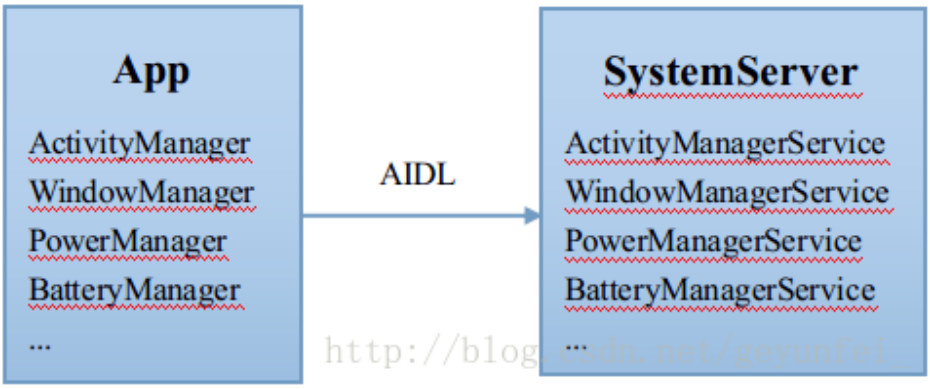
Android系统服务(SystemService)简介

什么是SystemService

我们在Android开发过程中经常会用到各种各样的系统管理服务，如进行窗口相关的操作会用到窗口管理服务WindowManager，进行电源相关的操作会用到电源管理服务PowerManager，还有很多其他的系统管理服务，如通知管理服务NotifacationManager、振动管理服务Vibrator、电池管理服务BatteryManager.....
这些Manager提供了很多对系统层的控制接口。对于App开发者，只需要了解这些接口的使用方式就可以方便的进行系统控制，获得系统各个服务的信息，而不需要了解这些接口的具体实现方式。而对于Framework开发者，则需要了解这些Manager服务的常用实现模式，维护这些Manager的接口，扩展这些接口，或者实现新的Manager。



一个简单的SystemService

我们从一个简单的系统服务Vibrator服务来看一下一个系统服务是怎样建立的。
Vibrator服务提供的控制手机振动的接口，应用可以调用Vibrator的接口来让手机产生振动，达到提醒用户的目的。
从Android的官方文档中可以看到Vibrator只是一个抽象类，只有4个抽象接口：

- `abstract void cancel()` 取消振动
- `abstract boolean hasVibrator()` 是否有振动功能
- `abstract void vibrate(long[] pattern, int repeat)` 按节奏重复振动
- `abstract void vibrate(long milliseconds)` 持续振动

应用中使用振动服务的方法也很简单，如让手机持续振动500毫秒：

```
Vibrator mVibrator = (Vibrator)
getSystemService(Context.VIBRATOR_SERVICE);
mVibrator.vibrate(500);
```

Vibrator使用起来很简单，我们再来看一下实现起来是不是也简单。
从文档中可以看到Vibrator只是定义在android.os
包里的一个抽象类，在源码里的位置即*frameworks/base/core/java/android/os/Vibrator.java*，那么应用中实际使用的是哪个实例呢？应用中使用的Vibrator实例是通过Context的一个方法getSystemService(Context.VIBRATOR_SERVICE)获得的，而Context的实现一般都在ContextImpl中，那我们就看一下ContextImpl是怎么实现getSystemService的：
frameworks/base/core/java/android/app/ContextImpl.java

```
@Override
public Object getSystemService(String name) {
    return SystemServiceRegistry.getSystemService(this, name);
}
```

frameworks/base/core/java/android/app/SystemServiceRegistry.java
(SystemServiceRegistry是 Android 6.0之后才有的，Android 6.0
之前的代码没有该类，下面的代码是直接写在ContextImpl里的)

```
 public static Object getSystemService(ContextImpl ctx, String
name) {
    ServiceFetcher<?> fetcher =
SYSTEM_SERVICE_FETCHERS.get(name);
    return fetcher != null ? fetcher.getService(ctx) : null;
}
```

