设计模式概述

基本概念
设计模式是一套被反复使用的、多数人知晓的、经过分类编目的代码设计经验的总结

优点
代码质量好,可靠性高
复用已有的代码
代码更规范,更容易理解

但是设计模式不是万能的,并不能解决所有问题,不要滥用

设计模式原则(SOLID)(内功心法)

单一职责原则(SRP)

每个类只需要负责自己的那部分，类的复杂度就会降低。如果职责划分得很清楚，那么代码维护起来也更加容易
事实上，由于一些其他的因素影响，类的单一职责在项目中是很难保证的。通常，接口和方法的单一职责更容易实现

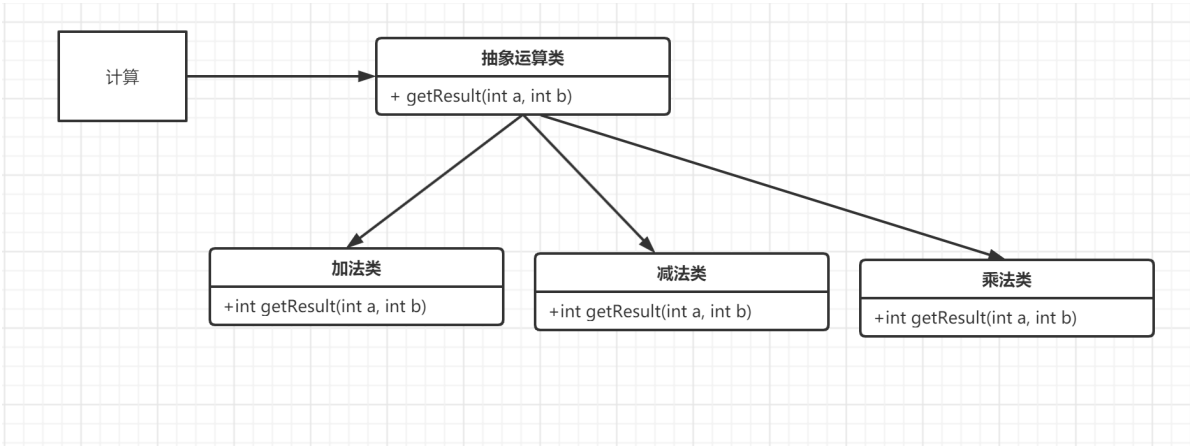
举例:

手机

摄像机

开闭原则(OCP)

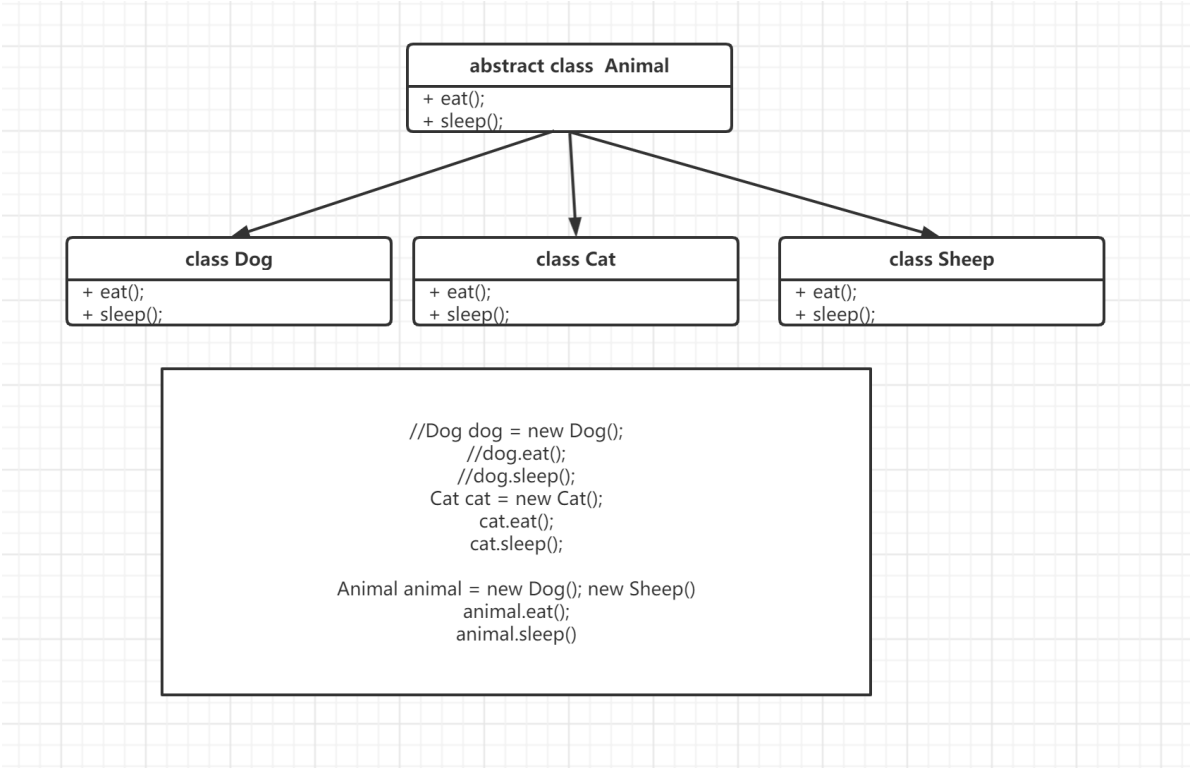
一个软件实体，如类、模块和函数应该对扩展开放，对修改关闭(实现热插拔功能)



