Earth Guardian

You are not LATE!You are not EARLY!

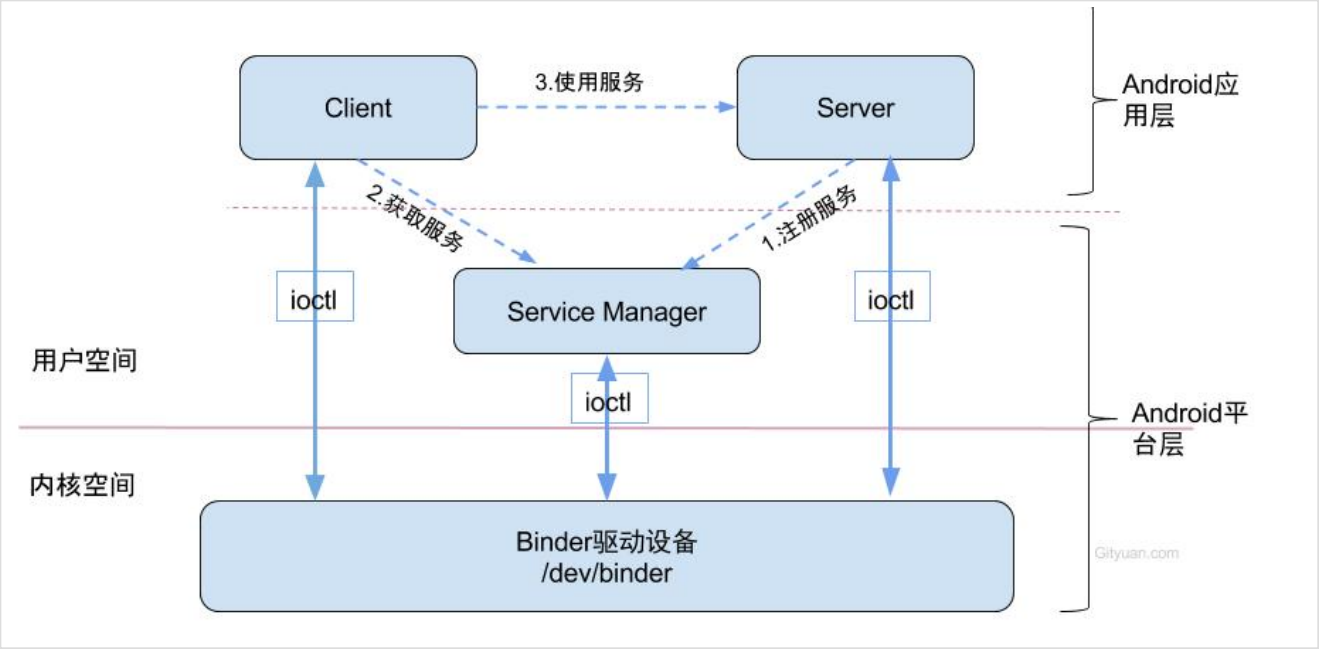
Android Binder 机制

2017-12-21 | Android | 阅读次数

Binder 是 Android 系统进程间通信（IPC）方式之一，它是基于开源的 OpenBinder 实现的。

基础

Binder 架构简图



所谓 Binder 通信机制：进程 A 和进程 B 分别直接和 Binder Driver 交互， Driver 来负责数据转发，从而间接实现进程 A 和进程 B 之间的数据交互。

Binder 架构中通信过程包含四个角色：Client，Server，ServiceManager，Driver 。可以从图中看到 Client，Server，ServiceManager 与 Driver 之间都是实线，他们通过 ioctl 直接交互。而 Client，Server，ServiceManager 分属于三个不同的进程，三者交互都是虚线，它们之间是通过 Binder Driver 串起来实现通信的，也就是使用了 Binder 的通信机制。

- `ServiceManager`

  中间人，对应一个 `service_manager` 开启的守护进程，维护所有服务的列表。

- `Server`

  服务端进程通过 `ServiceManager` 注册服务。服务端进程将服务名和 `IBinder` 写入到 `ServiceManager` 守护进程维护的服务列表中，是一次进程间通信。

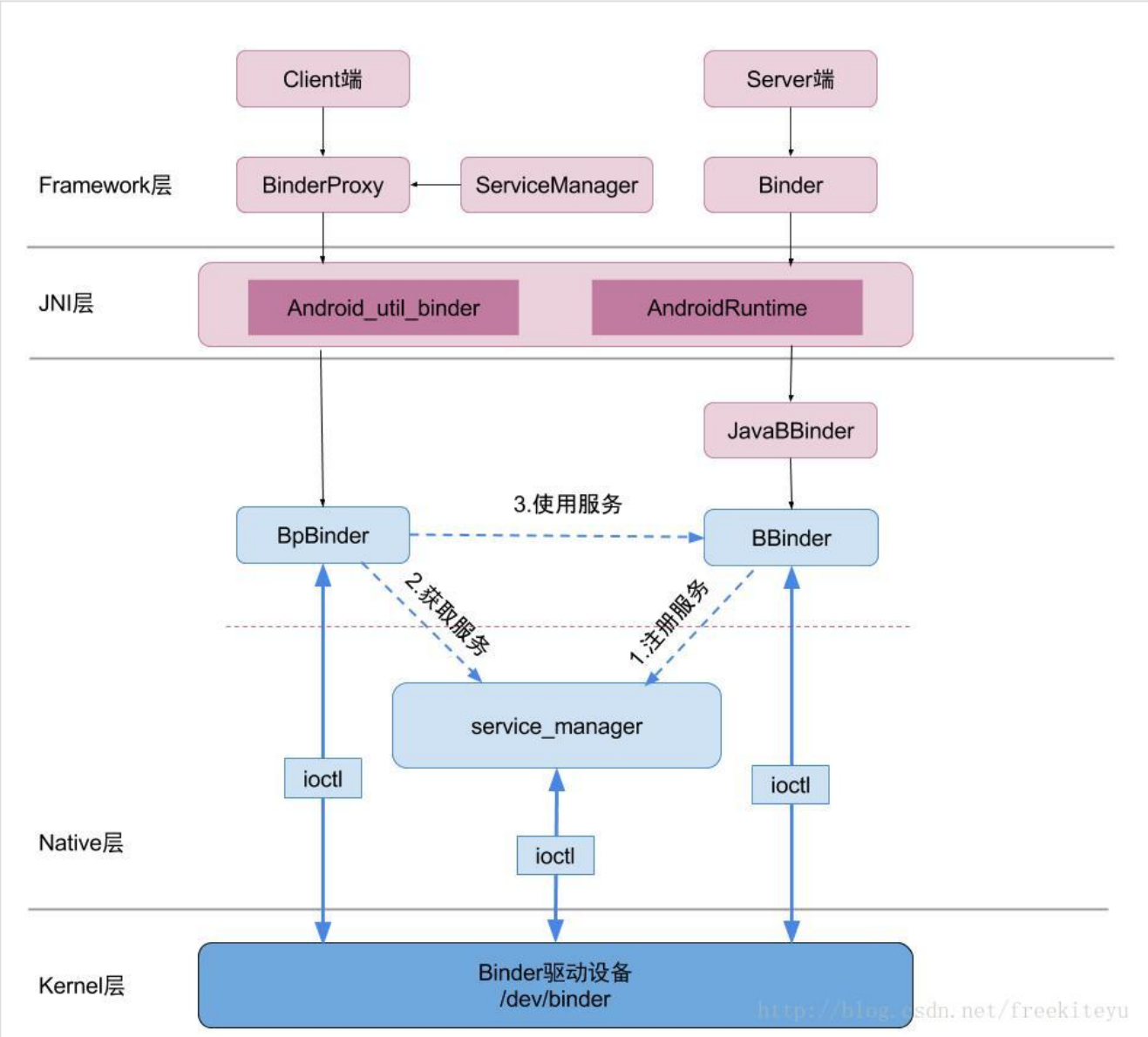
- `Client`

  客户端进程通过 `ServiceManager` 查询服务，即 `Client` 进程和 `ServiceManager` 守护进程间的通信。查询服务拿到服务端 `IBinder` 后，实现 `Client/Server` 进程间通信。

- `Driver`

  实现数据转发。上面三次进程间通信都是 `Binder` 通信机制实现的，也就是三个进程都直接和 `Driver` 交互，来实现数据转发。

`Binder` 详细架构图



- ServiceManager

  具有双重属性，细分为 ServiceManager/service_manager 。 ServiceManager 在和 Client/Server 交互时作为服务端；在和 service_manager 交互时作为客户端，此时 service_manager 守护进程为服务端，维护着一个服务链表。

- Server

  服务端通过 Java 或者 CPP 代码形式都可以注册。不管什么方式，都是通过的 Native 中 BBinder 向 service_manager 守护进程注册的，并写入守护进程维护的服务列表。

- Client

  客户端也可以通过 Java 或者 CPP 代码形式查询服务，最终都是通过 Native 中 BpBinder 从 service_manager 守护进程查询，并返回 IBinder 。

- Driver

  Server，Client，ServiceManager 运行于用户空间， Driver 属于内核空间，他们都是通过 IBinder 来通信的。 Driver 的作用就是用来转发数据：

  A： Client 持有 IBinder ，通过它来实现和 Driver 通信

  B： Server 本身就是 IBinder ，直接和 Driver 通信

  C： service_manager 守护进程可以直接下发 ioctl 和 Driver 通信；这四个角色的关系和互联网类比： Server 是服务器， Client 是客户终端， ServiceManager 是域名服务器（ DNS ），Driver 是路由器。

Binder 机制特点

- service_manager 维护一个链表，用来添加或查询服务
- Binder Driver 实现进程间的数据交互

Android Binder 机制在 Android 系统中江湖地位非常之高。在 Zygote 孵化出 system_server 进程后， system_server 进程会初始化支撑整个 Android Framework 的各种各样的 Service ，另外在 init.rc 中也会启动很多 Service ，这些服务几乎都是基于 Binder IPC 机制。

三个基本概念

- IBinder

  表示拥有被跨进程传输的能力。 IBinder 是远程对象的基本接口，定义了与远程对象交互的协议。 IBinder 是一种传输方式（类比 Socket ），代表 Binder 通信机制，只有它的对象才能通过 Binder 机制跨进程通信；也可以认为 IBinder 是一个数据类型（类比 Object，String 等），它能被写入 Parcel 中。

- IInterface

  定义远程接口，表示服务端拥有什么能力，能提供哪些服务，并提供了转换为 IBinder 的方法。

- Parcel

  是一个缓冲区，除了存储基本数据类型， Parcelable 数据类型等，还可以传递 IBinder 对象。区别于 Java 中 Serializable 可以将数据保存到存储介质上， Parcel 仅存储在内存中，属于轻量级序列化机制。 Binder 通信机制跨进程传输的数据，是存储在 Parcel 中的。

Client/Server 两个角色在使用过程中并不用关心 Binder 的通信过程，这些是 Android 系统已经实现了的。 Client/Server 只需要按照 Binder 机制中规定实现相应的 IBinder/IInterface ，系统将完成这个通信过程。

源码目录结构速查表

整个 Binder 框架目录结构

```
1   frameworks/base/core/java/              (Java)
2   frameworks/base/core/jni/               (JNI)
3   frameworks/native/libs/binder           (Native)
4   frameworks/native/cmds/servicemanager/  (Native)
5   kernel/drivers/staging/android          (Driver)
```

Java Framework

```
1   frameworks/base/core/java/android/os/
2       - IInterface.java
3       - IBinder.java
4       - Parcel.java
5       - IServiceManager.java
6       - ServiceManager.java
7       - ServiceManagerNative.java
8       - Binder.java
9
10
11  frameworks/base/core/jni/
12      - android_os_Parcel.cpp
13      - AndroidRuntime.cpp
14      - android_util_Binder.cpp   (核心类)
```

