Android源码学习之六——ActivityManager框架解析

标签： 框架 android null token list system

2010-11-26 10:29 39076人阅读 评论(15) 收藏 举报

分类：

Android（21）

目录(?) [+]

ActivityManager在操作系统中有重要的作用，本文利用操作系统源码，逐步理清ActivityManager的框架，并从静态类结构图和动态序列图两个角度分别进行剖析，从而帮助开发人员加强对系统框架及进程通信机制的理解。

ActivityManager的作用

参照SDK的说明，可见ActivityManager的功能是与系统中所有运行着的Activity交互提供了接口，主要的接口围绕着运行中的进程信息，任务信息，服务信息等。比如函数getRunningServices()的源码是：

```
public List<RunningServiceInfo> getRunningServices(int maxNum)
        throws SecurityException {
    try {
        return (List<RunningServiceInfo>)ActivityManagerNative.getDefault()
                .getServices(maxNum, 0);
    } catch (RemoteException e) {
        // System dead, we will be dead too soon!
        return null;
    }
}
```

从中可以看到，ActivityManager的大多数功能都是调用了ActivityManagerNative类接口来完成的，因此，我们寻迹来看ActivityManagerNative的代码，并以此揭示ActivityManager的整体框架。

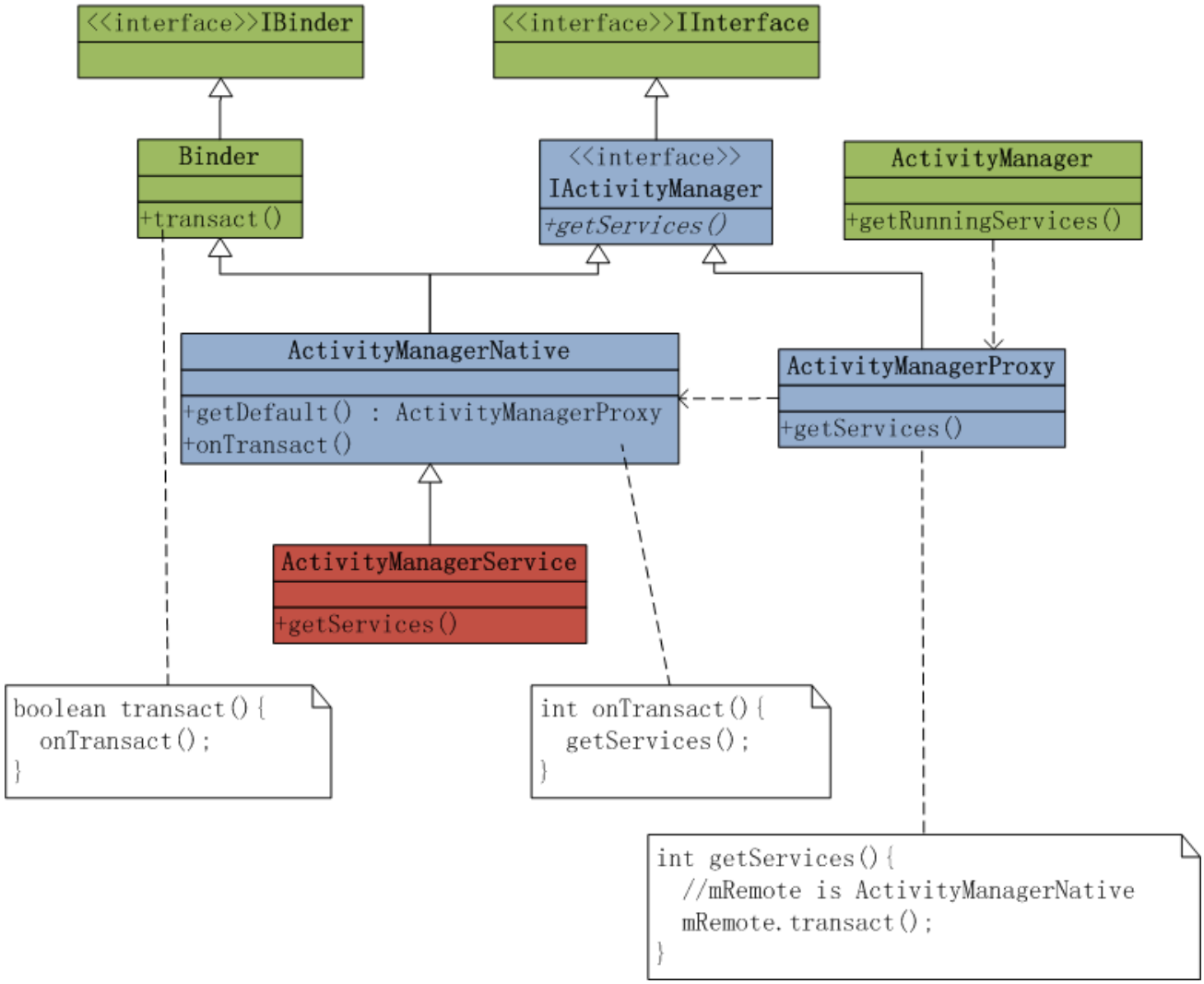
ActivityManager的静态类图

通过源吗，可以发现ActivityManagerNative类的继承关系如下：

