

# Android 动态代理以及利用动态代理实现 ServiceHook

原创

2017年02月25日 20:44:15

标签: android / java / 动态代理 / servicehoo

8670

这篇博客主要介绍使用 InvocationHandler 这个接口来达到 hook 系统 service , 从而实现一些很有趣的功能的详细步骤。

转载请注明出处: [http://blog.csdn.net/self\\_study/article/details/55050627](http://blog.csdn.net/self_study/article/details/55050627)

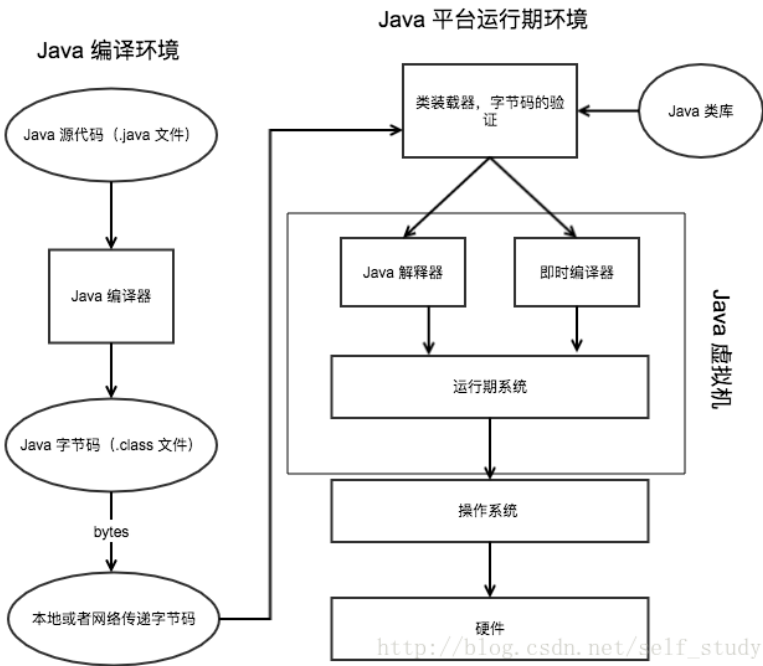
对感兴趣的童鞋加群 544645972 一起交流。

相关博文介绍:

- 不能在子线程中更新ui的讨论和分析: Activity 打开的过程分析;
- java/android 设计模式学习笔记 (9) —代理模式: AMS 的相关类图和介绍;
- WindowManager解析与骗取QQ密码案例分析: 界面 window 的创建过程;
- java/android 设计模式学习笔记 (8) —桥接模式: WMS 的相关类图和介绍;
- PC通信 (下) - AIDL: AIDL 以及 Binder 的相关介绍;
- 动态代理以及利用动态代理实现 ServiceHook: ServiceHook 的相关介绍;
- Android TransactionTooLargeException 解析, 思考与监控方案: TransactionTooLargeException 的解析以及监控方案。

## Java 的动态代理

首先我们要介绍的就是 Java 动态代理, Java 的动态代理涉及到两个类: InvocationHandler 接口和 Proxy 类, 下面我们会着重介绍一下这两个类, 并且结合实例来着重分析一下使用的正确姿势等。在这之前简单介绍一下 Java 中 class 文件的生成和加载过程, Java 编译器编译好 Java 文件之后会在磁盘产生 .class 文件。这种 .class 文件是二进制文件, 内容是只有 JVM 虚拟机才能识别的机器码, JVM 虚拟机读取字节码文件, 取出二进制数据, 加载到内存中, 解析 .class 文件内的信息, 使用相对应的 ClassLoader 类加载器生成对应的 Class 对象:



Shawn\_Dut

博客专家

原创	粉丝	喜欢	评论
92	779	312	384

等级: 博客 访问量: 48万+

积分: 5843 排名: 5381



Unable to Conn

The Proxy was unable to connect to the remote site. responding to requests. If you feel you have reached please submit a ticket via the link provided below.

URL: [http://pos.baidu.com/s?hei=250&wid=300&di=u%2Fblog.csdn.net%2Fself\\_study%2Farticle%2Fdetail](http://pos.baidu.com/s?hei=250&wid=300&di=u%2Fblog.csdn.net%2Fself_study%2Farticle%2Fdetail)

### 他的最新文章

- Android Native 开发之 NewString 与 New StringUtf 解析
- Android中图片压缩分析 (下)
- Android中图片压缩分析 (上)
- Android O新特性和行为变更总结
- android仿最新版本微信相册--附源码

### 文章分类

Android	92篇
Java	31篇
Android/Java 设计模式	25篇
开机	1篇
加密	1篇
签名	1篇

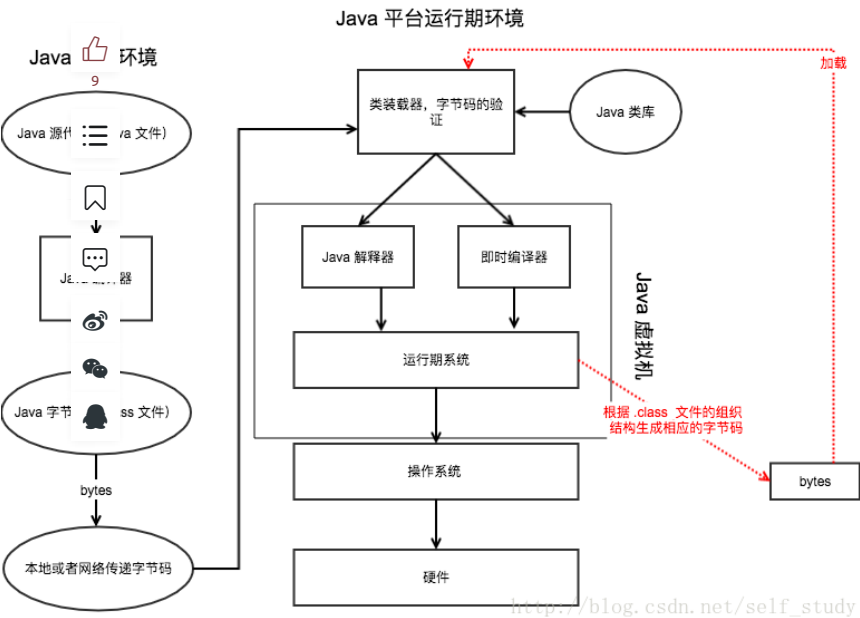
### 博主专栏

android

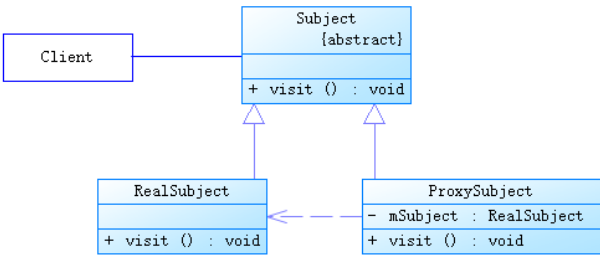
86820

介绍可以查看博客 [深入理解Java Class文件格式](#) 和 [Java 虚拟机规范](#)。

通过上面我们知道 JVM 是通过字节码的二进制信息加载类的，那么我们如果在运行期系统中，遵循 Java 编译系统组织 .class 文件的格式和结构，生成相应的二进制数据，然后再把这个二进制数据转换成对应的类，这样就可以在运行中动态生成一个我们想要的类了：



