Java代理模式讲解以及在Android上的应用

https://mp.weixin.qq.com/s?__biz=MzA5MzI3NjE2MA==&mid=2650242260&idx=1&sn=6fe14497cce4385c2c3a2d6fde339d14&chksm=88638dbbbf1404ad79f1a5b061c075ff61a5662f64f51c0834c983faaaaf2e0f06af49cc84f3&scene=38#wechat_redirect

Java代理模式讲解以及在Android上的应用

stormWen 郭霖 1月23日

今日科技快讯

小米近日已召开上市启动会，多个接近小米的市场人士透露，小米倾向于选择香港上市。但也有接近小米的手机行业人士亦称，关于上市时间和地点，小米公司内部目前暂无定论。

作者简介

本篇来自 **stormWen** 的投稿，分享了一个Java代理以及在Android中的一些简单应用实现技巧，希望大家会喜欢！

stormWen 的博客地址：

https://juejin.im/user/5743cb0ec26a38006c3b5c75

前言

Java中的代理设计模式(Proxy),提供了对目标对象另外的访问方式;即通过代理对象访问目标对象.这样做的好处是:可以在目标对象实现的基础上,增强额外的功能操作,即扩展目标对象的功能.代理设计模式在现实生活中无处不在，举个例子，一般明星都有个经纪人，这个经纪人的作用就是类似代理了，如果有谁想找明星，首先要通过经纪人，然后经纪人再转告明星去做某些事情，这里要注意的是:真正做事情的是明星本人，而不是经纪人，经纪人只是一个中间的角色而已，本身并不会做事情，比如唱歌是明星来唱歌，而不是经纪人来唱歌，当然了，经纪人可以做些额外的事情，比如收取一定的服务费用，然后才允许你见明星之类，这个理解很重要，因为经纪人可以做些额外的事情，"这个额外的事情"就为程序中提供了很多扩展的功能，下面的例子可以看到这些额外的功能，再举个例子，大家都买过房吧，一般情况下是由中介带着你去，而这个中介也是类似代理人的作用，你也要明白，真正的买房人是你，你负责出钱，中介只是负责传达你的要求而已，当然啦，一些额外的功能就是需要出点钱给中介，毕竟人家那么辛苦，坑点茶水费还是要的，大家都懂，上次在买房也被坑过一些钱，后来想起了这个代理模式，也就理解了，哈哈,大家会发现，设计模式都是源于生活，在生活中处处都有设计模式的影子,我觉得学习设计模式一定要理解，理解，理解，重要的事情说三遍。

正文

代理模式的分类

静态代理，指的是在编译的时候就已经存在了，需要定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类，下面举个例子，以买房为例子。

首先定义一个购买的接口和方法

```
public interface Buy {
    void buyHouse(long money);
}
```

然后假设一个用户去买房

```
public class User implements Buy {
    @Override
    public void buyHouse(long money) {
        System.out.println("买房了，用了"+money+" 钱 ");
    }
}
```

我们先来运行一下

```
public class ProxyClient {
    public static void main(String[] args){
        Buy buy=new User();
        buy.buyHouse(1000000);
    }
}
```

测试结果为

现在我们来弄一个中介来帮我们买房，顺便加上一些额外的功能,就是坑点我们的血汗钱，Fuck，万恶的中介，开玩笑,代码如下

```
public class UserProxy implements Buy {
    /**
     *这个是真实对象，买房一定是真实对象来买的，中介只是跑腿的
     */
    private Buy mBuy;
    public UserProxy(Buy mBuy) {
        this.mBuy = mBuy;
    }

    @Override
    public void buyHouse(long money) {
        long newMoney= (long) (money*0.99);
        System.out.println("这里坑点血汗钱，坑了我们:"+(money-newMoney)+"钱");
        /**
         * 这里是我们出钱去买房,中介只是帮忙
         */
        mBuy.buyHouse(newMoney);
    }
}

public class ProxyClient {
    public static void main(String[] args){
        Buy buy=new User();
        UserProxy proxy=new UserProxy(buy);
        proxy.buyHouse(1000000);
    }
}
```

运行结果如下

看到没，中介帮我们跑腿买房，被坑了10000元，结果皆大欢喜，我们买到房了，而"额外功能""中介也赚了点辛苦费，大家都开心，这就是静态代理的理解，可以看到，在编译时就已经决定的了.

