优秀借鉴

设计模式（四）——搞懂什么是代理模式

代理设计模式（refactoringguru.cn）

黑马程序员Java设计模式详解-设计模式-结构型模式-代理对象概述

《深入设计模式》-亚历山大·什韦茨(Alexander Shvets)

1、简介

代理模式(Proxy)是一种常见的设计模式，它允许通过**代理对象**控制对某个对象的访问。在代理模式中，代理类扮演着客户端和真正的目标对象之间的中介角色，代理类可以为目标对象提供额外的功能，例如远程访问、延迟加载、权限控制等。

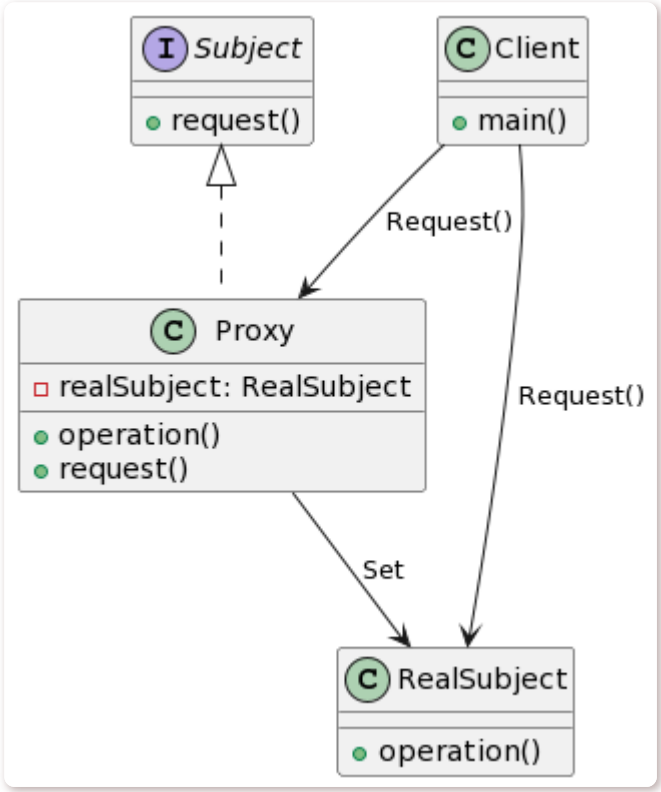
使用代理模式可以实现对象的封装，同时也能够降低系统耦合度，增强了系统的灵活性和可扩展性。如果在开发过程中需要对某个对象进行控制，并且希望保持系统的高内聚、低耦合特性，那么代理模式是一个不错的选择。

2、结构

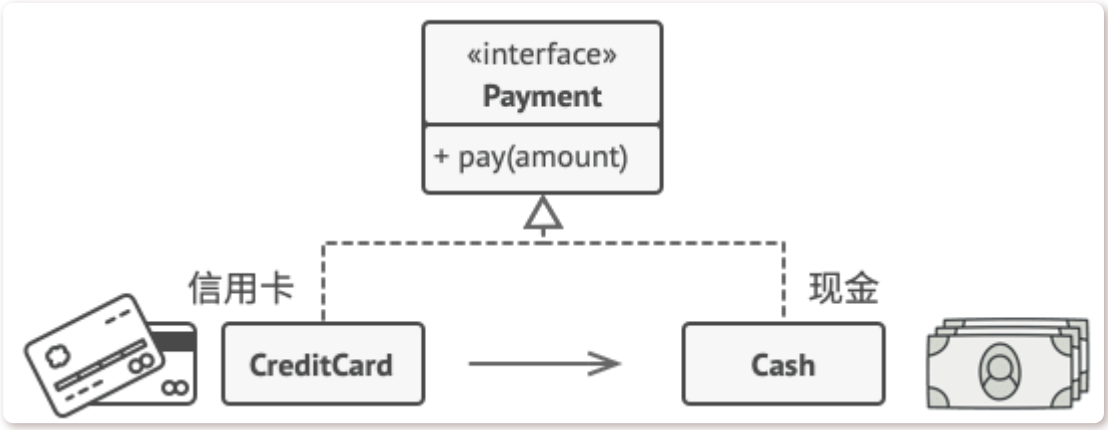
代理模式通常包括三个角色：**抽象主题(Subject)、真实主题(Real Subject)和代理主题(Proxy Subject)**：