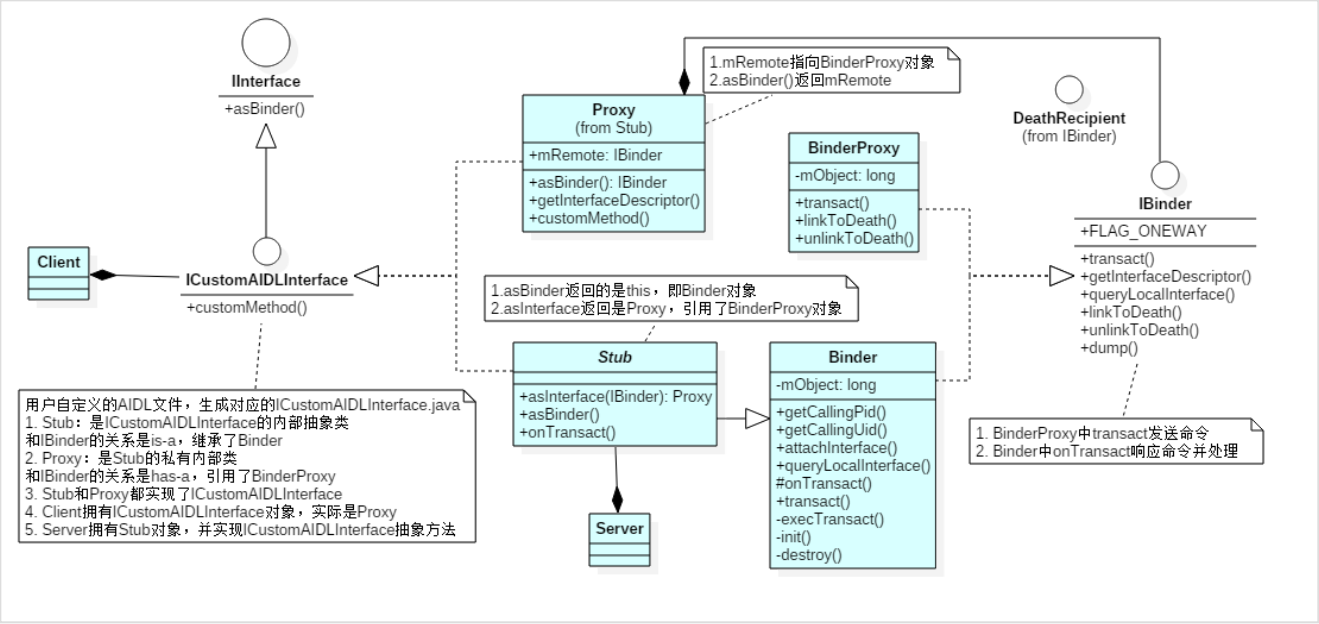
Native Framework

```
frameworks/native/libs/binder
    - IServiceManager.cpp
    - BpBinder.cpp
    - Binder.cpp
    - IPCThreadState.cpp        (核心类)
    - ProcessState.cpp          (核心类)

frameworks/native/include/binder/
    - IServiceManager.h
    - IInterface.h

frameworks/native/cmds/servicemanager/
    - service_manager.c
    - binder.c
```

kernel Driver

```
kernel/drivers/staging/android/
    - binder.c
    - uapi/binder.h
```

Binder Java 类图



通常来讲，对于 Server 进程， Binder 指的是 Binder 本地对象；对于 Client 进程， Binder 指的是 BinderProxy 对象，它只是 Binder 本地对象的一个远程代理。对 BinderProxy 对象的操作，会通过驱动最终转发到 Binder 本地对象上去完成。对于一个拥有 Binder 对象的使用者而言，它无须关心这是一

个 BinderProxy 对象还是 Binder 本地对象，对于代理对象的操作和对本地对象的操作对它来说没有区别。

在驱动中， Binder 本地对象的代表是一个叫做 binder_node 的数据结构， BinderProxy 对象是用 binder_ref 代表的。有的地方把 Binder 本地对象直接称作 Binder 实体，把 BinderProxy 对象直接称作 Binder 引用（句柄），其实指的是 Binder 对象在驱动里面的表现形式。

类和接口对应文件

```
IInterface.java: IInterface
IBinder.java: IBinder, DeathRecipient
Binder.java: Binder, BinderProxy
BinderInternal.java: BinderInternal, GcWatcher
ICustomAIDLInterface.java(AIDL 自动生成): ICustomAIDLInterface, Stub, Proxy
```

BinderProxy

Client 持有，也可以理解为远程端，下发命令。

- transact 发送命令
- mObject 保存 BpBinder 的引用

Binder

Server 持有，也可以理解为本地服务端，响应命令。

- Binder.onTransact 响应命令并处理
- mObject 保存 JavaBBinderHolder 的引用

AIDL 文件

自定义的 AIDL 文件，编译时会**自动生成**对应的 ICustomAIDLInterface.java 文件，这个文件包含 ICustomAIDLInterface，Stub，Proxy 三个类或接口。

- Stub 和 Proxy 都实现了 ICustomAIDLInterface
- Client 拥有 ICustomAIDLInterface 对象，实际是 Proxy
- Server 拥有 Stub 对象，并实现 CustomAIDLInterface 抽象方法

使用模板：

```
// Client
ICustomAIDLInterface mAIDLService =
        ICustomAIDLInterface.Stub.asInterface(IBinder);

// Server
private ICustomAIDLInterface.Stub mBinder =
        new ICustomAIDLInterface.Stub() {
    @Override
    public void customMethod() {...}
```

没有 ServiceManager 的参与，是因为所有的动作都是在 ActivityManager 中实现的。而 ActivityManager 完成了服务的注册和查询，它也是一次 Binder 通信。

Stub

客户辅助对象，常译为“桩”。它是 ICustomAIDLInterface 的内部抽象类。

- 和 IBinder 的关系是 is-a，继承了 Binder
- asBinder 返回的是 this，即 Binder 对象
- asInterface 返回是 Proxy，引用了 BinderProxy 对象

> Stub 同时实现了 IInterface 和 IBinder，所以它是整个 Binder Java 框架的中转站，通过 asInterface/asBinder 转换为需要的接口。

Proxy

Stub 的私有内部类。

- 和 IBinder 的关系是 has-a，引用了 BinderProxy 对象
- mRemote 指向 BinderProxy 对象
- asBinder() 返回 mRemote

IInterface

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

该接口只包含一个方法： asBinder ，即将 IInterface 转换为 IBinder 。客户端只有 IInterface 实例，需要转换为 IBinder 后才能跨进程通信。

IBinder

- FLAG_ONEWAY

  Binder 机制中客户端和服务端通信默认是阻塞式的，但如果设置了 FLAG_ONEWAY ，将成为非阻塞的调用，客户端能立即返回，服务端采用回调方式来通知客户端完成情况。

- DeathDecipient

  死亡通知，是一个回调接口。当进程不再持有 IBinder 时，会通过这个回调来通知。 IBinder 通过 linkToDeath/unlinkToDeath 来绑定和解绑。

BinderInternal

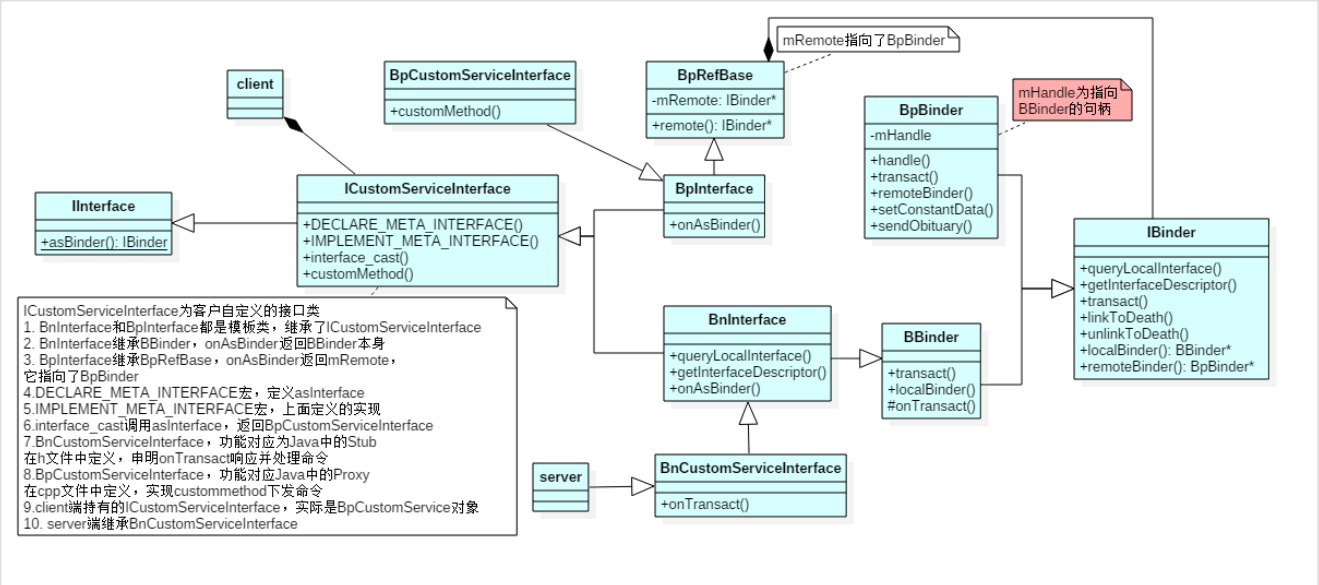
- getContextObject

  静态方法： BinderInternal.getContextObject() ，返回 IBinder ，**专供** ServiceManager 拿到 IServiceManager 的引用。

- GcWatcher

  内部类，用于处理和调试与 Binder 相关的垃圾回收。

Binder CPP 类图



Client 端持有 ICustomServiceInterface ，实际对应的是 BpCustomServiceInterface ，它会通过 BpRefBase.mRemote 指向的 BpBinder 拿到 mHandle ，而这个 mHandle 句柄指向 BBinder 。也就是

mHandle 将 Client 端和 Server 端连接起来。

Server 端继承 BnCustomServiceInterface，它继承 BBinder，直接注册服务。

类和接口对应文件

```
IBinder.h: IBinder
BpBinder.h/BpBinder.cpp: BpBinder, ObjectManager,
Binder.h/Binder.cpp: BBinder, BpRefBase
IInterface.h/IInterface.cpp: IInterface, BnInterface, BpInterface
android_util_Binder.cpp: JavaBBinder, JavaBBinderHolder, JavaDeathRecipient
```

命名规则

- Bp***

  Binder proxy 表示代理，是客户端持有的一个代理对象。

- Bn***

  Binder native 与 Bp 相对表示本地，是本地对象。**但 BBinder 是一个特例**，有点命名混乱的感觉。

BpBinder

- transact

  客户端持有后， BpBinder.transact() 用于发送命令。

- mHandle

  在构造函数中初始化，表示连接的 BBinder 的句柄（句柄：操作系统在进程的地址空间会存储一张句柄表，每个编号内都存储一个地址，这个地址指向实际的对象，而句柄就是这个编号。这么做系统不用暴露对象的实际地址给其他进程，可以认为句柄为指针的指针，但是句柄只能由系统来解析）。所以 mHandle 是 Driver 生成的，仅在 Driver 中有用。 Binder Driver 转发数据时，通过它能找到 BpBinder/BBinder 对象。

- BpBinder* remoteBinder();

  实现该方法，返回 this 。

BBinder

- onTransact

  本地服务端， BBinder.onTransact() 用于响应命令并处理。

- BBinder* localBinder();
  实现该方法，返回 this 。

IBinder 通过 remoteBinder/localBinder 来区分具体是代理还是实体实例。

BpRefBase

mRemote 指针指向 IBinder ，具体是 BpBinder 对象。

IInterface 及重要的宏

- asBinder
  返回 IBinder 的强指针。
- BpInterface
  模板类，同时继承了 ICustomServiceInterface 和 BpRefBase 。 onAsBinder 返回 mRemote ，它指向了 BpBinder 。
- BnInterface
  模板类，同时继承了 ICustomServiceInterface 和 BBinder 。 onAsBinder 返回 this ，即 BBinder 本身。
- DECLARE_META_INTERFACE 宏
  定义了 asInterface 和 getInterfaceDescriptor ，以及构造和析构函数，在 ICustomServiceInterface.h 文件中调用。
- IMPLEMENT_META_INTERFACE 宏
  实现了 asInterface 和 getInterfaceDescriptor ，以及构造和析构函数，在 ICustomServiceInterface.cpp 文件中调用。
- asInterface
  即上面两个宏实现的函数，将 IBinder(BpBinder) 转换为 BpCustomServiceInterface 。
- interface_cast
  模板函数，调用了上面宏定义的 asInterface ，即将 IBinder 转换为 ICustomServiceInterface 。

ICustomServiceInterface

客户自定义的接口类，继承了 IInterface 。**注意**：需要在 .h 文件中调用宏 DECLARE_META_INTERFACE ，在 cpp 文件中调用宏 IMPLEMENT_META_INTERFACE ，实现 asInterface 函数。