动态代理，顾名思义是动态的，Java中的动态一般指的是在运行时的状态，是相对编译时的静态来区分，就是在运行时生成一个代理对象帮我们干活，还是以买房为例子，如果静态代理是我们在还没有买房的时候(就是编译的时候)预先找好的中介，那么动态代理就是在买房过程中找的，注意:买房过程中说明是在买房这件事情的过程中，就是代码在运行的时候才找的一个中介，Java中的动态代理要求必须实现一个接口，InvocationHandler，动态代理也必须有一个真实的对象，不管是什么代理，只是帮忙传达指令，最终还是必须有原来的对象去干活的。

下面是代码

```
public class DynamiclProxy implements InvocationHandler {
    //真正要买房的对象
    private Buy mObject;

    public DynamiclProxy(Buy mObject) {
        this.mObject = mObject;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        if (method.getName().equals("buyHouse")){
            //如果方法是买房的话，那么先坑点钱
```

```
            long money= (long) args[0];//取出第一个参数，因为我们是知道参数的
            long newMoney= (long) (money*0.99);
            System.out.println("坑了我们:"+(money-newMoney)+"钱");
            args[0]=newMoney;//坑完再赋值给原来的参数
            /**
             * 调用真实对象去操作
             */
            return method.invoke(mObject,args);
        }
        //如果有其他方法，也可以跟上面这样判断
        return null;
    }
}
```

看到没，都是需要一个真实对象的引用，然后在接口方法中做自己的逻辑，最后接口方法帮我们去实现自己的逻辑，我前面已经说过了，不管是什么代理，都是真实的对象去做行为，代理只是帮忙做事情跑腿的， 测试代码:

```
public class ProxyClient {
    public static void main(String[] args){
        Buy buy=new User();
        UserProxy proxy=new UserProxy(buy);
        proxy.buyHouse(1000000);

        System.out.println("动态代理测试");
        Buy dynamicProxy= (Buy) Proxy.newProxyInstance(buy.getClass().getClassLoader(),
                buy.getClass().getInterfaces(),new DynamiclProxy(buy));
        dynamicProxy.buyHouse(1000000);
    }
}
```

下面是运行结果

可以看到运行的结果跟静态代理的是一样的,顺便提下，动态代理不仅在Java中有重要的作用，特别是AOP编程方面，更是在Android的插件话发挥了不可或缺的作用，我们前面说过Java层的Hook一般有反射和动态代理2个方面，一般情况下是成对出现的，反射是负责找出隐藏的对象，而动态代理则是生成目标接口的代理对象，然后再由反射替换掉，一起完成有意思的事情，下面我们简单来分析一下动态代理的内部原理实现:首先是生成class文件，如下代码:

```
public static void createProxyClassFile() {
        String name = "ProxyClass";
        byte[] data = ProxyGenerator.generateProxyClass(name, new Class[]{Buy.class});
        try {
            FileOutputStream out = new FileOutputStream(name + ".class");
            out.write(data);
            out.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

在项目根目录下面会有ProxyClass.class文件生成，我们简单分析下源码:

看到这里的继承关系，我想有些读者已经看懂了一个动态代理的缺陷了，因为生成的类已经有一个Proxy父类了，因此注定了不是接口的不能使用动态代理，因为java的继承机制所决定的，但通过第三方cglib可以实现，大家可以去试试，虽然有点遗憾，但不会影响其发挥的巨大作用，我们来看看是如何生成动态代理的子类，看方法:Proxy.newProxyInstance(),下面是代码:

```
public static Object newProxyInstance(ClassLoader loader,
                                       Class<?>[] interfaces,
                                       InvocationHandler h)
        throws IllegalArgumentException
    {
        Objects.requireNonNull(h);

        final Class<?>[] intfs = interfaces.clone();
        final SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
```

```
            checkProxyAccess(Reflection.getCallerClass(), loader, intfs);
        }

        /*
         * Look up or generate the designated proxy class.
         */
        Class<?> cl = getProxyClass0(loader, intfs);