Java 中有很多的框架可以在运行时根据 JVM 规范动态的生成对应的 .class 二进制字节码，比如 ASM 和 Javassist 等，这里就不详细介绍了，感兴趣的可以去查阅相关的资料。这里我们就以动态代理模式为例来介绍一下我们要用到这两个很重要的类，关于动态代理模式，我在 [java/android 设计模式学习笔记（9）—代理模式](#) 中已经介绍过了，但是当时并没有详细分析过 InvocationHandler 接口和 Proxy 类，这里就来详细介绍一下。在代理模式那篇博客中，我们提到了代理模式分为动态代理和静态代理：



上面就是静态代理模式的类图，当在代码阶段规定这种代理关系时，ProxySubject 类通过编译器生成 .class 字节码文件，当系统运行之前，这个 .class 文件就已经存在了。动态代理模式的结构和上面的静态代理模式的结构稍微有所不同，它引入了一个 InvocationHandler 接口和 Proxy 类。在静态代理模式中，代理类 ProxySubject 中的方法，都指定地调用到特定 RealSubject 中对应的方法，ProxySubject 所做的事情无非是调用触发 RealSubject 对应的方法；动态代理工作的基本模式就是将自己方法功能的实现交给 InvocationHandler 角色，外界对 Proxy 角色中每一个方法的调用，Proxy 角色都会交给 InvocationHandler 来处理，而 InvocationHandler 则调用 RealSubject 的方法，如下图所示：



java  
📄 142189  
31 篇



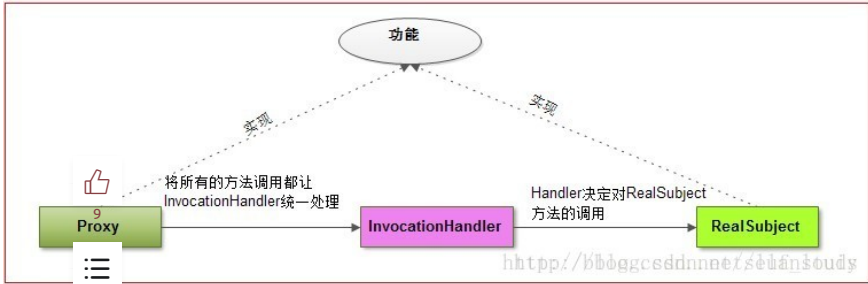
android进阶  
📄 205663  
28 篇

文章存档

2017年12月	1篇
2017年11月	2篇
2017年7月	1篇
2017年4月	4篇
2017年3月	1篇
2017年2月	5篇
展开	

他的热门文章

- android permission权限与安全机制解析（下）  
📄 33189
- Android 悬浮窗权限各机型各系统适配大全  
📄 25666
- android permission权限与安全机制解析（上）  
📄 16483
- android 特殊用户通知用法汇总--Notification源码分析  
📄 11474
- Android TransactionTooLargeException 解析，思考与监控方案  
📄 11375
- java/android 设计模式学习笔记（2）--- 观察者模式  
📄 11036
- android 自定义状态栏和导航栏分析与实现  
📄 10861
- android 不能在子线程中更新ui的讨论和分析  
📄 10389
- android WindowManager解析与骗取QQ密码案例分析  
📄 9925
- android scollview嵌套webview底部空白，高度无法自适应解决  
📄 9864



### InvocationHandler 接口和 Proxy 类

我们来分析一下动态代理模式中 ProxySubject 的生成步骤：

- 1. 获取 RealSubject 上的所有接口列表；
- 2. 确定生成的代理类的类名，系统默认生成的名字为：com.sun.proxy.\$ProxyXXXX；
- 3. 根据要实现的接口信息，在代码中动态创建该 ProxySubject 类的字节码；
- 4. 将对应的字节码转换为对应的 Class 对象；
- 5. 创建 InvocationHandler 的实例对象 h，用来处理 Proxy 角色的所有方法调用；
- 6. 以创建的 h 对象为参数，实例化一个 Proxy 角色对象。

具体的代码为：

#### Subject.java

```
1 public interface Subject {
2     String operation();
3 }
```

#### RealSubject.java

```
1 public class RealSubject implements Subject{
2     @Override
3     public String operation() {
4         return "operation by subject";
5     }
6 }
```

#### ProxySubject.java

```
1 public class ProxySubject implements InvocationHandler{
2     protected Subject subject;
3     public ProxySubject(Subject subject) {
4         this.subject = subject;
5     }
6
7     @Override
8     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
9         //do something before
10        return method.invoke(subject, args);
11    }
12 }
```

#### 测试代码



Unable to Conn

The Proxy was unable to connect to the remote site. responding to requests. If you feel you have reached please submit a ticket via the link provided below.

URL: http://pos.baidu.com/s?hei=250&wid=300&di=u %2Fblog.csdn.net/%2Fself\_study/%2Farticle/%2Fdetail

#### 联系我们



请扫描二维码联系客服  
webmaster@csdn.net  
400-660-0108  
QQ客服 客服论坛

关于 招聘 广告服务 百度  
©1999-2018 CSDN版权所有  
京ICP证09002463号

经营性网站备案信息  
网络110报警服务  
中国互联网举报中心  
北京互联网违法和不良信息举报中心

```
2 ProxySubject proxy = new ProxySubject(subject);
3 Subject sub = (Subject) Proxy.newProxyInstance(subject.getClass().getClassLoader(),
4         subject.getClass().getInterfaces(), proxy);
5 sub.operation();
```

以上就是动态代理模式的最简单实现代码，JDK 通过使用 `java.lang.reflect.Proxy` 包来支持动态代理，我们来看看这个类的表述：

```
1 Proxy provides static methods for creating dynamic proxy classes and instances, and it is
2 the base class of all dynamic proxy classes created by those methods.
```

一般情况下，我们使用下面的

```
1 public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)
2     throws IllegalArgumentException {
3     if (h == null) {
4         throw new NullPointerException();
5     }
6     // 获得与指定类装载器和一组接口相关的代理类类型对象
7     Class cl = getProxyClass(loader, interfaces);
8
9
10    // 通过反射获取构造函数对象并生成代理类实例
11    try {
12        Constructor cons = cl.getConstructor(ConstructorParams);
13        return (Object) cons.newInstance(new Object[] { h });
14    } catch (NoSuchMethodException e) { throw new InternalError(e.toString()); }
15    } catch (IllegalAccessException e) { throw new InternalError(e.toString()); }
16    } catch (InstantiationException e) { throw new InternalError(e.toString()); }
17    } catch (InvocationTargetException e) { throw new InternalError(e.toString()); }
18    }
19 }
```

Proxy 类的 `getProxyClass` 方法调用了 `ProxyGenerator` 的 `generatorProxyClass` 方法去生成动态类：

```
1 public static byte[] generateProxyClass(final String name, Class[] interfaces)
```

这个方法我们下面将会介绍到，这里先略过，生成这个动态类的字节码之后，通过反射去生成这个动态类的对象，通过 Proxy 类的这个静态函数生成了一个动态代理对象 sub 之后，调用 sub 代理对象的每一个方法，在代码内部，都是直接调用了 `InvocationHandler` 的 `invoke` 方法，而 `invoke` 方法根据代理类传递给自己的 `method` 参数来区分是什么方法，我们来看看 `InvocationHandler` 类的介绍：

```
1 InvocationHandler is the interface implemented by the invocation handler of a proxy instance.
2
3 Each proxy instance has an associated invocation handler. When a method is invoked on a proxy
4 instance, the method invocation is encoded and dispatched to the invoke method of its invocation handler.
```

Public methods	
abstract Object	invoke(Object proxy, Method method, Object[] args)Processes a method invocation on a proxy instance and returns the result.

