



北京动力节点教育科技有限公司

动力节点课程讲义

DONGLIJIEDIANKECHENGJIANGYI

www.bjpowernode.com

第1章 代理模式

1.1 什么是代理模式

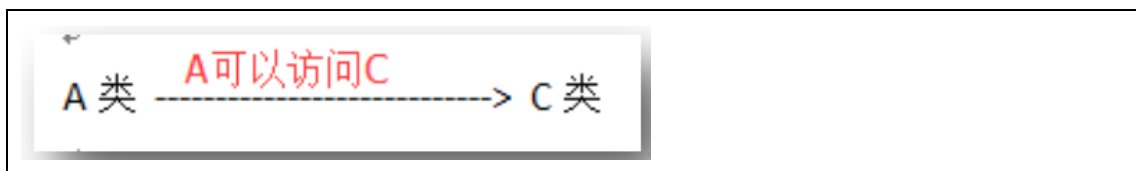
代理模式是指，为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户和目标对象之间起到中介的作用。

换句话说，使用代理对象，是为了在不修改目标对象的基础上，增强主业务逻辑。客户类真正的想要访问的对象是目标对象，但客户类真正可以访问的对象是代理对象。

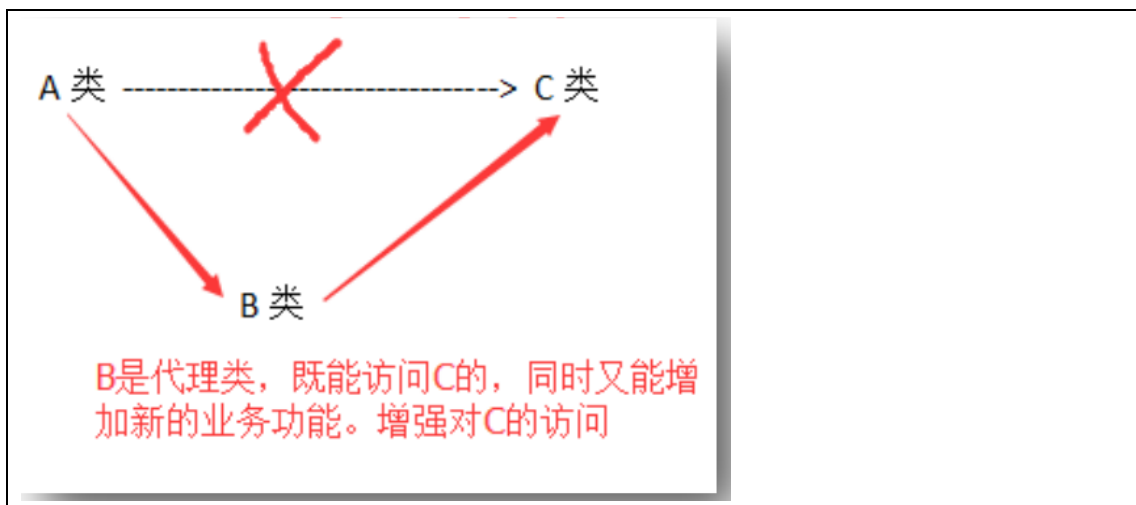
客户类对目标对象的访问是通过访问代理对象来实现的。当然，代理类与目标类要实现同一个接口。

例如：有 A, B, C 三个类，A 原来可以调用 C 类的方法，现在因为某种原因 C 类不允许 A 类调用其方法，但 B 类可以调用 C 类的方法。A 类通过 B 类调用 C 类的方法。这里 B 是 C 的代理。A 通过代理 B 访问 C。