public abstract class ActivityManagerNative extends Binder implements IActivityManager

继承自Binder类，同时实现了IActivityManager接口。

同样的，我们继续沿Binder和IActivityManager上溯，整理出如下图所示的类结构图。



在这张图中，绿色的部分是在SDK中开放给应用程序开发人员的接口，蓝色的部分是一个典型的Proxy模式，红色的部分是底层的服务实现，是真正的动作执行者。这里的一个核心思想是Proxy模式，我们接下来对此模式加以介绍。

Proxy模式

Proxy模式，也称代理模式，是经典设计模式中的一种结构型模式，其定义是为其他对象提供一种代理以控制对这个对象的访问，简单的说就是在访问和被访问对象中间加上的一个间接层，以隔离访问者和被访问者的实现细节。

结合上面的类结构图，其中ActivityManager是一个客户端，为了隔离它与ActivityManagerService，有效降低甚至消除二者的耦合度，在这中间使用了ActivityManagerProxy代理类，所有对ActivityManagerService的访问都转换成对代理类的访问，这样ActivityManager就与ActivityManagerService解耦了。这就是代理模式的典型应用场景。

为了让代理类与被代理类保持一致的接口，从而实现更加灵活的类结构，或者说完美的屏蔽实现细节，通常的作法是让代理类与被代理类实现一个公共的接口，这样对调用者来说，无法知道被调用的是代理类还是直接是被代理类，因为二者的接口是相同的。

这个思路在上面的类结构图里也有落实，IActivityManager接口类就是起的这个作用。

以上就是代理模式的思路，有时我们也称代理类为本地代理（Local Proxy），被代理类为远端代理（Remote Proxy）。

本地代理与远端代理的Binder

我们再来看一下Binder类的作用，Binder的含义可能译为粘合剂更为贴切，即将两侧的东西粘贴起来。在操作系统中，Binder的一大作用就是连接本地代理和远端代理。Binder中最重要的一个函数是：

