

代理模式

1 简介

1.1 为什么要学习代理模式？

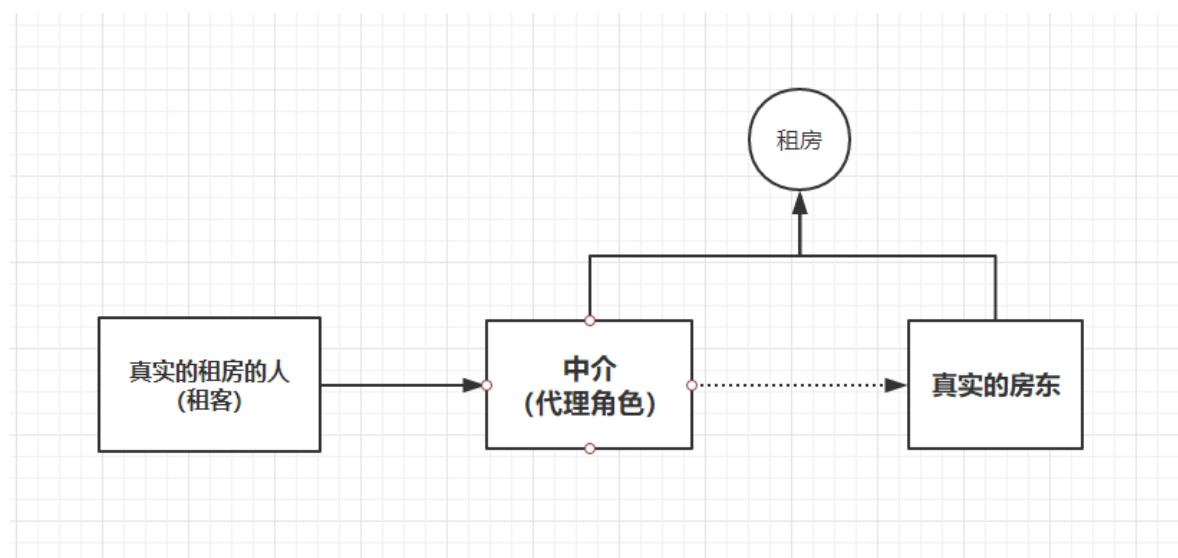
代理模式是SpringAOP的底层。

1.2 分类

- 静态代理
- 动态代理

1.3 图解代理模式

举个例子：租房



- 租客和房东都有一个共同的需求：租房
- 租客不再直接找房东租房，而是通过中介
- 房东不再自己去把房子租出去，而是交由中介处理
- 中介就是这里的**代理角色**
- 角色关系是代理模式的核心！

2 静态代理

2.1 角色分析

- 抽象角色：一般会使用接口或者抽象类来解决
- 真实角色：被代理的角色 —— **房东**
- 代理角色：代理真实角色【代理真实角色后，一般会做一些附属的操作】 —— **中介**
- 客户：访问代理对象 —— **租客**

2.2 通过案例理解

- 接口

```

1  /**
2   * 租房需求的接口
3   */
4  public interface Rent {
5      public void rent();
6  }

```

- 真实角色

```

1  /**
2   * 房东
3   */
4  public class Host implements Rent{
5      public void rent() {
6          System.out.println("房东出租房子");
7      }
8  }

```

- 代理角色

```

1  package com.kuang.demo1;
2
3  /**
4   * 代理
5   * 实现需求（接口） -- 租房
6   */
7  public class Proxy implements Rent {
8      //要帮房东出租房子
9      private Host host;
10
11      public Proxy() {
12      }
13
14      //传给中介哪个房东，就给哪个房东干活
15      public Proxy(Host host) {
16          this.host = host;
17      }
18
19      //出租房子
20      public void rent() {
21          //传给中介哪个房东，就给哪个房东租房子
22          host.rent();
23
24          seeHouse();
25
26          free();
27
28          contact();
29      }
30
31      //看房
32      public void seeHouse() {
33          System.out.println("中介带租客看房");
34      }
35
36      //收中介费

```

```

37     public void free() {
38         System.out.println("收中介费");
39     }
40
41     //签合同
42     public void contact() {
43         System.out.println("签合同");
44     }
45 }

```

- 客户端访问代理角色

```

1  package com.kuang.demo1;
2
3  /**
4   * 租客（租房子的人）
5   */
6  public class client {
7      public static void main(String[] args) {
8          //房东要租房子
9          Host host = new Host();
10
11          //代理，中介帮房东租房子
12          //代理角色一般会有附属操作
13          Proxy proxy = new Proxy(host);
14
15          //不用面对房东，直接面对中介租房
16          proxy.rent();
17      }
18  }