原来的访问关系：



通过代理的访问关系：



Window 系统的快捷方式也是一种代理模式。快捷方式代理的是真实的程序，双击快捷

方式是启动它代表的程序。

1.2 代理模式的作用

- 控制目标对象的访问
- 增强功能

1.3 代理模式的分类

- 静态代理
- 动态代理又分为 JDK 动态代理和 CGLib 动态代理

1.4 代理的实现方式

- 静态代理实现
- 动态代理的实现又分为 JDK 动态代理和 CGLib 动态代理

第2章 静态代理

2.1 静态代理的特点

静态代理要求目标对象和代理对象实现同一个业务接口。代理对象中的核心功能是由目标对象来完成，代理对象负责增强功能。

2.2 静态代理的实现

需求：有个明星(目标对象)很大腕，档期很满，我们想约这个明星来学校表演。我们只能通过他的助理来约他，助理就是（代理对象）。具体的时间、地点、场合、费用都只能跟助理来谈。助理完全负责明星的所有行程。并且安排明星来表演。

2.1.1 实现步骤

(1) 定义业务接口

定义业务接口 IService（目标对象和代理对象都要实现的业务接口）。

```
/**
 * 业务接口,规定业务功能
 */
public interface Service {
    void sing();
}
```

(2) 目标类实现接口

```
/**
 * 目标对象刘德华,要实现业务接口,并实现功能
 */
public class SuperStarLiu implements Service {
    @Override
    public void sing() {
        System.out.println("我是刘德华,我正在唱歌.....");
    }
}
```

(3) 代理类实现功能

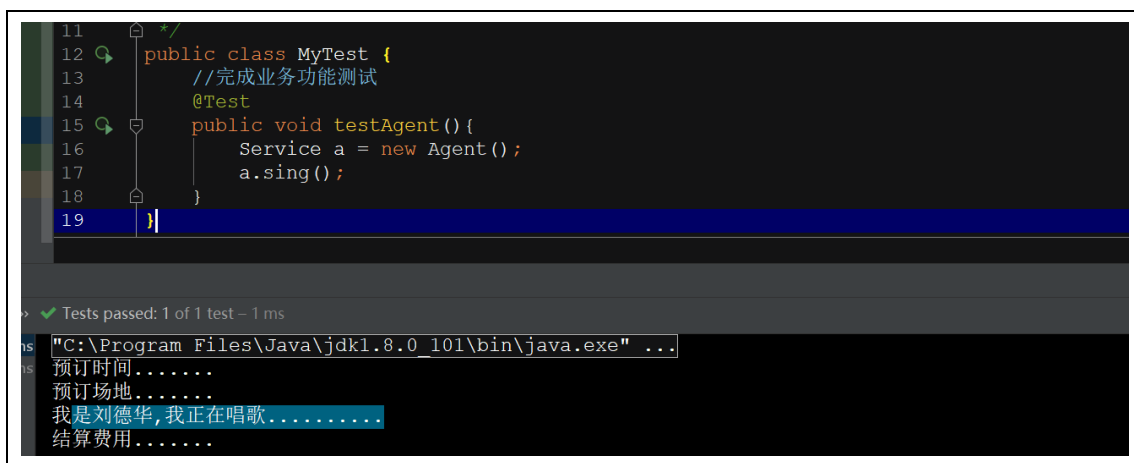
```
public class Agent implements Service {

    @Override
    public void sing() {
        System.out.println("预订时间.....");
        System.out.println("预订场地.....");

        //请目标对象来完成业务功能
        SuperStarLiu liu = new SuperStarLiu();
        liu.sing();

        System.out.println("结算费用.....");
    }
}
```

(4) 测试



```
11  /**
12  * public class MyTest {
13  *     //完成业务功能测试
14  *     @Test
15  *     public void testAgent(){
16  *         Service a = new Agent();
17  *         a.sing();
18  *     }
19  * }
```

Tests passed: 1 of 1 test - 1 ms

"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe" ...

预订时间.....

预订场地.....

我是刘德华,我正在唱歌.....

结算费用.....

(5) 代理功能改造

现在的代理类只能代理一个目标对象,不够灵活.如果需要代理多个目标对象,是可以使用面向接口编程.

