链滴

一步一步学设计模式——代理模式

作者：jesministrator

原文链接：https://ld246.com/article/1511491024380

来源网站：链滴

代理模式是一个我们在编程中经常用到的设计模式，它的**目的是为其它对象提供一种代理以控制对这对象的访问。**

1. 生活实例

每年到快要过年的时候，抢票都是一个十分艰辛的任务。在互联网技术与网络购票还没有特别成熟时大家都需要在售票点排队去买票。很多学校上学的学生都没有时间在刚售票的时候就去排队等票，因出现了代排队的黄牛，他们每次多收学生25%的钱然后替学生排队买票。

再回想一下我们代理的模式的概念，在上面的例子中就存在一个典型的代理模式。代理模式目的是为它对象(学生)提供一种代理(黄牛)以控制这个对象(火车票)的访问。

下面我们就将上面这个生活实例转变为代码。

2. 生活实例代码

● 首先我们建立火车票类：

```
package com.wxueyuan.DesignPettern.StaticProxy;

/**
 * Author: Jesmin
 * Description: 火车票实体类
 *
 */
public class Ticket {
    private double price;

    public Ticket(double price) {
        this.price = price;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

}
```

● 由于由于黄牛和学生都需要买票，我们抽象出一个Operation接口，提供一个公共方法buyTicket来表示买票操作

```
package com.wxueyuan.DesignPettern.StaticProxy;

/**
 * Author: Jesmin
 * Description: 由于黄牛和学生都需要买票，我们抽象出一个Operation接口，提供一个公共方法buy
icket
```

```
 *                用来表示买票操作
 *
 */
public interface Operation {
    void buyTicket(Ticket t);
}
```

- 建立黄牛实体类Scalper

```
package com.wxueyuan.DesignPettern.StaticProxy;


/**
 * Author: Jesmin
 * Description:  “黄牛”实体类，在黄牛的购票操作中，他实际上分成了4步，先收钱，然后排队，后购票，最后将票交给学生
 *
 */
public class Scalper implements Operation{

    private Student  realConsumer;

    public Scalper(Student s) {
        realConsumer = s;
    }

    @Override
    public void buyTicket(Ticket t) {
        // TODO Auto-generated method stub
        System.out.println("黄牛收取购票者的钱");
        System.out.println("黄牛连夜排队");
        realConsumer.buyTicket(t);
        System.out.println("黄牛将票交给学生");
    }

}
```

- 建立学生实体类Student

```
package com.wxueyuan.DesignPettern.StaticProxy;

public class TicketConsumer implements Operation{

    @Override
    public void Student(Ticket t) {
        // TODO Auto-generated method stub
        System.out.println("学生买到一张票,票价为"+t.getPrice());
    }

}
```

- 最后进行测试

```
package com.wxueyuan.DesignPettern.StaticProxy;

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //学生需要购买的ticket实例
        Ticket t = new Ticket(200);
        //学生黄牛代理实例
        Scalper scalper = new Scalper(new Student());
        //黄牛为学生执行买票操作
        scalper.buyTicket(t);
    }
}
```

执行结果为：

黄牛收取购票者的钱

黄牛连夜排队

购票者买到一张票,票价为200.0

黄牛将票交给学生

分析一下上面的代码，黄牛实体类中获得了购票者的实例，因此他能够在他的购票操作中执行购票者购票操作，以及一些额外的操作。

下面我们一起来分析一下在代理模式中有哪些角色：

- 抽象角色： **通常是根据代理对象与实际对象相同的行为所抽象出的接口，由于代理对象与实际对都实现了这个接口，代理对象就可以在任何实际对象调用方法的地方替换它。**
- 代理角色： **代理对象内部含有实际对象的引用，因此可以执行实际对象的操作。代理对象可以在行实际对象操作时，附加其他的操作，相当于对实际对象的操作进行封装。**
- 委托角色： **定义了代理对象所代表的目标对象。代理角色所代表的委托对象，是我们最终要引用对象。**

3.代理模式的优缺点

- 优点：能够在不修改方法源码的情况下，对方法进行附加操作；可以通过代理模式将核心业务代码非核心业务代码解耦。
- 缺点：1.由于代理类和委托类实现了相同的接口，如果接口增加一个方法，除了所有实现类需要实这个方法外，所有代理类也需要实现此方法，增加了代码维护的复杂度。 2.代理对象只服务于一种类的对象，以我们上面的黄牛类来说，它只能作为学生类的代理(因为代理类中需要持有委托类的引用)假如我们的业务情况突然发生了变化，老师类也需要通过黄牛来买票，那我们就不得不再建立另一个理类TeacherScalper，并持有Teacher类的引用，来帮助老师进行买票。因此在有大量的实际类需要代理的情况下，这种代理模式并不是很适合,会有大量的冗余代码。
- 老师黄牛代理

```
package com.wxueyuan.DesignPettern.StaticProxy;


/**
 * Author: Jesmin
```

```
 * Description: 老师黄牛代理类，在黄牛的购票操作中，他实际上分成了4步，先收钱，然后排队，后购票，最后将票交给老师
 *
 */
public class TeacherScalper implements Operation{

    private Teacher realConsumer ;

    public TeacherScalper(Teacher t) {
        realConsumer = t;
    }

    @Override
    public void buyTicket(Ticket t) {
        // TODO Auto-generated method stub
        System.out.println("黄牛收取购票者的钱");
        System.out.println("黄牛连夜排队");
        realConsumer.buyTicket(t);
        System.out.println("黄牛将票交给购票者");
    }

}
```