方法的参数和返回：

--	--

Parameters	
proxy	Object: the proxy instance that the method was invoked on
method	Method: the Method instance corresponding to the interface method invoked on the proxy instance. The declaring class of the Method object will be the interface that the method was declared in, which may be a superinterface of the proxy interface that the proxy class inherits the method through.
args	Object: an array of objects containing the values of the arguments passed in the method invocation on the proxy instance, or null if interface method takes no arguments. Arguments of primitive types are wrapped in instances of the appropriate primitive wrapper class, such as java.lang.Integer or java.lang.Boolean.
Return:	
Object	value to return from the method invocation on the proxy instance. If the declared return type of interface method is a primitive type, then the value returned by this method must be an instance of the corresponding primitive wrapper class; otherwise, it must be a type assignable to declared return type. If the value returned by this method is null and the interface method's return type is primitive, then a NullPointerException will be thrown by the method invocation on the proxy instance. If the value returned by this method is otherwise not compatible with the interface method's declared return type as described above, a ClassCastException will be thrown by the method invocation on the proxy instance.

上面提到的一点需要特别注意的是，如果 Subject 类中定义的方法返回值为 8 种基本数据类型，那么在 ProxySubject 类中必须要返回相应的基本类型包装类，即 int 对应的返回为 Integer 等等，还需要注意的是如果此时返回 null，则会抛出 NullPointerException，除此之外的其他情况下返回值的对象必须要和 Subject 类中定义方法的返回值一致，要不然会抛出 ClassCastException。

### 生成源码分析

那么通过 Proxy 类的 newInstance 方法动态生成的类是什么样子的呢，我们上面也提到了，JDK 为我们提供了一个方法 ProxyGenerator.generateProxyClass(String proxyName,class[] interfaces) 来产生动态代理类的字节码，这个类位于 sun.misc 包中，是属于特殊的 jar 包，于是问题又来了，android studio 创建的 android 工程是没法找到 ProxyGenerator 这个类的，这个类在 jre 目录下，就算我把这个类相关的 .jar 包拷贝到工程里面并且在 gradle 里面引用它，虽然最后能够找到这个类，但是编译时又会出现很奇葩的问题，所以，没办法喽，android studio 没办法创建普通的 java 工程，只能自己装一个 intellij idea 或者求助相关的同事了。创建好 java 工程之后，使用下面这段代码就可以将生成的类导出到指定路径下面：

```
1 public static void generateClassFile(Class clazz,String proxyName)
2 {
3     //根据类信息和提供的代理类名称，生成字节码
4     byte[] classFile = ProxyGenerator.generateProxyClass(proxyName, clazz.getInterfaces())
5     String paths = "D:\\\\"; // 这里写死路径为 D 盘，可以根据实际需要去修改
6     System.out.println(paths);
7     FileOutputStream out = null;
8
9     try {
10         //保留到硬盘中
11         out = new FileOutputStream(paths+proxyName+".class");
12         out.write(classFile);
13         out.flush();
14     } catch (Exception e) {
15         e.printStackTrace();
16     } finally {
17         try {
18             out.close();
19
20
```

```
21     }  
22 }  
23 }
```

调用代码的方式为：

```
1 writeClassFile(ProxySubject.class, "ProxySubject");
```

最后就会在 D 盘（如果没有修改路径）的根目录下面生成一个 ProxySubject.class 的文件，使用 jd-gui 就可打开该文件：

```
1 : java.lang.reflect.InvocationHandler;  
2 : java.lang.reflect.Method;  
3 : java.lang.reflect.Proxy;  
4 : java.lang.reflect.UndeclaredThrowableException;  
5  
6 : final class ProxySubject  
7     implements Proxy  
8         implements Subject  
9  
10     private static Method m1;  
11     private static Method m3;  
12     private static Method m2;  
13     private static Method m0;  
14  
15     public ProxySubject(InvocationHandler paramInvocationHandler)  
16     {  
17         super(paramInvocationHandler);  
18     }  
19  
20     public final boolean equals(Object paramObject)  
21     {  
22         try  
23         {  
24             return ((Boolean)this.h.invoke(this, m1, new Object[] { paramObject })).booleanValue  
25         }  
26         catch (Error|RuntimeException localError)  
27         {  
28             throw localError;  
29         }  
30         catch (Throwable localThrowable)  
31         {  
32             throw new UndeclaredThrowableException(localThrowable);  
33         }  
34     }  
35  
36     public final String operation()  
37     {  
38         try  
39         {  
40             return (String)this.h.invoke(this, m3, null);  
41         }  
42         catch (Error|RuntimeException localError)  
43         {  
44             throw localError;  
45         }  
46         catch (Throwable localThrowable)  
47         {  
48             throw new UndeclaredThrowableException(localThrowable);  
49         }  
50     }  
51 }
```

加入，享受更精准的内容推荐，与500万程序员共同成长！

[登录](#)

[注册](#)

```

53     {
54         try
55         {
56             return (String)this.h.invoke(this, m2, null);
57         }
58         catch (Error|RuntimeException localError)
59         {
60             throw localError;
61         }
62         catch (Throwable localThrowable)
63         {
64             throw new UndeclaredThrowableException(localThrowable);
65         }
66     }
67
68     public final int hashCode()
69     {
70         try
71         {
72             return ((Integer)this.h.invoke(this, m0, null)).intValue();
73         }
74         catch (Error|RuntimeException localError)
75         {
76             throw localError;
77         }
78         catch (Throwable localThrowable)
79         {
80             throw new UndeclaredThrowableException(localThrowable);
81         }
82     }
83
84     static
85     {
86         try
87         {
88             m1 = Class.forName("java.lang.Object").getMethod("equals", new Class[] { Class.forName("java.lang.Object")});
89             m3 = Class.forName("Subject").getMethod("operation", new Class[0]);
90             m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[0]);
91             m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]);
92             return;
93         }
94         catch (NoSuchMethodException localNoSuchMethodException)
95         {
96             throw new NoSuchMethodError(localNoSuchMethodException.getMessage());
97         }
98         catch (ClassNotFoundException localClassNotFoundException)
99         {
100             throw new NoClassDefFoundError(localClassNotFoundException.getMessage());
101         }
102     }
103 }