- **抽象主题**定义了一个共同的接口，也可被称为真实主题的规范，被代理类和真实的目标类都要实现该接口；
- **真实主题**即真正执行业务逻辑的对象；

- **代理主题**是代理类，可以代替真实主题来完成一些操作，同时也可以在完成操作前或者之后添加一些额外的逻辑，以实现对真实主题的控制。



上面结构在真实世界中可以通过一个银行的场景来类比：



代理模式三个角色在上面的实现分别如下：

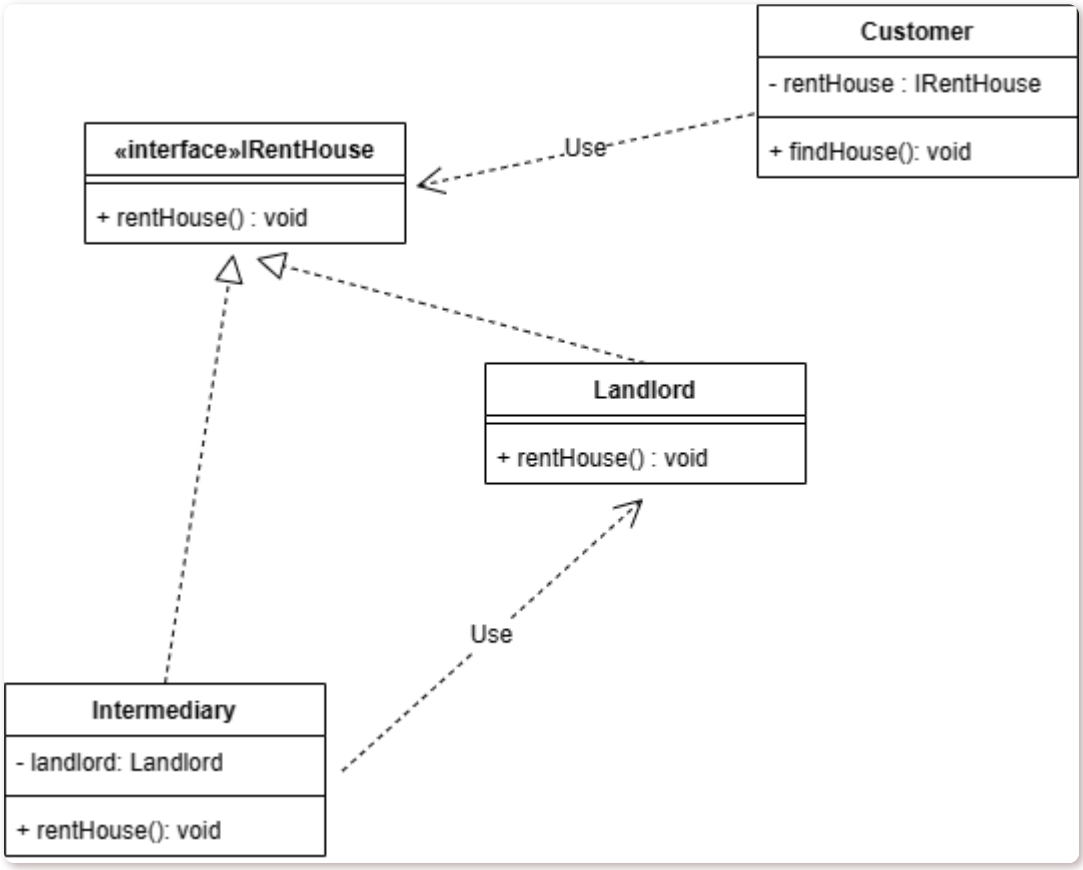
- **抽象主题**：体现在上面的**接口**中，定下一个支付的规范，实现该接口的类需要重写支付方法；
- **真实主题**：体现在上面的**现金**中，最直接的支付方式便是通过现金支付；
- **代理主题**：体现在上面的**信用卡**中，用户可以不用带着一大捆的现金出门，直接通过信用卡进行刷卡支付，有信用卡的厂商代付。

