# 概述

在代理模式（Proxy Pattern）中，一个类代表另一个类的功能

## 举例说明

去买一支钢笔，都会去文具店这类经销商进行购买，而不是直接去生产钢笔的工厂进行购买。 这里的商店就是钢笔厂商的代理

### 与其他几个设计模式的区别

1、和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。

2、和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制

## 代码

### 主体（Subject）接口： 图像

```
public interface Image {
   void display();
}
```

### 具体的主体类（realSubject）： 真实图像

```
public class RealImage implements Image {

   private String fileName;

   public RealImage(String fileName) {
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

    @Override
    public void display() {
       System.out.println("Displaying " + fileName);
    }

    private void loadFromDisk(String fileName) {
       System.out.println("Loading " + fileName);
    }
}
```

### 主体代理人(proxy)：图像代理类

```
public class ProxyImage implements Image {

   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName) {
      this.fileName = fileName;
   }

    @Override
    public void display() {
       if (realImage == null) {
          realImage = new RealImage(fileName);
       }
       realImage.display();
```

```
    }
}
```

### 客户端（client）

```
public class Main {
    public static void main(String[] args) {
        Image image = new ProxyImage("test_10mb.jpg");

        //图像将从磁盘加载
        image.display();
        System.out.println("");
        //图像将无法从磁盘加载
        // image.loadFromDisk();
     }
}
```

### 代码概述

- 图像类只提供一个展示图像的方法
- 具体的图像类，实际包含了，从磁盘加载图片，然后才能进行展示两个方法
- 对于用户来说，只需要关系图像的展示即可，不需要关心图像从磁盘怎么进行加载的
- 通过代理类来实现 图像的展示。隐藏加载细节

## 总结

### 优点

- 延迟加载，如果主题（subject）初始化时间很长，如果通过代理（proxy），实际上只需要初始化轻量级的代理（proxy）即可。当主行初始化
- 控制权限，对外控制使用权限。一个接口实际上有 10个方法，但是对外只能暴露5个方法。

### 缺点

- 增加复杂度

## 扩展 - 动态代理

### jdk

#### 代码流程

1、 新建一个接口
2、 为接口创建一个实现类
3、 创建代理类实现java.lang.reflect.InvocationHandler接口
4、 测试

#### 主体接口

```
public interface Subject {
    public String method1();
    public void method2();
    public int method3(int x);
}
```

**具体实现类**

```
public class RealSubject implements Subject {

    @Override
    public String method1() {
        System.out.println("method1 running...");
        return "aaa";
    }

    @Override
     public void method2() {
         System.out.println("method2 running...");
     }

     @Override
     public int method3(int x) {
         return x;
     }

}
```

**代理类**

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

public class MyJDKDynamicProxy implements InvocationHandler {

    private Object subject;

    public MyJDKDynamicProxy(Object subject) {
         this.subject = subject;
     }

     /**
      * 获取被代理接口实例对象
      *
      * @param <T>
      * @return
      */
     public <T> T getProxy() {
         return (T) Proxy.newProxyInstance(subject.getClass().getClassLoader(), subject.getClass().g
     }

     @Override
     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
         System.out.println("Do something before");
         Object invoke = method.invoke(subject, args);
         System.out.println("Do something after");
         return invoke;
     }
}
```

### 客户端调用

```
public class Main {

    public static void main(String[] args) {

        // jdk动态代理测试
        Subject subject = new MyJDKDynamicProxy(new RealSubject()).getProxy();
        String method1 = subject.method1();
        System.out.println(method1);
        System.out.println();


         subject.method2();
         System.out.println();


         int method3 = subject.method3(100);
         System.out.println(method3);
    }
}
```

### 动态代理大概流程

1、为接口创建代理类的字节码文件

2、使用ClassLoader将字节码文件加载到JVM

3、创建代理类实例对象，执行对象的目标方法

## CGLIB

### 代理类

```
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class MyCglibDynamicProxy implements MethodInterceptor {

    private static MyCglibDynamicProxy instance = new MyCglibDynamicProxy();

    public static MyCglibDynamicProxy getInstance() {
        return instance;
    }

    public <T> T getProxy(Class<T> cls) {
        return (T) Enhancer.create(cls, this);
    }

    @Override
    public Object intercept(Object target, Method method, Object[] args, MethodProxy methodProxy) t
        System.out.println("Do something before");
        Object result = methodProxy.invokeSuper(target, args);
        System.out.println("Do something after");
```

```
        return result;
    }
}
```

客户端

```
public class Main {

    public static void main(String[] args) {

        Subject subject = MyCglibDynamicProxy.getInstance().getProxy(RealSubject.class);

        String method1 = subject.method1();
        System.out.println(method1);
        System.out.println();


         subject.method2();
         System.out.println();


         int method3 = subject.method3(100);
         System.out.println(method3);

    }
}
```

## 应用实例

spring aop 通过通过代理的形式，在方法调用前后进行处理

hibernate 的 load 实现延迟加载， 根据传入的 Class， 生产代理对象，拦截 get 和 set方法，真正调用实际对象

## JDK动态代理与CGLIB动态代理的区别

JDK的动态代理机制只能代理实现了接口的类，而不能实现接口的类就不能实现JDK的动态代理

cglib是针对类来实现代理的，他的原理是对指定的目标类生成一个子类，并覆盖其中方法实现增强，**但因为采用的是继承，所以不能对final修**