面向接口编程的要点:

- A. 类中的成员变量设计为接口
- B. 方法的参数设计为接口
- C. 方法的返回值设计为接口
- D. 调用时接口指向实现类

切记:上了接口就是上灵活.



```
/**
 * 助理就是代理对象
 */
public class Agent implements Service {
    //类中的成员变量设计为接口,传进来谁就是谁的实现
    public Service target;
    //通过构造方法传入目标对象
    public Agent(Service target){
        this.target = target;
    }
    @Override
    public void sing() {
        System.out.println("预订时间.....");
        System.out.println("预订场地.....");
    }
}
```

```
//请目标对象来完成业务功能
//    SuperStarLiu liu = new SuperStarLiu();
//    liu.sing();
//    SuperStarZhou zhou = new SuperStarZhou();
//    zhou.sing();
target.sing();//谁来就调用谁的实现
System.out.println("结算费用.....");
}
}
```

2.3 静态代理的缺陷

2.3.1 代理复杂，难于管理

代理类和目标类实现了相同的接口，每个代理都需要实现目标类的方法，这样就出现了大量的代码重复。如果接口增加一个方法，除了所有目标类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。

2.3.2 代理类依赖目标类，代理类过多

代理类只服务于一种类型的目标类，如果要服务多个类型。势必要为每一种目标类都进行代理，静态代理在程序规模稍大时就无法胜任了，代理类数量过多

第3章 动态代理

动态代理是指代理类对象在程序运行时由 JVM 根据反射机制动态生成的。动态代理不需要定义代理类的.java 源文件。动态代理其实就是 jdk 运行期间，动态创建 class 字节码并加载到 JVM。动态代理的实现方式常用的有两种：使用 JDK 动态代理和 CGLIB 动态代理。

3.1 JDK 动态代理

JDK 动态代理是基于 Java 的反射机制实现的。使用 JDK 中接口和类实现代理对象的动态创建。JDK 的动态代理要求目标对象必须实现接口，而代理对象不必实现业务接口，这是 java 设计上的要求。从 jdk1.3 以来，java 语言通过 `java.lang.reflect` 包提供三个类和接口支持代理模式，它们分别 `Proxy`, `Method` 和 `InvocationHandler`。

3.1.1 InvocationHandler 接口

`InvocationHandler` 接口叫做调用处理器，负责完成调用目标方法，并增强功能。通过代理对象执行目标接口中的方法，会把方法的调用分派给调用处理器(`InvocationHandler`)的实现类，执行实现类中的 `invoke()`方法，我们需要把功能代理写在 `invoke()`方法中。此接口中只有一个方法。

```
public interface InvocationHandler {  
  
    public Object invoke(Object proxy, Method method, Object[] args)  
        throws Throwable;  
}
```

在 `invoke` 方法中可以截取对目标方法的调用。在这里进行功能增强。Java 的动态代理是建立在反射机制之上的。实现了 `InvocationHandler` 接口的类用于加强目标类的主业务逻辑。这个接口中有一个方法 `invoke()`，具体加强的代码逻辑就是定义在该方法中的。通过代理对象执行接口中的方法时，会自动调用 `invoke()`方法。

`invoke()`方法的介绍如下：

```
public Object invoke ( Object proxy, Method method, Object[] args)
```

proxy: 代表生成的代理对象

method: 代表目标方法

args: 代表目标方法的参数

第一个参数 `proxy` 是 `jdk` 在运行时赋值的，在方法中直接使用，第二个参数后面介绍，第三个参数是方法执行的参数，这三个参数都是 `jdk` 运行时赋值的，无需程序员给出。

3.1.2 Method 类

`invoke()`方法的第二个参数为 `Method` 类对象，该类有一个方法也叫 `invoke()`，可以调用目标方法。这两个 `invoke()`方法，虽然同名，但无关。

```
public Object invoke ( Object obj, Object... args)
```