信用卡是银行账户的代理，银行账户则是一大捆现金的代理。它们都实现了同样的接口，均可用于进行支付。消费者会非常满意，因为不必随身携带大量现金；商店老板同样会十分高兴，因为交易收入能以电子化的方式进入商店的银行账户中，无需担心存款时出现现金丢失或被抢劫的情况。

3、实现方式

3.1、案例引入

在下面的案例中，统一使用租房的案例进行介绍，整个场景的UML图如下：



代理模式三个角色在上面的实现分别如下：

- **抽象主题**：体现在上面的**接口IRentHouse**中，定下一个租房的规范，实现该接口的类需要重写租房方法；
- **真实主题**：体现在上面的**房东Landlord**中，最直接的租房方式便是直接找到房东租房；
- **代理主题**：体现在上面的**中介Intermediary**中，**租客Customer**不用为了房源大老远跑去找房东，而是通过手握房源的中介进行租房。

中介是房东的代理，中介和房东都实现了抽象主题接口，均可以找到他们去租房子。租客在租房过程中，可以直接找到距离自己近的中介租房而不用奔波到遥远的房东去租房

3.2、静态代理

在静态代理中，代理类和真实类都要实现相同的接口或者继承相同的抽象类。代理类负责将客户端请求转发给真实对象，并且可以在调用真实对象前后添加一些额外的逻辑。

> 🌈
>
> 值得注意的是，静态代理需要手动编写代理类，代码量较大，但是运行效率较高。

使用静态代理对案例进行实现如下：

```
/**
 * @author xbaozi
```

```
 * @version 1.0
 * @classname StaticProxy
 * @date 2023-04-09  12:42
 * @description 静态代理
 */
public class StaticProxy {
    public static void main(String[] args) {
        System.out.println("\n********** 直接找到房东租房 **********");
        Customer customerToLandlord = new Customer(new Landlord());
        customerToLandlord.findHouse();
        System.out.println("\n********** 找附近手握房源的中介租房 **********");
        Customer customerToIntermediary = new Customer(new Intermediary());
        customerToIntermediary.findHouse();
    }
}

/**
 * 抽象主题，对租房定下规范
 */
interface IRentHouse {
    void rantHouse();
}

/**
 * 真实主题，实现抽象主题
 */
class Landlord implements IRentHouse {

    @Override
    public void rantHouse() {
        System.out.println("[真实主题] 找到房东租房……");
    }
}

/**
 * 代理主题，实现抽象主题，同时对真实主题进行增强
 */
class Intermediary implements IRentHouse {

    private Landlord landlord = new Landlord();

    @Override
    public void rantHouse() {
        System.out.println("[代理主题] 找到手握房源的中介交中介费……");
        landlord.rantHouse();
        System.out.println("[代理主题] 和租户对接好后续工作");
```

```
    }
}

/**
 * 租户类
 */
class Customer {
    private IRentHouse rentHouse;

    public Customer(IRentHouse rentHouse) {
        this.rentHouse = rentHouse;
    }

    public void findHouse() {
        rentHouse.rantHouse();
    }
}
```

运行结果如下：

```
"D:\Program Files\Java\jdk-11.0.16.1\bin\java.exe" "-jav

********** 直接找到房东租房 **********
[真实主题] 找到房东租房……

********** 找附近手握房源的中介租房 **********
[代理主题] 找到手握房源的中介交中介费……
[真实主题] 找到房东租房……
[代理主题] 和租户对接好后续工作

Process finished with exit code 0
```

3.3、JDK动态代理

JDK动态代理是一种在运行时生成代理对象的技术。它允许我们在不修改源代码的情况下，通过代理对象来调用目标对象的方法。

其通常用于实现 **AOP(面向切面编程)** 和 **RPC(远程过程调用协议)** 等功能。在AOP中，代理对象可以在执行目标对象的方法前后进行一些额外的操作，如日志记录、事务管理等。而在远程方法调用中，代理对象可以隐藏底层的网络通信细节，使得远程调用看起来就像本地调用一样。