```

可以观察到这个生成的类继承自 `java.lang.reflect.Proxy`，实现了 `Subject` 接口，我们在看看生成动态类的代码：

```

1 Subject sub = (Subject) Proxy.newProxyInstance(subject.getClass().getClassLoader(),
2         subject.getClass().getInterfaces(), proxy);

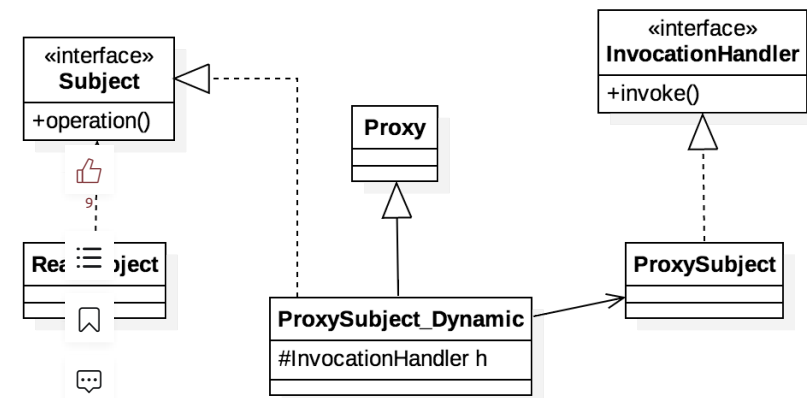
```

可见这个动态生成类会实现 `subject.getClass().getInterfaces()` 中的所有接口，并且还有一点是类中所有的方法都是 `final` 的，而且该类也是 `final`，所以该类不可继承，最后就是所有的方法都会调用到 `InvocationHandler` 接口上的 `invoke` 方法。这也就是为什么最后会调用到 `InvocationHandler` 接口上的 `invoke` 方法。

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

[登录](#)

[注册](#)

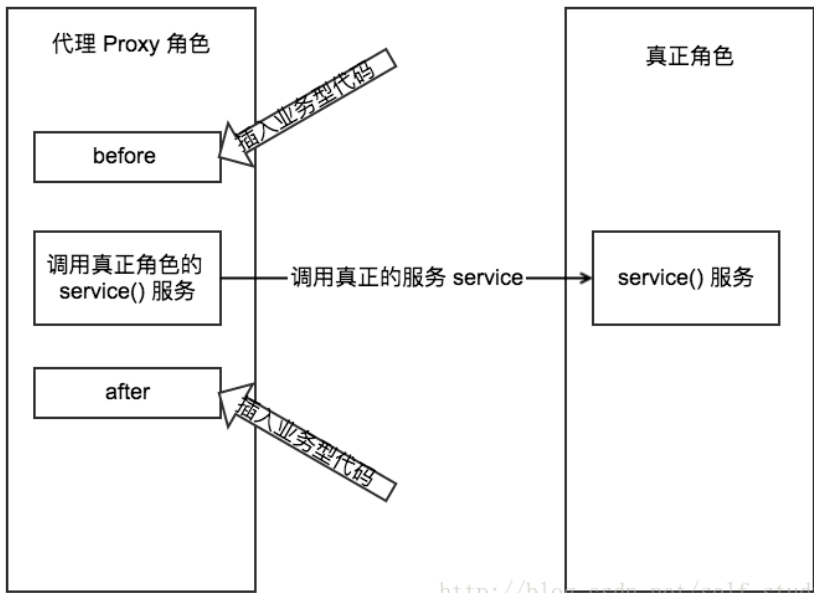


http://blog.csdn.net/self\_study

从这个图可以很清楚的看明白，动态生成的类 ProxySubject（同名，所以后面加上了 Dynamic）持有了 InvocationHandler 接口的 ProxySubject 类的对象 h，然后调用代理对象的 operation 方法时，就会调用到对象 h 的 invoke 方法中，invoke 方法中根据 method 的名字来区分到底是什么方法，然后通过 method.invoke() 方法来调用具体对象的对应方法。

## Android 中利用动态代理实现 ServiceHook

通过上面对 InvocationHandler 的介绍，我们对这个接口应该有了大体的了解，但是在运行时动态生成的代理类有什么作用呢，其实它的作用就是在调用真正业务之前或者之后插入一些额外的操作：



http://blog.csdn.net/self\_study

所以简而言之，代理类的处理逻辑很简单，就是在调用某个方法前及方法后插入一些额外的业务。而我们在 Android 中的实践例子就是在真正调用系统的某个 Service 之前和之后选择性的做一些自己特殊的处理，这种思想在插件化框架上也是很重要的。那么我们具体怎么去实现 hook 系统的 Service，在真正调用系统 Service 的时候附上我们需要的业务呢，这就需要介绍 ServiceManager



## ServiceManager 介绍以及 hook 的步骤

### 第一步

关于 ServiceManager 的详细介绍在我的博客：[android IPC通信（下） - AIDL](#) 中已经介绍过了，这里就不赘述了，强烈建议大家去看一下那篇博客，我们这里就着重看一下 ServiceManager 的 getService(String name) 方法：

```
1  public final class ServiceManager {
2      private static final String TAG = "ServiceManager";
3
4      private static IServiceManager sServiceManager;
5      private static HashMap<String, IBinder> sCache = new HashMap<String, IBinder>();
6
7      private static IServiceManager getIServiceManager() {
8          if (sServiceManager != null) {
9              return sServiceManager;
10         }
11
12         // Find the service manager
13         sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());
14         return sServiceManager;
15     }
16
17     /**
18      * Returns a reference to a service with the given name.
19      *
20      * @param name the name of the service to get
21      * @return a reference to the service, or null if the service doesn't exist
22      */
23     public static IBinder getService(String name) {
24         try {
25             IBinder service = sCache.get(name);
26             if (service != null) {
27                 return service;
28             } else {
29                 return getIServiceManager().getService(name);
30             }
31         } catch (RemoteException e) {
32             Log.e(TAG, "error in getService", e);
33         }
34         return null;
35     }
36
37     public static void addService(String name, IBinder service) {
38         ...
39     }
40     ....
41 }
```

我们可以看到，getService 方法第一步会去 sCache 这个 map 中根据 Service 的名字获取这个 Service 的 IBinder 对象，如果获取到为空，则会通过 ServiceManagerNative 通过跨进程通信获取这个 Service 的 IBinder 对象，所以我们就以 sCache 这个 map 为切入点，反射该对象，然后修改该对象，由于系统的 android.os.ServiceManager 类是 @hide 的，所以只能使用反射，根据这个初步思路，写下第一步的代码：

```
1  Class c_ServiceManager = Class.forName("android.os.ServiceManager");
2  if (c_ServiceManager == null) {
3      return;
4  }
5
6  N, 享受更精准的内容推荐，与500万程序员共同成长！
7
```

[登录](#)[注册](#)

```

8         Field sCache = c_ServiceManager.getDeclaredField("sCache");
9         sCache.setAccessible(true);
10        sCacheService = (Map<String, IBinder>) sCache.get(null);
11    } catch (Exception e) {
12        e.printStackTrace();
13    }
14 }
15 sService.remove(serviceName);
16 sService.put(serviceName, service);