        /*
         * Invoke its constructor with the designated invocation handler.
         */
        try {


            final Constructor<?> cons = cl.getConstructor(constructorParams);
            final InvocationHandler ih = h;
            if (!Modifier.isPublic(cl.getModifiers())) {
                AccessController.doPrivileged(new PrivilegedAction<Void>() {
                    public Void run() {
                        cons.setAccessible(true);
                        return null;
                    }
                });
            }
            return cons.newInstance(new Object[]{h});
        }
        ...省略了一些非关键代码
    }
```

可以看到，是通过反射构造函数来创建子类的，而构造函数里面的参数constructorParams正是接口类型

```
/** parameter types of a proxy class constructor */
private static final Class<?>[] constructorParams =
        { InvocationHandler.class };
```

下面我们分析一下是如何生成字节码的:

```
public static byte[] generateProxyClass(final String name,
                                        Class[] interfaces)
    {
        ProxyGenerator gen = new ProxyGenerator(name, interfaces);
    // 这里是关键
        final byte[] classFile = gen.generateClassFile();

    if(saveGeneratedFiles) {
            AccessController.doPrivileged(new PrivilegedAction() {
                public Void run() {
                    try {
                        int var1 = var0.lastIndexOf(46);
                        Path var2;
                        if(var1 > 0) {
                            Path var3 = Paths.get(var0.substring(0, var1).replace('.',
File.separatorChar), new String[0]);
                            Files.createDirectories(var3, new FileAttribute[0]);
                            var2 = var3.resolve(var0.substring(var1 + 1, var0.length()) + ".class");
                        } else {
                            var2 = Paths.get(var0 + ".class", new String[0]);
                        }
                        //这个是重点，写在本地磁盘上
                        Files.write(var2, var4, new OpenOption[0]);
                        return null;
                    } catch (IOException var4x) {
                        throw new InternalError("I/O exception saving generated file: " + var4x);
                    }
                }
```

```
            });

    // 返回代理类的字节码
        return classFile;
        }

        //generateClassFile方法比较多，都是一些字节码的编写
        我们重点看下最终写到哪里去了

         ByteArrayOutputStream var13 = new ByteArrayOutputStream();
            DataOutputStream var14 = new DataOutputStream(var13);

            try {
                var14.writeInt(-889275714);
                var14.writeShort(0);
                var14.writeShort(49);
                this.cp.write(var14);
                var14.writeShort(this.accessFlags);
                var14.writeShort(this.cp.getClass(dotToSlash(this.className)));
                var14.writeShort(this.cp.getClass("java/lang/reflect/Proxy"));
                var14.writeShort(this.interfaces.length);
                Class[] var17 = this.interfaces;
                int var18 = var17.length;

                for(int var19 = 0; var19 < var18; ++var19) {
                    Class var22 = var17[var19];
                    var14.writeShort(this.cp.getClass(dotToSlash(var22.getName())));
                }

                var14.writeShort(this.fields.size());
                var15 = this.fields.iterator();

                while(var15.hasNext()) {
                    ProxyGenerator.FieldInfo var20 = (ProxyGenerator.FieldInfo)var15.next();
                    var20.write(var14);
                }

                var14.writeShort(this.methods.size());
                var15 = this.methods.iterator();

                while(var15.hasNext()) {
                    ProxyGenerator.MethodInfo var21 = (ProxyGenerator.MethodInfo)var15.next();
                    var21.write(var14);
                }

                var14.writeShort(0);
                return var13.toByteArray();
            } catch (IOException var9) {
                throw new InternalError("unexpected I/O Exception", var9);
            }
            很明显了，最终是写在本地磁盘上来1
```

现在我们知道了是写到本地磁盘上的字节码，然后生成一个代理的对象，那么是如何调用具体的方法呢，在刚才那个文件ProxyClass.class告诉了我们，比如我们的接口方法:

```
public final void buyHouse(long var1) throws  {
    try {
        super.h.invoke(this, m3, new Object[]{Long.valueOf(var1)});
    } catch (RuntimeException | Error var4) {
        throw var4;
    } catch (Throwable var5) {
        throw new UndeclaredThrowableException(var5);
    }
```