SYSTEM_SERVICE_MAP是一个HashMap，通过我们服务的名字name字符串，从这个HashMap里取出一个ServiceFetcher，再return这个ServiceFetcher的getService()。
ServiceFetcher是什么？它的getService()又是什么？既然他是从SYSTEM_SERVICE_MAP这个HashMap里get出来的，那就找一找这个HashMap都put了什么。
通过搜索SystemServiceRegistry可以找到如下代码：

```
private static <T> void registerService(String serviceName,
Class<T> serviceClass,
        ServiceFetcher<T> serviceFetcher) {
    SYSTEM_SERVICE_NAMES.put(serviceClass, serviceName);
    SYSTEM_SERVICE_FETCHERS.put(serviceName, serviceFetcher);
}
```

这里往SYSTEM_SERVICE_MAP里put了一对String与ServiceFetcher组成的key/value对，registerService()又是从哪里调用的？继续搜索可以发现很多类似下面的代码：

```
static {
```

```
    registerService(Context.ACCESSIBILITY_SERVICE,
AccessibilityManager.class,
            new CachedServiceFetcher<AccessibilityManager>() {
        @Override
        public AccessibilityManager createService(ContextImpl
ctx) {
            return AccessibilityManager.getInstance(ctx);
        }});

    ...
    registerService(Context.VIBRATOR_SERVICE, Vibrator.class,
            new CachedServiceFetcher<Vibrator>() {
        @Override
        public Vibrator createService(ContextImpl ctx) {
            return new SystemVibrator(ctx);
        }});
    ...
}
```

SystemServiceRegistry的static代码块里通过registerService注册了很多的系统服务，其中就包括我们正在调查的VIBRATOR_SERVICE，通过结合上面的分析代码可以可以知道getSystemService(Context.VIBRATOR_SERVICE)得到的是一个SystemVibrator的实例，通过查看SystemVibrator的代码也可以发现SystemVibrator确实是继承自Vibrator：

```
public class SystemVibrator extends Vibrator {
    ...
}
```

我们再从SystemVibrator看一下系统的振动控制是怎么实现的。以hasVibrator()为例，这个是查询当前系统是否能够振动，在SystemVibrator中它的实现如下：

```
public boolean hasVibrator() {
    ...
    try {
        return mService.hasVibrator();
    } catch (RemoteException e) {
    }
    ...
}
```

这里直接调用了一个mService.hasVibrator()。mService是什么？哪来的？搜索一下可以发现：

```
private final IVibratorService mService;
public SystemVibrator() {
```

```
    ...
    mService = IVibratorService.Stub.asInterface(
            ServiceManager.getService("vibrator"));
}
```

mService
是一个IVibratorService，我们先不去管IVibratorService.Stub.asInterface是怎么回事，先看一下IVibratorService是什么。搜索一下代码发现这并不是一个java文件，而是一个aidl文件：
frameworks/base/core/java/android/os/IVibratorService.aidl
AIDL (Android Interface Definition Language)
是Android中的接口定义文件，为系统提供了一种简单跨进程通信方法。
IVibratorService
中定义了几个接口，SystemVibrator中使用的也是这几个接口，包括我们刚才使用的hasVibrator()

```
interface IVibratorService
{
    boolean hasVibrator();
    void vibrate(...);
    void vibratePattern(...);
    void cancelVibrate(IBinder token);
}
```

这里又只是接口定义，接口实现在哪呢？通过在*frameworks/base*目录下进行grep搜索，或者在AndroidXRef搜索，可以发现IVibratorService接口的实现在*frameworks/base/services/java/com/android/server/VibratorService.java*

```
public class VibratorService extends IVibratorService.Stub
```