```

反射 sC 这个变量，移除系统 Service，然后将我们自己改造过的 Service put 进去，这样就能实现当调用 ServiceManager 的 getService(String name) 方法的时候，返回的是我们改造过的 Service 而不是系统的原生 Service。

## 第二步

知道了如何去将改造过后的 Service put 进系统的 ServiceManager 中，第二步就是去生成一个 hook Service 了，怎么去生成呢？这就要用到我们上面介绍到的 InvocationHandler 类，我们先获取系统的 Service，然后通过 InvocationHandler 去构造一个 hook Service，最后通过第一步的步骤将 sCache 这个变量即可，第二步代码：

```

1 class ServiceHook implements InvocationHandler {
2     private static final String TAG = "ServiceHook";
3
4     private IBinder mBase;
5     private Class<?> mStub;
6     private Class<?> mInterface;
7     private InvocationHandler mInvocationHandler;
8
9     public ServiceHook(IBinder mBase, String iInterfaceName, boolean isStub, InvocationHan
10        this.mBase = mBase;
11        this.mInvocationHandler = InvocationHandler;
12
13        try {
14            this.mInterface = Class.forName(iInterfaceName);
15            this.mStub = Class.forName(String.format("%s%s", iInterfaceName, isStub ? "$St
16        } catch (ClassNotFoundException e) {
17            e.printStackTrace();
18        }
19    }
20
21    @Override public Object invoke(Object proxy, Method method, Object[] args) throws Thro
22        if ("queryLocalInterface".equals(method.getName())) {
23            return Proxy.newProxyInstance(proxy.getClass().getClassLoader(), new Class[] {
24                new HookHandler(mBase, mStub, mInvocationHandler));
25        }
26
27        Log.e(TAG, "ERROR!!!! method:name = " + method.getName());
28        return method.invoke(mBase, args);
29    }
30
31    private static class HookHandler implements InvocationHandler {
32        private Object mBase;
33        private InvocationHandler mInvocationHandler;
34
35        public HookHandler(IBinder base, Class<?> stubClass,
36            InvocationHandler invocationHandler) {
37            mInvocationHandler = invocationHandler;
38
39            try {
40                Method asInterface = stubClass.getDeclaredMethod("asInterface", IBinder.cl
41                this.mBase = asInterface.invoke(null, base);
42
43    }

```

加入 N，享受更精准的内容推荐，与500万程序员共同成长！

登录

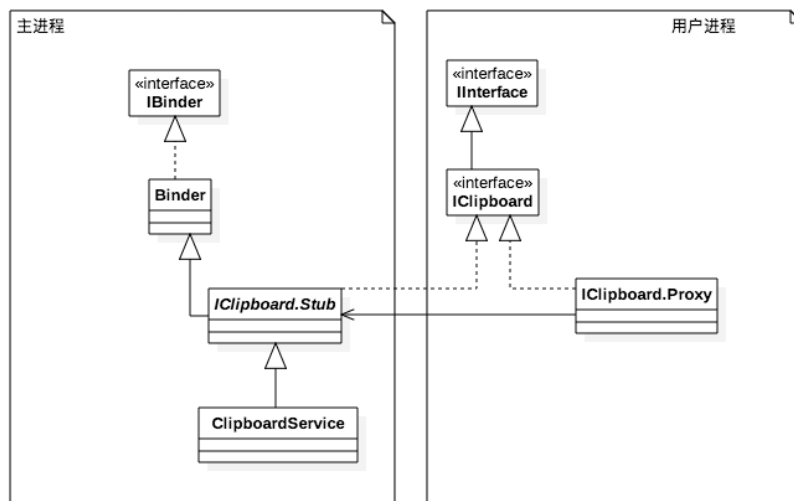
注册

```
44         }
45     }
46
47     @Override public Object invoke(Object proxy, Method method, Object[] args) throws
48         if (mInvocationHandler != null) {
49             return mInvocationHandler.invoke(mBase, method, args);
50         }
51         return method.invoke(mBase, args);
52     }
53 }
54
```

这里我们以 clipboardService 的调用代码为例：

```
1  ClipboardService clipboardService = ServiceManager.getService(Context.CLIPBOARD_SERVICE);
2  IClipboard iClipboard = "android.content.IClipboard";
3
4  if (clipboardService != null) {
5      IBinder hookClipboardService =
6          (IBinder) Proxy.newProxyInstance(clipboardService.getClass().getClassLoader(),
7              clipboardService.getClass().getInterfaces(),
8              new ServiceHook(clipboardService, IClipboard, true, new ClipboardHookH
9          //调用第一步的方法
10     ServiceManager.setService(Context.CLIPBOARD_SERVICE, hookClipboardService);
11 } else {
12     Log.e(TAG, "ClipboardService hook failed!");
13 }
```

分析一下上面的这段代码，分析之前，先要看一下 ClipboardService 的相关类图：



[http://blog.csdn.net/self\\_study](http://blog.csdn.net/self_study)

从这张类图我们可以清晰的看见 ClipboardService 的相关继承关系，接下来就是分析代码了：

- 调用代码中，`Proxy.newProxyInstance` 函数的第二个参数需要注意，由于 `ClipboardService` 继承自三个接口，所以这里需要把所有的接口传递进去，但是如果将第二个参数变更为 `new Class[] { IBinder.class }` 其实也是没有问题的（感兴趣的可以试一下，第一个参数修改为 `IBinder.class.getClassLoader()`，第二个参数修改为 `new Class[] { IBinder.class }`，也是可以的），因为实际使用的时候，我们只是用到了 `IBinder` 类的 `queryLocalInterface` 方法，其他的方法都没有使用到。接下来我们会说明 `queryLocalInterface` 这个函数的作用：

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录

注册

ke 方法中；

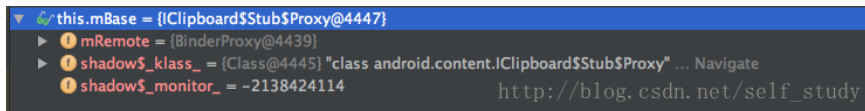
- 接着我们来看看 ServiceHook 的 invoke 函数，很简单，当函数为 queryLocalInterface 方法的时候返回一个 HookHandler 的对象，其他的情况直接调用 method.invoke 原生系统的 ClipboardService 功能，为什么只处理 queryLocalInterface 方法呢，这个我在博客：[java/android 设计模式学习笔记（9）—代理模式](#) 中分析 AMS 的时候已经提到了，asInterface 方法最终会调用到 queryLocalInterface 方法，queryLocalInterface 方法最后的返回结果会作为 asInterface 的结果 ① 回给 Service 的调用方，所以 queryLocalInterface 方法的最后返回的对象是会被外部直接调用的，这也解释了为什么调用代码中的第二个参数变更为 `new Class[] { IBinder.class }` 也是没有问题，因为第一次调用到 queryLocalInterface 函数之后，后续的所有调用都到了 HookHandler 对象中，动态生成的对象中只需要有 IBinder 的 queryLocalInterface 方法即可，而不需要 Clipboard 接口的其他方法；
- 接下来 HookHandler 类，首先我们看看这个类的构造函数，第一个参数为系统的 ClipboardService 第二个参数为



```
1 forName(String.format("%s%s", iInterfaceName, isStub ? "$Stub" : ""))//android.cont
```