里式替换原则(LSP)

所有引用基类的地方必须能透明地使用其子类的对象
子类可以实现父类的抽象方法,但不能覆盖父类的非抽象方法

```
Father father = new Son()
```



接口隔离原则(ISP)

一个接口不需要提供太多的行为,一个接口应该只提供一种对外的功能,不应该把所有的操作都封装到一个接口(不同的功能放入不同的接口中,避免写大接口)

依赖倒置原则(DIP)

上层模块不应该依赖底层模块，它们都应该依赖于抽象
抽象不应该依赖于细节(具体类)，细节应该依赖于抽象(面向接口编程,抽象类型)

```
package _26design_pattern.com.cskaoyan._01principle;

/**
 * @description:
 * @author: 景天
 * @date: 2022/8/2 15:20
 **/
/*
依赖倒置
 */
public class DIPTest {
    public static void main(String[] args) {
        Person person = new Person();
        //person.receiveEmail(new Email());
        //person.receiveQQ(new QQ());
        person.receiveMsg(new QQ());
    }
}

interface IReceiver{
    String receiveMsg();
}
```

```
class Email implements IReceiver{

    @Override
    public String receiveMsg() {
        return "Email:msg";
    }
}


class QQ implements IReceiver{

    @Override
    public String receiveMsg() {
        return "QQ:msg";
    }
}

class Person{
    // 定义专门接收邮件的方法
    // 依赖的就是具体的实现类
    public void receiveEmail(Email email) {
        String s = email.receiveMsg();
        System.out.println(s);
    }

    // 定义一个接收QQ消息的方法
    // 依赖的就是具体的实现类
    public void receiveQQ(QQ qq) {
        String s = qq.receiveMsg();
        System.out.println(s);
    }

    // 接收微信消息 接收微博消息  接收抖音
    // 通用的方法
    // 依赖的就是抽象
    public void receiveMsg(IReceiver iReceiver) {
        String s = iReceiver.receiveMsg();
        System.out.println(s);
    }

}
```

具体的设计模式(武功招数)

单例设计模式

什么是单例设计模式?

在应用程序中维护实例的方式,保证1个类仅有1个唯一的实例化对象

单例有什么好处?

内存中只有1个对象,节约空间
避免频繁的创建,销毁对象,提高性能
避免对共享资源的多重占用
为整个系统提供1个全局访问点

如何实现单例模式?