可以看到
VibratorService实现了IVibratorService定义的所有接口，并通过JNI调用到native层，进行更底层的实现。更底层的实现不是这篇文档讨论的内容，我们需要分析的是VibratorService怎么成为系统服务的。那么VibratorService是怎么注册为系统服务的呢？在SystemServer里面：

```
VibratorService vibrator = null;
...
//实例化VibratorService并添加到ServiceManager
traceBeginAndSlog("StartVibratorService");
vibrator = new VibratorService(context);
ServiceManager.addService("vibrator", vibrator);
Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);
...
//通知服务系统启动完成
```

```
Trace.traceBegin(Trace.TRACE_TAG_SYSTEM_SERVER,
"MakeVibratorServiceReady");
try {
    vibrator.systemReady();
} catch (Throwable e) {
    reportWtf("making Vibrator Service ready", e);
}
Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);
```

这样在SystemVibrator里就可以通过下面的代码连接到VibratorService，与底层的系统服务进行通信了：

```
IVibratorService.Stub.asInterface(ServiceManager.getService("vib
rator"));
```

mService相当于IVibratorService在应用层的一个代理，所有的实现还是在SystemServer的VibratorService里。
看代码时可以发现registerService是在static代码块里静态调用的，所以getSystemServcr获得的各个Manager也都是单例的。

System Service实现流程

从上面的分析，我们可以总结出Vibrator服务的整个实现流程：

1. 定义一个抽象类Vibrator，定义了应用中可以访问的一些抽象方法
   frameworks/base/core/java/android/os/Vibrator.java
2. 定义具体的类SystemVibrator继承Vibrator，实现抽象方法
   frameworks/base/core/java/android/os/SystemVibrator.java
3. 定义一个AIDL接口文件IVibratorService，定义系统服务接口
   frameworks/base/core/java/android/os/IVibratorService.aidl
4. 定义服务VibratorService，实现IVibratorService定义的接口
   frameworks/base/services/java/com/android/server/VibratorService.java

   ```
   public class VibratorService extends IVibratorService.Stub
   ```

5. 将VibratorServicey添加到系统服务
   frameworks/base/services/java/com/android/server/SystemServer.java

```
6. VibratorService vibrator = null;
7. ...
8. //实例化VibratorService并添加到ServiceManager
9. Slog.i(TAG, "Vibrator Service");
10. vibrator = new VibratorService(context);
11. ServiceManager.addService("vibrator", vibrator);
12. ...
13. //通知服务系统启动完成
```

```
14. try {
15.     vibrator.systemReady();
16. } catch (Throwable e) {
17.     reportWtf("making Vibrator Service ready", e);
  }
```

18. 在SystemVibrator中通过IVibratorService的代理连接到VibratorService，这样SystemVibrator的接口实现里就可以调用IVibratorService的接口：
frameworks/base/core/java/android/os/SystemVibrator.java

```
19. private final IVibratorService mService;
20. ...
21. public SystemVibrator() {
22.     ...
23.     mService = IVibratorService.Stub.asInterface(
24.             ServiceManager.getService("vibrator"));
25.     ...
26.     public boolean hasVibrator() {
27.         ...
28.         try {
29.             return mService.hasVibrator();
30.         } catch (RemoteException e) {
31.         }
32.         ...
33.     }
  }
```

34. 在Context里定义一个代表Vibrator服务的字符串
frameworks/base/core/java/android/content/Context.java

```
  public static final String VIBRATOR_SERVICE = "vibrator";
```

35. 在ContextImpl里添加SystemVibrator的实例化过程
frameworks/base/core/java/android/app/ContextImpl.java

```
36. registerService(VIBRATOR_SERVICE, new ServiceFetcher() {
37. public Object createService(ContextImpl ctx) {
38.     return new SystemVibrator(ctx);
  }});
```

39. 在应用中使用Vibrator的接口

```
40. Vibrator mVibrator = (Vibrator)
  getSystemService(Context.VIBRATOR_SERVICE);
  mVibrator.vibrate(500);
```

41. 为保证编译正常，还需要将AIDL文件添加到编译配置里
frameworks/base/Android.mk

```
42. LOCAL_SRC_FILES += \
43. ...
```

```
    core/java/android/os/IVibratorService.aidl \
```

