**Hook技术**

hook技术中，就是运用动态代理与反射实现的。hook技术一般是去获取系统framework 层的一些类，通过代理去在调用某个方法前及方法后插入一些额外的业务。

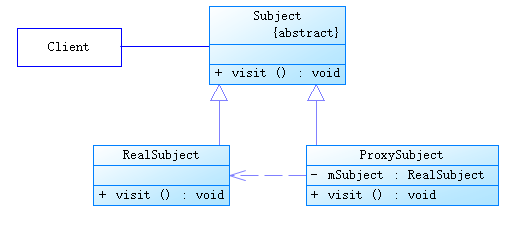
hook技术关键的地方是在找到hook点，即找到需要动态代理的那个类，一般需要通过反射获取，获取之后，通过动态代理去生成一个代理类，这样就可以重新去执行这个类的方法。

所以hook技术的核心就是反射和代理。反射在之前已经分析过了，就不做赘述。

**一．Java的代理模式**

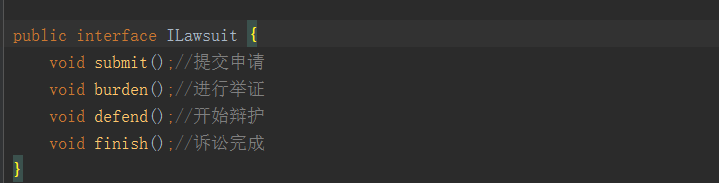
代理模式为另一个对象提供一个代理以控制对这个对象的访问。代理模式可以大致分为静态代理和动态代理。静态代理模式的代码由程序员自己或通过一些自动化工具生成固定的代码再对其进行编译，也就是说我们的代码在运行前代理类的 class 编译文件就已经存在；而动态代理则与静态代理相反，在 Java 或者 Android 中通过反射机制动态地生成代理者的对象，也就是说我们在 code 阶段完全不需要知道代理谁，代理谁我们将会在执行阶段决定，在 Java 中，也提供了相关的动态代理接口 InvocationHandler 类。

1. 静态代理

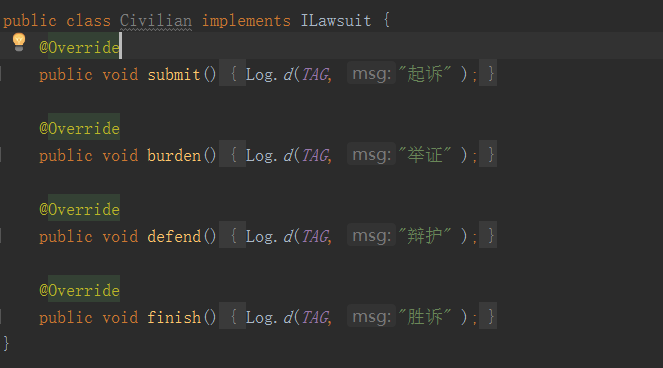


据此我们可以写出代理模式的例子，一个律师帮诉讼人代理申诉的例子

Subject.class 相当于例子中申诉这一件事，他一般是抽象类或者接口,相当于例子中的ILawsuit.



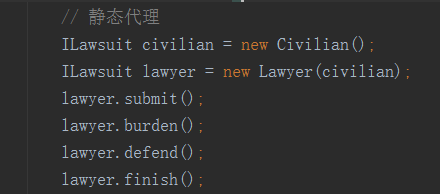
RealSubject.class相当于诉讼人，是真正要做申诉的人，相当于例子中的Civilian,需要实现ILawsuit接口。