- 老师实体类

```
package com.wxueyuan.DesignPettern.StaticProxy;

public class Teacher implements Operation{
    @Override
    public void buyTicket(Ticket t) {
        // TODO Auto-generated method stub
        System.out.println("老师买到一张票,票价为"+t.getPrice());
    }
}
```

- 测试类

```
package com.wxueyuan.DesignPettern.StaticProxy;

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //学生需要购买的ticket实例
        Ticket t = new Ticket(200);
        //学生黄牛代理实例
        Scalper scalper = new Scalper(new Student());
        //黄牛为学生执行买票操作
        scalper.buyTicket(t);

        //老师需要购买的ticket实例
        Ticket adultTicket = new Ticket(300);
        //老师黄牛代理实例
```

```
        TeacherScalper ts = new TeacherScalper(new Teacher());
        //黄牛为老师执行买票操作
        ts.buyTicket(adultTicket);
    }

}
```

由于这种代理模式，提前已经抽象好了代理角色与委托角色共同的行为接口，并且代理角色在编译时经确定了与委托类之间的委托关系，因此这种代理模式也被称为**静态代理**。

考虑到由多个委托类时，需要有多个代理类的情况，我们自然会想有没有更加好的办法，能够通过一代理类完成全部的代理功能呢？答案是有的，它就是**动态代理**。

4.动态代理

在静态代理中，一个代理只能代理一种类型，而且是在编译期间就已经确定被代理的对象。而动态代是在运行时，通过反射机制实现动态代理，并且能够代理各种类型的对象。

下面我们以上面的老师，学生买火车票为例，使用动态代理去完成这个例子，下面例子中使用到的Tiet实体类，Teacher实体类，Student实体类与Operation抽象接口均与静态代理中的相同，故不再赘。

- 首先我们需要实现InvocationHandler接口：

```
package com.wxueyuan.DesignPettern.DynamicProxy;

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

public class TicketOperationInvocationHandler implements InvocationHandler {

    //将需要代理的委托对象传入Handler中
    private Object target;

    public TicketOperationInvocationHandler(Object target) {
        this.target = target;
    }

    //获得帮助购票者买票的代理
    public Object getProxy() {
        return Proxy.newProxyInstance(Thread.currentThread()
                .getContextClassLoader(), target.getClass().getInterfaces(),
                this);
    }

    //实际上黄牛执行的购票操作
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // TODO Auto-generated method stub
        System.out.println("黄牛收取购票者的钱");
        System.out.println("黄牛连夜排队");
        Object ret = method.invoke(target, args);
        System.out.println("黄牛将票交给购票者");
```

```
        return ret;
    }

}
```

● 接下来我们来测试一下我们的代理：

```
package com.wxueyuan.DesignPettern.DynamicProxy;

import com.wxueyuan.DesignPettern.StaticProxy.Operation;
import com.wxueyuan.DesignPettern.StaticProxy.Student;
import com.wxueyuan.DesignPettern.StaticProxy.Teacher;
import com.wxueyuan.DesignPettern.StaticProxy.Ticket;

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        //学生需要购买的ticket实例
        Ticket studentTicket = new Ticket(200);
        //老师需要购买的ticket实例
        Ticket adultTicket = new Ticket(300);

        //创建为学生买票的黄牛代理
        Operation studentProxy = (Operation) new TicketOperationInvocationHandler(new Stud
nt()).getProxy();
        studentProxy.buyTicket(studentTicket);

        System.out.println("---------------------------");

        //创建为老师买票的黄牛代理
        Operation teacherProxy = (Operation) new TicketOperationInvocationHandler(new Teac
er()).getProxy();
        teacherProxy.buyTicket(adultTicket);
    }

}
```

执行结果为：

黄牛收取购票者的钱
黄牛连夜排队
学生买到一张票,票价为200.0
黄牛将票交给购票者

黄牛收取购票者的钱
黄牛连夜排队
老师买到一张票,票价为300.0
黄牛将票交给购票者

现在让我们分析一下**动态代理**的优点：1.与**静态代理**相比我们不需要为每一个需要被代理的委托类去一个对应的代理类(上例中的TeacherScalper和Scalper)，我们可以直接用抽象接口生成代理实例来替不同的委托类去完成任务。2.**动态代理**另一个优点就是将公共接口中声明的所有的方法都被转移到个集中的方法中去处理(invocke()方法)，也就是说我们可以在invoke方法中为所有接口中的方法附加同的额外操作，比如记录在方法执行前记录当前时间，方法执行后记录当前时间，用差值来获得每个法的实际执行时间等等，**这其实就是AOP(面向切面编程)的基本原理。**

到这里，博主就为大家介绍了代理设计模式，以及静态代理与动态代理之间的优缺点，有的同学可能动态代理的使用以及原理还有一些困惑，欢迎大家关注博主的另一篇文章《Java源码剖析——动态代的实现原理》。