System Service 新加接口

如果我们需要实现一个新的系统服务，就可以按照上面的步骤在系统中扩展出一个新的服务，并给应用层提供出使用接口。如果想在Vibrator里添加一个新的接口，需要下面3步：

1. 在IVibratorService添加接口；
2. 在VibratorService添加接口的实现；
3. 在Vibrator及SystemVibrator里扩展新的接口；

这样应用中就可以使用Vibrator的新接口了。

应用层与 System Service 通信

上面的实现我们看到的只是从应用层通过服务代理，调用系统服务的接口，如果我们想反过来，将系统服务的状态通知给应用层，该怎么做呢？

- 方法一：使用Broadcast

我们知道使用Broadcast广播可以实现跨进程的消息传递，一些系统服务也使用了这种方法。如电池管理服务BatteryManagerService，收到底层上报的电池状态变化信息时，就将当前的电池状态封装在一个Intent里，action为android.intent.action.BATTERY_CHANGED。应用只要注册一个对应的BroadcastReceiver就可以收到BatterManagerService发送的电池状态信息。

- 方法二：使用AIDL
  从上面我们可以知道，通过AIDL定义一套接口，由系统服务端实现这些接口，应用端使用一个相应的代理就可以访问系统服务的接口，那反过来让应用端实现AIDL接口，系统服务端使用代理调用应用端的接口可不可以呢？答案是YES。那么接下来的问题是怎么让系统服务得到这个代理。我们再来看一个LocationManager的例子。

LocationManager是系统的定位服务，应用通过LocationManager可以获得设备当前的地理位置信息。下面是LocationManager的使用代码片段：

```
//获得定位服务
LocationManager locationManager =
        (LocationManager)
getSystemService(Context.LOCATION_SERVICE);

//定义定位监听器
LocationListener locationListener = new LocationListener() {
```

```
    public void onLocationChanged(Location location) {
        //监听到位置信息
    }
    ...
};

//注册监听器
locationManager.requestLocationUpdates(LocationManager.NETWORK_P
ROVIDER,
        0, 0, locationListener);
```

从上面的代码可以看到，我们创建了一个位置监听器LocationListener，并将这个监听器在LocationManager里进行了注册。当系统定位到系统的位置后，就会回调监听器的onLocationChanged()，将位置信息通知给监听器。LocationListener就是一个系统服务调用应用层接口的例子，我们就研究一下LocationListener的实现方式。
我们先从LocationManager怎么注册LocationListener开始研究：
frameworks/base/location/java/android/location/LocationManager.java

```
private final ILocationManager mService;
...
private void requestLocationUpdates(LocationRequest request,
        LocationListener listener, Looper looper, PendingIntent
intent) {
    ...
    // wrap the listener class
    ListenerTransport transport = wrapListener(listener,
looper);
    try {
        mService.requestLocationUpdates(request, transport,
                intent, packageName);
   } catch (RemoteException e) {
       Log.e(TAG, "RemoteException", e);
   }
}
```

可以看到LocationListener被重新封装成了一个ListenerTransport，然后传递给了ILocationManager
，从前面的分析可以猜测到这个ILocationManager应该就是LocationManagerService的一个代理。那么ListenerTransport又是什么呢？搜索LocationManager.java可以找到：

```
private class ListenerTransport extends ILocationListener.Stub {
    ...
    @Override
    public void onLocationChanged(Location location) {
        ...
```

```
    }
}
```

原来是ILocationListener.Stub的一个继承实现，那么ILocationListener应该就是一个AIDL接口定义：
frameworks/base/location/java/android/location/ILocationListener.aidl

```
oneway interface ILocationListener
{
    void onLocationChanged(in Location location);
    ...
}
```

而在LocationManagerService里只要调用ILocationListener的方法就可以将消息传递给应用层的监听：

```
mListener.onLocationChanged(new Location(location));
```