ProxySubject.class相当于律师，代替诉讼人去做申诉的操作，相当于例子中的Lawyer



客户端使用：



Civilian相当于做了一个授权，让lawyer去完成所有的申诉操作。这就是静态代理

代理模式的角色：

Subject：抽象主题类

该类的主要职责是声明真实主题与代理的共同接口方法，该类既可以是一个抽象类，也可以是一个接口；

RealSubjct：真实主题类

该类也称为被委托类或被代理类，该类定义了代理所表示的真实对象，由其执行具体的业务逻辑方法，而客户端则通过代理类间接地调用真实主题类中定义的方法；

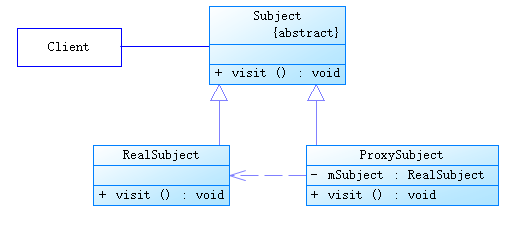
ProxySubject：代理类

该类也称为委托类或代理类，该类持有一个对真实主题类的引用，在其所实现的接口方法中调用真实主题类中对应的接口方法，以此起到代理的作用；

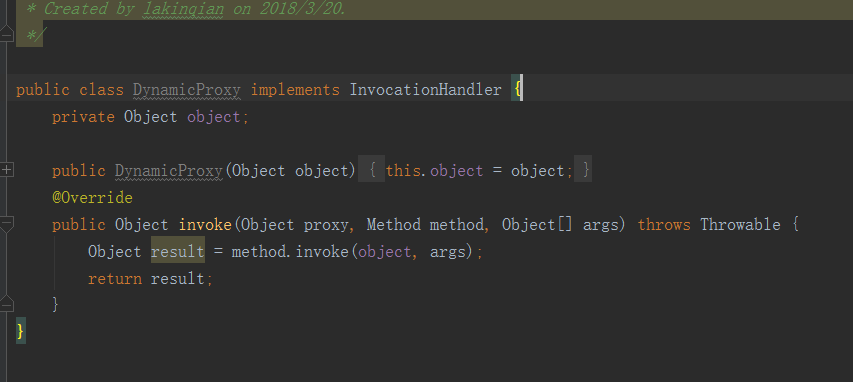
Client：客户类，即使用代理类的部分。

1. 动态代理

静态代理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。ProxySubject只能代理某个特定的类。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。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 则通过反射调用 RealSubject 的方法，这是规范动态的生成对应的 .class 二进制的过程，所以动态代理用过反射可以代理任意一个类。



动态代理类：上图中的ProxySubject相当于DynamicProxy，RealSubject相当于实现了ILawsuit接口的Civilian.



客户端使用：



当前是代理的Civilian类，当情况需要是，该代理类也可以代理实现任意其他接口的类。

其中Proxy.newProxyInstance（）的源码如下：



通过这个方法，先获取到代理类的对象，在生成代理类的实例，这样便可以去完成代理类的工作了。

[InvocationHandler](https://developer.android.com/reference/java/lang/reflect/InvocationHandler.html) 接口和 [Proxy](https://developer.android.com/reference/java/lang/reflect/Proxy.html) 类