```

```

房东出租房子
中介带租客看房
收中介费
签合同

Process finished with exit code 0

```

2.3 好处

- 可以使真实角色的操作更加纯粹（只租房子），不用去关注一些公共的业务
- 公共的业务交给代理角色，实现了业务的分工
- 公共业务发生拓展的时候，方便集中管理

2.4 缺点

- 一个真实角色就得产生一个代理角色，代码量会翻倍，开发效率变低（后面会使用办法解决）

2.5 聊聊AOP

AOP: 面向切面编程

先来一个例子：给每一个业务的实现添加日志（Log）功能

- 业务

```
1 package com.kuang.demo2;
2
3 /**
4  * 业务
5  */
6 public interface UserService {
7     public void add();
8
9     public void delete();
10
11     public void update();
12
13     public void query();
14 }
```

- 业务的实现（未添加业务）

```
1 package com.kuang.demo2;
2
3 /**
4  * 真实对象
5  */
6 public class UserServiceImpl implements UserService {
7     public void add() {
8         System.out.println("增加了一个用户");
9     }
10
11     public void delete() {
12         System.out.println("删除一个用户");
13     }
14
15     public void update() {
16         System.out.println("修改了一个用户的信息");
17     }
18
19     public void query() {
20         System.out.println("查询一个用户");
21     }
22 }
```

- 当我们想给每一个功能（增删改查）前都加上“使用了xxx方法”应该怎么做？
一开始我们都是直接在每个功能前加上 `System.out.println("[Debug]使用了" + msg + "方法");`；
但是这样子有个问题：我们改动了源代码，这在公司开发是大忌，不可通的。
于是，我们采用代理，使用代理给每个增删改查加上打印日志这个功能。

- 代理对象

```
1 package com.kuang.demo2;
2
3 public class UserServiceProxy implements UserService {
```

```

4     private UserServiceImpl userService;
5
6     public void setUserService(UserServiceImpl userService) {
7         this.userService = userService;
8     }
9
10    public UserServiceProxy() {
11    }
12
13    public void add() {
14        log("add");
15        userService.add();
16    }
17
18    public void delete() {
19        log("delete");
20        userService.delete();
21    }
22
23    public void update() {
24        log("update");
25        userService.update();
26    }
27
28    public void query() {
29        log("query");
30        userService.query();
31    }
32
33    //日志功能
34    public void log(String msg) {
35        System.out.println("[Debug]使用了" + msg + "方法");
36    }
37 }

```

代理对象实现业务实现类的接口，并在代理对象中增加了新的功能——日志。我们在代理类中就可以实现我们的需求。

- 测试运行

```

1 package com.kuang.demo2;
2
3 public class Client {
4     public static void main(String[] args) {
5         UserServiceImpl userService = new UserServiceImpl();
6
7         UserServiceProxy userServiceProxy = new UserServiceProxy();
8         userServiceProxy.setUserService(userService);
9
10        userServiceProxy.add();
11    }
12 }

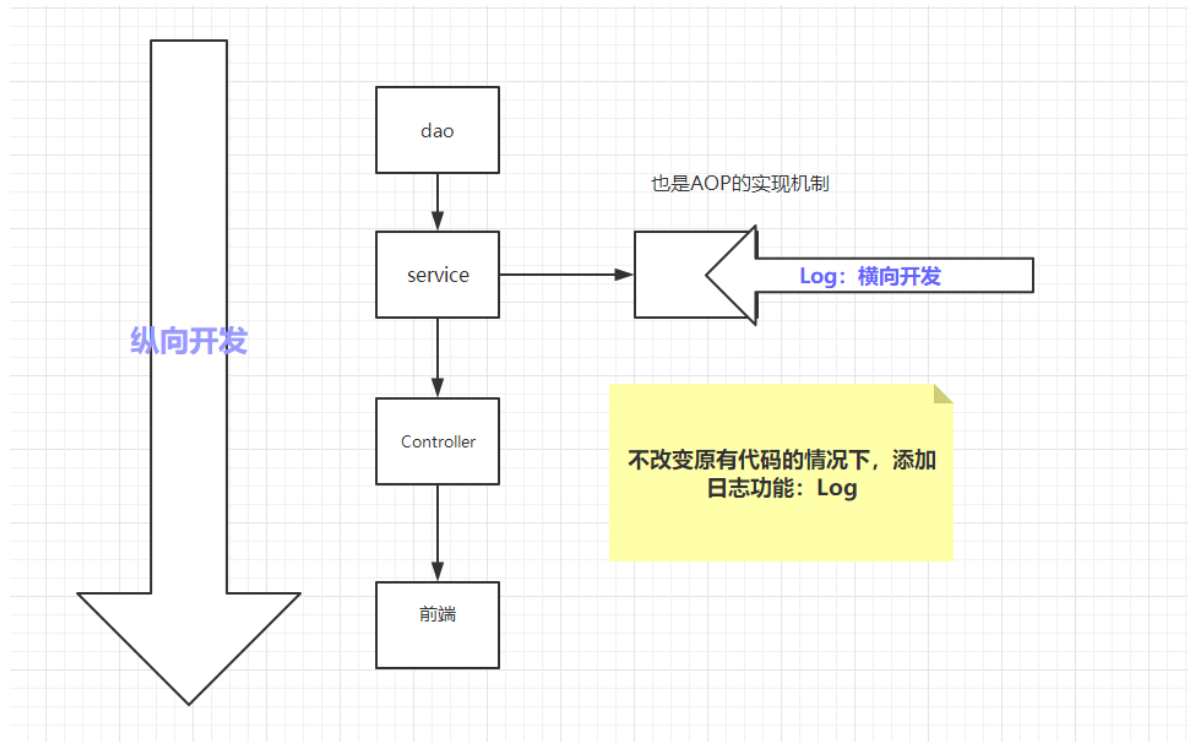
```

```
使用了add方法  
增加了一个用户
```

```
Process finished with exit code 0
```