- 构造方法私有
- 提供静态方法,返回实例
- **提供自身类型的全局的成员变量**

懒加载与立即加载(懒汉,饿汉)

- 懒加载: 用到的时候才加载
- 立即加载: 不管用不用,先创建好,用的时候直接返回

线程不安全的懒加载单例

```
package _26design_pattern.com.cskaoyan._02singleton;

/**
 * @description:
 * @author: 景天
 * @date: 2022/8/2 15:55
 **/
/*
线程不安全的懒加载单例
 */
public class Singleton1 {
    //-  **提供自身类型的全局的成员变量**
    private static Singleton1 instance;

    // -  构造方法私有
    private Singleton1() {

    }

    //-  提供静态方法,返回实例
    public static Singleton1 getInstance() {
        // 返回唯一实例
        // 完成赋值
        // 进行判断
        // AB2个线程 A先执行
        // 没赋值
        // B执行
        if (instance == null) {
            // A进来 切换到B线程
            // B进来
            instance = new Singleton1();
        }
        return instance;
    }

}
```

线程安全的懒加载单例

同步方法

```
package _26design_pattern.com.cskaoyan._02singleton;

/**
 * @description:
 * @author: 景天
 * @date: 2022/8/2 15:55
 **/
/*
线程安全的懒加载单例
 */
public class Singleton2 {
    //-  **提供自身类型的全局的成员变量**
    private static Singleton2 instance;

    // -  构造方法私有
    private Singleton2() {

    }

    //-  提供静态方法,返回实例
    public static synchronized Singleton2 getInstance() {
        // 返回唯一实例
        // 完成赋值
        // 进行判断
        // AB2个线程 A先执行
        // 没赋值
        // B执行
        if (instance == null) {
            // A进来 切换到B线程
            // B进来
            instance = new Singleton2();
        }
        return instance;
    }

}
```

Double Check 同步代码块

```
package _26design_pattern.com.cskaoyan._02singleton;

/**
```

```
 * @description:
 * @author: 景天
 * @date: 2022/8/2 15:55
 **/
/*
线程安全的懒加载单例  double check
 */
public class Singleton3 {

    //-  **提供自身类型的全局的成员变量**
    private static Singleton3 instance;

    // -  构造方法私有
    private Singleton3() {

    }

    //-  提供静态方法,返回实例
    public static Singleton3 getInstance() {

        // 第一次校验
        if (instance == null) {
            // A进来
            // B进来
            synchronized (Singleton3.class) {
                // A进来
                // 第二次校验
                if (instance == null) {
                    instance = new Singleton3();
                }
            }
        }
        return instance;
    }

}
```

线程安全的立即加载单例

```
package _26design_pattern.com.cskaoyan._02singleton;

/**
 * @description:
 * @author: 景天
 * @date: 2022/8/2 15:55
 **/
/*
线程安全的立即加载单例
 */
public class Singleton4 {
```

```
    static {
        instance = new Singleton4();
    }
    //-  **提供自身类型的全局的成员变量**
    //private static Singleton4 instance = new Singleton4();
    private static Singleton4 instance;

    // -  构造方法私有
    private Singleton4() {

    }

    //-  提供静态方法,返回实例
    public static Singleton4 getInstance() {

        return instance;
    }

}
```

线程安全的懒加载单例(静态内部类实现)

```
package _26design_pattern.com.cskaoyan._02singleton;

/**
 * @description:
 * @author: 景天
 * @date: 2022/8/2 16:16
 **/
/*
线程安全的懒加载单例 通过静态内部类实现
 */
public class Singleton5 {
    // 提供自身的全局的成员变量
    private static Singleton5 instance;

    // 构造方法私有
    private Singleton5() {
    }

    // 返回实例的方法
    public static Singleton5 getInstance() {
        return Inner.innerMethod();
    }


    static class Inner{
        static {
            // 完成赋值
            instance = new Singleton5();
        }

        static Singleton5 innerMethod() {
            return instance;
        }
```

```
    }
}
```