- BpCustomServiceInterface

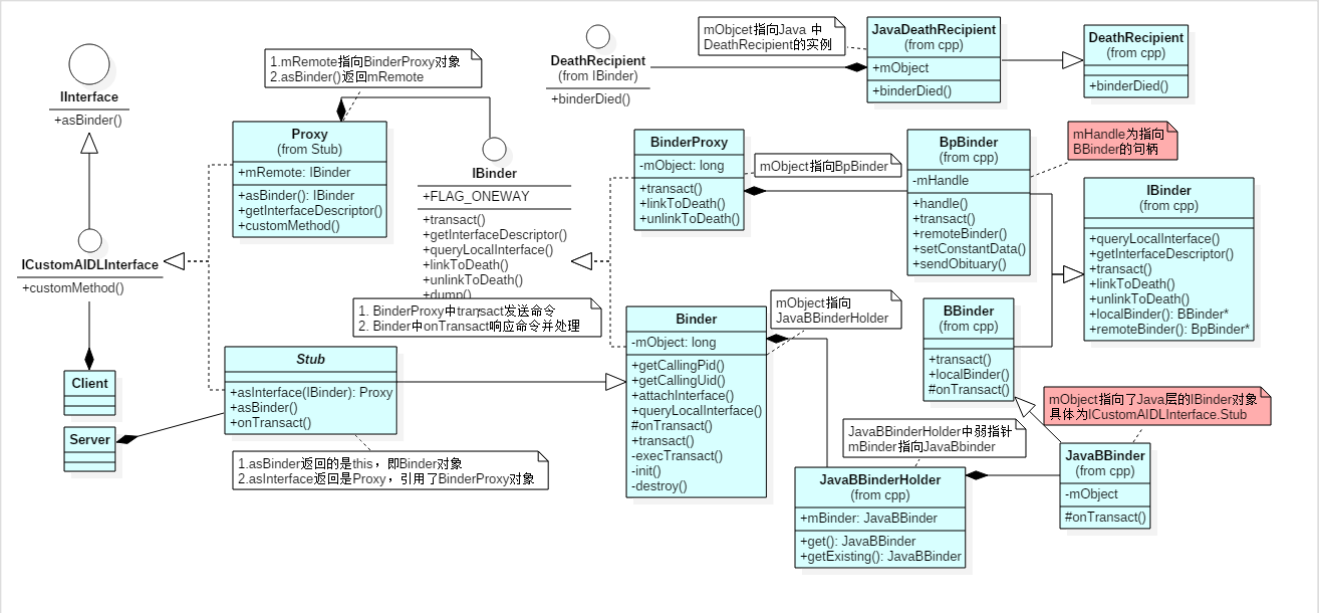
  BpInterface 的具体实现，功能对应 Java 中的 Proxy，在 cpp 文件中定义，实现接口文件中的具体方法，通过 BpBinder.transact 下发命令。

- BnCustomServiceInterface

  BnInterface 的具体实现，功能对应 Java 中的 Stub，在 h 文件中定义，申明 onTransact 响应并处理命令。

> BnCustomServiceInterface 同时继承了 IInterface 和 IBinder，同理它是 Binder CPP 的中转站，通过 onAsBinder/asInterface 来转换。

Binder Java/CPP 转换对应类图



Binder Java 最终都会转换为 Binder CPP 来实现整个 Binder 通信机制。

JavaBBinderHolder

用来管理 JavaBBinder 的实例，使用弱指针指向了该实例。

JavaBinder

- mObject

  保存了服务端注册服务时的 IBinder 引用，也就是说实际指向的是 ICustomAIDLInterface.Stub（可以查看 server_init 序列图）。

android_util_Binder.cpp

Java/CPP 衔接文件：android_util_Binder.cpp: JavaBBinder, JavaBBinderHolder, JavaDeathRecipient，仅在通过 Java 代码注册服务时才会使用到。

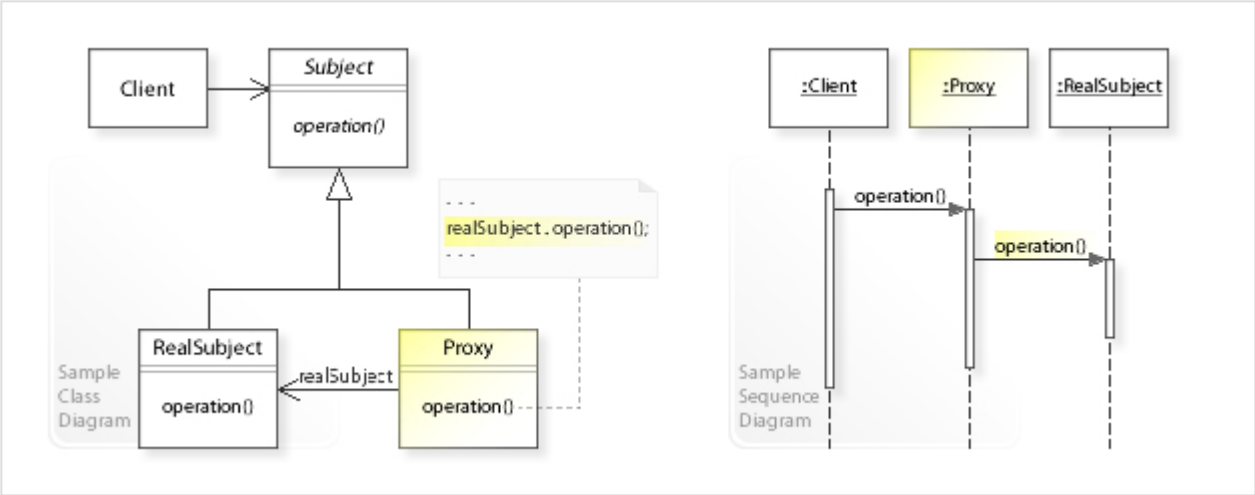
Android Runtime 在开启时，注册了 REG_JNI(register_android_os_Binder)，而 android_util_Binder::register_android_os_Binder 实现了对 Binder Java/CPP 的关联，即对相关 mObject 赋值，以及 Java native 代码的映射。

```
// android_util_Binder.cpp
int register_android_os_Binder(JNIEnv* env)
{
    if (int_register_android_os_Binder(env) < 0)
        return -1;
    if (int_register_android_os_BinderInternal(env) < 0)
        return -1;
    if (int_register_android_os_BinderProxy(env) < 0)
        return -1;
    ...
}

// BinderProxy.mObject 赋值
jobject javaObjectForIBinder(JNIEnv* env, const sp<IBinder>& val)
{
    ...
    object = env->NewObject(gBinderProxyOffsets.mClass, gBinderProxyOffsets.mConstruct
    if (object != NULL) {
        LOGDEATH("objectForBinder %p: created new proxy %p !\n", val.get(), object);
        // The proxy holds a reference to the native object.
        env->SetLongField(object, gBinderProxyOffsets.mObject, (jlong)val.get());
        val->incStrong((void*)javaObjectForIBinder);

        // The native object needs to hold a weak reference back to the
        // proxy, so we can retrieve the same proxy if it is still active.
        jobject refObject = env->NewGlobalRef(
                env->GetObjectField(object, gBinderProxyOffsets.mSelf));
        val->attachObject(&gBinderProxyOffsets, refObject,
                jnienv_to_javavm(env), proxy_cleanup);

        // Also remember the death recipients registered on this proxy
        sp<DeathRecipientList> drl = new DeathRecipientList;
        drl->incStrong((void*)javaObjectForIBinder);
        env->SetLongField(object, gBinderProxyOffsets.mOrgue, reinterpret_cast<jlong>(
        ...
    }

    return object;
```

```
39  }
40
41  // Binder.mObject 指向 JavaBBinderHolder
42  static void android_os_Binder_init(JNIEnv* env, jobject obj)
43  {
44      JavaBBinderHolder* jbh = new JavaBBinderHolder();
45      ...
46      env->SetLongField(obj, gBinderOffsets.mObject, (jlong)jbh);
47  }
```

Binder 机制中的设计模式

代理模式



先看代理模式的定义：**为其他对象提供一种代理以控制对这个对象的访问。**代理模式中，代理和被代理对象继承相同的接口，实现相同的方法。控制被代理对象的访问权限或者隐藏被代理对象的远程操作等等。代理模式类似经纪人角色，可以起到保护明星的功能。

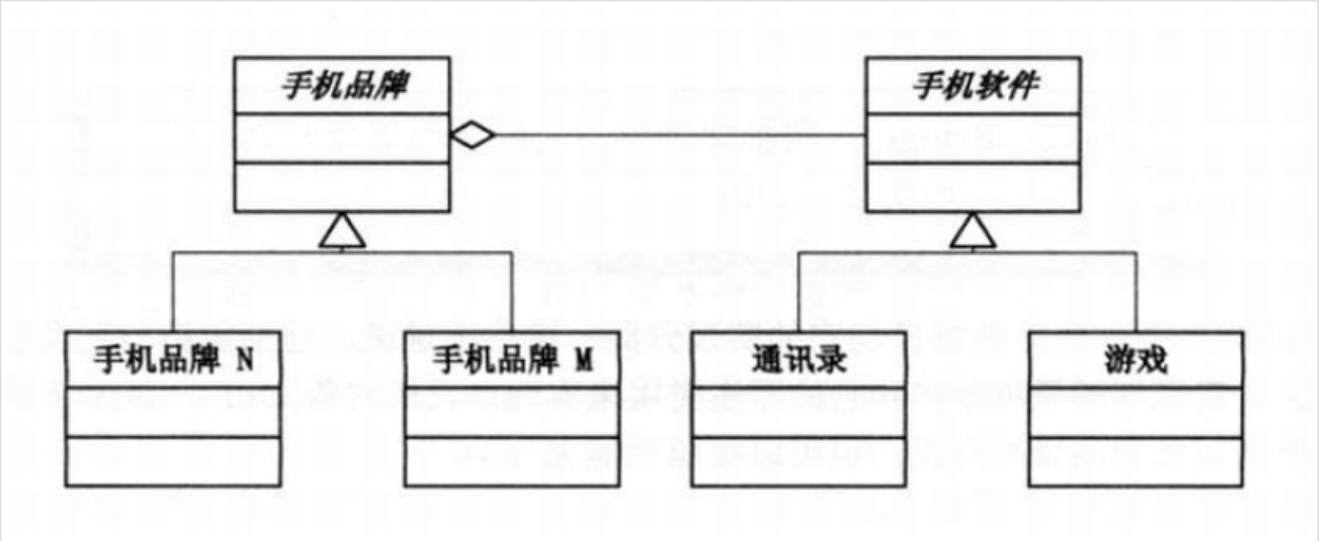
整个 Binder 通信机制都是基于代理模式，远程代理就好比“远程对象的本地代表”，所以跨进程交互或者说 C/S 结构，非常适合使用代理模式。因为是跨进程通信，客户端进程并不能直接拿到服务端的实例对象，只能通过远程代理（BpBinder）来访问服务端（BBinder），这样相互通信看起来像是两个本地对象在交互，而远程代理在幕后默默的和服务端通信，客户端并不清楚这个过程。另外，通过代理模式，Binder 机制能够控制访问权限，大大提供安全性。

单例模式

ProcessState, IPCThreadState, IServiceManager 都是使用的单例模式。

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

// IPCThreadState.cpp
IPCThreadState* IPCThreadState::self()
{
    if (gHaveTLS) {
restart:
        const pthread_key_t k = gTLS;
        IPCThreadState* st = (IPCThreadState*)pthread_getspecific(k);
        if (st) return st;
        return new IPCThreadState;
    }
    ...
}

// IServiceManager.cpp
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;

    {
        AutoMutex _l(gDefaultServiceManagerLock);
        while (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
            if (gDefaultServiceManager == NULL)
                sleep(1);
        }
    }

    return gDefaultServiceManager;
}
```

桥接模式



桥接模式连接着不同分类的两端，比如 `Proxy` 连接了 `IInterface, IBinder` 。

```
1  private static class Proxy implements com.***.ICustomAIDLInterface {
2      private android.os.IBinder mRemote;
3
4      Proxy(android.os.IBinder remote) {
5          mRemote = remote;
6      }
7      ...
8      @Override
9      public void customMethod(int anInt) throws android.os.RemoteException {
10         android.os.Parcel _data = android.os.Parcel.obtain();
11         android.os.Parcel _reply = android.os.Parcel.obtain();
12         try {
13             _data.writeInterfaceToken(DESCRIPTOR);
14             _data.writeInt(anInt);
15             mRemote.transact(Stub.TRANSACTION_basicTypes, _data, _reply, 0);
16             _reply.readException();
17         } finally {
18             _reply.recycle();
19             _data.recycle();
20         }
21     }
22 }
```

`Parcel` 在 `Binder` 机制中的作用

定义

先看一段 `Parcel.java` 中的注释：