实现 **System Service** 的注意事项

1. 注意防止阻塞
   应用层访问系统服务提供的接口时会有两种情况：
   - 一种是应用调用端需要等待服务实现端处理完成，返回处理结果，这样如果服务端发生阻塞，那么应用端也会发生阻塞，因此在实现服务端的实现时要注意不要发生阻塞。
   - 另一种是调用端不需要等待服务端返回结果，调用完成后直接返回void，这样服务端发生阻塞不会影响到应用端，这样的单向的接口在AIDL里定义时需要添加oneway关键字，如：

     ```
     oneway void statusBarVisibilityChanged(int visibility);
     ```

     对于需要在服务端调用，在应用端实现的接口，考虑到系统的稳定性以及安全性，一般都会设计成上面的第二种，即AIDL里所有的接口都是单向的，如上面的ILocationListener

     ```
     oneway interface ILocationListener
     ```

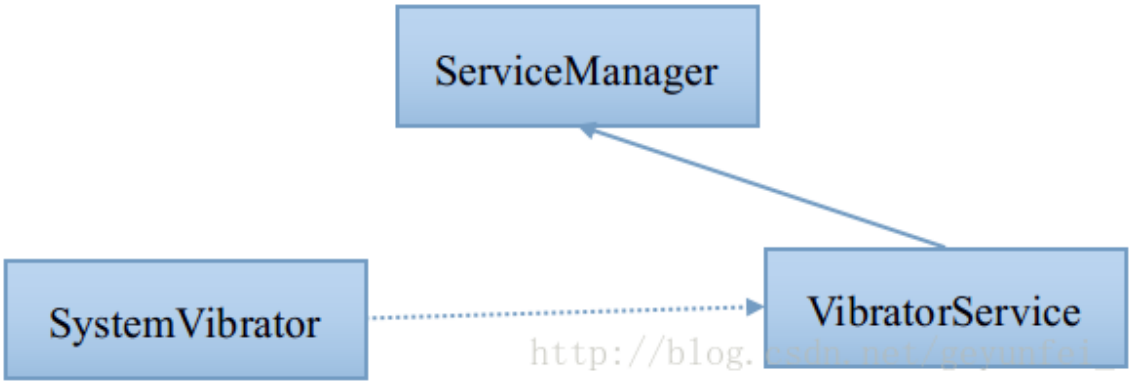
2. 注意多线程访问
   每个系统服务在系统进程中只有一个实例，而且应用中系统服务的代理也是单例的，而且应用端的访问，在系统进程都是使用独立的线程进行响应，所以访问同一个系统服务的接口时必然会出现多个线程或者多个进程同时访问的情况。为保证系统服务的线程安全，需要对系统服务的进程进行多线程访问的保护，目前主要有两种实现线程安全的方法：
   - 一种是通过同步锁机制，锁住一个对象实例（一般是这个服务对象本身），这样这个服务同一时间只能响应一个访问请求，如LocationManagerService里：

```
3. public boolean callStatusChangedLocked(...) {
4.     ...
5.     synchronized (this) {
6.     ...
7.     }
   }
```

- 另一种方法就是使用Handler机制，这种服务一般会创建一个单独的线程，当有应用端访问请求到来时会向服务线程的Handler里发送一个Message，利用单线程顺序执行的特性，保证所有的访问都按顺序进行处理，但这种方法只适合单向的访问，不适合需要返回的双向访问。

ServiceManager原理

从上面流程的分析我们可以看到，所有的系统服务最终都是由ServiceManager来管理的，那么ServiceManager是怎么管理这些系统服务的呢？还是先看上面的VibratorService的实现：