这个方法 ② 对照上面的类图，这个类为 ClipboardService 的父类，它里面有一个 asInterface 的方法，通过 asInterface 方法然后将 IBinder 对象变成 IInterface 对象，为什么要这么做，可以去看看我的博客：[java/android 设计模式学习笔记（9）—代理模式](#) 中的最后总结，通过 ServiceManager.getService 方法获取一个 IBinder 对象，但是这个 IBinder 对象不能直接调用，必须要通过 asInterface 方法转成对应的 IInterface 对象才可以使用，所以 mBase 对象其实是一个 IInterface 对象：



最后也证实了这个结果，为什么是 Proxy 对象这就不用我解释了吧；

最后是 HookHandler 的 invoke 方法，这个方法调用到了 ClipboardHookHandler 对象，我们来看看这个类的实现：

```

1 public class ClipboardHook {
2
3     private static final String TAG = ClipboardHook.class.getSimpleName();
4
5     public static void hookService(Context context) {
6         IBinder clipboardService = ServiceManager.getService(Context.CLIPBOARD_SERVICE);
7         String IClipboard = "android.content.IClipboard";
8
9         if (clipboardService != null) {
10             IBinder hookClipboardService =
11                 (IBinder) Proxy.newProxyInstance(IBinder.class.getClassLoader(),
12                     new Class[]{IBinder.class},
13                     new ServiceHook(clipboardService, IClipboard, true, new Clipbo
14             ServiceManager.setService(Context.CLIPBOARD_SERVICE, hookClipboardService);
15         } else {
16             Log.e(TAG, "ClipboardService hook failed!");
17         }
18     }
19
20     public static class ClipboardHookHandler implements InvocationHandler {
21
22         @Override
23         public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
24             String methodName = method.getName();
25
26             N, 享受更精准的内容推荐，与500万程序员共同成长！
27
28             登录 注册
29
30             第12页 共20页
31
32             2018/3/22 下午4:27
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

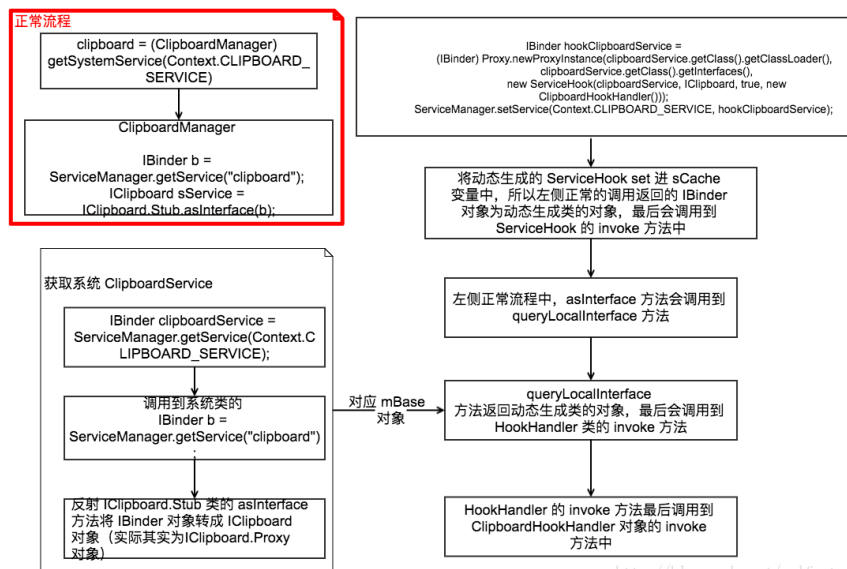
```
28         if (argLength >= 2 && args[0] instanceof ClipData) {
29             ClipData data = (ClipData) args[0];
30             String text = data.getItemAt(0).getText().toString();
31             text += "this is shared from ServiceHook-----by Shawn_Dut";
32             args[0] = ClipData.newPlainText(data.getDescription().getLabel(), text);
33         }
34     }
35     return method.invoke(proxy, args);
36 }
37 }
38 }
```

所以 ClipdHookHandler 类的 invoke 方法最终获取到了要 hook 的 Service 的 IInterface 对象（即为 IClipboard.Proxy 对象，最后通过 Binder Driver 调用到了系统的 ClipboardService 中），调用函数的 IInterface 对象和参数列表对象，获取到了这些之后，不用我说了，就可以尽情的去做一些额外的操作了。我这里是仿照知乎复制文字时，在后面加上类似的版权声明。

## 问题

上文：ServiceHook 的详细步骤了，了解它必须对 InvocationHandler 有详细的了解，并且还要扒下 AOSP 源码，比如要去 hook ClipboardService，那么就要去看看 ClipboardService 的源码，看看这个类中每个函数的名字和作用，参数列表中每个参数的顺序和作用，而且有时候这还远远不够，我们知道，随着 Android 每个版本的更新，这些类可能也会被更新修改甚至删除，很有可能对于新版本来说老的 hook 方法就不管用了，这时候必须要去了解最新的源码，看看更新修改的地方，针对新版本去重新制定 hook 的步骤，这是一点需要慎重对待考虑的地方。

## 步骤



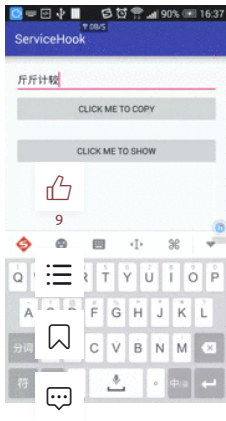
[http://blog.csdn.net/self\\_study](http://blog.csdn.net/self_study)

## 源码

此为我们公司某位大神代码，经过整理修改而出，不知道有没有版权问题，哈哈哈，谢谢周杰大神，虽然已经不在公司，感谢感谢～～

源码下载地址：<https://github.com/zhaozepeng/ServiceHook>

先来看看运行的效果：



最后 有的代码贴出来：

ServiceManager.java

```
1  : class ServiceManager {
2
3      private static Method sGetServiceMethod;
4      private static Map<String, IBinder> sCacheService;
5      private static Class c_ServiceManager;
6
7      static {
8          try {
9              c_ServiceManager = Class.forName("android.os.ServiceManager");
10         } catch (Exception e) {
11             e.printStackTrace();
12         }
13     }
14
15     public static IBinder getService(String serviceName) {
16         if (c_ServiceManager == null) {
17             return null;
18         }
19
20         if (sGetServiceMethod == null) {
21             try {
22                 sGetServiceMethod = c_ServiceManager.getDeclaredMethod("getService", String.class);
23                 sGetServiceMethod.setAccessible(true);
24             } catch (NoSuchMethodException e) {
25                 e.printStackTrace();
26             }
27         }
28
29         if (sGetServiceMethod != null) {
30             try {
31                 return (IBinder) sGetServiceMethod.invoke(null, serviceName);
32             } catch (Exception e) {
33                 e.printStackTrace();
34             }
35         }
36
37         return null;
38     }
39
40     public static void setService(String serviceName, IBinder service) {
41         if (c_ServiceManager == null) {
42             return;
43         }
44     }
45 }
```

加 N, 享受更精准的内容推荐, 与500万程序员共同成长!

登录

注册

