# Android插件化原理解析——Hook机制之Binder Hook

发表于    |   [60](http://weishu.me/2016/02/16/understand-plugin-framework-binder-hook/" \l "comments)条评论   |   26012次阅读

Android系统通过Binder机制给应用程序提供了一系列的系统服务，诸如ActivityManagerService，ClipboardManager， AudioManager等；这些广泛存在系统服务给应用程序提供了诸如任务管理，音频，视频等异常强大的功能。

插件框架作为各个插件的管理者，为了使得插件能够**无缝地**使用这些系统服务，自然会对这些系统服务做出一定的改造(Hook)，使得插件的开发和使用更加方便，从而大大降低插件的开发和维护成本。比如，Hook住ActivityManagerService可以让插件无缝地使用startActivity方法而不是使用特定的方式(比如that语法)来启动插件或者主程序的任意界面。

我们把这种Hook系统服务的机制称之为Binder Hook，因为本质上这些服务提供者都是存在于系统各个进程的Binder对象。因此，要理解接下来的内容必须了解Android的Binder机制，可以参考我之前的文章[Binder学习指南](http://weishu.me/2016/01/12/binder-index-for-newer/)

阅读本文之前，可以先clone一份 [understand-plugin-framework](https://github.com/tiann/understand-plugin-framework)，参考此项目的binder-hook 模块。另外，插件框架原理解析系列文章见[索引](http://weishu.me/2016/01/28/understand-plugin-framework-overview/)。

## **系统服务的获取过程**

我们知道系统的各个远程service对象都是以Binder的形式存在的，而这些Binder有一个管理者，那就是ServiceManager；我们要Hook掉这些service，自然要从这个ServiceManager下手，不然星罗棋布的Binder广泛存在于系统的各个角落，要一个个找出来还真是大海捞针。

回想一下我们使用系统服务的时候是怎么干的，想必这个大家一定再熟悉不过了：通过Context对象的getSystemService方法；比如要使用ActivityManager：

|  |  |
| --- | --- |
| 1 | ActivityManager am = (ActivityManager) context.getSystemService(Context.ACTIVITY\_SERVICE); |

可是这个貌似跟ServiceManager没有什么关系啊？我们再查看getSystemService方法；(Context的实现在ContextImpl里面)：

|  |  |
| --- | --- |
| 1  2  3  4 | public Object getSystemService(String name) {  ServiceFetcher fetcher = SYSTEM\_SERVICE\_MAP.get(name);  return fetcher == null ? null : fetcher.getService(this);  } |

很简单，所有的service对象都保存在一张map里面，我们再看这个map是怎么初始化的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | registerService(ACCOUNT\_SERVICE, new ServiceFetcher() {  public Object createService(ContextImpl ctx) {  IBinder b = ServiceManager.getService(ACCOUNT\_SERVICE);  IAccountManager service = IAccountManager.Stub.asInterface(b);  return new AccountManager(ctx, service);  }}); |

在ContextImpl的静态初始化块里面，有的Service是像上面这样初始化的；可以看到，确实使用了ServiceManager；当然还有一些service并没有直接使用ServiceManager，而是做了一层包装并返回了这个包装对象，比如我们的ActivityManager，它返回的是ActivityManager这个包装对象：

|  |  |
| --- | --- |
| 1  2  3  4 | registerService(ACTIVITY\_SERVICE, new ServiceFetcher() {  public Object createService(ContextImpl ctx) {  return new ActivityManager(ctx.getOuterContext(), ctx.mMainThread.getHandler());  }}); |

但是在ActivityManager这个类内部，也使用了ServiceManager；具体来说，因为ActivityManager里面所有的核心操作都是使用ActivityManagerNative.getDefault()完成的。那么这个语句干了什么呢？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | private static final Singleton<IActivityManager> gDefault = new Singleton<IActivityManager>() {  protected IActivityManager create() {  IBinder b = ServiceManager.getService("activity");  IActivityManager am = asInterface(b);  return am;  }  }; |

因此，通过分析我们得知，系统Service的使用其实就分为两步：

|  |  |
| --- | --- |
| 1  2 | IBinder b = ServiceManager.getService("service\_name"); // 获取原始的IBinder对象  IXXInterface in = IXXInterface.Stub.asInterface(b); // 转换为Service接口 |

