DesignPattern-Proxy

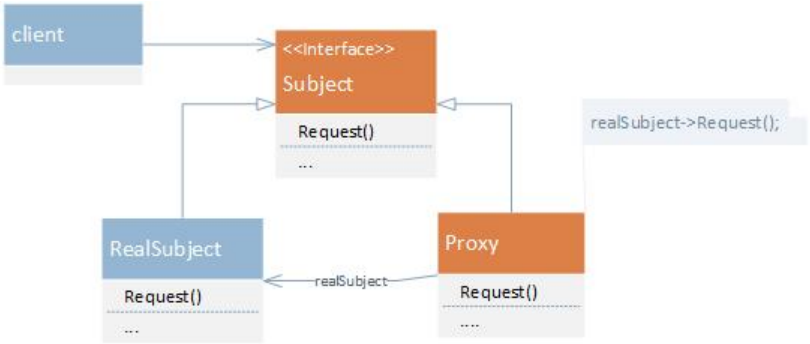
作者：wbq813
原文链接：https://ld246.com/article/1569334383695
来源网站：链滴

- **动机**

在面向对象系统中，有些对象由于某种原因（比如对象创建的开销很大，或者某些操作需要安全控制或者需要进程外的访问等），直接访问会给使用者、或者系统结构带来很多麻烦。如何在不是去透明作对象的同时来管理/控制这些对象特有的复杂性？增加一层间接层是软件开发中常见的解决方式。

- **定义**

为其他对象提供一种代理以控制（隔离，使用接口）对这个对象的访问。



- **总结**

1. “增加一层间接层”是软件系统中对许多复杂问题的一种常见解决方法。在面向对象系统中，直接用某些对象会带来许多问题，作为间接层的proxy对象便是解决这一问题的常用手段。

2. 具体proxy设计模式的实现方法、实现粒度都相差很大，有些可能对单个对象做细粒度的控制，如cpy-on-write技术，有些对象可能对组件模块提供抽象代理层，在架构层次对对象做proxy。

3. Proxy 并不一定要求保持接口完整的一致性，只要能够实现间接控制，有时候损及一些透明性是可接受的。

Java 动态代理机制

- **静态代理模式**

以下代码基本符合顶部设计模式类图的内容，静态代理类使得在委托类内部变化的情况下，Client无关注，同时在代理类可以实现自定义的修改（过滤判断，权限控制等）无需修改委托类。

```
public interface ISubject {
    String request();
}

class RealSubject implements ISubject{
    @java.lang.Override
    public String request() {
        return "hello world!";
    }
}

class SubStaticProxy implements ISubject{
    private RealSubject reaSubject;

    public SubStaticProxy(RealSubject reaSubject) {
        this.reaSubject = reaSubject;
    }
```

```
    @Override
    public String request() {
        return "before request.\n"+reaSubject.request()+"\nAfter request.";
    }
}
```

但是，如果需要代理多个接口就需要提供对应的代理类，或者修改代理类的代码（一个代理类代理多接口），如何让代理类在委托类数量增加的情况下固定不变？

- **动态代理**

下面的代码实现了动态代理，Client可以根据需要绑定不同的委托类，系统将会自动根据绑定的委托动态生成代理类。

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class SubDynamicProxy implements InvocationHandler {
    private Object realObject;

    public Object bind(Object realObject) {
        this.realObject = realObject;
        return Proxy.newProxyInstance(
                realObject.getClass().getClassLoader(),
                realObject.getClass().getInterfaces(),
                this
        );
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("addition process.");
        return method.invoke(realObject, args);
    }
}

class Client{
    public static void main(String[] args) {
        // static proxy
        ISubject sub = new SubStaticProxy(new RealSubject());
        System.out.println(sub.request());

        // dynamic proxy
        //加上这句将会产生一个$Proxy0.class文件，这个文件即为动态生成的代理类文件
        System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles","true");
        ISubject dySub = (ISubject) new SubDynamicProxy().bind(new RealSubject());
        System.out.println(dySub.request());
    }
}
```

代理类是如何生成，又是如何调用 Override 的invoke函数的：

设置属性sun.misc.ProxyGenerator.saveGeneratedFiles为true 可以在运行时看到生成的代理类 $Proxy0.class.

查看newProxyInstance的源码，下面是关键部分：

```
// 复制接口
final Class<?>[] intfs = interfaces.clone();
// 生成代理类
Class<?> cl = getProxyClass0(loader, intfs);
// 用生成的代理类生成实例并返回
final Constructor<?> cons = cl.getConstructor(constructorParams);
return cons.newInstance(new Object[]{h});
```

getProxyClass0() 最终调用ProxyGenerator中的generateClassFile() 函数实现class文件的生成：

1. 调用 generateMethod()添加equal、toString、hashCode这样的默认方法；

2. 循环获取每个接口内的每个方法，调用 generateMethod()添加到代理类。

3. cons.newInstance(new Object[]{h}) 分为两个过程，括号内部就是SubDynamicProxy，也就是nwProxyInstance时传入的this；生成类的父类就是Proxy，Proxy在构造函数设置了h属性为构造传入参数。

4. generateMethod() 向方法中注入了 (String)super.h.invoke(this, method, (Object[])args); 这里 method 是 m3 = Class.forName("ISubject").getMethod("request")，结合上一步，我们发现都用到了SubDynamicProxy的invoke方法。在这里做完额外的处理之后再执行委托类的对应方法。最类之间的依赖关系如下图所示。