```
}
```

看到没是h来调用的，那么h从哪里来的呢？构造函数，这个构造函数就是我们代码中的 new DynamiclProxy(buy))中buy就是这里的参数var1,也就是真正需要代理的对象赋值给了父类Proxy.中的h;对象。这个类里面的其他方法的执行都是这样的流程走的，比如object类的HasCode方法等。

看到这里，我想基本明白了吧，动态代理就一句话:系统帮我们生成了字节码文件保存在本地并生成一个InvocationHandler的代理子类，然后通过我们传进去的真实对象的引用，再帮忙调用各种接口方法，最终所有的方法都走

```
public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable
```

从而我们可以在里面根据实际情况做不同的业务处理，比如统计耗时，替换参数(这些就是所谓的额外的功能)等等,大家有时间可以去多看那些涉及到的源码就好。

除了上面提到的静态代理和动态代理，还有一种代理称为远程代理的，这个在Android比较广泛应用，特别是在IPC过程或者Binder机制中不可或缺，这种交互一般发生在不同的进程之间，所以一般称为远程代理模式。

代理模式在Android中的应用

我们前面说了，Java中代理模式一般三种，其中动态代理用的比较多，比较灵活，而且有时候由于接口是隐藏的，也不好用静态代理，因此大多数时候都是用动态代理比较多，远程代理一般在不同进程之间使用，这里先简单介绍一下远程代理，比如我们熟悉的Activity的启动过程，其实就隐藏了远程代理的使用，由于APP本地进程和AMS(ActivityManagerService)进程分别属于不同的进程，因此在APP进程内，所有的AMS的实例其实都是经过Binder驱动处理的代理而已，大家要明白，真正的实例只有一个的，就是在AMS进程以内，其他进程之外的都不过是经过Binder处理的代理傀儡而已，还是先拿出这个启动图看看:

可以看到APP进程和AMS进程之间可以相互调用，其实就是靠各自的远程代理对象进行调用的，而不可能之间调用(进程隔离的存在) 就是APP本地进程有AMS的远端代理ActivityManagerProxy，有了这个代理，就可以调用AMS的方法了，而AMS也一样，有了ActivityThread的代理对象ApplicationThreadProxy，也可以调用APP本地进程的方法了，大家要明白，这些代理对象都是一个傀儡而已，只是Binder驱动处理之后的真实对象的引用，跟买房中介一样的性质，实际上所有Binder机制中的所谓的获取到的"远程对象"，都不过是远程真实对象的代理对象，只不过这个过程是驱动处理，对我们透明而已，有兴趣的同学可以去看看Native的源码，相信体会的更深.下面我们利用动态代理来有意义的事情。

现在大家的项目中估计都有引入了好多个第三方的库吧，大部分是远程依赖的，有些引用库会乱发通知的，但是这些代码因为对我们不可见，为了方便对通知的统一管理，我们有必要对系统中的所有通知进行统一的控制，我们知道，通知是用NotificationManager来管理的，实际上这个不过是服务端对象在客户端对象的一个代理对象的包装， 也就是说最终的管理还是在远端进程里面，客户端的作用只是包装一下参数，通过Binder机制发到服务端进行处理而已，我们先看一下代码:

```
private static INotificationManager sService;
/** @hide */
static public INotificationManager getService()  {
    if (sService != null) {
        return sService;
    }
    IBinder b = ServiceManager.getService("notification");
    sService = INotificationManager.Stub.asInterface(b);
    return sService;
}
```

这是Binder机制的内容，首先ServiceManager通过 getService方法获取了一个原生的裸的IBinder对象，然后通过AIDL机制的asInterface方法转换成了本地的代理对象，而我们在通知中的所有的操作都是有这个sService发起的，当然了，sService也是什么事情都干不了，只是跑腿，包装参数发送给真正的远程服务对象去做真正的事情，顺便提一下，Android系统中的绝大多数服务都在以这样形式而存在的，只有少数的比如AMS,PMS是以单列形式存在，因为AMS,PMS比较常用，按照常规的套路，先反射出sService字段，然后我们利用动态代理生成一个伪造的sService对象替换掉，代替我们的工作，这样所有的方法调用都会走动态代理的方法，这个我们前面已经说过了

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
```