```
1  * Container for a message (data and object references) that can
2  * be sent through an IBinder.  A Parcel can contain both flattened data
3  * that will be unflattened on the other side of the IPC (using the various
4  * methods here for writing specific types, or the general
5  * {@link Parcelable} interface), and references to live {@link IBinder}
6  * objects that will result in the other side receiving a proxy IBinder
7  * connected with the original IBinder in the Parcel.
```

Parcel 是一个容器包含了数据或对象的引用，它能够通过 IBinder 来传输。 Parcel 能够包含序列化的数据，这些数据可以被 IPC 通信的另一端反序列化（通过各种 write 方法或者 Parcelable 接口）；并且可以包含一个 IBinder 对象的引用，这个引用会让对方接受到一个和该 IBinder 对象已经连接好的代理。

Parcel 是整个 Binder 机制中，数据传输的载体，存储了所有需要跨进程通信的数据，包含 IBinder 也可存储到 Parcel 中。这个读写都是基于内存的，所以效率会比 Java Serializable 基于外部存储的要高。

文件路径

```
1  frameworks/base/core/java/android/os/Parcel.java
2  frameworks/base/core/jni/android_os_Parcel.cpp
3  frameworks/native/libs/binder/Parcel.cpp
```

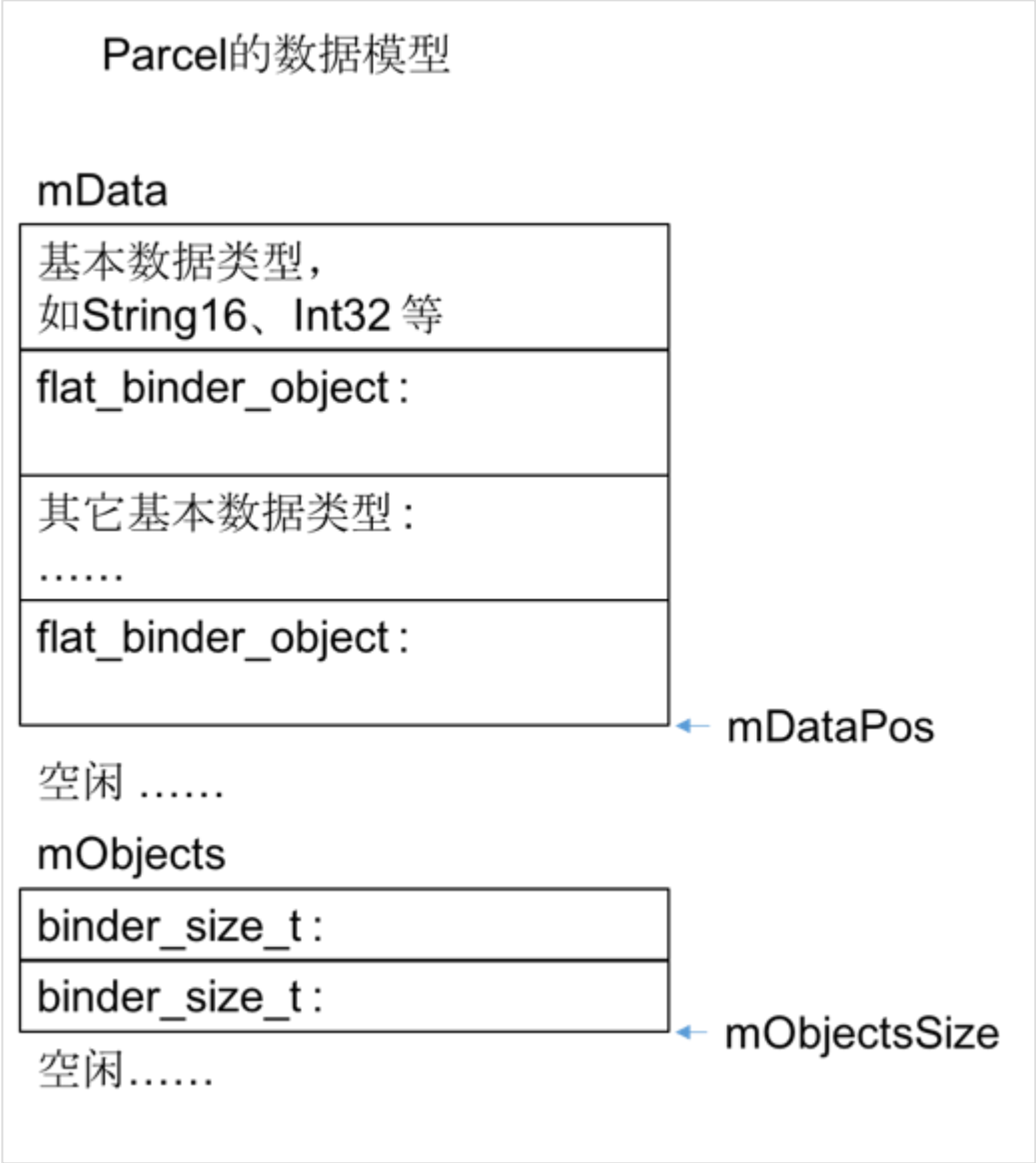
通过 jni 实现 native 的方法， jni 是在 AndroidRuntime 运行时初始化。

Binder 相关 API

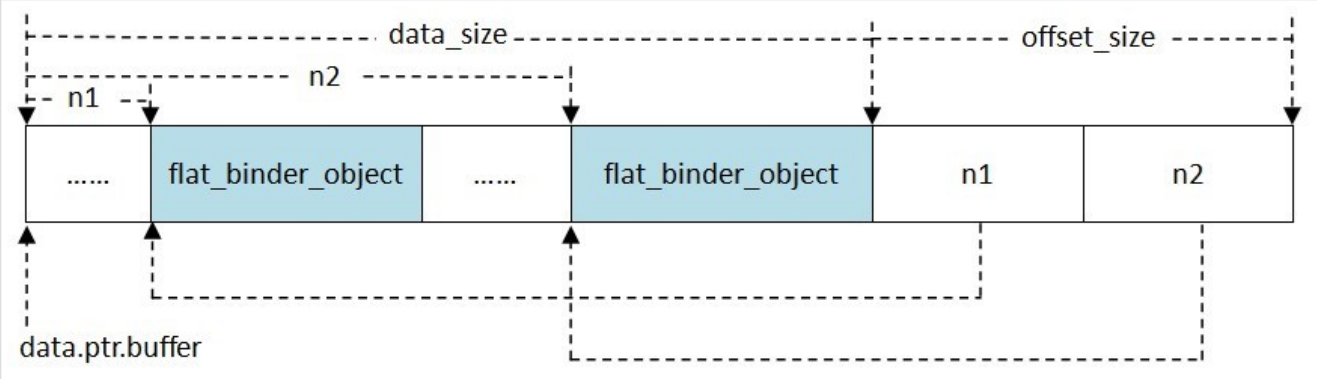
```
1  // Parcel.java
2  public final IBinder[] createBinderArray() {...}
3  public final void readBinderArray(IBinder[] val) {...}
4  public final void readBinderList(List<IBinder> list) {...}
5  public final void writeBinderArray(IBinder[] val) {...}
6  public final void writeBinderList(List<IBinder> val) {...}
7  public final ArrayList<IBinder> createBinderArrayList() {...}
8  /**
9   * Write an object into the parcel at the current dataPosition(),
10  * growing dataCapacity() if needed.
11  */
12 public final void writeStrongBinder(IBinder val) {
13     nativeWriteStrongBinder(mNativePtr, val);
14 }
15
```

```
16  /**
17   * Read an object from the parcel at the current dataPosition().
18   */
19  public final IBinder readStrongBinder() {
20      return nativeReadStrongBinder(mNativePtr);
21  }
22
23  /**
24   * Store or read an IBinder interface token in the parcel at the current
25   * {@link #dataPosition}.  This is used to validate that the marshalled
26   * transaction is intended for the target interface.
27   */
28  public final void writeInterfaceToken(String interfaceName) {
29      nativeWriteInterfaceToken(mNativePtr, interfaceName);
30  }
31
32  public final void enforceInterface(String interfaceName) {
33      nativeEnforceInterface(mNativePtr, interfaceName);
34  }
35  /**
36   * Write an object into the parcel at the current dataPosition(),
37   * growing dataCapacity() if needed.
38   */
39  public final void writeStrongInterface(IInterface val) {
40      writeStrongBinder(val == null ? null : val.asBinder());
41  }
```

Parcel 数据模型



Parcel 的数据区域分两个部分： mData 和 mObjects ，所有的数据不管是基础数据类型还是对象实体，全都追加到 mData 里， mObjects 是一个偏移量数组，记录所有存放在 mData 中的 flat_binder_object 实体的偏移量。



offsets_size，data.offsets 两个成员是 Binder 通信有别于其它 IPC 的地方。 Binder 采用面向对象的设计思想，一个 Binder 实体可以发送给其它进程从而建立许多跨进程的引用；另外这些引用也可以在进程之间传递，就象 Java 里将一个引用赋给另一个引用一样。为 Binder 在不同进程中建立引用必须有驱动的参与，由驱动在内核创建并注册相关的数据结构后接收方才能使用该引用。而且这些引用可以是强类型，需要驱动为其维护引用计数。然而这些跨进程传递的 Binder 混杂在应用程序发送的数据包里，数据格式由用户定义，如果不把它们一一标记出来告知驱动，驱动将无法从数据中将它们提取出来。于是就使用数组 data.offsets 存放用户数据中每个 Binder 相对 data.buffer 的偏移量，用 offsets_size 表示这个数组的大小。驱动在发送数据包时会根据 data.offsets 和 offset_size 将散落于 data.buffer 中的 Binder 找出来并一一为它们创建相关的数据结构。在数据包中传输的 Binder 是类型为 struct flat_binder_object 的结构体。对于接收方来说，该结构只相当于一个定长的消息头，真正的用户数据存放在 data.buffer 所指向的缓存区中。如果发送方在数据中内嵌了一个或多个 Binder ，接收到的数据包中同样会用 data.offsets，offset_size 指出每个 Binder 的位置和总个数。不过通常接收方可以忽略这些信息，因为接收方是知道数据格式的，参考双方约定的格式定义就能知道这些 Binder 在什么位置。

Binder 进程和线程管理

文件路径

```
1   frameworks/native/libs/binder/ProcessState.cpp
2   framework/native/libs/binder/IPCThreadState.cpp
```

概述

Android 系统特别为程序进程使用 Binder 机制封装了两个实现类，即 ProcessState/IPCThreadState 。

- ProcessState

  是进程相关的，负责打开 Binder 驱动设备，进行 mmap() 等准备工作。

- IPCThreadState

  是线程相关的，负责与 Binder 驱动进行具体的命令通信。

ProcessState.cpp

- 单例模式特性

  只能通过单例模式获取 ProcessState 对象，用于创建 Binder 线程： sp<ProcessState> proc =

`ProcessState::self();` 。而构造函数中会打开 `Binder` 设备，单例模式的设计可以确保每个进程的 `Binder` 设备只会被打开一次。

```
// 单例模式
sp<ProcessState> ProcessState::self()
{
    Mutex::Autolock _l(gProcessMutex);
    if (gProcess != NULL) {
        return gProcess;
    }
    // gProcess在 Static.cpp 中定义的全局变量
    gProcess = new ProcessState;
    return gProcess;
}
```

- 线程数

  每个 `APP` 在启动时都会创建名称为 `Binder_X` 的线程，最少会创建 2 个（`Binder` 主线程和当前加入的线程），最多会创建 15 个，可以通过命令查看：

  命令： `adb shell; ps -t | grep -irs "binder*"`

```
#define DEFAULT_MAX_BINDER_THREADS 15
static int open_driver()
{
    int fd = open("/dev/binder", O_RDWR | O_CLOEXEC);
    if (fd >= 0) {
        ...
        size_t maxThreads = DEFAULT_MAX_BINDER_THREADS;
        result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
        ...
    }
    return fd;
}
```

- 创建线程

  使用 `ProcessState` 来创建线程池，并且确保每个进程的线程池只会被创建一次，并且会创建第一个 `PoolThread` 主线程。

```
proc->startThreadPool();  //或者
ProcessState::self()->startThreadPool();

void ProcessState::startThreadPool()
{
```

```
6       AutoMutex _l(mLock);
7       if (!mThreadPoolStarted) {
8           mThreadPoolStarted = true;
9           spawnPooledThread(true);
10      }
11  }
12  void ProcessState::spawnPooledThread(bool isMain)
13  {
14      if (mThreadPoolStarted) {
15          String8 name = makeBinderThreadName();
16          ALOGV("Spawning new pooled thread, name=%s\n", name.string());
17          sp<Thread> t = new PoolThread(isMain);
18          t->run(name.string());
19      }
20  }
```

- `PoolThread`

  线程池在开启时，会创建一个主线程 `PoolThread` 。这个类很简单，仅仅是作为主线程加入了线程池： `IPCThreadState::self()->joinThreadPool(mIsMain);` 。