## **寻找Hook点**

在[插件框架原理解析——Hook机制之动态代理](http://weishu.me/2016/01/28/understand-plugin-framework-proxy-hook/)里面我们说过，Hook分为三步，最关键的一步就是寻找Hook点。我们现在已经搞清楚了系统服务的使用过程，那么就需要找出在这个过程中，在哪个环节是最合适hook的。

由于系统服务的使用者都是对第二步获取到的IXXInterface进行操作，因此如果我们要hook掉某个系统服务，**只需要把第二步的asInterface方法返回的对象修改为为我们Hook过的对象就可以了。**

### **asInterface过程**

接下来我们分析asInterface方法，然后想办法把这个方法的返回值修改为我们Hook过的系统服务对象。这里我们以系统剪切版服务为例，源码位置为android.content.IClipboard,IClipboard.Stub.asInterface方法代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public static android.content.IClipboard asInterface(android.os.IBinder obj) {  if ((obj == null)) {  return null;  }  android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR); // Hook点  if (((iin != null) && (iin instanceof android.content.IClipboard))) {  return ((android.content.IClipboard) iin);  }  return new android.content.IClipboard.Stub.Proxy(obj);  } |

这个方法的意思就是：先查看本进程是否存在这个Binder对象，如果有那么直接就是本进程调用了；如果不存在那么创建一个代理对象，让代理对象委托驱动完成跨进程调用。

观察这个方法，前面的那个if语句判空返回肯定动不了手脚；最后一句调用构造函数然后直接返回我们也是无从下手，要修改asInterface方法的返回值，我们唯一能做的就是从这一句下手：

|  |  |
| --- | --- |
| 1 | android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR); // Hook点 |

我们可以尝试修改这个obj对象的queryLocalInterface方法的返回值，并保证这个返回值符合接下来的if条件检测，那么就达到了修改asInterface方法返回值的目的。

而这个obj对象刚好是我们第一步返回的IBinder对象，接下来我们尝试对这个IBinder对象的queryLocalInterface方法进行hook。

### **getService过程**

上文分析得知，我们想要修改IBinder对象的queryLocalInterface方法；获取IBinder对象的过程如下：

|  |  |
| --- | --- |
| 1 | IBinder b = ServiceManager.getService("service\_name"); |

因此，我们希望能修改这个getService方法的返回值，让这个方法返回一个我们伪造过的IBinder对象；这样，我们可以在自己伪造的IBinder对象的queryLocalInterface方法作处理，进而使得asInterface方法返回在queryLocalInterface方法里面处理过的值，最终实现hook系统服务的目的。

在跟踪这个getService方法之前我们思考一下，由于系统服务是一系列的远程Service，它们的本体，也就是Binder本地对象一般都存在于某个单独的进程，在这个进程之外的其他进程存在的都是这些Binder本地对象的代理。因此在我们的进程里面，存在的也只是这个Binder代理对象，我们也只能对这些Binder代理对象下手。(如果这一段看不懂，建议不要往下看了，先看[Binder学习指南](http://weishu.me/2016/01/12/binder-index-for-newer/))

然后，这个getService是一个静态方法，如果此方法什么都不做，拿到Binder代理对象之后直接返回；那么我们就无能为力了：我们没有办法拦截一个静态方法，也没有办法获取到这个静态方法里面的局部变量(即我们希望修改的那个Binder代理对象)。

接下来就可以看这个getService的代码了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public static IBinder getService(String name) {  try {  IBinder service = sCache.get(name);  if (service != null) {  return service;  } else {  return getIServiceManager().getService(name);  }  } catch (RemoteException e) {  Log.e(TAG, "error in getService", e);  }  return null;  } |

天无绝人之路！ServiceManager为了避免每次都进行跨进程通信，把这些Binder代理对象缓存在一张map里面。

我们可以替换这个map里面的内容为Hook过的IBinder对象，由于系统在getService的时候每次都会优先查找缓存，因此返回给使用者的都是被我们修改过的对象，从而达到瞒天过海的目的。