VibratorService通过addService()将自己注册到ServiceManager里，SystemVibrator通过getService()获得一个服务代理，并与服务进行通信交互，那么ServiceManager又是什么？它是怎么管理服务的注册与代理的呢？我们先从addService()与getService()开始，分析一下ServiceManager:

frameworks/base/core/java/android/os/ServiceManager.java

```
public final class ServiceManager {
    public static void addService(String name, IBinder service)
{
        ...
        getIServiceManager().addService(name, service, false);
    }

    public static IBinder getService(String name) {
        ...
        return getIServiceManager().getService(name);
    }
```

```
}
```

addService()和getService()都是直接调用了getIServiceManager()的方法，getIServiceManager()返回的又是什么呢？

```
private static IServiceManager getIServiceManager() {
    if (sServiceManager != null) {
        return sServiceManager;
    }
    // Find the service manager
    sServiceManager =
ServiceManagerNative.asInterface(BinderInternal.getContextObject
());
    return sServiceManager;
}
```

这里是创建了一个IServiceManager类型的单实例，具体的实例又是通过ServiceManagerNative创建的:

```
ServiceManagerNative.asInterface(BinderInternal.getContextObject
());
```

先来看BinderInternal.getContextObject()
frameworks/base/core/java/com/android/internal/os/BinderInternal.java

```
public static final native IBinder getContextObject();
```

getContextObject()是在native层实现的
frameworks/base/core/jni/android_util_Binder.cpp

```
static jobject
android_os_BinderInternal_getContextObject(JNIEnv* env,
        jobject clazz)
{
    sp<IBinder> b = ProcessState::self()-
>getContextObject(NULL);
    return javaObjectForIBinder(env, b);
}
```

后面的实现过程逻辑比较复杂，这里就不详细分析了，只是说明一下大致流程，感兴趣的可以自己详细研究一下binder的实现机制。
ProcessState从名字可以看出这应该是一个保存进程状态的类，最好应该用单实例的对象保存，所以ProcessState::self()得到的也就是ProcessState的一个单实例对象，它的getContextObject()会继续调用getStrongProxyForHandle(int32_t handle)返回一个IBinder
frameworks/native/libs/binder/ProcessState.cpp

```
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>&
caller)
{
    return getStrongProxyForHandle(0);
}
```

在getStrongProxyForHandle()中会根据传入的参数handle创建一个BpBinder，这个BpBinder会保存在一个数组mHandleToObject中，下次再用同样的handle请求时不会再重新创建。由于我们传入的handle=0，这里创建的BpBinder也就相当于第0号BpBinder。之后的javaObjectForIBinder()会将C++的BpBinder对象封装成Java的BinderProxy对象并返回。所以BinderInternal.getContextObject()得到的是一个BinderProxy对象，并关联了native层的第0号BpBinder。

```
ServiceManagerNative.asInterface(BinderInternal.getContextObject
());
```

相当于

```
ServiceManagerNative.asInterface(new BinderProxy());
```

ServiceManagerNative.asInterface()又做了些什么呢？
frameworks/base/core/java/android/os/ServiceManagerNative.java

```
static public IServiceManager asInterface(IBinder obj)
{
    ...
    return new ServiceManagerProxy(obj);
}
```

这里会将BinderProxy再封装成一个ServiceManagerProxy()，所以getIServiceManager()得到的其实是一个ServiceManagerProxy，但是底层指向的是一个BpBinder(0)。
ServiceManagerProxy、BinderProxy以及BpBinder都是代理模式中的proxy端，真正的实现应该在对应的native端。我们接着看。
addService()和getService()在代理端的实现应该是在ServiceManagerProxy()里：

```
public IBinder getService(String name) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IServiceManager.descriptor);
    data.writeString(name);
    mRemote.transact(GET_SERVICE_TRANSACTION, data, reply, 0);
    IBinder binder = reply.readStrongBinder();
    reply.recycle();
    data.recycle();
```

```
    return binder;
}
...
public void addService(String name, IBinder service, boolean
allowIsolated)
        throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IServiceManager.descriptor);
    data.writeString(name);
    data.writeStrongBinder(service);
    data.writeInt(allowIsolated ? 1 : 0);
    mRemote.transact(ADD_SERVICE_TRANSACTION, data, reply, 0);
    reply.recycle();
    data.recycle();
}
```