```
1   class PoolThread : public Thread
2   {
3   public:
4       PoolThread(bool isMain)
5           : mIsMain(isMain)
6       {
7       }
8
9   protected:
10      virtual bool threadLoop()
11      {
12          IPCThreadState::self()->joinThreadPool(mIsMain);
13          return false;
14      }
15
16      const bool mIsMain;
17  };
```

`Binder` 进程大小不超过 `1M`

`Binder` 是轻量级进程间通信机制，传输的数据大小不能超过 `1M` 。

```
// ProcessState.cpp
// Binder 虚拟机默认大小为 1M - 8K 大小的内存
#define BINDER_VM_SIZE ((1*1024*1024) - (4096 *2))
ProcessState::ProcessState()
    : mDriverFD(open_driver())
    ,...
{
    ...
    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_NORESERVE, mDriverFD, 0);
    ...
}
```

ProcessState 常用 API

```
// 单例，获取对象
sp<ProcessState> ProcessState::self()
// 开启线程池
ProcessState::self()->startThreadPool();
// 获取 handle 为 0 的 IBinder，handle 为 0 表示是 service_manager 守护进程
ProcessState::self()->getContextObject(NULL)
//
ProcessState::self()->getContextObject(const String16& name, const sp<IBinder>& caller
// 设置为上下文管理员
bool ProcessState::becomeContextManager(context_check_func checkFunc, void* userData)
```

IPCThreadState.cpp

- 万众归一 `joinThreadPool`

  可以看到不管是 `ProcessState` 创建的线程，还是其他应用线程，最终都是通过 `joinThreadPool` 来加入 `Binder` 线程池的。

```
// 头文件定义，默认为 true
void joinThreadPool(bool isMain = true);

void IPCThreadState::joinThreadPool(bool isMain)
{
    ...
    // 主线程和其他线程 BC 码不一样
    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
```

```
    ...
    // 如果是主线程将进入无限循环，处理请求信息
    do {
        processPendingDerefs();
        // now get the next command to be processed, waiting if necessary
        result = getAndExecuteCommand();
        ...
        if(result == TIMED_OUT && !isMain) {
            break;
        }
    } while (result != -ECONNREFUSED && result != -EBADF);
    ...
    mOut.writeInt32(BC_EXIT_LOOPER);
    talkWithDriver(false);
}
```

- 两个重要数据存储

  mIn, mOut： mIn 用来接收来自 Binder Driver 的数据， mOut 用来存储发往 Binder Driver 的数据。

- 请求码和响应码

  BINDER_COMMAND_PROTOCOL：请求码，以 BC_ 开头，简称 BC 码，请求命令用于用户空间向内核空间发出请求。

  BINDER_RETURN_PROTOCOL：响应码，以 BR_ 开头，简称 BR 码，用于响应返回命令，内核空间向用户空间返回响应。

- 向驱动发送请求码

  IPCThreadState 各个 API 会将请求码写入 mOut，最终都会通过 talkWithDriver 写入 Binder Driver。

- 处理驱动返回的响应码

  响应信息都会通过 mIn 传递回 Native 层。

```
status_t IPCThreadState::executeCommand(int32_t cmd)
status_t IPCThreadState::waitForResponse(Parcel *reply,
            status_t *acquireResult)
```

- AIDL 中的 oneway 关键字处理

  AIDL 中是否定义 oneway 关键字，主要是在传递的过程中会体现：

```
// 没有使用 oneway
mRemote.transact(***, _data, null, 0);
// 使用了 oneway
mRemote.transact(***, _data, null, android.os.IBinder.FLAG_ONEWAY);
```

而这个标记最终会在这里解析：

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    ...
    // 没有设置 oneway
    if ((flags & TF_ONE_WAY) == 0) {
        ...
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        ...
    // 设置 oneway
    } else {
        err = waitForResponse(NULL, NULL);
    }

    return err;
}

status_t IPCThreadState::executeCommand(int32_t cmd)
{
    ...
    case BR_TRANSACTION:
        {
            ...
            // 没有设置 oneway，发送回执
            if ((tr.flags & TF_ONE_WAY) == 0) {
                LOG_ONEWAY("Sending reply to %d!", mCallingPid);
                if (error < NO_ERROR) reply.setError(error);
                sendReply(reply, 0);
            // 设置 oneway，忽略
            } else {
                LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
            }
```

```
            ...
        }
    ...
}
```

代码中可以看到， oneway 关键字决定了是否阻塞等待 waitForResponse 以及响应时是否发送回执 sendReply 。

Binder 机制驱动交互

真正与 Binder Driver 交互的地方是 talkWithDriver 中的 ioctl() ，通过它 BINDER_WRITE_READ 命令写入 Binder Driver 。

```
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    ...
    binder_write_read bwr;
    ...
    bwr.write_size = outAvail;
    bwr.write_buffer = (uintptr_t)mOut.data();

    // This is what we'll read.
    if (doReceive && needRead) {
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (uintptr_t)mIn.data();
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }
    ...
    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
    status_t err;
    do {
        IF_LOG_COMMANDS() {
            alog << "About to read/write, write size = " << mOut.dataSize() << endl;
        }
#if defined(__ANDROID__)
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
        ...
    } while (err == -EINTR);
```

```
30      ...
31  }
```

IPCThreadState 常用 API

```
1   // 加入 Binder 线程池
2   IPCThreadState::self()->joinThreadPool();
```

至少 2 个 Binder 线程

在所有使用 Binder 机制的示例中，都能看到初始化时至少会执行如下两句：

```
1   int main(...){
2       ...
3       ProcessState::self()->startThreadPool();    // Binde_1 主线程
4       IPCThreadState::self()->joinThreadPool();   // 当前主线程变成 Binder 线程
5       ...
6   }
```

从前面的分析可以看到：

- `ProcessState::self()->startThreadPool();`
  开启线程池，也就是新建一个 Binder 主线程，名称为 Binder_1 。
- `IPCThreadState::self()->joinThreadPool();`
  当前线程加入线程池，也就是将当前线程变为 Binder 线程。

我们在分析 startThreadPool() 时可以看到，新建了一个 PoolThread 异步调用 joinThreadPool()，同时应用主线程同步调用了 joinThreadPool，阻塞等待。代码中可以看出，这两个都是 Binder 主线程，但是线程名不一样，同步调用 joinThreadPool() 的目的之一是确保 startThreadPool 异步产生的线程不会因为执行到了 main 函数结尾而被迫退出；目的之二可能是为了提高 Binder 线程处理的吞吐量，都可以等待并处理请求。

Binder 线程总结

Binder 系统中可分为 3 类线程：

- Binder 主线程
  ProcessState::self()->startThreadPool(); 创建 Binder 主线程。编号从 1 开始，即主线程名

为 Binder_1 ，并且主线程是不会退出的。

- Binder 普通线程

  由 Binder Driver 来决定是否创建 Binder 线程，发回消息 BR_SPAWN_LOOPER 后回调 spawnPooledThread(false) 创建普通线程，该线程名格式为 Binder_X 。

- Binder 其他线程

  其他线程是指并没有调用 spawnPooledThread 方法，而是直接调用 IPCThreadState::self()->joinThreadPool ，将当前线程直接加入 Binder 线程队列。

ServiceManager

双重属性

ServiceManager 它既是客户端也是服务端。

- 作为服务端

  IServiceManager.java/IServiceManager.cpp ： 其他进程都是通过它们来查询或注册服务的。不过在 Java 代码中， ServiceManager.java 是对 IServiceManager.java 的一个封装，同时保存了一个 Cache ，即 Java 层通常是通过 ServiceManager.java 来访问的，而不是直接调用 IServiceManager.java 。

- 作为客户端

  service_manager.c ： 手机开机时 init.rc 会开启一个名称是小写的 servicemanager 服务，它是由 service_manager.c 实现的守护进程，为了做区分本文将守护进程服务命名重命名为 service_manager 。守护进程开启后会进入无限循环，只有两个功能：注册服务和查询服务。

  IServiceManager.cpp 和 service_manager.c 是一个完整的 Binder 通信流程。

  ServiceManager 可以看做客户端， service_manager 守护进程可以看做是服务端。这部分的通信过程是系统实现的，可以认为对用户透明，所以通常将 ServiceManager 和 service_manager 合二为一，统称为 ServiceManager 。

ServiceManager 存在的意义

在 Android 系统中，所有 Service 都需要加入到 ServiceManager 来集中管理。这样客户端可以很方便的通过服务名称从系统查询服务，同时 ServiceManager 会向客户端提供服务端的 IBinder ，用于客户端和服务端的 Binder 通信。而且这个过程都是系统自动完成，系统屏蔽了整个通信机制，只开放两个接口：

```
1   public static void addService(String name, IBinder service) {...}
```

```
2   public static IBinder getService(String name) {...}
```

C/S 模型

- 注册服务 addService

  Server 进程向 ServiceManager 注册服务。该过程： Server 是客户端， ServiceManager 是服务端。

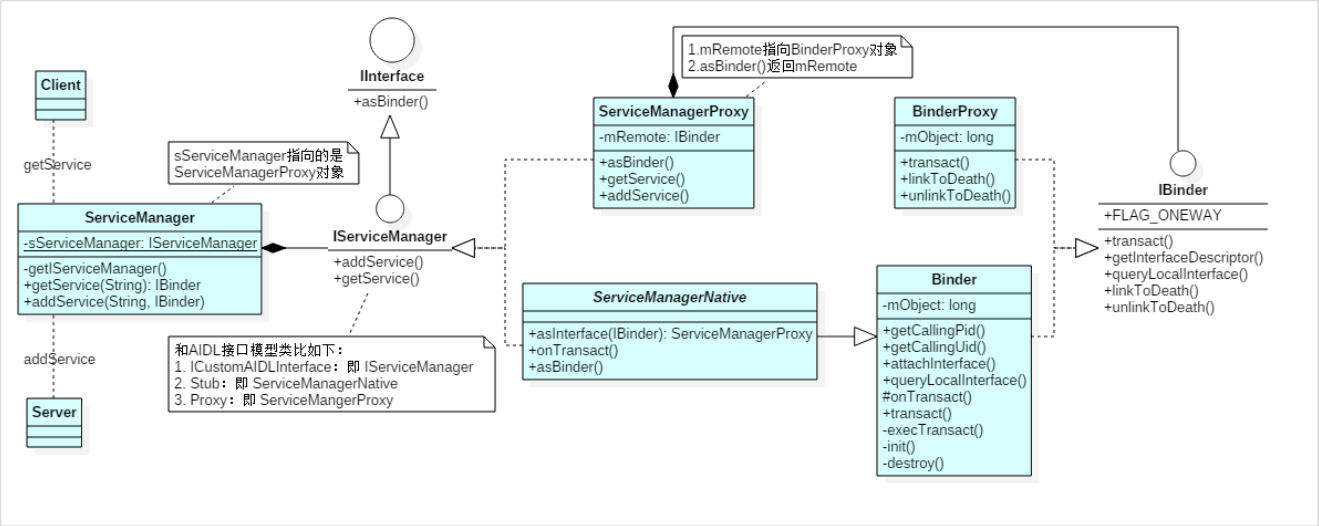
- 获取服务 getService

  Client 进程向 ServiceManager 获取相应的服务。该过程： Client 是客户端， ServiceManager 是服务端。

- 使用服务

  Client 得到服务的 IBinder 与 Server 进程通信，然后就可以通过 Binder Driver 交互数据。该过程： Client 是客户端， Server 是服务端。

Java 层的类图



- ServiceManagerProxy

  其成员变量 mRemote 指向 BinderProxy 对象， ServiceManagerProxy:addService，getService 方法最终是交由 mRemote 来完成。