工厂设计模式

什么是工厂设计模式?
顾名思义,用来产对象的

工厂设计模式有什么好处?
隐藏起来一些代码细节.可以标准化的产生实例

简单工厂

简单工厂模式就是把对类的创建全都交给一个工厂来执行，而用户不需要去关心创建的过程是什么样的，只用告诉工厂我想要什么就行了

```
package _26design_pattern.com.cskaoyan._03factory;

/**
 * @description:
 * @author: 景天
 * @date: 2022/8/2 16:30
 **/

public class FruitFactory {
    // 定义一个静态的方法 负责产生水果对象
    public static Fruit getInstance(String name) {
        Fruit fruit = null;
        if ("apple".equals(name)) {
            fruit = new Apple();
        } else if ("orange".equals(name)) {
            fruit = new Orange();
            // 违反开闭原则
        } else if ("pear".equals(name)) {
            fruit = new Pear();
        } else {
            System.out.println("生产不了");
        }
        // 最终目的返回水果对象
        return fruit;
    }


    //
    public static Fruit getInstance2(String className) throws Exception{
        Class<?> c = Class.forName(className);
        Fruit fruit = (Fruit) c.newInstance();

        // 返回一个水果对象
        return fruit;
    }
}
```

工厂方法

工厂方法模式中抽象工厂类负责定义创建对象的接口,具体对象的创建工作由继承抽象工厂的具体类实现

```
package _26design_pattern.com.cskaoyan._03factory;

/**
 * @description:
 * @author: 景天
 * @date: 2022/8/2 16:43
 **/

abstract public class CarFactory {
    abstract Car createCar();
}

class BMWFactory extends CarFactory {

    @Override
    Car createCar() {
        return new BMW();
    }
}

class TeslaFactory extends CarFactory {

    @Override
    Car createCar() {
        return new Tesla();
    }
}
```

代理设计模式

什么是代理?

房产中介

早上来上班碰见了长风,都没吃饭. 我让长风帮我买个包子

长风是代理人， 我是委托人

静态代理与动态代理的区别是什么?

静态代理与动态代理的核心区别就是静态代理类需要自己写代理类,动态代理不需要

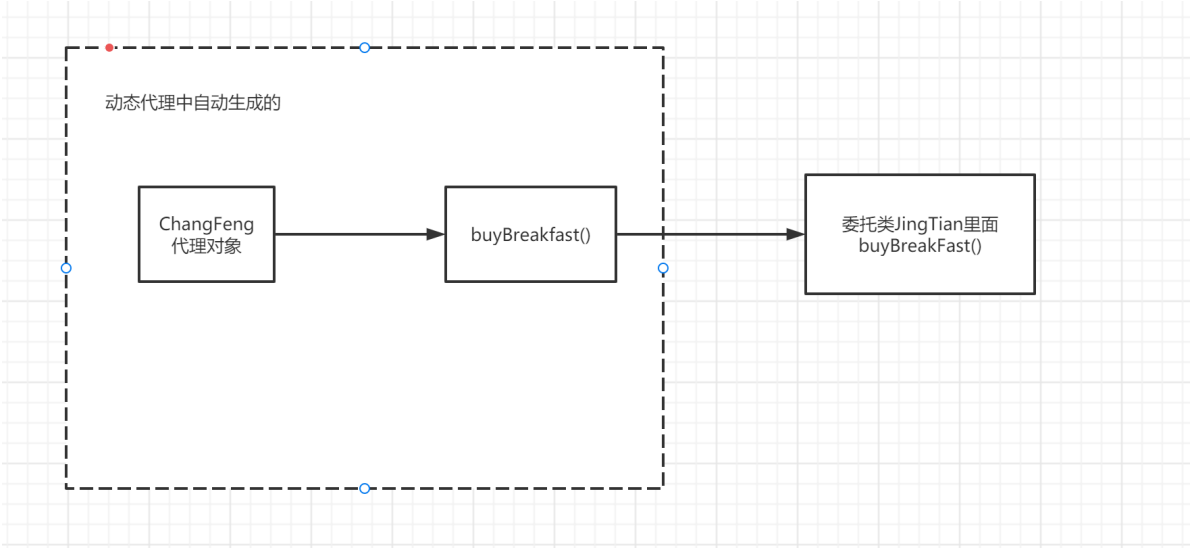
静态代理

方式一:成员变量方式

```
package _26design_pattern.com.cskaoyan._04proxy.bean;

/**
 * @description: 代理类
 * @author: 景天
 * @date: 2022/8/2 16:50
 **/
// 成员变量方式
public class ChangFeng implements BuyFood{
    // 定义成员变量
    JingTian delegate = new JingTian();
    @Override
    public void buyBreakFast() {
        System.out.println("买个鸡蛋");
        // 执行的是委托对象的方法
        delegate.buyBreakFast();
        System.out.println("买个奶茶");

    }
}
```