JDK动态代理的原理是基于**反射机制和接口**实现的。通过获取目标对象的接口信息和实现类，然后创建一个新的代理类并实现相同的接口，并在代理类中处理特定的逻辑操作。

```
/**
 * @author xbaozi
 * @version 1.0
 * @classname JavaDynamicProxy
```

```
 * @date 2023-04-09  13:48
 * @description Java动态代理
 */
public class JavaDynamicProxy {
    public static void main(String[] args) {
        System.out.println("\n********** 直接找到房东租房 **********");
        Customer customerToLandlord = new Customer(new Landlord());
        customerToLandlord.findHouse();
        System.out.println("\n********** 找附近手握房源的中介租房 **********");
        IRentHouse proxyIntermediary = ProxyFactory.getProxy();
        Customer customerToIntermediary = new Customer(proxyIntermediary);
        customerToIntermediary.findHouse();
    }
}

/**
 * 抽象主题类
 */
interface IRentHouse {
    void rantHouse();
}

/**
 * 真实主题，实现抽象主题
 */
class Landlord implements IRentHouse {

    @Override
    public void rantHouse() {
        System.out.println("[Java动态代理-真实主题] 找到房东租房……");
    }
}

/**
 * 代理工厂，用于生成代理主题类
 */
class ProxyFactory {
    private static Landlord landlord = new Landlord();

    public static IRentHouse getProxy() {
        /*
            newProxyInstance()方法参数说明:
                ClassLoader loader :  类加载器，用于加载代理类，使用真实对象的类加载器即可
                Class<?>[] interfaces :  真实对象所实现的接口，代理模式真实对象和代理对象实现相同的接口
```

```
                    InvocationHandler h :  代理对象的调用处理程序
             */
            return (IRentHouse) Proxy.newProxyInstance(
                    landlord.getClass().getClassLoader(),
                    landlord.getClass().getInterfaces(),
                    new InvocationHandler() {
                        /*
                            InvocationHandler中invoke方法参数说明:
                                proxy :  代理对象，newProxyInstance方法的返回对象
                                method :  对应于在代理对象上调用的接口方法的 Method 实例
                                args :  代理对象调用接口方法时传递的实际参数
                         */
                        @Override
                        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                            System.out.println("[Java动态代理-代理主题]去中介公司交中介费获取中介服务");
                            Object returnArg = method.invoke(landlord, args);
                            System.out.println("[Java动态代理-代理主题]和租户对接好后续工作");
                            return returnArg;
                        }
                    }
            );
        }
    }

    /**
     * 租户类
     */
    class Customer {
        private IRentHouse rentHouse;

        public Customer(IRentHouse rentHouse) {
            this.rentHouse = rentHouse;
        }

        public void findHouse() {
            rentHouse.rantHouse();
        }
    }
```

运行结果如下：

```
"D:\Program Files\Java\jdk-11.0.16.1\bin\java.exe" "-javaag

********** 直接找到房东租房 **********
[Java动态代理-真实主题] 找到房东租房.....

********** 找附近手握房源的中介租房 **********
[Java动态代理-代理主题] 去中介公司交中介费获取中介服务
[Java动态代理-真实主题] 找到房东租房.....
[Java动态代理-代理主题] 和租户对接好后续工作

Process finished with exit code 0
```

> 这里不知道有没有小伙伴有疑问：ProxyFactory代理工厂是我们代理模式中的代理主题即代理类吗?

答案为并不是。ProxyFactory只是一个动态生成代理类的一个工厂，而代理类是程序在运行过程中动态的在内存中生成的类。这可以类比成ProxyFactory是一个中介公司，其并不是要真正为租客找房子的那个人，真正为租客代理租房的是中介公司派出(生成)的中介，即真正的代理类。

在动态代理中，底层通过**反射**获取到目标调用的方法，然后通过自定义的 `InvocationHandler` 中的 `invoke` 方法实现对目标方法的增强。这里可以通过阿里巴巴开源的 Java 诊断工具（Arthas【阿尔萨斯】）查看生成代理类的结构(精简版)：

```
//程序运行过程中动态生成的代理类
public final class $Proxy0 extends Proxy implements IRentHouse {
    private static Method m3;

    public $Proxy0(InvocationHandler invocationHandler) {
        super(invocationHandler);
    }

    static {
        m3 =
Class.forName("com.xbaoziplus.proxy.dynamic.jdk.IRentHouse").getMet
hod("rantHouse", new Class[0]);
    }

    public final void rantHouse() {
        this.h.invoke(this, m3, null);
    }
}
```