- ServiceManager

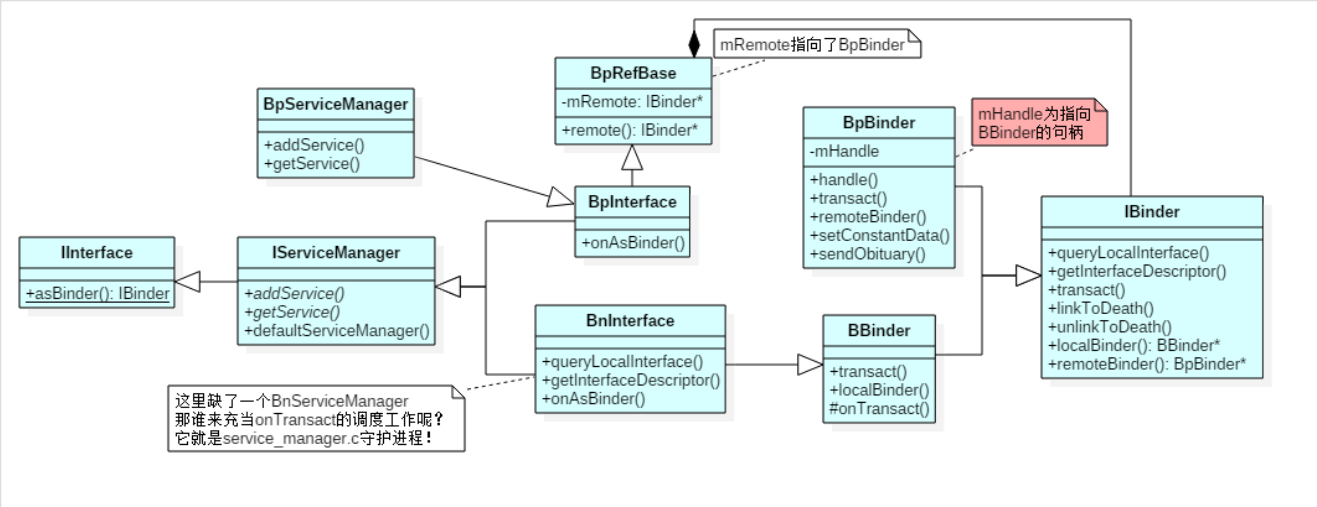
  通过 getIServiceManager 方法获取的是 ServiceManagerProxy 对象。ServiceManager:addService，getService 实际工作都交由 ServiceManagerProxy 的相应方法来处理。

- ServiceManagerNative

  asInterface() 返回的是 ServiceManagerProxy 对象， ServiceManager 是通过

ServiceManagerNative 类来找到 ServiceManagerProxy 的。

CPP 层的类图



- BpServiceManager

  同时继承了 IServiceManager, BpInterface，其中 BpInterface 继承 BpRefBase，而 BpRefBase.mRemote 指向了 BpBinder。BpBinder.mHandle 为指向 BBinder 的句柄，通过这个句柄实现 Binder 间的通信。

- IServiceManager

  单例模式 IServiceManager::defaultServiceManager 获取到 BpServiceManager 实例。

对比 Binder_CPP 核心类图，缺失了 BnServiceManager 这个类。那谁来充当 onTransact 的调度工作呢？它就是 service_manager.c 守护进程！

IServiceManager::defaultServiceManager

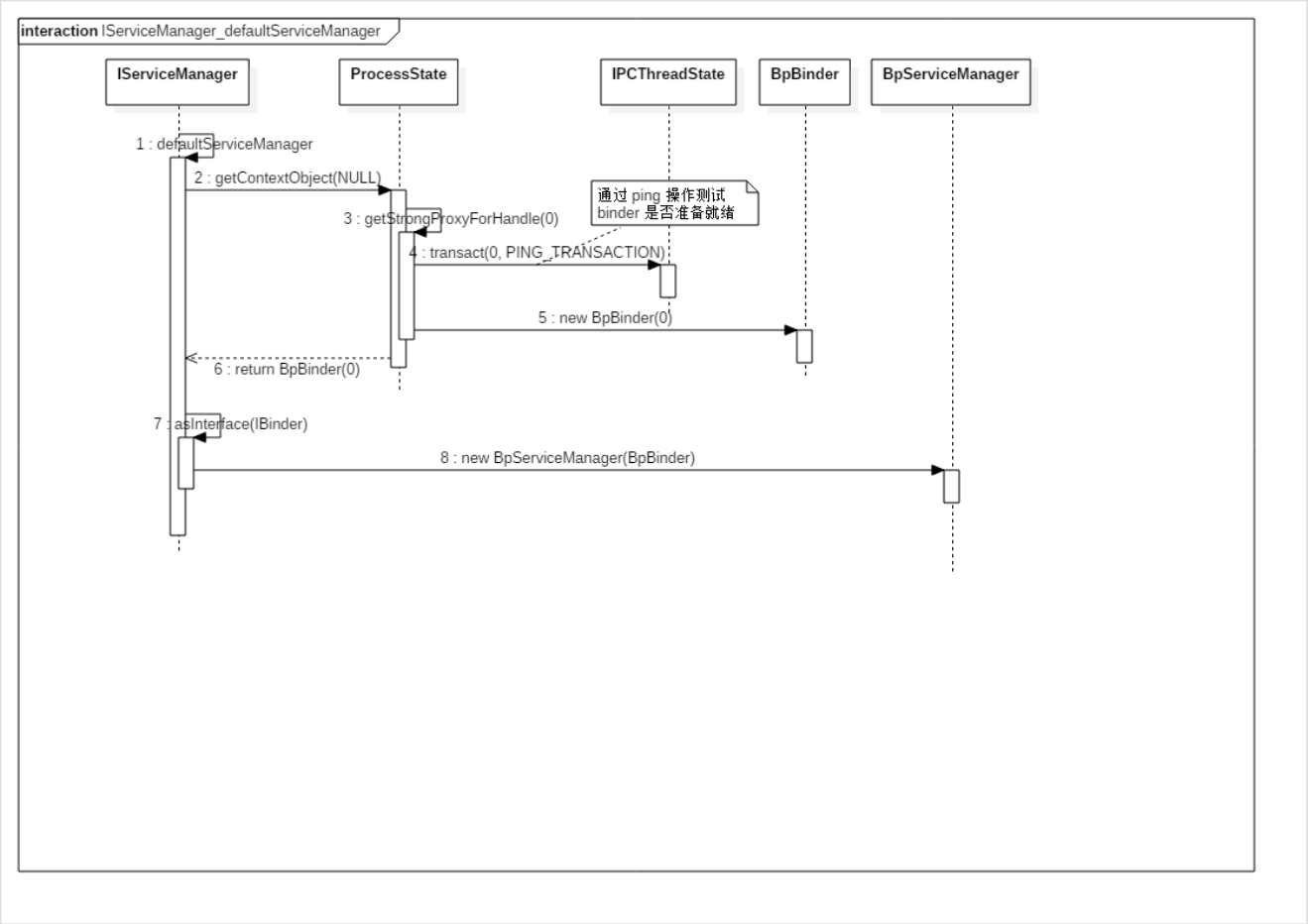
从类图中可以看到，IServiceManager 并没有对应的注册服务，只提供了查询即 defaultServiceManager。得到 handle 句柄为 0 的 BpBinder，而 0 号句柄对应的 BBinder 实际为 service_manager 守护进程。

```
1  // Static.cpp 中定义
2  sp<IServiceManager> gDefaultServiceManager;
3  // IServiceManager.cpp，单例模式获取
4  sp<IServiceManager> defaultServiceManager()
5  {
6      if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
7
8      {
```

```
        AutoMutex _l(gDefaultServiceManagerLock);
        while (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
            if (gDefaultServiceManager == NULL)
                sleep(1);
        }
    }

    return gDefaultServiceManager;
}

// ProcessState.cpp
// 句柄为 0 时，先去 ping 一下看是否准备好
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t 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)
```

```
56                      return NULL;
57              }
58
59              b = new BpBinder(handle);
60              e->binder = b;
61              if (b) e->refs = b->getWeakRefs();
62              result = b;
63          } ...
64      return result;
65  }
```

时序图：



总结：

- `defaultServiceManager` 等价于 `new BpServiceManager(new BpBinder(0));`
- `handle` 为 0 的句柄，代表 `ServiceManager` 所对应的 `BBinder`

示例：

```
//获取service manager引用
sp < IServiceManager > sm = defaultServiceManager();
//注册名为"service.myservice"的服务到service manager
sm->addService(String16("service.myservice"), new BnMyService());

//获取service manager引用
sp < IServiceManager > sm = defaultServiceManager();
//获取名为"service.myservice"的binder接口
sp < IBinder > binder = sm->getService(String16("service.myservice"));
//将biner对象转换为强引用类型的IMyService
sp<IMyService> cs = interface_cast < IMyService > (binder);
```

`service_manager` 守护进程

文件路径

```
frameworks/native/cmds/servicemanager/servicemanager.rc
frameworks/native/cmds/servicemanager/service_manager.c
frameworks/native/cmds/servicemanager/binder.c
```

守护进程

由 `init.rc` 开启的守护进程，对应 `service_manager.c` 文件。

```
// 对应 rc 文件：Servicemanager.rc
service servicemanager /system/bin/servicemanager
    class core
    user system
    group system readproc
    critical
    onrestart restart healthd
    onrestart restart zygote
    onrestart restart audioserver
    onrestart restart media
    onrestart restart surfaceflinger
    onrestart restart inputflinger
    onrestart restart drm
    onrestart restart cameraserver
    writepid /dev/cpuset/system-background/tasks
```

主程序

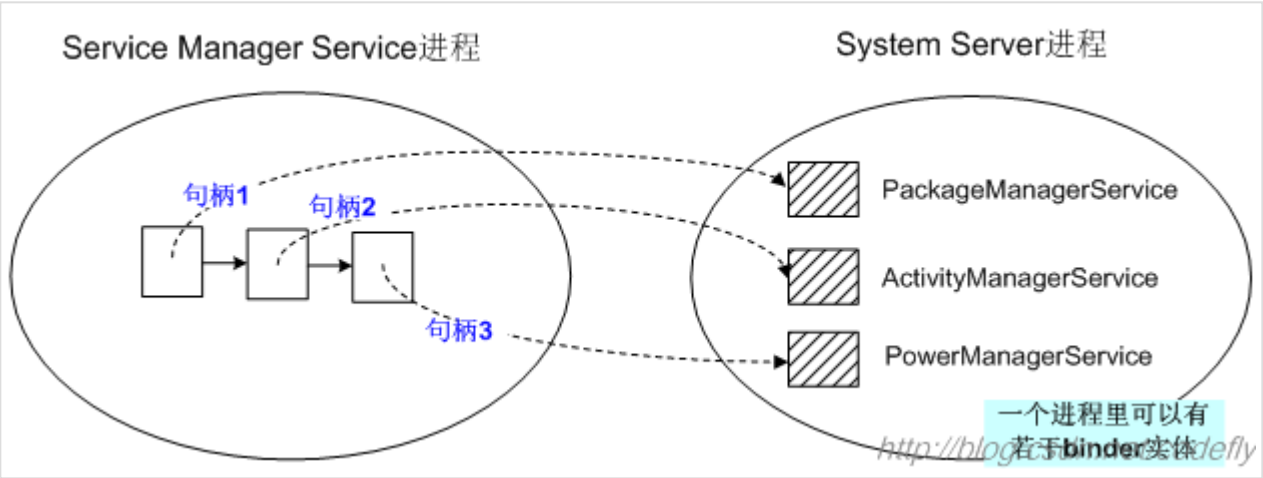
```
// service_manager.c
int main()
{
    ...
    // 开启 Binder 驱动
    bs = binder_open(128*1024);
    ...
    // 成为 Binder 服务的大管家
    if (binder_become_context_manager(bs)) {
        ALOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }
    ...
    // 进入无限循环，处理客户端请求
    binder_loop(bs, svcmgr_handler);
    ...
}
```

主程序逻辑很简单：

- 打开 `Binder Driver`，申请 128k 字节大小的内存空间
- 注册成为 `Binder` 服务的大管家，也就是对应 `BBinder` 句柄为 0
- 进入无限循环，处理客户端发来的请求

服务链表

无论调用 `Java/CPP API`，每个服务最终加入 `svclist` 单向链表中保存。我们也可以看到 `svcinfo` 这个结构体实际只保存了服务的名称和句柄（这个句柄就是 `BpBinder.mHandle`）。



> 查看系统已经注册了的所有服务： `adb shell service list`

```
struct svcinfo
{
    struct svcinfo *next;
    uint32_t handle;
    struct binder_death death;
    int allow_isolated;
    size_t len;
    uint16_t name[0];
};

struct svcinfo *svclist = NULL;

// 注册和查询服务入口
int svcmgr_handler(struct binder_state *bs,
                   struct binder_transaction_data *txn,
                   struct binder_io *msg,
                   struct binder_io *reply)
{
    ...
    uint32_t handle;
    ...
    switch(txn->code) {
    case SVC_MGR_GET_SERVICE:
    case SVC_MGR_CHECK_SERVICE:
        s = bio_get_string16(msg, &len);
        if (s == NULL) {
            return -1;
        }
        handle = do_find_service(s, len, txn->sender_euid, txn->sender_pid);
        if (!handle)
            break;
        bio_put_ref(reply, handle);
        return 0;
    case SVC_MGR_ADD_SERVICE:
        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;
    ...
}
```

守护进程注册为大管家

binder_become_context_manager 在 Native 层，很简单仅仅是下发了 BINDER_SET_CONTEXT_MGR 的 ioctl 命令，具体见 Driver 部分分析。

```
1  int binder_become_context_manager(struct binder_state *bs)
2  {
3      return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
4  }
```