这样的话，我们就可以通过选择某些方法来做些自己想要的事情，比如判断参数，然后选择屏蔽之类，好了，我们写一波代码先:

```
public static void hookNotificationManager(Context context) {
    try {
        NotificationManager notificationManager = (NotificationManager)
```

```
context.getSystemService(Context.NOTIFICATION_SERVICE);
        Method method = notificationManager.getClass().getDeclaredMethod("getService");
        method.setAccessible(true);
        //获取代理对象
        final Object sService = method.invoke(notificationManager);
        Log.d("[app]", "sService=" + sService);
        Class<?> INotificationManagerClazz = Class.forName("android.app.INotificationManager");
        Object proxy = Proxy.newProxyInstance(INotificationManagerClazz.getClassLoader(),
                        new Class[]{INotificationManagerClazz},new NotifictionProxy(sService));
        //获取原来的对象
        Field mServiceField = notificationManager.getClass().getDeclaredField("sService");
        mServiceField.setAccessible(true);
        //替换
        mServiceField.set(notificationManager, proxy);
        Log.d("[app]", "Hook NoticeManager成功");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public class NotifictionProxy implements InvocationHandler {
    private Object mObject;

    public NotifictionProxy(Object mObject) {
        this.mObject = mObject;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Log.d("[app]", "方法为:" + method.getName());
        /**
         * 做一些业务上的判断
         * 这里以发送通知为准,发送通知最终的调用了enqueueNotificationWithTag
         */
        if (method.getName().equals("enqueueNotificationWithTag")) {
            //具体的逻辑
            for (int i = 0; i < args.length; i++) {
                if (args[i]!=null){
                    Log.d("[app]", "参数为:" + args[i].toString());
                }
            }
            //做些其他事情，然后替换参数之类
            return method.invoke(mObject, args);
        }
        return null;
    }
}
```

好了，我们在Application里面的attachBaseContext()方法里面注入就好，为什么要在这里注入呢，因为attachBaseContext()在四大组件中的方法是最先执行的，比ContentProvider的onCreate()方法都先执行，而ContrentProvider的onCreate()方法比Application的onCreate()都先执行，大家可以去测试一下,因此如果Hook的地方是涉及到ContentProvider的话，那么最好在这个地方执行，我们在页面发送通知试试:,代码如下:

```
Intent intent=new Intent();
Notification build = new NotificationCompat.Builder(MotionActivity.this)
                        .setContentTitle("测试通知")
                        .setContentText("测试通知内容")
                        .setAutoCancel(true)
                        .setDefaults(Notification.DEFAULT_SOUND|Notification.DEFAULT_VIBRATE)
                        .setPriority(NotificationCompat.PRIORITY_MAX)
                        .setSmallIcon(R.mipmap.ic_launcher)
                        .setWhen(System.currentTimeMillis())
                        .setLargeIcon(BitmapFactory.decodeResource(getResources(), R.mipmap.ic_launcher))
```

```
                        .setContentIntent(PendingIntent.getService(MotionActivity.this, 0, intent,
PendingIntent.FLAG_UPDATE_CURRENT))
                        .build();
NotificationManager manager = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
manager.notify((int) (System.currentTimeMillis()/1000L), build);
```

好了，我们看一下结果吧:

结语

看到结果了吧，已经成功检测到被Hook的方法了，而具体如何执行就看具体的业务了。至此Java中的常用Hook手段:反射和动态代理就到此为止了，但实际上他们还有很多地方值得去使用，研究，只是限于篇幅，不在一一说明，以后如果有涉及到这方面的会再次提起的，大家有空可以研究源码，还是那句话，源码就是最好的学习资料

实际上Android上的很多服务都可以用类似的手段去处理，除了在Hook之外的应用外，在动态代理里面也有广泛的应用的，在以后写性能优化的时候会提出来的，感谢大家阅读，欢迎提出改进意见，不甚感谢。

项目地址：

https://juejin.im/post/5a2e4e9a51882559e2259ad3

如果有什么问题欢迎留言哈，谢谢！