从上面两个方法的实现可以看到，首先是创建了两个可序列化的Parcel data、reply，传入的参数被放到了data里，data、reply又一起传给了mRemote.transact()，之后又从reply里读取结果。addService()的实现里还通过data.writeStrongBinder(service)写入了一个IBinder的实例。同时注意到getService()和addService()里面调用mRemote.transact()传递的第一个参数分别为GET_SERVICE_TRANSACTION 和 ADD_SERVICE_TRANSACTION，我们可以在IServiceManager里看到这是两个int值，分别为1和3

frameworks/base/core/java/android/os/IServiceManager.java

```
int GET_SERVICE_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION;
//值为1
int CHECK_SERVICE_TRANSACTION =
IBinder.FIRST_CALL_TRANSACTION+1;
int ADD_SERVICE_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION+2;
```

mRemote就是BinderProxy，真正的实现是C++里的BpBinder

frameworks/native/libs/binder/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) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }
```

```
    return DEAD_OBJECT;
}
```

IPCThreadState里的transact会将proxy端的请求通过binder机制写入到一块共享内存里，然后会有一个native端从共享内存里读出请求，并执行相应的操作。那么这个native端在哪里呢？是不是ServiceManagerNative呢？但是ServiceManagerNative只是一个抽象类，我们并没有找到继承自它的子类。实际上ServiceManagerNative只是架构方面的一个抽象定义，并没有真正的一个实现，真正实现ServiceManager
的native端功能的是在native层的service_manager.c中。
servicemanager是底层的一个独立进程，主要的实现代码就在service_manager.c中。
frameworks/native/cmds/servicemanager/binder.h

```
enum {
    /* Must match definitions in IBinder.h and IServiceManager.h
*/
    PING_TRANSACTION  = B_PACK_CHARS('_','P','N','G'),
    SVC_MGR_GET_SERVICE = 1,
    SVC_MGR_CHECK_SERVICE,
    SVC_MGR_ADD_SERVICE,
    SVC_MGR_LIST_SERVICES,
};
```

frameworks/native/cmds/servicemanager/service_manager.c

```
#include "binder.h"
...
int svcmgr_handler(struct binder_state *bs,
                   struct binder_transaction_data *txn,
                   struct binder_io *msg,
                   struct binder_io *reply)
{
    ...
    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;
        ...
    }
}
```

service_manager.c的svcmgr_handler函数就是监听代理端请求命令的txn->code就是mRemote.transact()里传过来的第一个参数。SVC_MGR_GET_SERVICE和SVC_MGR_ADD_SERVICE是在头文件binder.h里定义的，它们的值与IServiceManager.java里定义的一致，也是1和3。
我们先看SVC_MGR_ADD_SERVICE的响应：

- 首先通过s = bio_get_string16(msg, &len)获得了service的名称，
- 然后通过handle = bio_get_ref(msg)获得了一个handle，这个handle就是我们之前通过writeStrongBinder写入的IBinder，
- 最后通过do_add_service()添加注册service

do_add_service()的实现如下：

```
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;
    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));
        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;
    }
    ...
}

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;
}
```

- 首先声明了一个struct svcinfo *si
- 通过find_svc(s, len)查找一下同样名称的service之前是不是注册过，防止重复注册。find_svc()的实现里可以看到是通过遍历一个svclist链表来查重的，svclist链表就是用来保存所有注册过的service的
- 如果确认没有重复注册service，就重新构造一个svcinfo添加到svclist链表的头部。

我们再看SVC_MGR_GET_SERVICE的响应，主要是通过do_find_service()查找到对应的service，并通过bio_put_ref(reply, handle)将查找到的handle返回。do_find_service()的实现主要也是通过find_svc()去svclist链表中查找

```
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;
```

}

通过上面的流程梳理我们最终了解到：

- 每个System Service通过调用`ServiceManager.addService()`将自己的名字以及`IBinder`引用保存到servicemanager进程的一个链表里
- 每个使用该System Service的进程通过调用`ServiceManager.getService()`从servicemanager进程获得该System Service对应的`IBinder`，就可以与该System Service进行通信了。