- 将Binder驱动的fd赋值 `ProcessState` 对象中的变量 `mDriverFD` ，用于交互操作。

3. BpServiceManager巧妙将通信层与业务层逻辑合二为一
   - 通过继承IServiceManager，实现了接口中的业务逻辑函数；
   - 其成员变量mRemote = new BpBinder(0)，通过成员变量进行Binder通信工作。
4. BpBinder通过handler来对应BBinder, 在整个Binder系统中，handle=0代表ServiceManager所对应的BBinder。

---

gityuan@gmail.com (mailto:gityuan@gmail.com) · Github (https://github.com/yuanhuihui)