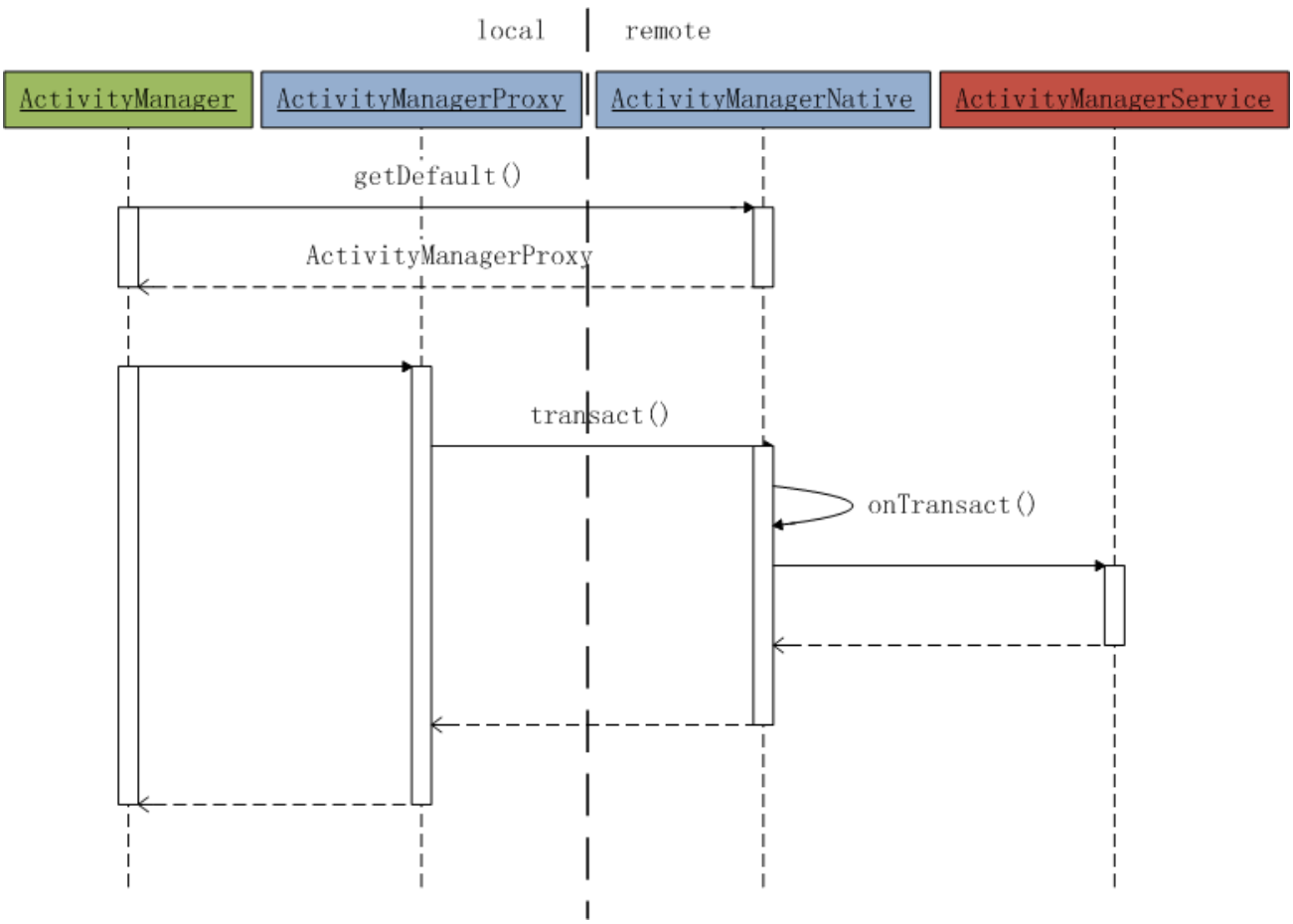
```
public final boolean transact(int code, Parcel data, Parcel reply,
        int flags) throws RemoteException {
            ......
    boolean r = onTransact(code, data, reply, flags);
    if (reply != null) {
        reply.setDataPosition(0);
    }
    return r;
}
```

它的作用就在于通过code来表示请求的命令标识，通过data和reply进行数据传递，只要远端代理能实现onTransact()函数，即可做出正确的动作，远端的执行接口被完全屏蔽了。

当然，Binder的实现还是很复杂的，不仅是类型转换，还要透过Binder驱动进入KERNEL层来完成进程通信，这些内容不在本文的范围之内，故此处不再深入解析相应的机制。此处我们只要知道Binder的transact()函数实现就可以了。

到此为止，我们对ActivityManager的静态类结构就分析完了，但这还不足以搞清在系统运行中的调用过程，因此，我们以下图的序列图为基础，结合源码探索一下ActivityManager运行时的机制。

动态序列图



我们以ActivityManager的getRunningServices()函数为例，对上述序列图进行解析。

```
public List<RunningServiceInfo> getRunningServices(int maxNum)
        throws SecurityException {
    try {
        return (List<RunningServiceInfo>)ActivityManagerNative.getDefault()
                .getServices(maxNum, 0);
    } catch (RemoteException e) {
        // System dead, we will be dead too soon!
        return null;
    }
}
```

