# 静态代理模式

接口:

|  |
| --- |
| package lw.pers.test;  public interface Count {  void queryCount();  void updateCount();  } |

实现类:

|  |
| --- |
| package lw.pers.test;  public class CountImpl implements Count {  @Override  public void queryCount() {  System.out.println("查询账户");  }  @Override  public void updateCount() {  System.out.println("更改账户");  }  } |

代理类:

|  |
| --- |
| package lw.pers.test;  public class CountProxy implements Count{  private CountImpl count;  public CountProxy(CountImpl count) {  this.count = count;  }  @Override  public void queryCount() {  System.out.println("查询账户预处理");  count.queryCount();  System.out.println("查询账户预之后");  }  @Override  public void updateCount() {  System.out.println("修改账户预处理");  count.updateCount();  System.out.println("修改账户预之后");  }  } |

测试:

|  |
| --- |
| CountImpl count = new CountImpl();  CountProxy countProxy = new CountProxy(count);  countProxy.updateCount();  countProxy.queryCount(); |

静态代理的缺点很明显：一个代理类只能对一个业务接口的实现类进行包装，如果有多个业务接口的话就要定义很多实现类和代理类才行。而且，如果代理类对业务方法的预处理、调用后操作都是一样的（比如：调用前输出提示、调用后自动关闭连接），则多个代理类就会有很多重复代码。这时我们可以定义这样一个代理类，它能代理所有实现类的方法调用：根据传进来的业务实现类和方法名进行具体调用。——那就是动态代理。

# 动态代理—jdk动态代理

接口:

|  |
| --- |
| package lw.pers.test;  public interface BookFacade {  void addBook();  } |

实现类:

|  |
| --- |
| package lw.pers.test;  public class BookFacadeImpl implements BookFacade {  @Override  public void addBook() {  System.out.println("增加图书的方法");  }  } |

代理类:

|  |
| --- |
| package lw.pers.test;  import java.lang.reflect.InvocationHandler;  import java.lang.reflect.Method;  import java.lang.reflect.Proxy;  public class BookFacadeProxy implements InvocationHandler{  private Object target;  public Object bind(final Object target) {  this.target = target;  return Proxy.newProxyInstance(target.getClass().getClassLoader(),target.getClass().getInterfaces(),this);  }  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  Object result = null;  System.out.println("预处理");  result = method.invoke(target,args);  System.out.println("预处理之后");  return result;  }  } |

测试:

|  |
| --- |
| BookFacadeProxy bookFacadeProxy = new BookFacadeProxy();  BookFacadeImpl bookFacade = new BookFacadeImpl();  BookFacade bind = (BookFacade) bookFacadeProxy.bind(bookFacade);  bind.addBook(); |

缺点:需要有接口,不然要报错

# 动态代理—cglib动态代理

业务类:

|  |
| --- |
| package lw.pers.test;  public class BookFacadeImpl {  public void addBook() {  System.out.println("增加图书的普通方法...");  }  } |

代理类:

|  |
| --- |
| package lw.pers.test;  import org.springframework.cglib.proxy.Enhancer;  import org.springframework.cglib.proxy.MethodInterceptor;  import org.springframework.cglib.proxy.MethodProxy;  import java.lang.reflect.Method;  public class BookFacadeCglib implements MethodInterceptor {  private Object target; //业务对象  public Object getInstance(Object target){  this.target = target;  Enhancer enhancer = new Enhancer(); //创建加强器:用来创建动态代理类  enhancer.setSuperclass(this.target.getClass()); //给加强器指定要代理的业务类  //设置回调：对于代理类上所有方法的调用，都会调用CallBack，而Callback则需要实现intercept()方法进行拦  enhancer.setCallback(this);  //创建动态代理类对象并返回  return enhancer.create();  }  @Override  public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  System.out.println("预处理");  methodProxy.invokeSuper(o,objects);//调用业务（父类中）的方法  System.out.println("预处理之后");  return null;  }  } |

测试:

|  |
| --- |
| BookFacadeImpl bookFacade = new BookFacadeImpl();  BookFacadeCglib bookFacadeCglib = new BookFacadeCglib();  BookFacadeImpl instance = (BookFacadeImpl )bookFacadeCglib.getInstance(bookFacade);  instance.addBook(); |

cglib是针对类来实现代理的，原理是对指定的业务类生成一个子类，并覆盖其中业务方法实现代理。因为采用的是继承，所以不能对final修饰的类进行代理

# 4.总结

静态代理是通过在代码中显式定义一个业务实现类一个代理，在代理类中对同名的业务方法进行包装，用户通过代理类调用被包装过的业务方法；

JDK动态代理是通过接口中的方法名，在动态生成的代理类中调用业务实现类的同名方法；

CGlib动态代理是通过继承业务类，生成的动态代理类是业务类的子类，通过重写业务方法进行代理；