我们来分析一下动态代理模式中 ProxySubject（DynamicProxy） 的生成步骤：

获取 RealSubject 上的所有接口列表；

确定要生成的代理类的类名，系统默认生成的名字为：com.sun.proxy.$ProxyXX；

根据需要实现的接口信息，在代码中动态创建该 ProxySubject 类的字节码；

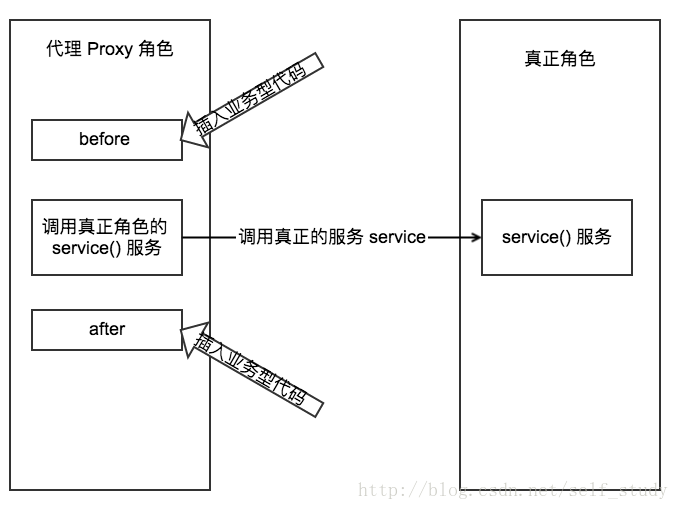
将对应的字节码转换为对应的 Class 对象；

创建 InvocationHandler 的实例对象 h，用来处理 Proxy 角色的所有方法调用；

以创建的 h 对象为参数，实例化一个 Proxy 角色对象。

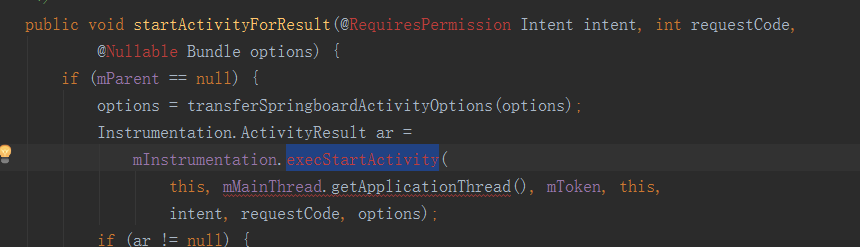
**二．Android 中利用动态代理实现 Hook 机制**

什么是hook机制，hook机制就是对系统framework层的一些类通过反射实现动态代理，来实现在某个方法前后去插入一些自己的业务，甚至直接修改该方法的实现。



Hook实现的第一步就是寻找hook点，也就是要去寻找你需要动态代理的那个类，下面通过hook StartActivity这个方法来分析hook的过程。

要hook StartActivity这个方法，我们首先应该分析StartActivity的最终实现。

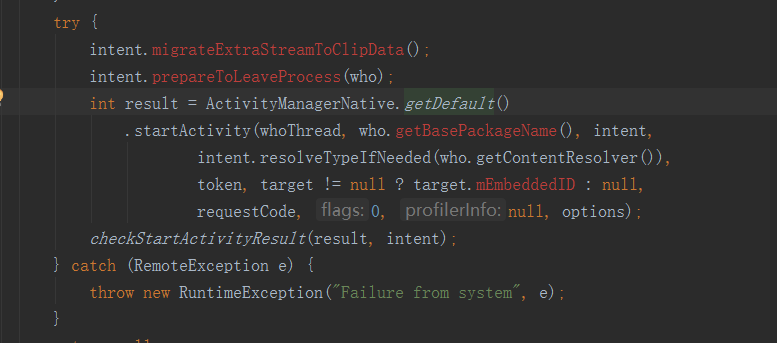


StartActivity最终是通过Instrumentation类的execStartActivity方法实现启动某个Activity的，而在execStartActivity方法中，启动activity的操作如下：

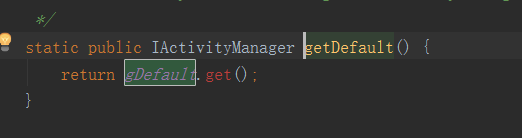
可以发现我们的hook点就是IActivityManager的实现类。

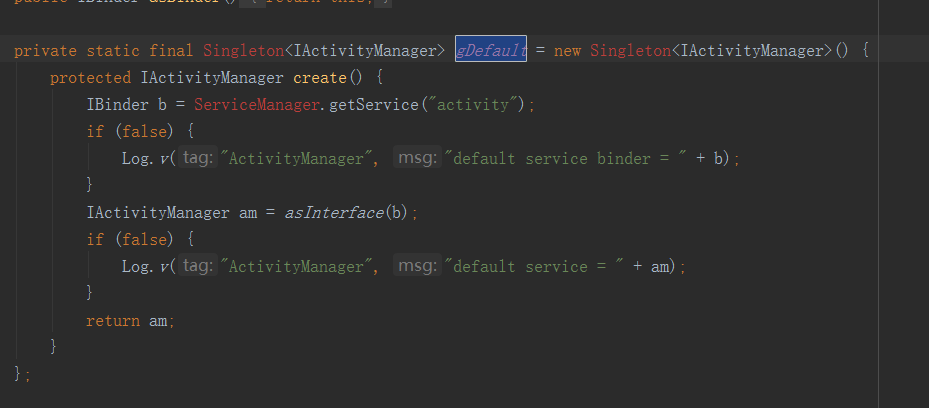
这个地方有一个坑需要注意，在android 8.0前后的实现IActivityManager实现类的方式是获取不一样的。也就是说反射路径不同：

Android 8.0以前：

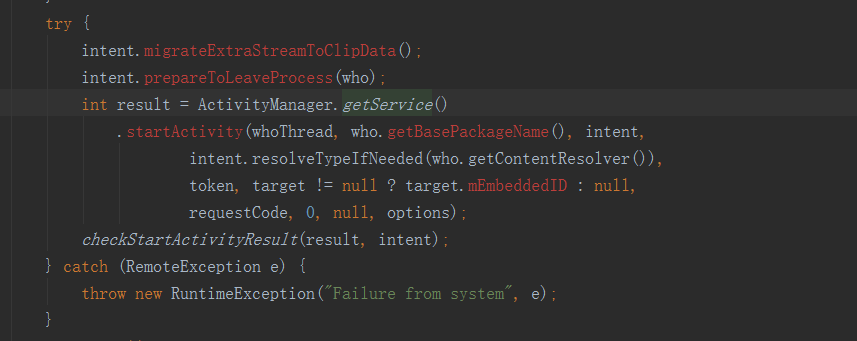


其中，获取IActivityManager实现类是通过ActivityManagerNative.getDefault()的实现，我们需要反射 ActivityManagerNative 的静态成员变量gDefault，最后获取我们需要代理的那个IActivityManager实现类。

