# Android插件化原理解析——Hook机制之动态代理

发表于    |   [141](http://weishu.me/2016/01/28/understand-plugin-framework-proxy-hook/" \l "comments)条评论   |   32484次阅读

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统API从而达到插件化的目的。本文将带你了解基于动态代理的Hook机制。

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dynamic-proxy-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/)。

## **代理是什么**

为什么需要代理呢？其实这个代理与日常生活中的“代理”，“中介”差不多；比如你想海淘买东西，总不可能亲自飞到国外去购物吧，这时候我们使用第三方海淘服务比如惠惠购物助手等；同样拿购物为例，有时候第三方购物会有折扣比如当初的米折网，这时候我们可以少花点钱；当然有时候这个“代理”比较坑，坑我们的钱，坑我们的货。

从这个例子可以看出来，代理可以实现**方法增强**，比如常用的日志,缓存等；也可以实现方法拦截，通过代理方法修改原方法的参数和返回值，从而实现某种不可告人的目的～接下来我们用代码解释一下。

## **静态代理**

## **注意：静态代理只适用于目标对象的目标方法是公有或者继承，而如果目标方法隐藏则没有办法，包括目标方法带有隐藏参数也没有办法使用这种方式**

静态代理，是最原始的代理方式；假设我们有一个购物的接口，如下：

|  |  |
| --- | --- |
| 1  2  3 | public interface Shopping {  Object[] doShopping(long money);  } |

它有一个原始的实现，我们可以理解为亲自，直接去商店购物：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class ShoppingImpl implements Shopping {  @Override  public Object[] doShopping(long money) {  System.out.println("逛淘宝 ,逛商场,买买买!!");  System.out.println(String.format("花了%s块钱", money));  return new Object[] { "鞋子", "衣服", "零食" };  }  } |

好了，现在我们自己没时间但是需要买东西，于是我们就找了个代理帮我们买：

|  |  |
| --- | --- |
| 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 | public class ProxyShopping implements Shopping {  Shopping base;  ProxyShopping(Shopping base) {  this.base = base;  }  @Override  public Object[] doShopping(long money) {  // 先黑点钱(修改输入参数)  long readCost = (long) (money \* 0.5);  System.out.println(String.format("花了%s块钱", readCost));  // 帮忙买东西  Object[] things = base.doShopping(readCost);  // 偷梁换柱(修改返回值)  if (things != null && things.length > 1) {  things[0] = "被掉包的东西!!";  }  return things;  } |

很不幸，我们找的这个代理有点坑，坑了我们的钱还坑了我们的货；先忍忍。

## **动态代理**

## **注意：这种方法只适用于目标对象是一个接口类型，因为父类存在的地方子类总能存在。比如目标对象是IBase obj其引用的本来是一个BaseImp这种情况就可以动态代理，而如果是AbsBase或者BaseImp这种情况下就不行了，AbsBase是一个虚基类,BaseImp是实现类，不是接口。**

传统的静态代理模式需要为每一个需要代理的类写一个代理类，如果需要代理的类有几百个那不是要累死？为了更优雅地实现代理模式，JDK提供了动态代理方式，可以简单理解为JVM可以在运行时帮我们动态生成一系列的代理类，这样我们就不需要手写每一个静态的代理类了。依然以购物为例，用动态代理实现如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public static void main(String[] args) {  Shopping women = new ShoppingImpl();  // 正常购物  System.out.println(Arrays.toString(women.doShopping(100)));  // 招代理  women = (Shopping) Proxy.newProxyInstance(Shopping.class.getClassLoader(),  women.getClass().getInterfaces(), new ShoppingHandler(women));  System.out.println(Arrays.toString(women.doShopping(100)));  } |

动态代理主要处理InvocationHandler和Proxy类；完整代码可以见[github](https://github.com/tiann/understand-plugin-framework)

## **代理Hook**

我们知道代理有比原始对象更强大的能力，比如飞到国外买东西，比如坑钱坑货；那么很自然，如果我们自己创建代理对象，然后把原始对象替换为我们的代理对象，那么就可以在这个代理对象为所欲为了；修改参数，替换返回值，我们称之为Hook。

下面我们Hook掉startActivity这个方法，使得每次调用这个方法之前输出一条日志；（当然，这个输入日志有点点弱，只是为了展示原理；只要你想，你想可以替换参数，拦截这个startActivity过程，使得调用它导致启动某个别的Activity，指鹿为马！）

首先我们得找到被Hook的对象，我称之为Hook点；什么样的对象比较好Hook呢？自然是**容易找到的对象**。什么样的对象容易找到？**静态变量和单例**；在一个进程之内，静态变量和单例变量是相对不容易发生变化的，因此非常容易定位，而普通的对象则要么无法标志，要么容易改变。我们根据这个原则找到所谓的Hook点。

然后我们分析一下startActivity的调用链，找出合适的Hook点。我们知道对于Context.startActivity（Activity.startActivity的调用链与之不同），由于Context的实现实际上是ContextImpl;我们看ConetxtImpl类的startActivity方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @Override  public void startActivity(Intent intent, Bundle options) {  warnIfCallingFromSystemProcess();  if ((intent.getFlags()&Intent.FLAG\_ACTIVITY\_NEW\_TASK) == 0) {  throw new AndroidRuntimeException(  "Calling startActivity() from outside of an Activity "  + " context requires the FLAG\_ACTIVITY\_NEW\_TASK flag."  + " Is this really what you want?");  }  mMainThread.getInstrumentation().execStartActivity(  getOuterContext(), mMainThread.getApplicationThread(), null,  (Activity)null, intent, -1, options);  } |

