1. 为什么要使用动态代理？

动态代理：在不改变原有代码的情况下上进行对象功能增强 使用代理对象代替原来的对象完成功能 进而达到拓展功能的目的

2.JDK Proxy 动态代理面向接口的动态代理

特点：

1. 一定要有接口和实现类的存在 代理对象增强的是实现类 在实现接口的方法重写的方法
2. 生成的代理对象只能转换成 接口的不能转换成 被代理类
3. 代理对象只能增强接口中定义的方法 实现类中其他和接口无关的方法是无法增强的
4. 代理对象只能读取到接口中方法上的注解 不能读取到实现类方法上的注解
   使用方法：

```
public class Test1 {
    public static void main(String[] args) {
        Dinner dinner=new Person("张三");
        // 通过Porxy动态代理获得一个代理对象,在代理对象中,对某个方法进行增强
//        ClassLoader loader,被代理的对象的类加载器
        ClassLoader classLoader = dinner.getClass().getClassLoader();
//        Class<?>[] interfaces,被代理对象所实现的所有接口
        Class[] interaces= dinner.getClass().getInterfaces();
//        InvocationHandler h,执行处理器对象,专门用于定义增强的规则
        InvocationHandler handler = new InvocationHandler(){
            // invoke 当我们让代理对象调用任何方法时,都会触发invoke方法的执行
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
//                Object proxy, 代理对象
//                Method method,被代理的方法
//                Object[] args,被代理方法运行时的实参
                Object res=null;
                if(method.getName().equals("eat")){
                    System.out.println("饭前洗手");
                    // 让原有的eat的方法去运行
                    res =method.invoke(dinner, args);
                    System.out.println("饭后刷碗");
                }else{
                    // 如果是其他方法,那么正常执行就可以了
                    res =method.invoke(dinner, args);
                }
                 return res;
            }
        };
        Dinner dinnerProxy =(Dinner)
Proxy.newProxyInstance(classLoader,interaces,handler);
        //dinnerProxy.eat("包子");
        dinnerProxy.drink();
    }
}
interface Dinner{
    void eat(String foodName);
    void drink();
}
```

```
class Person implements Dinner{
    private String name;
    public Person(String name) {
        this.name = name;
    }
    @Override
    public void eat(String foodName) {
        System.out.println(name+"正在吃"+foodName);
    }
    @Override
    public void drink( ) {
        System.out.println(name+"正在喝茶");
    }
}
class Student implements Dinner{
    private String name;
    public Student(String name) {
        this.name = name;
    }
    @Override
    public void eat(String foodName) {
        System.out.println(name+"正在食堂吃"+foodName);
    }
    @Override
    public void drink( ) {
        System.out.println(name+"正在喝可乐");
    }
}
```

3.CGlib动态代理

cglib动态代理模式是面向父类

特点：

1. 面向父类的和接口没有直接关系
   2.不仅可以增强接口中定义的方法还可以增强其他方法
   3.可以读取父类中方法上的所有注解
2. 使用实例

```
public class Test1 {
    @Test
    public void testCglib(){
        Person person =new Person();
        // 获取一个Person的代理对象
        // 1 获得一个Enhancer对象
        Enhancer enhancer=new Enhancer();
        // 2 设置父类字节码
        enhancer.setSuperclass(person.getClass());
        // 3 获取MethodIntercepter对象 用于定义增强规则
        MethodInterceptor methodInterceptor=new MethodInterceptor() {
            @Override
            public Object intercept(Object o, Method method, Object[]
objects, MethodProxy methodProxy) throws Throwable {
                /*Object o,  生成之后的代理对象 personProxy
```

```
                Method method,  父类中原本要执行的方法  Person>>> eat()
                Object[] objects, 方法在调用时传入的实参数组
                MethodProxy methodProxy  子类中重写父类的方法 personProxy >>>
eat()
                */
                Object res =null;
                if(method.getName().equals("eat")){
                    // 如果是eat方法 则增强并运行
                    System.out.println("饭前洗手");
                    res=methodProxy.invokeSuper(o,objects);
                    System.out.println("饭后刷碗");
                }else{
                    // 如果是其他方法 不增强运行
                    res=methodProxy.invokeSuper(o,objects); // 子类对象方法在执
行,默认会调用父类对应被重写的方法
                }
                return res;
            }
        };
        // 4 设置methodInterceptor
        enhancer.setCallback(methodInterceptor);
        // 5 获得代理对象
        Person personProxy = (Person)enhancer.create();
        // 6 使用代理对象完成功能
        personProxy.eat("包子");
    }
}
class Person  {
    public Person( ) {
    }
    public void eat(String foodName) {
        System.out.println("张三正在吃"+foodName);
    }
}
```

4. 两个动态代理的区别

1. JDK动态代理是面向接口的，只能增强实现类中接口中存在的方法。CGlib是面向父类的，可以增强父类的所有方法
2. JDK得到的对象是JDK代理对象实例，而CGlib得到的对象是被代理对象的子类