```
44
45     if (sCacheService == null) {
46         try {
47             Field sCache = c_ServiceManager.getDeclaredField("sCache");
48             sCache.setAccessible(true);
49             sCacheService = (Map<String, IBinder>) sCache.get(null);
50         } catch (Exception e) {
51             e.printStackTrace();
52         }
53     }
54     sCacheService.remove(serviceName);
55     sCacheService.put(serviceName, service);
56
57
```

ServiceManager 这个类就是使用反射的方式去获取对应 Service（这里不能使用 Context.getSystemService，因为它的返回不是 IBinder 对象，比如对于 ClipboardService，它就是 ClipboardManager），并设置 service 到 sCache 变量中；

ServiceHook.java

```
1 public class ServiceHook implements InvocationHandler {
2     private static final String TAG = "ServiceHook";
3
4     private IBinder mBase;
5     private Class<?> mStub;
6     private Class<?> mInterface;
7     private InvocationHandler mInvocationHandler;
8
9     public ServiceHook(IBinder mBase, String iInterfaceName, boolean isStub, InvocationHandler mInvocationHandler) {
10         this.mBase = mBase;
11         this.mInvocationHandler = mInvocationHandler;
12
13         try {
14             this.mInterface = Class.forName(iInterfaceName);
15             this.mStub = Class.forName(String.format("%s%s", iInterfaceName, isStub ? "$Stub" : ""));
16         } catch (ClassNotFoundException e) {
17             e.printStackTrace();
18         }
19     }
20
21     @Override public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
22         if ("queryLocalInterface".equals(method.getName())) {
23             return Proxy.newProxyInstance(proxy.getClass().getClassLoader(), new Class[] {
24                 mInterface
25             }, new HookHandler(mBase, mStub, mInvocationHandler));
26         }
27         Log.e(TAG, "ERROR!!!!!! method:name = " + method.getName());
28         return method.invoke(mBase, args);
29     }
30
31     private static class HookHandler implements InvocationHandler {
32         private Object mBase;
33         private InvocationHandler mInvocationHandler;
34
35         public HookHandler(IBinder base, Class<?> stubClass,
36             InvocationHandler invocationHandler) {
37             mBase = base;
38             mInvocationHandler = invocationHandler;
39
40             try {
41                 Method asInterface = stubClass.getDeclaredMethod("asInterface", IBinder.class);
42
```

42 N，享受更精准的内容推荐，与500万程序员共同成长！

[登录](#)

[注册](#)

```

43         e.printStackTrace();
44     }
45 }
46
47 @Override public Object invoke(Object proxy, Method method, Object[] args) throws
48     if (mInvocationHandler != null) {
49         return mInvocationHandler.invoke(mBase, method, args);
50     }
51     return method.invoke(mBase, args);
52 }
53
54

```

这个类上介绍的很详细了，在这里就不继续介绍了；

#### ClipboardHook.java

```

1  class ClipboardHook {
2
3  private static final String TAG = ClipboardHook.class.getSimpleName();
4
5  public static void hookService(Context context) {
6      IBinder clipboardService = ServiceManager.getService(Context.CLIPBOARD_SERVICE);
7      String IClipboard = "android.content.IClipboard";
8
9      if (clipboardService != null) {
10         IBinder hookClipboardService =
11             (IBinder) Proxy.newProxyInstance(IBinder.class.getClassLoader(),
12                 new Class[]{IBinder.class},
13                 new ServiceHook(clipboardService, IClipboard, true, new Clipboa
14             ServiceManager.setService(Context.CLIPBOARD_SERVICE, hookClipboardService);
15     } else {
16         Log.e(TAG, "ClipboardService hook failed!");
17     }
18 }
19
20 public static class ClipboardHookHandler implements InvocationHandler {
21
22     @Override
23     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
24         String methodName = method.getName();
25         int argsLength = args.length;
26         //每次从本应用复制的文本，后面都加上分享的出处
27         if ("setPrimaryClip".equals(methodName)) {
28             if (argsLength >= 2 && args[0] instanceof ClipData) {
29                 ClipData data = (ClipData) args[0];
30                 String text = data.getItemAt(0).getText().toString();
31                 text += "this is shared from ServiceHook-----by Shawn_Dut";
32                 args[0] = ClipData.newPlainText(data.getDescription().getLabel(), text
33             }
34         }
35         return method.invoke(proxy, args);
36     }
37 }
38 }

```

这里的 hook ClipboardService 使用的是，将复制的文本取出，在结尾处添加我们自定义的文本，用户再粘贴到其他地方的时候，就会出现我们添加上我们自定义文本后的文字了，还有需要注意的是这只会影响应用进程的 ClipboardService，并不会影响主进程的相关 Service，因为不管怎么 hook，修改的都是应用进程的 ServiceManager 里面的 sCache 变量，应用进程的 ServiceManager 其实



ent，其实是不会影响 Binder Server的，不明白的建议还是看看：[android IPC通信（下） - AIDL](#)。

测试代码

```
1  switch (v.getId()) {
2      case R.id.btn_copy:
3          String input = mEtInput.getText().toString().trim();
4          if (TextUtils.isEmpty(input)) {
5              Toast.makeText(this, "input不能为空", Toast.LENGTH_SHORT).show();
6              return;
7          }
8      case R.id.btn_show_paste:
9          //复制
10         ClipData clip = ClipData.newPlainText("simple text", mEtInput.getText().toString());
11         clipboard.setPrimaryClip(clip);
12         break;
13     case R.id.btn_show_paste:
14         //黏贴
15         clip = clipboard.getPrimaryClip();
16         if (clip != null && clip.getItemCount() > 0) {
17             Toast.makeText(this, clip.getItemAt(0).getText(), Toast.LENGTH_SHORT).show();
18         }
19         break;
20 }
```

测试代码，我这里就不需要说明了，Clipboard 的简单使用而已。

这里只演示了 hook ClipboardService 的例子，其他的使用方式比如插件化等等在这里就不一一介绍了，感兴趣的可以去网上查阅相关的资料。

转载请注明出处：[http://blog.csdn.net/self\\_study/article/details/55050627](http://blog.csdn.net/self_study/article/details/55050627)

## 引用

<http://blog.csdn.net/luanlouis/article/details/24589193>  
<http://www.cnblogs.com/xiaoluo501395377/p/3383130.html>  
[http://blog.csdn.net/self\\_study/article/details/51628486](http://blog.csdn.net/self_study/article/details/51628486)  
<http://paddy-w.iteye.com/blog/841798>  
<http://www.cnblogs.com/flyyoung2008/archive/2013/08/11/3251148.html>

版权声明：转载请标明出处[http://blog.csdn.net/self\\_study](http://blog.csdn.net/self_study)，对技术感兴趣的童鞋加群544645972一起交流  
[http://blog.csdn.net/zhao\\_zepeng/article/details/55050627](http://blog.csdn.net/zhao_zepeng/article/details/55050627)

本文已收录于以下专栏：[android进阶](#)

👤 目前您尚未登录，请 [登录](#) 或 [注册](#) 后进行评论



z437955114 2017-11-06 17:25

[回复](#) 4楼

生成文件处：

-----

调用代码的方式为：

generateClassFile(proxySubject.class, "ProxySubject");


-----

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

[登录](#)

[注册](#)