这里，实际上使用了ActivityThread类的mInstrumentation成员的execStartActivity方法；注意到，ActivityThread 实际上是主线程，而主线程一个进程只有一个，因此这里是一个良好的Hook点。

接下来就是想要Hook掉我们的主线程对象，也就是把这个主线程对象里面的mInstrumentation给替换成我们修改过的代理对象；要替换主线程对象里面的字段，首先我们得拿到主线程对象的引用，如何获取呢？ActivityThread类里面有一个静态方法currentActivityThread可以帮助我们拿到这个对象类；但是ActivityThread是一个隐藏类，我们需要用反射去获取，代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | // 先获取到当前的ActivityThread对象  Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");  Method currentActivityThreadMethod = activityThreadClass.getDeclaredMethod("currentActivityThread");  currentActivityThreadMethod.setAccessible(true);  Object currentActivityThread = currentActivityThreadMethod.invoke(null); |

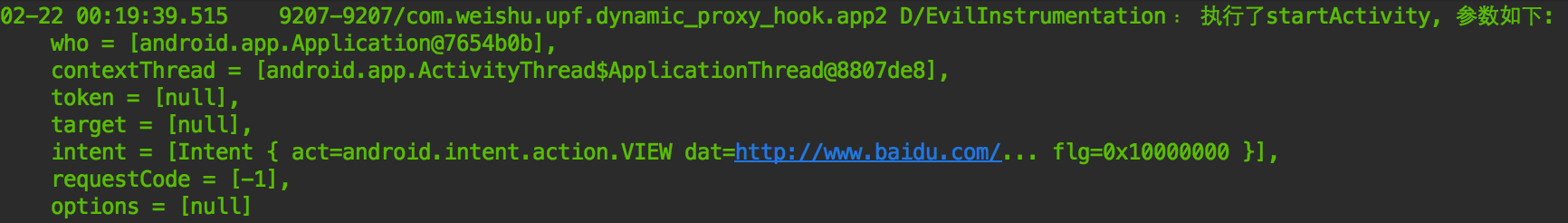
拿到这个currentActivityThread之后，我们需要修改它的mInstrumentation这个字段为我们的代理对象，我们先实现这个代理对象，由于JDK动态代理只支持接口，而这个Instrumentation是一个类，没办法，我们只有手动写静态代理类，覆盖掉原始的方法即可。（cglib可以做到基于类的动态代理，这里先不介绍）

|  |  |
| --- | --- |
| 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 | public class EvilInstrumentation extends Instrumentation {  private static final String TAG = "EvilInstrumentation";  // ActivityThread中原始的对象, 保存起来  Instrumentation mBase;  public EvilInstrumentation(Instrumentation base) {  mBase = base;  }  public ActivityResult execStartActivity(  Context who, IBinder contextThread, IBinder token, Activity target,  Intent intent, int requestCode, Bundle options) {  // Hook之前, XXX到此一游!  Log.d(TAG, "\n执行了startActivity, 参数如下: \n" + "who = [" + who + "], " +  "\ncontextThread = [" + contextThread + "], \ntoken = [" + token + "], " +  "\ntarget = [" + target + "], \nintent = [" + intent +  "], \nrequestCode = [" + requestCode + "], \noptions = [" + options + "]");  // 开始调用原始的方法, 调不调用随你,但是不调用的话, 所有的startActivity都失效了.  // 由于这个方法是隐藏的,因此需要使用反射调用;首先找到这个方法  try {  Method execStartActivity = Instrumentation.class.getDeclaredMethod(  "execStartActivity",  Context.class, IBinder.class, IBinder.class, Activity.class,  Intent.class, int.class, Bundle.class);  execStartActivity.setAccessible(true);  return (ActivityResult) execStartActivity.invoke(mBase, who,  contextThread, token, target, intent, requestCode, options);  } catch (Exception e) {  // 某该死的rom修改了 需要手动适配  throw new RuntimeException("do not support!!! pls adapt it");  }  }  } |

Ok，有了代理对象，我们要做的就是偷梁换柱！代码比较简单，采用反射直接修改：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public static void attachContext() throws Exception{  // 先获取到当前的ActivityThread对象  Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");  Method currentActivityThreadMethod = activityThreadClass.getDeclaredMethod("currentActivityThread");  currentActivityThreadMethod.setAccessible(true);  Object currentActivityThread = currentActivityThreadMethod.invoke(null);  // 拿到原始的 mInstrumentation字段  Field mInstrumentationField = activityThreadClass.getDeclaredField("mInstrumentation");  mInstrumentationField.setAccessible(true);  Instrumentation mInstrumentation = (Instrumentation) mInstrumentationField.get(currentActivityThread);  // 创建代理对象  Instrumentation evilInstrumentation = new EvilInstrumentation(mInstrumentation);  // 偷梁换柱  mInstrumentationField.set(currentActivityThread, evilInstrumentation);  } |

好了，我们启动一个Activity测试一下，结果如下：



可见，Hook确实成功了！这就是使用代理进行Hook的原理——偷梁换柱。整个Hook过程简要总结如下：

1. 寻找Hook点，原则是静态变量或者单例对象，尽量Hook pulic的对象和方法，非public不保证每个版本都一样，需要适配。
2. 选择合适的代理方式，如果是接口可以用动态代理；如果是类可以手动写代理也可以使用cglib。
3. 偷梁换柱——用代理对象替换原始对象

完整代码参照：[understand-plugin-framework](https://github.com/tiann/understand-plugin-framework)；里面留有一个作业：我们目前仅Hook了Context类的startActivity方法，但是Activity类却使用了自己的mInstrumentation；你可以尝试Hook掉Activity类的startActivity方法。

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/)!