Arthas 生成的完整的代码如下，感兴趣的小伙伴可以自行查看：

```
package com.sun.proxy;

```

```java
import com.xbaoziplus.proxy.dynamic.jdk.IRentHouse;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;

public final class $Proxy0 extends Proxy implements IRentHouse {
    private static Method m1;
    private static Method m2;
    private static Method m3;
    private static Method m0;

    public $Proxy0(InvocationHandler invocationHandler) {
        super(invocationHandler);
    }

    static {
        try {
            m1 =
Class.forName("java.lang.Object").getMethod("equals",
Class.forName("java.lang.Object"));
            m2 =
Class.forName("java.lang.Object").getMethod("toString", new
Class[0]);
            m3 =
Class.forName("com.xbaoziplus.proxy.dynamic.jdk.IRentHouse").getMet
hod("rantHouse", new Class[0]);
            m0 =
Class.forName("java.lang.Object").getMethod("hashCode", new
Class[0]);
            return;
        }
        catch (NoSuchMethodException noSuchMethodException) {
            throw new
NoSuchMethodError(noSuchMethodException.getMessage());
        }
        catch (ClassNotFoundException classNotFoundException) {
            throw new
NoClassDefFoundError(classNotFoundException.getMessage());
        }
    }

    public final boolean equals(Object object) {
        try {
            return (Boolean)this.h.invoke(this, m1, new Object[]
{object});
        }
        catch (Error | RuntimeException throwable) {
            throw throwable;
```

```
        }
        catch (Throwable throwable) {
            throw new UndeclaredThrowableException(throwable);
        }
    }

    public final String toString() {
        try {
            return (String)this.h.invoke(this, m2, null);
        }
        catch (Error | RuntimeException throwable) {
            throw throwable;
        }
        catch (Throwable throwable) {
            throw new UndeclaredThrowableException(throwable);
        }
    }

    public final int hashCode() {
        try {
            return (Integer)this.h.invoke(this, m0, null);
        }
        catch (Error | RuntimeException throwable) {
            throw throwable;
        }
        catch (Throwable throwable) {
            throw new UndeclaredThrowableException(throwable);
        }
    }

    public final void rantHouse() {
        try {
            this.h.invoke(this, m3, null);
            return;
        }
        catch (Error | RuntimeException throwable) {
            throw throwable;
        }
        catch (Throwable throwable) {
            throw new UndeclaredThrowableException(throwable);
        }
    }
}
```

3.4、CGLIB动态代理

CGLIB动态代理是一种Java动态代理技术，它可以在运行时动态地生成一个子类来作为被代理对象的代理。相比于JDK自带的动态代理，CGLIB动态代理使用**更加灵活**，它为没有实现接口的类提供代理，为JDK的动态代理提供了很好的补充。

> 🌈
>
> CGLIB动态代理和JDK动态代理最大的区别就是前者使用的是第三方包，不需要有抽象主题的接口，后者是JDK自带的，必须要有抽象主题接口。

CGLIB动态代理的原理是通过**继承**被代理类，然后重写其中的方法实现代理功能。当调用被代理类的方法时，实际上是调用了代理类中重写的方法。这样就可以对被代理类的方法进行增强或拦截，从而实现**AOP(面向切面编程)**的功能。

CGLIB是第三方提供的资源包，所以在使用之前需要引入jar包依赖：

```
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>2.2.2</version>
</dependency>
```