`obj`: 表示目标对象

`args`: 表示目标方法参数，就是其上一层 `invoke` 方法的第三个参数

该方法的作用是：调用执行 `obj` 对象所属类的方法，这个方法由其调用者 `Method` 对象确定。在代码中，一般的写法为

```
method.invoke(target, args);
```

其中，`method` 为上一层 `invoke` 方法的第二个参数。这样，即可调用了目标类的目标方法。

3.1.3 Proxy 类

通过JDK的`java.lang.reflect.Proxy`类实现动态代理，会使用其静态方法`newProxyInstance()`，依据目标对象、业务接口及调用处理器三者，自动生成一个动态代理对象。

```
public static newProxyInstance ( ClassLoader loader, Class<?>[] interfaces,  
InvocationHandler handler)
```

`loader`: 目标类的类加载器，通过目标对象的反射可获取

`interfaces`: 目标类实现的接口数组，通过目标对象的反射可获取

`handler`: 调用处理器。

3.1.4 实现步骤

- (1) 代理对象不需要实现接口
- (2) 代理对象的生成是利用JDKAPI中的`Proxy`类， 动态的在内存中构建代理对象
- (3) 代码实现结构



(4) ProxyFactory.java 代理实例生成工厂

```
public class ProxyFactory {  
    //任何的代理对象，都要清楚目标对象，在此得设置一个目标对象，  
    private IService superStar;  
    //传入目标对象  
    public ProxyFactory(IService superStar){  
        this.superStar=superStar;  
    }  
    //给目标对象生成代理实例  
    public Object getProxyInstance(){  
        return Proxy.newProxyInstance(  
            //指定当前目标对象，使用类加载器获得  
            superStar.getClass().getClassLoader(),  
            //获得目标对象实现的所有接口  
            superStar.getClass().getInterfaces(),  
            //处理代理实例上的方法并返回调用结果  
            new InvocationHandler(){  
                @Override  
                public Object invoke(  
                    //代理对象的实例  
                    Object proxy,  
                    //代理的目标对象的实现方法  
                    Method method,  
                    //代理的目标对象实现方法的参数  
                    Object[] args) throws Throwable {  
                        System.out.println("预定场地.....");  
                        //目标对象执行自己的方法  
                        Object returnValue=method.invoke(superStar, args);  
                        System.out.println("结帐走人.....");  
                        return returnValue;  
                    }  
                });  
    }  
}
```

(5) 测试类

```
@Test
public void testDynamicProxy(){
    //先创建目标对象
    IService superStar=new SuperStar();
    //创建代理对象
    IService agent=(IService) new ProxyFactory(superStar).getProxyInstance();
    agent.sing();
}
```