总结一下，要达到修改系统服务的目的，我们需要如下两步：

1. 首先肯定需要**伪造一个系统服务对象**，接下来就要想办法让asInterface能够返回我们的这个伪造对象而不是原始的系统服务对象。
2. 通过上文分析我们知道，只要让getService返回IBinder对象的queryLocalInterface方法直接返回我们伪造过的系统服务对象就能达到目的。所以，我们需要**伪造一个IBinder对象**，主要是修改它的queryLocalInterface方法，让它返回我们伪造的系统服务对象；然后把这个伪造对象放置在ServiceManager的缓存map里面即可。

我们通过Binder机制的优先查找本地Binder对象的这个特性达到了Hook掉系统服务对象的目的。因此queryLocalInterface也失去了它原本的意义(只查找本地Binder对象，没有本地对象返回null)，这个方法只是一个傀儡，是我们实现hook系统对象的桥梁：我们通过这个“漏洞”让asInterface永远都返回我们伪造过的对象。由于我们接管了asInterface这个方法的全部，我们伪造过的这个系统服务对象不能是只拥有本地Binder对象(原始queryLocalInterface方法返回的对象)的能力，还要有Binder代理对象操纵驱动的能力。

接下来我们就以Hook系统的剪切版服务为例，用实际代码来说明，如何Hook掉系统服务。

## **Hook系统剪切版服务**

### **伪造剪切版服务对象**

首先我们用代理的方式伪造一个剪切版服务对象，关于如何使用代理的方式进行hook以及其中的原理，可以查看[插件框架原理解析——Hook机制之动态代理](http://weishu.me/2016/01/28/understand-plugin-framework-proxy-hook/)。

具体代码如下，我们用动态代理的方式Hook掉了hasPrimaryClip()，getPrimaryClip()这两个方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class BinderHookHandler implements InvocationHandler {  private static final String TAG = "BinderHookHandler";  // 原始的Service对象 (IInterface)  Object base;  public BinderHookHandler(IBinder base, Class<?> stubClass) {  try {  Method asInterfaceMethod = stubClass.getDeclaredMethod("asInterface", IBinder.class);  // IClipboard.Stub.asInterface(base);  this.base = asInterfaceMethod.invoke(null, base);  } catch (Exception e) {  throw new RuntimeException("hooked failed!");  }  }  @TargetApi(Build.VERSION\_CODES.HONEYCOMB)  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  // 把剪切版的内容替换为 "you are hooked"  if ("getPrimaryClip".equals(method.getName())) {  Log.d(TAG, "hook getPrimaryClip");  return ClipData.newPlainText(null, "you are hooked");  }  // 欺骗系统,使之认为剪切版上一直有内容  if ("hasPrimaryClip".equals(method.getName())) {  return true;  }  return method.invoke(base, args);  }  } |

注意，我们拿到原始的IBinder对象之后，如果我们希望使用被Hook之前的系统服务，并不能直接使用这个IBinder对象，而是需要使用asInterface方法将它转换为IClipboard接口；因为getService方法返回的IBinder实际上是一个**裸Binder代理对象**，它只有与驱动打交道的能力，但是它并不能独立工作，需要人指挥它；asInterface方法返回的IClipboard.Stub.Proxy类的对象通过操纵这个裸BinderProxy对象从而实现了具体的IClipboard接口定义的操作。