具体实现步骤如下：

1. **创建 Enhancer 实例**：`Enhancer` 是 CGLIB 中的主要类，用于生成代理对象。需要使用 `Enhancer` 创建一个新的代理对象；
2. **设置父类**：CGLIB生成的代理对象是目标类的子类，因此需要设置父类。这可以通过调用 `Enhancer.setSuperclass()` 方法来完成；
3. **设置回调**：回调是代理对象将要执行的操作。可以使用 `MethodInterceptor` 或 `CallbackFilter` 等类来设置回调；
4. **创建代理对象**：最后一步是使用 `Enhancer.create()` 方法创建代理对象。

```
/**
 * @author xbaozi
 * @version 1.0
 * @classname CGLIBDynamicProxy
 * @date 2023-04-10  15:22
 * @description CGLIB动态代理
 */
public class CGLIBDynamicProxy {
    public static void main(String[] args) {
        System.out.println("\n********** 直接找到房东租房 **********");
        Customer customerToLandlord = new Customer(new Landlord());
        customerToLandlord.findHouse();
        System.out.println("\n********** 找附近手握房源的中介租房 **********");
```

```
        Landlord proxyIntermediary = new ProxyFactory().getProxy();
        Customer customerToIntermediary = new
Customer(proxyIntermediary);
        customerToIntermediary.findHouse();
    }
}

/**
 * 真实主题，实现抽象主题
 */
class Landlord {

    public void rantHouse() {
        System.out.println("[CGLIB动态代理-真实主题] 找到房东租房……");
    }
}

/**
 * 代理工厂，用于生成代理主题类
 */
class ProxyFactory implements MethodInterceptor {
    private Landlord landlordSuper = new Landlord();

    public Landlord getProxy() {
        // 1. 创建Enhancer实例
        Enhancer enhancer = new Enhancer();
        // 2. 设置父类
        enhancer.setSuperclass(landlordSuper.getClass());
        // 3. 设置回调
        enhancer.setCallback(this);
        // 4. 创建代理对象
        Landlord landlord = (Landlord) enhancer.create();
        return landlord;
    }

    @Override
    public Landlord intercept(Object o, Method method, Object[]
args, MethodProxy methodProxy) throws Throwable {
        System.out.println("[CGLIB动态代理-代理主题] 去中介公司交中介费
获取中介服务");
        Landlord landlord = (Landlord) methodProxy.invokeSuper(o,
args);
        System.out.println("[CGLIB动态代理-代理主题] 和租户对接好后续工
作");
        return landlord;
    }
}

/**
```

```
 * 租户类
 */
class Customer {
    private Landlord rentHouse;

    public Customer(Landlord rentHouse) {
        this.rentHouse = rentHouse;
    }

    public void findHouse() {
        rentHouse.rantHouse();
    }
}
```

运行结果如下：

```
"D:\Program Files\Java\jdk-11.0.16.1\bin\java.exe" "-javaagent:D:\Program File

********** 直接找到房东租房 **********
[CGLIB动态代理-真实主题] 找到房东租房……

********** 找附近手握房源的中介租房 **********
[CGLIB动态代理-代理主题] 去中介公司交中介费获取中介服务
[CGLIB动态代理-真实主题] 找到房东租房……
[CGLIB动态代理-代理主题] 和租户对接好后续工作

Process finished with exit code 0
```

4、区别对比

4.1、静态代理和动态代理

动态代理和静态代理是代理模式中两种不同的实现方式，它们之间的区别主要体现在以下几个方面：

1. **代理类生成时期不同。**静态代理在**编译期**就已经确定了代理类与委托类的关系，即代理类和委托类是早已确定并且固定的。而动态代理则是在**运行时**通过反射机制动态地生成代理类，使其具有与委托类相同的接口和方法；
2. **灵活性不同。**由于静态代理在编译期就确定了代理类和委托类的关系，因此它的**灵活性较差**，无法在运行时改变代理类和委托类的关系。而动态代理则可以根据需要在运行时生成代理类，并动态地指定具体的委托类对象，从而具有**更高的灵活性**；

3. **实现原理不同。**静态代理在程序编写和编译时，需要开发人员**手动编写**代理类和委托类的代码，较为繁琐。而动态代理是通过**Java反射机制**生成代理类的字节码，并加载到JVM中，然后动态创建代理实例。这种方式大大简化了代理类的开发工作。

4.2、JDK动态代理和CGLIB动态代理

JDK动态代理和CGLIB动态代理都是Java中的动态代理技术，它们的主要区别在于实现方式和适用场景。

JDK动态代理是通过**反射机制**来实现的，在**运行时**动态地创建一个实现了**指定接口的代理类**，代理类中的方法调用会被转发到 `InvocationHandler` 进行处理。因此，JDK动态代理**只能代理实现了接口的类**，并且生成的代理类**只能代理接口中声明的方法**，对于其他方法则无法代理。