这个命令在驱动中实现了如下功能：

- 告诉驱动，当前进程即 Binder 上下文管理者
- 新建对应于 Context Manager 的 binder_node （即 BBinder 对应的驱动结构体）
- 新建 binder_ref，设置句柄为 0，并设置引用地址为上面这个 Binder 实体

核心工作

service_manager 会无限循环读取和处理：服务注册或查询请求（由 IServiceManager 发出），和 IPCThreadState::talkWithDriver 一样，向驱动下发 BINDER_WRITE_READ 命令，读取并解析驱动返回的信息。

```
1  void binder_loop(struct binder_state *bs, binder_handler func)
2  {
3      int res;
4      struct binder_write_read bwr;
5      uint32_t readbuf[32];
6
7      bwr.write_size = 0;
8      bwr.write_consumed = 0;
9      bwr.write_buffer = 0;
10
11     readbuf[0] = BC_ENTER_LOOPER;
12     binder_write(bs, readbuf, sizeof(uint32_t));
13
14     for (;;) {
15         bwr.read_size = sizeof(readbuf);
16         bwr.read_consumed = 0;
17         bwr.read_buffer = (uintptr_t) readbuf;
18
```

```
        // 和 IPCThreadState::talkWithDriver 一样，下发 BINDER_WRITE_READ 命令。
        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
        ...
        // 解析驱动返回信息
        res = binder_parse(bs, 0, (uintptr_t) readbuf, bwr.read_consumed, func);
        ...
    }
}

// func 即为 svcmgr_handler， 用于查询和注册
int binder_parse(struct binder_state *bs, struct binder_io *bio,
                 uintptr_t ptr, size_t size, binder_handler func)
{
    ...
    switch(cmd) {
    ...
    case BR_TRANSACTION: {
        struct binder_transaction_data *txn = (struct binder_transaction_data *) ptr;
        ...
        if (func) {
            unsigned rdata[256/4];
            struct binder_io msg;
            struct binder_io reply;
            int res;

            bio_init(&reply, rdata, sizeof(rdata), 4);
            bio_init_from_txn(&msg, txn);
            // 调用 svcmgr_handler
            res = func(bs, txn, &msg, &reply);
            if (txn->flags & TF_ONE_WAY) {
                binder_free_buffer(bs, txn->data.ptr.buffer);
            } else {
                binder_send_reply(bs, &reply, txn->data.ptr.buffer, res);
            }
        }
        ptr += sizeof(*txn);
        break;
    }
    ...
}
```

服务注册

系统所有服务，最终会在这里实现注册。从 `svcmgr_handler` 可以看到，服务注册调用的是 `do_add_service`，新建一个 `svcinfo` 保存基本的名称和句柄，并加入链表。句柄是驱动创建 `Binder` 实体对象时生成，同时还会生成一个 `Binder` 引用对象指向它。

```
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;

    ...
    {
        si = malloc(sizeof(*si) + (len + 1) * sizeof(uint16_t));
        ...
        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;
        si->death.ptr = si;
        si->allow_isolated = allow_isolated;
        si->next = svclist;
        svclist = si;
    }

    binder_acquire(bs, handle);
    binder_link_to_death(bs, handle, &si->death);
    return 0;
}
```

服务查询

系统中查询对应的服务，从 `svcmgr_handler` 可以看到，服务查询和检查都是调用的 `do_find_service`，通过服务的名称来匹配，并返回句柄。为什么只需要返回句柄就可以了？从后面的驱动分析中可以了解到，驱动通过句柄可以找到相应的 `Binder` 引用对象，而 `Binder` 引用对象的结构体中保存了 `Binder` 实体对象。也就是说通过句柄能同时找到 `Binder` 的引用和实体对象。

```
uint32_t do_find_service(const uint16_t *s, size_t len, uid_t uid, pid_t spid)
{
```

```
    struct svcinfo *si = find_svc(s, len);
    ...
    return si->handle;
}

struct svcinfo *find_svc(const uint16_t *s16, size_t len)
{
    struct svcinfo *si;

    for (si = svclist; si; si = si->next) {
        if ((len == si->len) &&
            !memcmp(s16, si->name, len * sizeof(uint16_t))) {
            return si;
        }
    }
    return NULL;
}
```

IServiceManager.cpp/service_manager.c 通信流程

在前面的分析中，IServiceManager.cpp 只对应生成了 BpServiceManager 类，而没有 BnServiceManager 类的存在，也就是说在 IServiceManager 中并没有注册服务，只提供了查询服务 defaultServiceManager。而守护进程 service_manager.c 只注册成为了上下文大管家，并在驱动中新建了 binder_node 并赋值给变量 binder_context_mgr_node。IServiceManager.cpp/service_manager.c 之间是如果串起来的呢？也就是 service_manager.c 如何完成了 BnServiceManager 的功能？这些都是在 Binder Driver 中实现的，下面先做个简要分析。在 Binder 通信机制中，BpBinder.mHandle 找到对应的 BBinder，是在 Binder Driver 的 binder_transaction 中实现的。

路由逻辑是：如果 handle 为真，则通过 handle 在红黑树中找到 bind_ref（即 Native 层中的 BpBinder），而这个结构体中保存了通信对应的 binder_node（即 Native 层中的 BBinder）；如果 handle 不为真，即句柄为 0，则返回 binder_context_mgr_node。

也就是说句柄为 0 时，对应的就是和 service_manager 通信，而 IServiceManager::defaultServiceManager 查询服务时，也就明白为什么要直接赋值句柄为 0 了。

```
driver: binder.c
static void binder_transaction(struct binder_proc *proc,
                   struct binder_thread *thread,
                   struct binder_transaction_data *tr, int reply)
{
    ...
    struct binder_node *target_node = NULL;
    ...
```

```
        if (tr->target.handle) {
            struct binder_ref *ref;
            ref = binder_get_ref(proc, tr->target.handle, true);
            if (ref == NULL) {
                binder_user_error("%d:%d got transaction to invalid handle\n",
                    proc->pid, thread->pid);
                return_error = BR_FAILED_REPLY;
                goto err_invalid_target_handle;
            }
            target_node = ref->node;
        } else {
            target_node = binder_context_mgr_node;
            if (target_node == NULL) {
                return_error = BR_DEAD_REPLY;
                goto err_no_context_mgr_node;
            }
        }
    ...
}
```

Binder Driver 驱动

Binder Driver 是整个 Binder 通信机制的核心，它工作于内核态，负责进程之间通信的建立，数据在进程之间转换和传递，每个进程中最大线程数为 15 个。

文件路径

```
./drivers/staging/android/binder.h
./drivers/staging/android/binder.c
./drivers/staging/android/uapi/binder.h
./drivers/staging/android/binder_trace.h
```

在 kernel 3.19 之后，默认已经合入到 kernel master 分支中了。

master 分支

```
./drivers/android/binder_alloc.h
./drivers/android/binder_alloc.c
./drivers/android/binder.c
./drivers/android/binder_trace.h
./drivers/android/binder_alloc_selftest.c
./include/uapi/linux/android/binder.h
```

数据结构

- binder_proc： Binder 进程

  对应于用户空间的 ProcessState，每个进程调用 open() 打开 Binder 驱动都会创建该结构体，用于管理 IPC 所需的各种信息。
- binder_thread： Binder 线程

  对应于上层的 Binder 线程。
- binder_node： Binder 实体

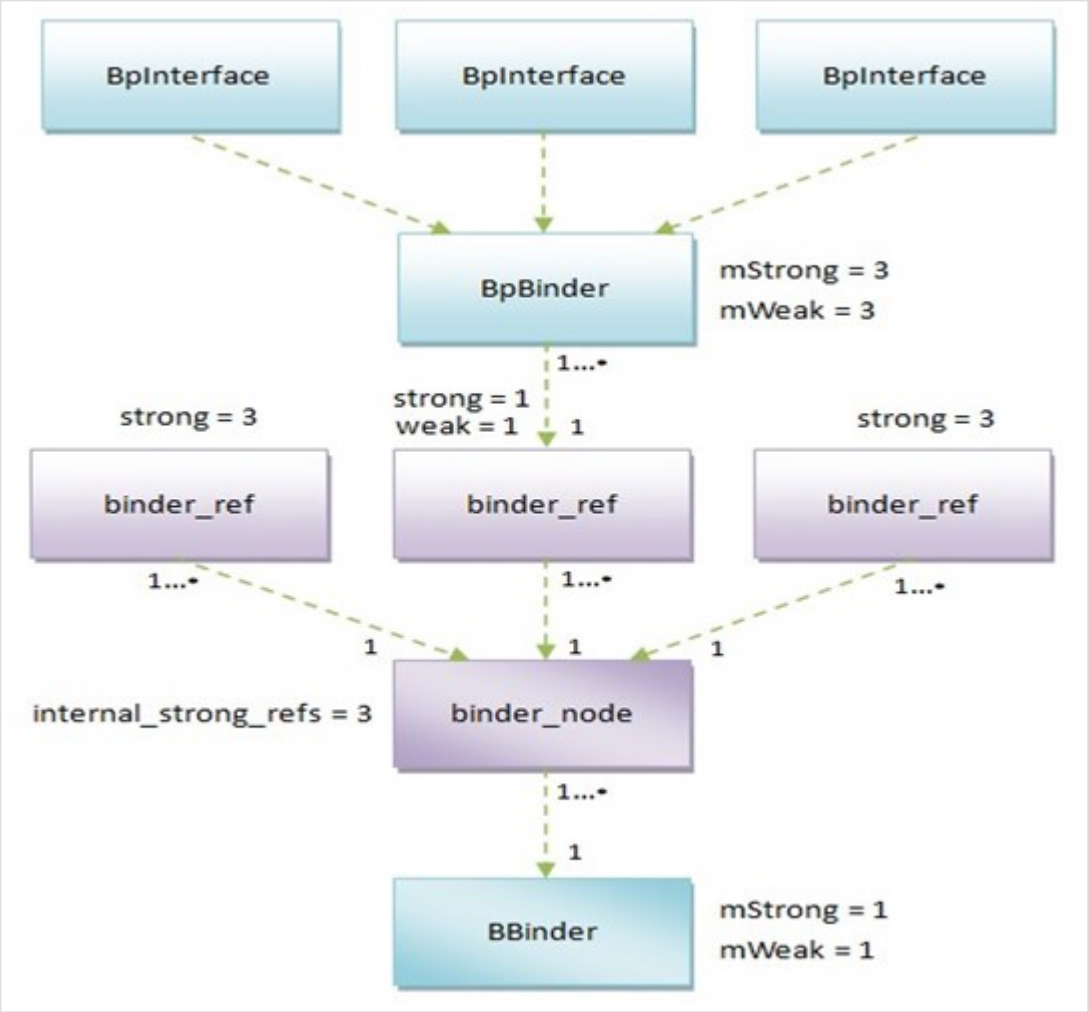
  对应于 BBinder 对象，记录 BBinder 的进程、指针、引用计数等。
- binder_ref： Binder 引用

  binder_node 实体对象的引用，对应于 BpBinder 对象，记录 BpBinder 的引用计数、死亡通知、BBinder 指针等。
- binder_ref_death： Binder 死亡引用

  记录 Binder 死亡的引用信息。
- flat_binder_object： IBinder 扁平对象

  IBinder 对象在两个进程间传递的扁平结构。

Binder 对象之间的引用关系



- `BBinder` 被 `binder_node` 引用
- `binder_node` 被 `binder_ref` 引用
- `binder_ref` 被 `BpBinder` 引用
- `BBinder` 和 `BpBinder` 运行在用户空间
- `binder_node` 和 `binder_ref` 运行在内核空间