### **伪造IBinder 对象**

在上一步中，我们已经伪造好了系统服务对象，现在要做的就是想办法让asInterface方法返回我们伪造的对象了；我们伪造一个IBinder对象：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | public class BinderProxyHookHandler implements InvocationHandler {  private static final String TAG = "BinderProxyHookHandler";  // 绝大部分情况下,这是一个BinderProxy对象  // 只有当Service和我们在同一个进程的时候才是Binder本地对象  // 这个基本不可能  IBinder base;  Class<?> stub;  Class<?> iinterface;  public BinderProxyHookHandler(IBinder base) {  this.base = base;  try {  this.stub = Class.forName("android.content.IClipboard$Stub");  this.iinterface = Class.forName("android.content.IClipboard");  } catch (ClassNotFoundException e) {  e.printStackTrace();  }  }  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  if ("queryLocalInterface".equals(method.getName())) {  Log.d(TAG, "hook queryLocalInterface");  // 这里直接返回真正被Hook掉的Service接口  // 这里的 queryLocalInterface 就不是原本的意思了  // 我们肯定不会真的返回一个本地接口, 因为我们接管了 asInterface方法的作用  // 因此必须是一个完整的 asInterface 过的 IInterface对象, 既要处理本地对象,也要处理代理对象  // 这只是一个Hook点而已, 它原始的含义已经被我们重定义了; 因为我们会永远确保这个方法不返回null  // 让 IClipboard.Stub.asInterface 永远走到if语句的else分支里面  return Proxy.newProxyInstance(proxy.getClass().getClassLoader(),  // asInterface 的时候会检测是否是特定类型的接口然后进行强制转换  // 因此这里的动态代理生成的类型信息的类型必须是正确的  new Class[] { IBinder.class, IInterface.class, this.iinterface },  new BinderHookHandler(base, stub));  }  Log.d(TAG, "method:" + method.getName());  return method.invoke(base, args);  }  } |

我们使用动态代理的方式伪造了一个跟原始IBinder一模一样的对象，然后在这个伪造的IBinder对象的queryLocalInterface方法里面返回了我们第一步创建的**伪造过的系统服务对象**；注意看注释，详细解释可以看[代码](https://github.com/tiann/understand-plugin-framework)

### **替换ServiceManager的IBinder对象**

现在就是万事具备，只欠东风了；我们使用反射的方式修改ServiceManager类里面缓存的Binder对象，使得getService方法返回我们伪造的IBinder对象，进而asInterface方法使用伪造IBinder对象的queryLocalInterface方法返回了我们伪造的系统服务对象。代码较简单，如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | final String CLIPBOARD\_SERVICE = "clipboard";  // 下面这一段的意思实际就是: ServiceManager.getService("clipboard");  // 只不过 ServiceManager这个类是@hide的  Class<?> serviceManager = Class.forName("android.os.ServiceManager");  Method getService = serviceManager.getDeclaredMethod("getService", String.class);  // ServiceManager里面管理的原始的Clipboard Binder对象  // 一般来说这是一个Binder代理对象  IBinder rawBinder = (IBinder) getService.invoke(null, CLIPBOARD\_SERVICE);  // Hook 掉这个Binder代理对象的 queryLocalInterface 方法  // 然后在 queryLocalInterface 返回一个IInterface对象, hook掉我们感兴趣的方法即可.  IBinder hookedBinder = (IBinder) Proxy.newProxyInstance(serviceManager.getClassLoader(),  new Class<?>[] { IBinder.class },  new BinderProxyHookHandler(rawBinder));  // 把这个hook过的Binder代理对象放进ServiceManager的cache里面  // 以后查询的时候 会优先查询缓存里面的Binder, 这样就会使用被我们修改过的Binder了  Field cacheField = serviceManager.getDeclaredField("sCache");  cacheField.setAccessible(true);  Map<String, IBinder> cache = (Map) cacheField.get(null);  cache.put(CLIPBOARD\_SERVICE, hookedBinder); |

接下来，在app里面使用剪切版，比如长按进行粘贴之后，剪切版的内容永远都是you are hooked了；这样，我们Hook系统服务的目的宣告完成！详细的代码参见 [github](https://github.com/tiann/understand-plugin-framework)。

也许你会问，插件框架会这么hook吗？如果不是那么插件框架hook这些干什么？插件框架当然不会做替换文本这么无聊的事情，DroidPlugin插件框架管理插件使得插件就像是主程序一样，因此插件需要使用主程序的剪切版，插件之间也会共用剪切版；其他的一些系统服务也类似，这样就可以达到插件和宿主程序之间的天衣服缝，水乳交融！另外，ActivityManager以及PackageManager这两个系统服务虽然也可以通过这种方式hook，但是由于它们的重要性和特殊性，DroidPlugin使用了另外一种方式，我们会单独讲解。

喜欢就点个赞吧～持续更新，请关注github项目 [understand-plugin-framework](https://github.com/tiann/understand-plugin-framework) 和我的 [博客](http://weishu.me/)!