CGLIB动态代理则是通过继承目标类来实现的，它创建的代理类是目标类的子类，重写了目标类中的非final方法，并将它们分派到 `Callback` 中定义的拦截器中去处理。因此，CGLIB动态代理**可以代理没有实现接口的类**，并且**可以代理目标类中所有非final方法。**

> 🌈
>
> 简而言之，JDK动态代理适用于代理有接口的类，而CGLIB动态代理则适用于代理没有接口或者需要代理目标类中所有非final方法的类。因此**大部分情况下有接口用JDK，无接口用CGLIB。**

除了上述区别之外，JDK动态代理和CGLIB动态代理还有一些其他的差异：

1. **性能**：一般情况下，后者性能比前者要好。JDK动态代理使用反射机制动态创建代理类，生成代理对象的效率相对较低；而CGLIB动态代理则是直接生成目标类的子类，因此生成代理对象的效率较高。
2. **内存占用**：由于CGLIB动态代理创建的代理类是目标类的子类，所以代理类会继承目标类的所有非私有属性和方法，导致代理类的内存占用比较大；而JDK动态代理生成的代理类只包含需要代理的接口方法，因此内存占用相对较小。
3. **依赖性**：JDK动态代理是Java原生的API，不需要引入第三方库，而CGLIB动态代理需要引入cglib库进行支持。
4. **版本兼容性**：JDK动态代理是Java原生API，因此具有很好的版本兼容性；而CGLIB动态代理在不同版本的Java环境下可能存在兼容性问题，推荐使用的是 `2.2.2` 及以下版本。

> 🌈
>
> 这里补充一个性能相关的小知识，在Java中，JDK在5、6、7、8等版本中都对动态代理进行了优化，使得在JDK8及之后，JDK动态代理的性能与CGLIB动态代理性能持平甚至反超。
>
> - 在**JDK5**中，Java引入了新的虚拟机指令——" `invokedynamic` "，该指令的出现为动态语言的实现提供了更广泛的支持。这项技术的引入也为Java的动态代理提供了更好的性能和灵活性。
> - 在**JDK6和7**中，Java对**反射机制**进行了一系列优化，使得动态代理的创建和调用效率得到了显著提升。
> - 在**JDK8**中，Java**引入了默认方法和Lambda表达式**等新特性，这些特性进一步提升了动态代理的性能和效率。同时，JDK8还引入了MethodHandle类，可以更高效地调用方法。

5、代理模式优缺点

优点	缺点
可以在客户端毫无察觉的情况下控制服务对象	代码可能会变得复杂， 因为需要新建许多类
如果客户端对服务对象的生命周期没有特殊要求， 可以对生命周期进行管理	服务响应可能会延迟
即使服务对象还未准备好或不存在， 代理也可以正常工作	
符合*开闭原则*。 可以在不对服务或客户端做出修改的情况下创建新代理	

6、应用场景

代理模式是一种结构型设计模式，它通过增加一个代理对象来控制对原始对象的访问。代理对象可以在不改变原始对象的前提下，实现额外的功能或者控制访问级别。

下面是一些代理模式中常见的应用场景：

1. **AOP**（面向切面编程）：通过动态代理，在方法前后自动添加日志记录、权限控制、性能统计等通用功能，避免了代码冗余，提高了代码的复用性和可维护性；
2. **RPC**（远程过程调用）：在分布式系统中，动态代理可以将远程方法调用封装成本地方法调用，使得远程调用像本地调用一样简单，同时也支持负载均衡、容错等功能；
3. **数据库连接池**：数据库连接池可以通过动态代理来实现，每次获取连接时，动态代理会检查当前连接是否可用，如果已经关闭或者超时，则重新创建连接返回给用户；
4. **缓存框架**：缓存框架可以通过动态代理来实现，当一个对象需要缓存时，动态代理可以根据缓存配置来判断是否需要从缓存中获取数据，还是直接从数据库中获取数据并更新缓存；
5. **日志框架**：动态代理可以用于实现日志框架，例如Spring AOP中的日志切面，可以动态地在方法前后添加日志记录代码，以此来监控系统运行情况，方便问题排查；
6. **虚拟代理**：虚拟代理是一种延迟加载技术，它允许对象在真正需要时才被创建。例如，在需要显示大量图片的应用程序中，可以使用虚拟代理来延迟加载图片，只有当用户需要查看图片时才会加载。