这样子和AOP有什么关系呢？切面编程又是什么意思？

这里给出一张上面添加功能的图：



如图所示：

- 纵向开发是我们正常的开发步骤
- 当我们遇到需求的时候（比如上面的添加日志），我们如果修改源码，直接在业务接口实现类中修改，就是在纵向中进行修改，一旦修改后出现了Bug，这个修改就十分地致命（原本代码没问题，被你这么一改反而出问题）。
- 于是，我们进行“横向开发”，用代理的方式对原先类进行“增强”，添加了我们想要的方法。
- 横向开发，从旁边“切”进去的感觉，切面编程暂且可以这么理解。（后面会做详细的介绍）

3 动态代理

3.1 区别静态代理

- 动态代理和静态代理的角色是一样的
- 动态代理的代理类是动态生成的，不是像静态代理那样自己编写
- 动态代理分为两大类：
 - 基于接口的动态代理 —— jdk动态代理
 - 基于类的动态代理 —— cglib
 - java字节码 —— javassist
- 需要了解两个类
 - Proxy —— 代理

| Modifier and Type | Method and Description |
|--------------------------|--|
| static InvocationHandler | <code>getInvocationHandler(Object proxy)</code> 返回指定代理实例的调用处理程序。 |
| static 类<?> | <code>getProxyClass(ClassLoader loader, 类<?>... interfaces)</code> 给出类加载器和接口数组的代理类的 <code>java.lang.Class</code> 对象。 |
| static boolean | <code>isProxyClass(类<?> cl)</code> 如果且仅当使用 <code>getProxyClass</code> 方法或 <code>newProxyInstance</code> 方法将指定的类动态生成成为代理类时，则返回 <code>true</code> 。 |
| static Object | <code>newProxyInstance(ClassLoader loader, 类<?>[] interfaces, InvocationHandler h)</code> 返回指定接口的代理类的实例，该接口将方法调用分派给指定的调用处理程序。 |

◦ InvocationHandler —— 调用处理程序

| Modifier and Type | Method and Description |
|-------------------|---|
| Object | <code>invoke(Object proxy, 方法 method, Object[] args)</code> 处理代理实例上的方法调用并返回结果。 |

- 一个动态代理，代理的是一个接口，一般对应的就是一类业务
- 一个动态代理类可以代理很多个类，只要是实现了同一个接口

3.2 案例解析

一样是租客和房东、中介。

- 需求（租房）

```

1  /**
2   * 租房需求的接口
3   */
4  public interface Rent {
5      public void rent();
6  }
```

- 真实角色（房东）

```

1  /**
2   * 房东
3   */
4  public class Host implements Rent {
5      public void rent() {
6          System.out.println("房东出租房子");
7      }
8  }
```