可以看到，调用被委托到了ActivatyManagerNative.getDefault()。

```
static public IActivityManager asInterface(IBinder obj)
    {
                ……
    return new ActivityManagerProxy(obj);
}

static public IActivityManager getDefault()
    {
            ……
    IBinder b = ServiceManager.getService("activity");
    gDefault = asInterface(b);
    return gDefault;
}
```

从上述简化后的源码可以看到，getDefault()函数返回的是一个ActivityManagerProxy对象的引用，也就是说，ActivityManager得到了一个本地代理。

因为在IActivityManager接口中已经定义了getServices()函数，所以我们来看这个本地代理对该函数的实现。

```
public List getServices(int maxNum, int flags) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
            ……
    mRemote.transact(GET_SERVICES_TRANSACTION, data, reply, 0);
    ……
}
```

从这个代码版段我们看到，调用远端代理的transact()函数，而这个mRemote就是ActivityManagerNative的Binder接口。

接下来我们看一下ActivityManagerNative的代码，因为该类是继承于Binder类的，所以transact的机制此前我们已经展示了代码，对于该类而言，重要的是对onTransact()函数的实现。

```
public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
        throws RemoteException {
    switch (code) {
    case GET_SERVICES_TRANSACTION: {
                ……
        List list = getServices(maxNum, fl);
                ……
        return true;
    }
            ……
    }
    return super.onTransact(code, data, reply, flags);
}
```

在onTrasact()函数内，虽然代码特别多，但就是一个switch语句，根据不同的code命令进行不同的处理，比如对于GET_SERVICES_TRANSACTION命令，只是调用了getServices()函数。而该函数的实现是在ActivityManagerService类中，它是ActivityManagerNative的子类，对于该函数的实现细节，不在本文中详细分析。

Activity启动

在经过前文的学习以后，我们一起来整理一下Activity的启动机制。就从Activity的startActivity()函数开始吧。

startActivity()函数调用了startActivityForResult()函数，该函数有源码如下：

```
public void startActivityForResult(Intent intent, int requestCode) {
    ……
        Instrumentation.ActivityResult ar =
            mInstrumentation.execStartActivity(
                this, mMainThread.getApplicationThread(), mToken, this,
                intent, requestCode);
                ……
    }
```

可见，功能被委托给Instrumentation对象来执行了。这个类的功能是辅助Activity的监控和测试，在此我们不详细描述，我们来看它的execStartActivity()函数。

```
public ActivityResult execStartActivity(
    Context who, IBinder contextThread, IBinder token, Activity target,
    Intent intent, int requestCode) {
```

```
            ……
        try {
            int result = ActivityManagerNative.getDefault()
                .startActivity(whoThread, intent,
                        intent.resolveTypeIfNeeded(who.getContentResolver()),
                        null, 0, token, target != null ? target.mEmbeddedID : null,
                        requestCode, false, false);
            checkStartActivityResult(result, intent);
        } catch (RemoteException e) {
        }
        return null;
    }
```

在这个函数里，我们看到了前文熟悉的ActivityManagerNative.getDefault()，没错，利用了ActivityManagerService。通过前文的线索，利用Proxy模式，我们可以透过ActivityManagerProxy，通过Binder的transact机制，找到真正的动作执行者，即ActivityManagerService类的startActivity()函数，并沿此线索继续追踪源码，在startActivityLocked()函数里边看到了mWindowManager.setAppStartingWindow的语句调用，mWindowManager是WindowManagerService对象，用于负责界面上的具体窗口调试。

通过这样的源码追踪，我们了解到了Activity启动的底层实现机制，也加深了对Proxy模式和Binder机制的理解。从而为学习其他框架打下了基础。

总结

本文从静态类结构和动态类结构两个角度分析了ActivityManager的框架，兼顾了Binder机制和代理模式在进程间通信的机理，对帮助开发人员深化操作系统的结构和框架具有一定的指导作用。