Android设计模式源码解析之Proxy模式

> 本文为 Android 设计模式源码解析 中 Proxy模式 分析
> Android系统版本： 5.0
> 分析者：singwhatiwanna，分析状态：完成，校对者：Mr.Simple，校对状态：未校对

Binder中的代理模式

再说Binder中的代理模式之前，我们需要先看看代理模式的简单实现，这一部分内容采用了《JAVA与模式》之代理模式这篇文章中的代码示例和uml类图。

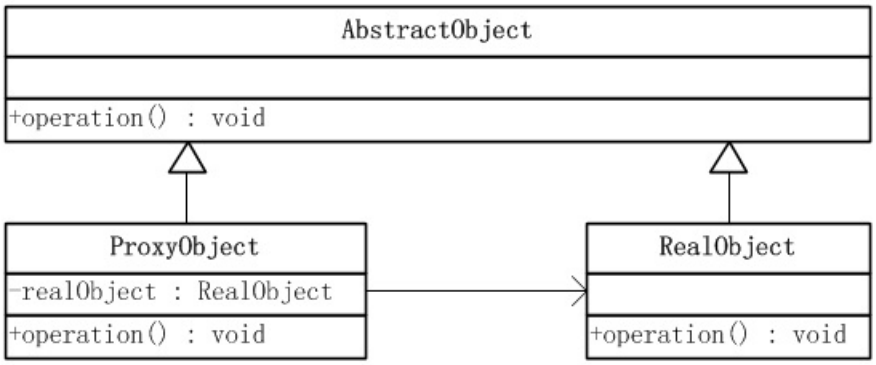
1. 模式介绍

代理模式是对象的结构模式。代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。

模式的使用场景

就是一个人或者机构代表另一个人或者机构采取行动。在一些情况下，一个客户不想或者不能够直接引用一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

2. UML类图



角色介绍

- 抽象对象角色：声明了目标对象和代理对象的共同接口，这样一来在任何可以使用目标对象的地方都可以使用代理对象。
- 目标对象角色：定义了代理对象所代表的目标对象。
- 代理对象角色：代理对象内部含有目标对象的引用，从而可以在任何时候操作目标对象；代理对象提供一个与目标对象相同的接口，以便可以在任何时候替代目标对象。代理对象通常在客户端调用传递给目标对象之前或之后，执行某个操作，而不是单纯地将调用传递给目标对象。

3. 模式的简单实现

简单实现的介绍

下面通过一种抽象的方式来实现下代理模式

实现源码

抽象对象角色

```
public abstract class AbstractObject {
    //操作
    public abstract void operation();
}
```

目标对象角色

```
public class RealObject extends AbstractObject {
    @Override
    public void operation() {
        //一些操作
        System.out.println("一些操作");
    }
}
```

代理对象角色

```
public class ProxyObject extends AbstractObject{
    RealObject realObject = new RealObject();
    @Override
    public void operation() {
        //调用目标对象之前可以做相关操作
        System.out.println("before");
        realObject.operation();
        //调用目标对象之后可以做相关操作
        System.out.println("after");
    }
}
```

客户端

```
public class Client {
    public static void main(String[] args) {
        AbstractObject obj = new ProxyObject();
        obj.operation();
    }
}
```

4. 代理模式在Binder中的使用

直观来说，Binder是Android中的一个类，它继承了IBinder接口。从IPC角度来说，Binder是Android中的一种跨进程通信方式，Binder还可以理解为一种虚拟的物理设备，它的设备驱动是/dev/binder，该通信方式在linux中没有；从Android Framework角度来说，Binder是ServiceManager连接各种Manager（ActivityManager、WindowManager，etc）和相应ManagerService的桥梁；从Android应用层来说，Binder是客户端和服务端进行通信的媒介，当你bindService的时候，服务端会返回一个包含了服务端业务调用的Binder对象，通过这个Binder对象，客户端就可以获取服务端提供的服务或者数据，这里的服务包括普通服务和基于AIDL的服务。

Binder一个很重要的作用是：将客户端的请求参数通过Parcel包装后传到远程服务端，远程服务端解析数据并执行对应的操作，同时客户端线程挂起，当服务端方法执行完毕后，再将返回结果写入到另外一个Parcel中并将其通过Binder传回到客户端，客户端接收到返回数据的Parcel后，Binder会解析数据包中的内容并将原始结果返回给客户端，至此，整个Binder的工作过程就完成了。由此可见，Binder更像一个数据通道，Parcel对象就在这个通道中跨进程传输，至于双方如何通信，这并不负责，只需要双方按照约定好的规范去打包和解包数据即可。