注意：JDK 动态代理中，代理对象不需要实现接口，但是目标对象一定要实现接口，否则不能用 JDK 动态代理。

3.2 CGLib（Code Generation Library）动态代理

想要功能扩展，但目标对象没有实现接口，怎样功能扩展？

解决方案：子类的方式

```
Class subclass extends UserDao{
```

以子类的方式实现(cglib 代理)，在内存中构建一个子类对象从而实现对目标对象功能的扩展。

3.2.1 CGLib 动态代理的特点

(1) JDK 的动态代理有一个限制，就是使用动态代理的目标对象必须实现一个或多个接口。如果想代理没有实现接口的类，就可以使用 CGLIB 实现。

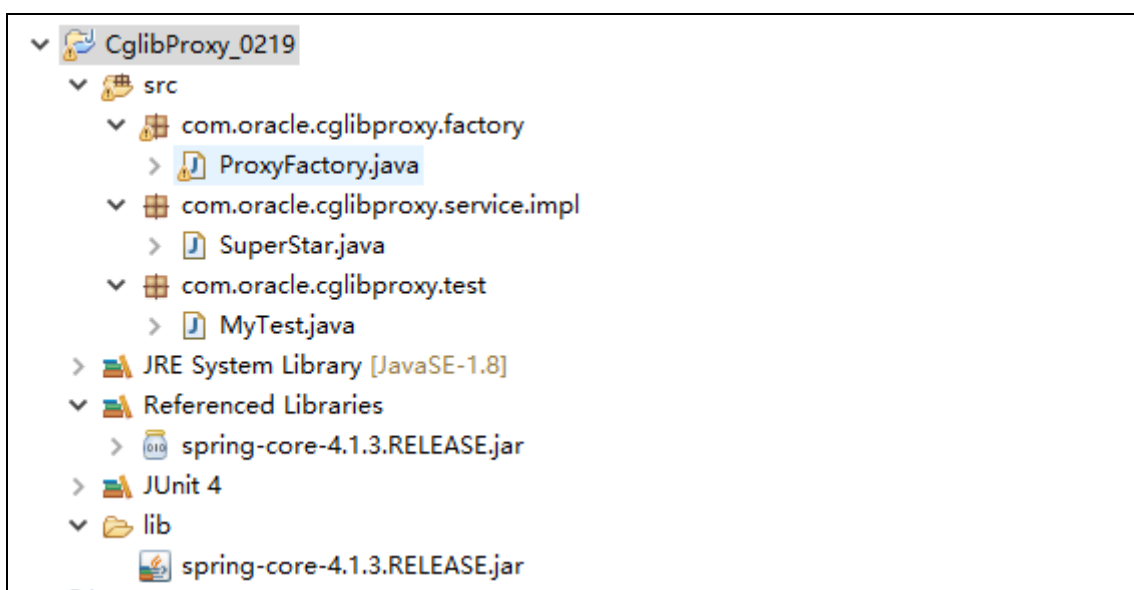
(2) CGLIB 是一个强大的高性能的代码生成包，它可以在运行期扩展 Java 类与实现 Java 接口。它广泛的被许多 AOP 的框架使用，例如 Spring AOP 和 dynaop，为他们提供方法的 interception。

(3) CGLIB 包的底层是通过使用一个小而快的字节码处理框架 ASM，来转换字节码并生成新的类。不鼓励直接使用 ASM，因为它要求你必须对 JVM 内部结构包括 class 文件的格式和指令集都很熟悉。

3.2.2 CGLib 实现的步骤

- (1) 需要 spring-core-5.2.5.jar 依赖即可。
- (2) 引入功能包后，就可以在内存中动态构建子类
- (3) 被代理的类不能为 final， 否则报错。
- (4) 目标对象的方法如果为 final/static， 那么就不会被拦截，即不会执行目标对象额外的业务方法。

- (5) 代码实现结构



ProxyFactory.java

```
public class ProxyFactory implements MethodInterceptor {
    //目标对象
    private Object target;
    //传入目标对象
    public ProxyFactory(Object target){
        this.target=target;
    }
    //Cglib 采用底层的字节码技术，在子类中采用方法拦截的技术，拦截父类指定方法的调用，并顺势植入代理功能的代码
    @Override
    public Object intercept(Object obj, Method method, Object[] arg2, MethodProxy proxy)
    throws Throwable {
        //代理对象的功能
        System.out.println("预定场地.....");
        //调用目标对象的方法
        Object returnValue=method.invoke(target, arg2);
    }
}
```

```
//代理对象的功能
System.out.println("结帐走人.....");
return returnValue;
}
//生成代理对象
public Object getProxyInstance(){
    //1.使用工具类
    Enhancer en=new Enhancer();
    //2.设置父类
    en.setSuperclass(target.getClass());
    //3.设置回调函数
    en.setCallback(this);
    //4.创建子类（代理）对象
    return en.create();
}
```

测试类

```
@Test
public void testCglibProxy(){
    SuperStar superStar=new SuperStar();
    System.out.println(superStar.getClass());
    SuperStar proxy=(SuperStar) new ProxyFactory(superStar).getProxyInstance();
    System.out.println(proxy.getClass());
    proxy.sing();
}
```