```
generateClassFile(RealSubject.class, &quot;RealSubject&quot;);
-----
-----
```

 **u013377211** 2017-08-10 18:14 1条回复 回复 3楼

加群管理员直接拒绝，连个拒绝理由都不给。

 **haorser** 2017-08-01 18:26 1条回复 回复 2楼

请教一下，这样hook的优点在什么地方？





查看 7 条热评




### Android开发中无处不在的设计模式——动态代理模式


继续更新设计模式系列，写这个模式的主要原因是最近看到了动态代理的代码。先来回顾一下前5个模式： - Android开发中无处不在的设计模式——单例模式 - Android开发中无处不在的设...

 **sbsujj** 2016年01月21日 11:37 7912




**android动态代理机制**  **u012439416** 2017年04月22日 21:50 984


-----本文转载自 **Android插件化原理解析——Hook机制之动态代理** 这一系列的文章实在是写的好! 1, 概述使用代理机制进行API Hook进而达到方法增强是框架...

**Android插件化开发-hook动态代理**  **u013022222** 2016年04月10日 11:19 6502

首先，我们阐述为什么android需要插件化： 1: 由于业务的增长，app的方法数逐渐达到65535(有人说用于检索方法数的列表大小使用short存储的，其实我看了源码之后并没有发现相关信息，并对此...

**Android10--Android之动态代理详解**  **u012954720** 2016年09月01日 14:33 1987


<http://my.oschina.net/rksi5/blog/224931>

**Android实用技巧-动态代理**  **ynztlxdei** 2017年07月03日 10:04 419

应用场景在Android的代码维护当中,经常会涉及到逻辑变更,但是又并不是整个逻辑变更了,往往是类似在之前的操作前面追加逻辑,或者是在之后追加逻辑.对于这样的逻辑,往往是每个类型的操作里面都要变更. ...


**Android设计模式之动态代理，实现方法拦截功能**

动态代理的好处： 1.代理方式多样，自由定义，比如可以查看被代理类的各方法执行时间。 2.当被代理对象改变其内部实现时，不影响代理规则。动态代理的局限性： 1.只能代理interface方法以拦...

 **qq\_21146289** 2016年12月11日 20:21 249

**Android插件化原理解析——Hook机制之动态代理**


转发必注明出处：Hook机制之动态代理使用代理机制进行API Hook进而达到方法增强是框架的常用手段，比如J2EE框架Spring通过动态代理优雅地实现了AOP编程，极大地提升了Web开发效率； ...


 **leilba** 2016年05月03日 23:09 1180

就是，使用反射完成的，写个小小的动态代理，第一步首先需要一个bean...


### Android插件化原理解析——Hook机制之动态代理

使用代理机制进行API Hook进而达到方法增强是框架的常用手段，比如J2EE框架Spring通过动态代理优雅地实现了AOP编程，极大地提升了Web开发效率；同样，插件框架也广泛使用了代理机制来增强系...

 wz245131 2016年10月26日 21:38 446

将cglib动态代理思想带入Android开发  zhangke3016 2017年05月08日 23:34 984

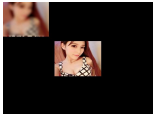
动态代理在Android实际开发中用的并不是很多，但在设计框架的时候用的就比较多多了，最近在看J2EE一些东西，像Spring，Hibernate等都有通过动态代理来实现方法增强、方法拦截等需要，通过代...


Android设计模式之代理模式 Proxy  l2show 2015年07月28日 22:14 20939

一.概述代理模式也是平时比较常用的设计模式之一,代理模式其实就是提供了一个新的对象,实现了对真实对象的操作,或成为真实对象的替身.在日常生活中也是很常见的.例如A要租房,为了省麻烦A...

### 技术外文文献看不懂？教你一个公式秒懂英语

不背单词和语法，一个公式学好英语



国内免代理下载android源码  feiniao8651 2016年12月14日 19:50 982


android程序猿痛苦的事情之一，就是Google的各种资源都被墙掉了，下载资源的话一定要翻墙。当然，由于国内的一些资源分享网站(<http://www.androiddevtools.cn/> ht...

Android开发资源获取国内代理（转载）  zerokkqq 2016年10月29日 22:39 2080

Android Dev Tools官网地址：[www.androiddevtools.cn](http://www.androiddevtools.cn) 收集整理Android开发所需的Android SDK、开发中用到的工具、Android开发教程、Android...


### Android SDK代理服务解决国内不能更新下载问题


Android SDK代理服务解决国内Android SDK不能更新下载问题，经常会遇到Fitch fail URL错误，要不就是Nothing was installed。目下Google遭受在中...

 boonya 2014年08月22日 11:23 142491


### Android 开发之避免被第三方使用代理抓包

Android 避免被第三方使用代理抓包

 a807891033 2016年12月14日 18:19 4489

教你在Android手机上使用全局代理！  testcs\_dn 2017年11月14日 07:43 3593

前言：在Android上使用系统自带的代理，限制灰常大，仅支持系统自带的浏览器。这样像QQ、飞信、微博等这些单独的App都不能使用系统的代理。如何让所有软件都能正常代理呢？ProxyDroid这个软件...

android代理  lgd5979 2012年03月12日 15:10 1794


HTC Magic Mozilla/5.0 (Linux; U; Android 1.5; en-ca; Build/CUPCAKE) AppleWebKit/528.5+ (KHTML, like ...



Unable to Connect to Site

### Android 一篇充篇之----Hook系统的AMS服务实现应用启动的拦截功能

在之前的一篇文章中已经介绍了Android中的应用启动流程，这个流程一定要理解透彻，这样我们才可以进行后续的Hook。在之前还介绍了Android中如何Hook系统的剪切板服务实现方法的拦截效果，...

 jiangweibao 0410003 2016年09月27日 20:51 6750


### Android 牛化开发-hook 系统服务（通过binder修改粘贴板服务行为）

如果您还读第一部分的内容，这篇文章不需往下读，在阅读第一部分后才能继续下面的内容：Hook动态代理基于... 一篇博客，我们学习了代理的概念，以及如何寻找Hook点。本篇博客将继续拓展前文，不过这...

 u0136 2016年04月11日 13:52 3812

### android service 的动态更改UI和service重启问题

最近在写项目的时候遇到了service的多个问题，下面跟大家分享一下，简单说一下。再写音乐播放器时，音乐播放器写在服务中，虽然有时不把mediaplayer写在service中，有时也能实现后台播...

 gongzhiyao3739124 2016年07月11日 20:31 965