为了更好地说明Binder，这里我们先手动实现了一个Binder。为了使得逻辑更清晰，这里简化一下，我们来模拟一个银行系统，这个银行提供的功能只有一个：即查询余额，只有传递一个int的id过来，银行就会将你的余额设置为id*10，满足下大家的发财梦。

1. 先定义一个Binder接口 ``` package com.ryg.design.manualbinder;

import android.os.IBinder; import android.os.IInterface; import android.os.RemoteException;

public interface IBank extends IInterface {

```
static final String DESCRIPTOR = "com.ryg.design.manualbinder.IBank";

static final int TRANSACTION_queryMoney = (IBinder.FIRST_CALL_TRANSACTION + 0);

public long queryMoney(int uid) throws RemoteException;
```

}

```
2.创建一个Binder并实现这个上述接口
```

2.创建一个Binder并实现IBank接口

package com.ryg.design.manualbinder;

import android.os.Binder; import android.os.IBinder; import android.os.Parcel; import android.os.RemoteException;

public class BankImpl extends Binder implements IBank {

```
public BankImpl() {
    this.attachInterface(this, DESCRIPTOR);
}

public static IBank asInterface(IBinder obj) {
    if ((obj == null)) {
        return null;
    }
    android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
    if (((iin != null) && (iin instanceof IBank))) {
        return ((IBank) iin);
    }
    return new BankImpl.Proxy(obj);
}

@Override
public IBinder asBinder() {
    return this;
}

@Override
public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
        throws RemoteException {
    switch (code) {
    case INTERFACE_TRANSACTION: {
        reply.writeString(DESCRIPTOR);
        return true;
    }
    case TRANSACTION_queryMoney: {
        data.enforceInterface(DESCRIPTOR);
        int uid = data.readInt();
        long result = this.queryMoney(uid);
        reply.writeNoException();
        reply.writeLong(result);
        return true;
    }
    }
    return super.onTransact(code, data, reply, flags);
}

@Override
public long queryMoney(int uid) throws RemoteException {
    return uid * 10l;
}

private static class Proxy implements IBank {
    private IBinder mRemote;

    Proxy(IBinder remote) {
        mRemote = remote;
    }

    @Override
    public IBinder asBinder() {
        return mRemote;
    }

    public java.lang.String getInterfaceDescriptor() {
        return DESCRIPTOR;
    }

    @Override
```

```
    public long queryMoney(int uid) throws RemoteException {
        Parcel data = Parcel.obtain();
        Parcel reply = Parcel.obtain();
        long result;
        try {
            data.writeInterfaceToken(DESCRIPTOR);
            data.writeInt(uid);
            mRemote.transact(TRANSACTION_queryMoney, data, reply, 0);
            reply.readException();
            result = reply.readLong();
        } finally {
            reply.recycle();
            data.recycle();
        }
        return result;
    }

}
```

}

ok，到此为止，我们的Binder就完成了，这里只要创建服务端和客户端，二者就能通过我们的Binder来通信了。这里就不做这个示例了，我们的目的是分析代理模式在Binder中的使用。

我们看上述Binder的实现中，有一个叫做“Proxy”的类，它的构造方法如下：

Proxy(IBinder remote) { mRemote = remote; }

Proxy类接收一个IBinder参数，这个参数实际上就是服务端Service中的onBind方法返回的Binder对象在客户端重新打包后的结果，因为客户端无法直接通过这个打包的Binder和服务端通信，因此客户端必须借助Proxy类来和服务端通信，这里Proxy的作用就是代理的作用，客户端所有的请求全部通过Proxy来代理，具体工作流程为：Proxy接收到客户端的请求后，会将客户端的请求参数打包到Parcel对象中，然后将Parcel对象通过它内部持有的Ibinder对象传送到服务端，服务端接收数据、执行方法后返回结果给客户端的Proxy，Proxy解析数据后返回给客户端的真正调用者。很显然，上述所分析的就是典型的代理模式。至于Binder如何传输数据，这涉及到很底层的知识，这个很难搞懂，但是数据传输的核心思想是共享内存。

5. 杂谈
优点与缺点
优点
* 给对象增加了本地化的扩展性，增加了存取操作控制

缺点
* 会产生多余的代理类