- 动态代理生成代理类

```

1  package com.kuang.demo3;
2
3  import java.lang.reflect.InvocationHandler;
4  import java.lang.reflect.Method;
5  import java.lang.reflect.Proxy;
6
7  /**
8   * 使用这个类，自动生成代理类
9   */
10 public class ProxyInvocationHandler implements InvocationHandler {
11
12     /**
13      * 被代理的接口
14      */
15     private Rent rent;
```

```

16
17     public void setRent(Rent rent) {
18         this.rent = rent;
19     }
20
21     /**
22      * 官网例子:
23      Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),
24      new Class<?>[] { Foo.class },
25      handler);
26      */
27
28     /**
29      * 生成得到代理类
30      * newProxyInstance :生成代理对象
31      * this.getClass().getClassLoader() : 加载类到哪个位置
32      * rent.getClass().getInterfaces() : 要代理的是哪个接口
33      * this : 自己本身的类（已经implements InvocationHandler,
34      *          这样子才可以通过InvocationHandler来处理）
35      *
36      * @return
37      */
38     public Object getProxy() {
39         return Proxy.newProxyInstance(this.getClass().getClassLoader(),
40         rent.getClass().getInterfaces(), this);
41     }
42
43     /**
44      * 处理代理实例，并返回结果
45      * 使用invoke去执行
46      *
47      * @param proxy
48      * @param method
49      * @param args
50      * @return
51      * @throws Throwable
52      */
53     public Object invoke(Object proxy, Method method, Object[] args)
54     throws Throwable {
55         seehouse(); //方法执行前输出
56
57         //动态代理的本质，就是使用反射机制实现
58         //rent : 代理的接口
59         Object result = method.invoke(rent, args); // 接口中方法执行
60
61         concat();//方法执行后输出
62
63         return result;
64     }
65
66     public void seehouse() {
67         System.out.println("中介带看房子");
68     }
69
70     public void concat() {
71         System.out.println("签合同");
72     }

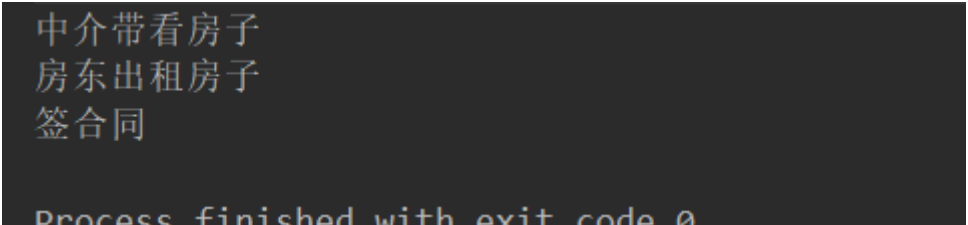
```



```
73 |
74 | }
```

- 测试运行

```
1 public static void main(String[] args) {
2     //真实角色（房东）
3     Host host = new Host();
4
5     //代理角色：目前还没有
6     ProxyInvocationHandler pih = new ProxyInvocationHandler();
7
8     //通过调用程序来处理我们要调用的接口对象（实现需求接口）
9     pih.setRent(host); //设置一个要代理的角色
10
11     Rent proxy = (Rent) pih.getProxy(); //这里的proxy就是动态代理生成的，我们
    并没有写
12
13     proxy.rent();
14 }
```



```
中介带看房子
房东出租房子
签合同
Process finished with exit code 0
```

3.3 万能动态代理类

```
1 package com.kuang.demo4;
2
3 import java.lang.reflect.InvocationHandler;
4 import java.lang.reflect.Method;
5 import java.lang.reflect.Proxy;
6
7 /**
8  * 万能的动态代理接口类
9  * 使用这个类，自动生成代理类
10  */
11 public class ProxyInvocationHandler implements InvocationHandler {
12
13     /**
14      * 被代理的接口
15      */
16     private Object target;
17
18     public void setTarget(Object target) {
19         this.target = target;
20     }
21
22
23     /**
24      * 生成得到代理类
25      * newProxyInstance :生成代理对象
26      * this.getClass().getClassLoader() : 加载类到哪个位置
```

```

27     * rent.getClass().getInterfaces() : 要代理的是哪个接口
28     * this : 自己本身的类 (已经implements InvocationHandler,
29     * 这样子才可以通过InvocationHandler来处理)
30     *
31     * @return
32     */
33     public Object getProxy() {
34         return Proxy.newProxyInstance(this.getClass().getClassLoader(),
35             target.getClass().getInterfaces(), this);
36     }
37
38     /**
39     * 处理代理实例, 并返回结果
40     * 使用invoke去执行
41     *
42     * @param proxy
43     * @param method
44     * @param args
45     * @return
46     * @throws Throwable
47     */
48     public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
49         log(method.getName());
50
51         //动态代理的本质, 就是使用反射机制实现
52         //rent : 代理的接口
53         Object result = method.invoke(target, args); // 接口中方法执行
54
55         return result;
56     }
57
58     //需求: 加一个日志功能
59     public void log(String msg) {
60         system.out.println("执行了" + msg + "方法");
61     }
62
63
64 }

```