调用顺序： `Client -> handle -> binder_ref -> binder_node -> Service` 。

Binder 通信机制高效原理

- 用户空间和内核空间简介

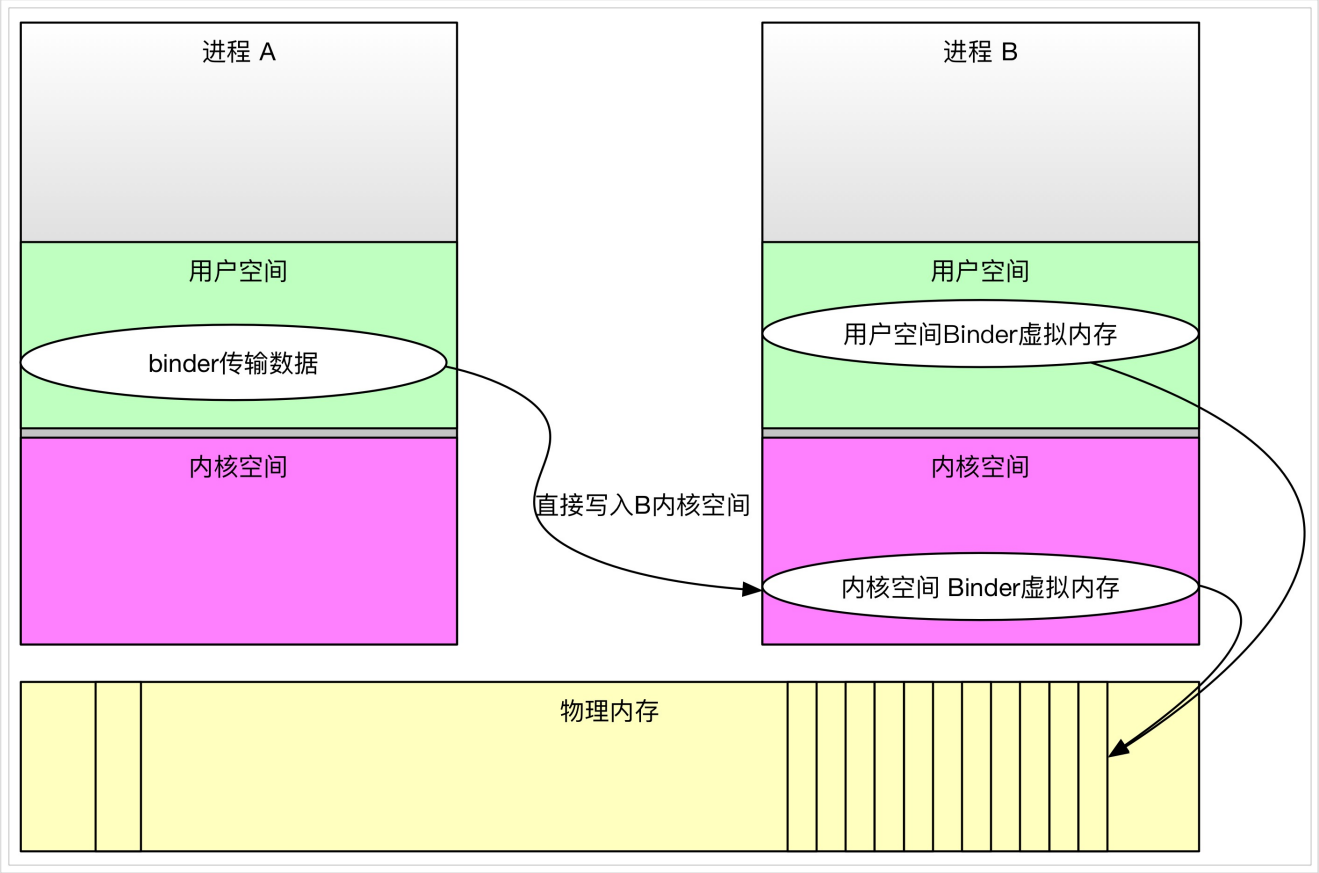
  `Linux` 操作系统和驱动程序运行在内核空间，应用程序运行在用户空间，两者不能简单地使用指针传递数据，因为 `Linux` 使用的虚拟内存机制，用户空间的数据可能被换出，当内核空间使用用户空间指针时，对应的数据可能不在内存中。用户空间的内存映射采用段页式，而内核空间有自己的规则。
- 虚拟进程空间

  通常 32 位 `Linux` 内核虚拟地址空间总共为 4G：划分 0~~3G 为用户空间，~~3~~4~~G 为内核空间(注意，内

核可以使用的线性地址只有 1G)。注意这里是 32 位内核地址空间划分，64 位内核地址空间划分是不同的。也就是说每个进程可以使用 4G 的虚拟内存，但是实际物理内存可能只用了几兆。

- 内核和用户虚拟内存空间映射

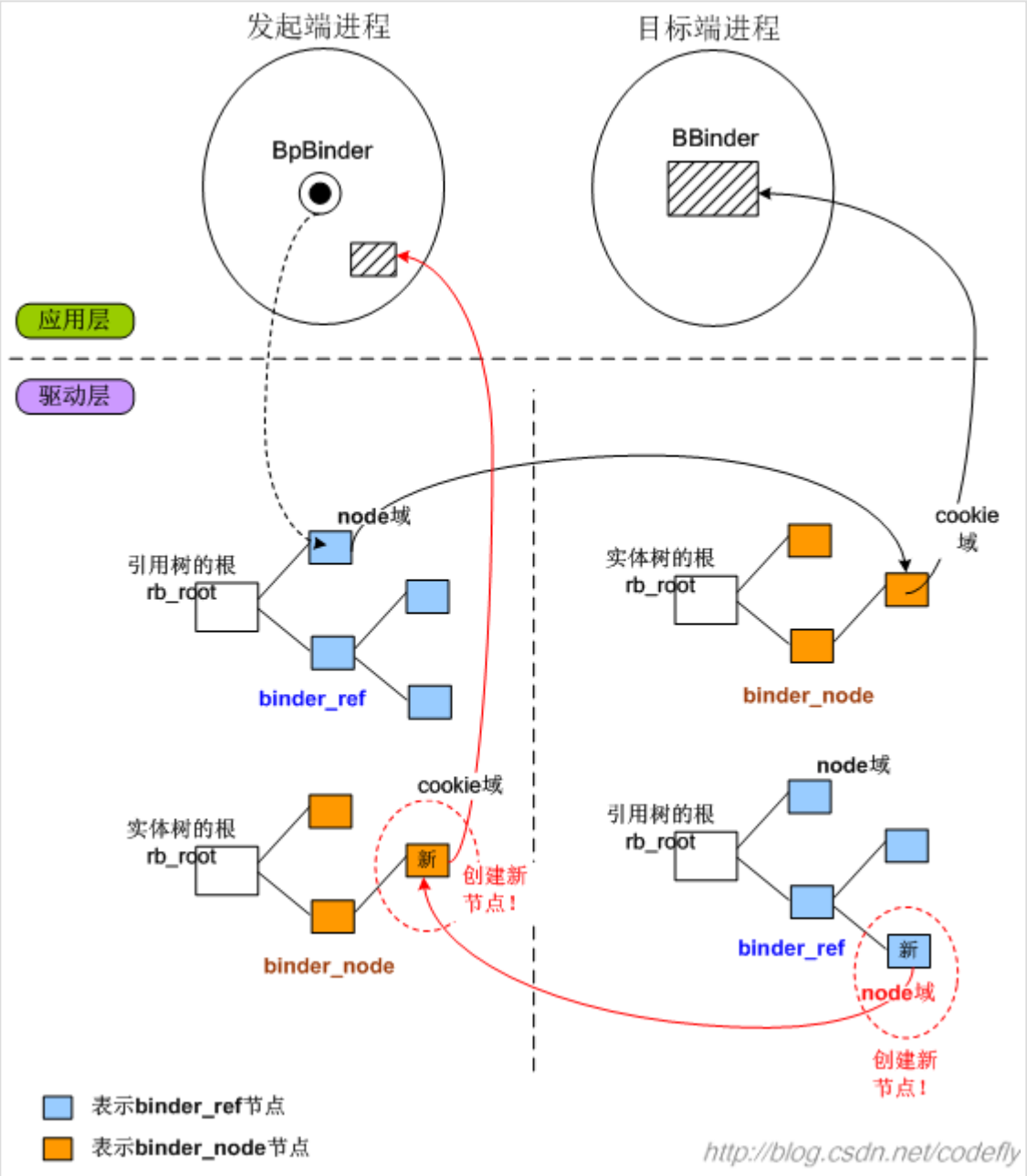
  首先在内核虚拟地址空间，申请一块与用户虚拟内存相同大小的内存；然后申请 1 个页大小的物理内存，再将同一块物理内存分别映射到内核虚拟地址空间和用户虚拟内存空间，从而实现了用户空间的和内核空间的 `Buffer` 同步操作的功能。而这种同时映射的方法，使得用户空间和内核空间将不需要做数据拷贝了。就是 `Binder` 进程间通信机制的精髓所在了， `Server` 进程内核空间会将 `Client` 进程的数据从用户空间拷贝到内核空间，进程间仅仅需要一次数据拷贝，大大提高了通信效率。




```
// ProcessState.cpp
// `Binder` 进程大小不超过 `1M` 中分析，构造函数中调用了 mmap
// driver binder.c
// 最终会调用驱动的 binder_mmap
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
{
    ...
    // 进程 4M 保护，实际用户空间只会申请 1M
    if ((vma->vm_end - vma->vm_start) > SZ_4M)
        vma->vm_end = vma->vm_start + SZ_4M;
    ...
```

handle 句柄的生成

服务注册时（addService），在驱动中生成对应的 binder_node 实体对象，以及 binder_ref 引用对象，此时会根据红黑树来生成对应的 handle 值。




```
// 用户调用到驱动的流程
// ioctl -> binder_ioctl -> binder_ioctl_write_read ->
// binder_thread_write -> binder_transaction
static void binder_transaction(struct binder_proc *proc,
                   struct binder_thread *thread,
                   struct binder_transaction_data *tr, int reply)
{
    ...
```

```
        switch (fp->type) {
        case BINDER_TYPE_BINDER:
        case BINDER_TYPE_WEAK_BINDER: {
            // 定义实体和引用对象
            struct binder_ref *ref;
            struct binder_node *node = binder_get_node(proc, fp->binder);

            if (node == NULL) {
                // 创建一个 Binder 实体对象 node
                node = binder_new_node(proc, fp->binder, fp->cookie);
                ...
            }
            ...
            // 创建一个 Binder 引用对象
            ref = binder_get_ref_for_node(target_proc, node);
            ...
            if (fp->type == BINDER_TYPE_BINDER)
                // 修改结构体fp的类型
                // 当驱动将进程间数据传递到目标进程时，进程间通信
                // 数据中的 Binder 实体对象就变成了Binder引用对象
                fp->type = BINDER_TYPE_HANDLE;
            else
                fp->type = BINDER_TYPE_WEAK_HANDLE;
            fp->binder = 0;
            // 句柄赋值
            fp->handle = ref->desc;
            fp->cookie = 0;
            ...
        } break;
        ...
}

// 创建引用对象
static struct binder_ref *binder_get_ref_for_node(struct binder_proc *proc,
                          struct binder_node *node)
{
    ...
    // 首先判断是否已经在目标进程 proc 中为 Binder 实体对象
    // 创建过一个 Binder 引用对象
    while (*p) {
        parent = *p;
        ref = rb_entry(parent, struct binder_ref, rb_node_node);

        if (node < ref->node)
            p = &(*p)->rb_left;
        else if (node > ref->node)
            p = &(*p)->rb_right;
```

```
            else
                return ref;
        }
        // 为 proc 创建一个 Binder 引用对象 new_ref
        new_ref = kzalloc(sizeof(*ref), GFP_KERNEL);
        ...
        // 指向 Binder 实体对象
        new_ref->node = node;
        ...
        // 为新创建的 Binder 引用对象 new_ref 分配句柄值
        // 检查是否引用了 service manager 的 Binder 实体对象 binder_context_mgr_node
        new_ref->desc = (node == binder_context_mgr_node) ? 0 : 1;
        for (n = rb_first(&proc->refs_by_desc); n != NULL; n = rb_next(n)) {
            // 在 proc 中找到一个未使用的最小句柄值
            // 作为新创建的 Binder 引用对象 new_ref 的句柄值
            ref = rb_entry(n, struct binder_ref, rb_node_desc);
            if (ref->desc > new_ref->desc)
                break;
            new_ref->desc = ref->desc + 1;
        }
        // 至此句柄创建完毕!
        ...
        return new_ref;
    }
```

> 这里我们也可以看到，当 binder_node 节点为上下文大管家对象 binder_context_mgr_node 时，句柄赋值为 0 时。也就解释了为什么 IServiceManager::defaultServiceManager 对应 0 的问题了。

查询服务时（ getService ），得到的是句柄，在 IServiceManager::defaultServiceManager 的分析中，可以看到 ProcessState::getStrongProxyForHandle 会通过句柄初始化一个 BpBinder 返回给客户端。

参考文档

- 设计篇
- Android深入浅出之Binder机制
- 深入理解Binder机制
- 老罗Binder机制
- **Binder机制系统介绍**
- **红茶一杯话Binder**
- **Binder机制常见问题**

- Android Binder 进程间通讯机制
- Binder学习指南
- Binder框架 – 用户空间和驱动的交互
- Parcel数据打包
- Parcel数据传输
- Android - Binder驱动

Android # IPC # Binder

‹ Java Socket NIO

Java HashMap 简介 ›