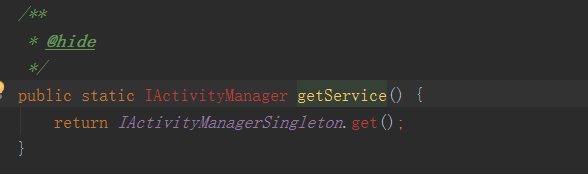


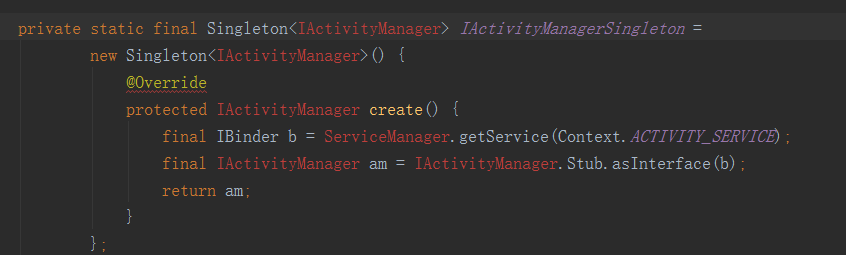


而对于API25以上，实现方式做了修改：



其中，但是他获取IActivityManager实现类的方法变了，通过ActivityManager.getService的实现，我们需要反射 ActivityManager的静态成员变量IActivityManagerSingleton，最后获取我们需要代理的那个IActivityManager实现类。

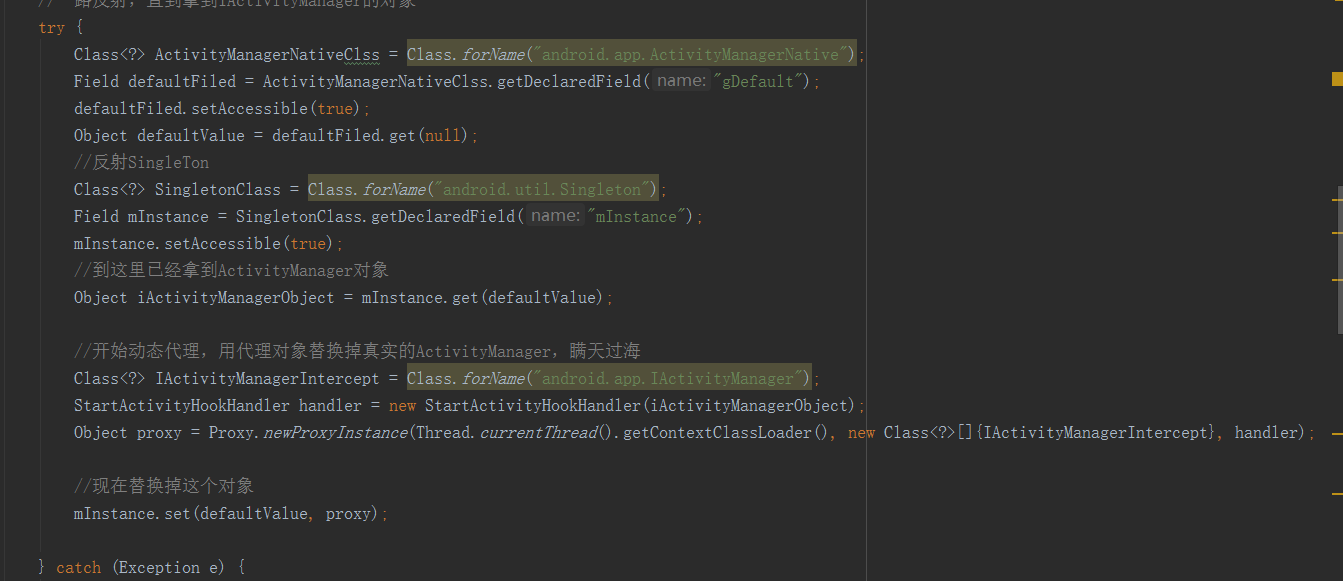




这两个方式都是通过反射去获取IActivityManager实现类，只是反射的路径变了，获取到实现类以后，我们就可以动态代理该类，然后再将他设置会原来的静态成员变量即可。

两个实现方式分别如下：

低于android 26



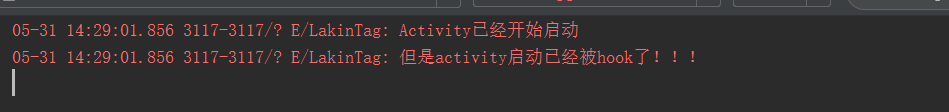
Android 26以上：



IActivityManager代理类：



最后在startActivity时，会打印如下日志：



这就是hook IActivityManager实现类，最后实现拦截startActivity的过程。