方式二:继承的方式

```
package _26design_pattern.com.cskaoyan._04proxy.bean;

/**
 * @description: 代理类
 * @author: 景天
 * @date: 2022/8/2 16:50
 **/
// 继承方式
public class ChangFeng2 extends JingTian implements BuyFood{

    @Override
    public void buyBreakFast() {
        System.out.println("买个鸡蛋");
        // 执行的是委托对象的方法
        super.buyBreakFast();
        System.out.println("买个奶茶");

    }
}
```



动态代理

jdk动态代理

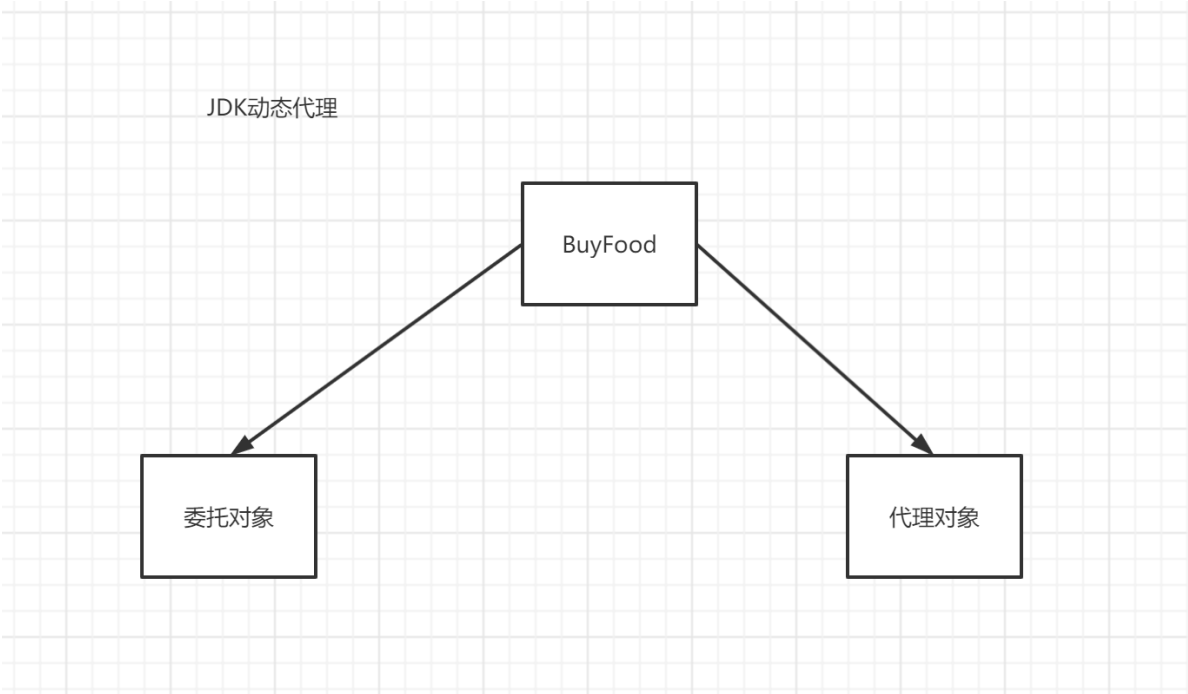
`Proxy` 提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类

static Object	**newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) 返回一个指定接口的代理类实例，该接口可以将方法调用指派到指定的调用处理程序。**
	loader委托类的类加载器, interfaces委托类实现的接口

接口 InvocationHandler

每个代理实例都具有一个关联的调用处理程序。对代理实例调用方法时，将对方法调用进行编码并将其指派到它的调用处理程序的 `invoke` 方法。

Object	**invoke(Object proxy, Method method, Object[] args) 在代理实例上处理方法调用并返回结果。**
	proxy代理对象 method是执行的方法 args:方法参数



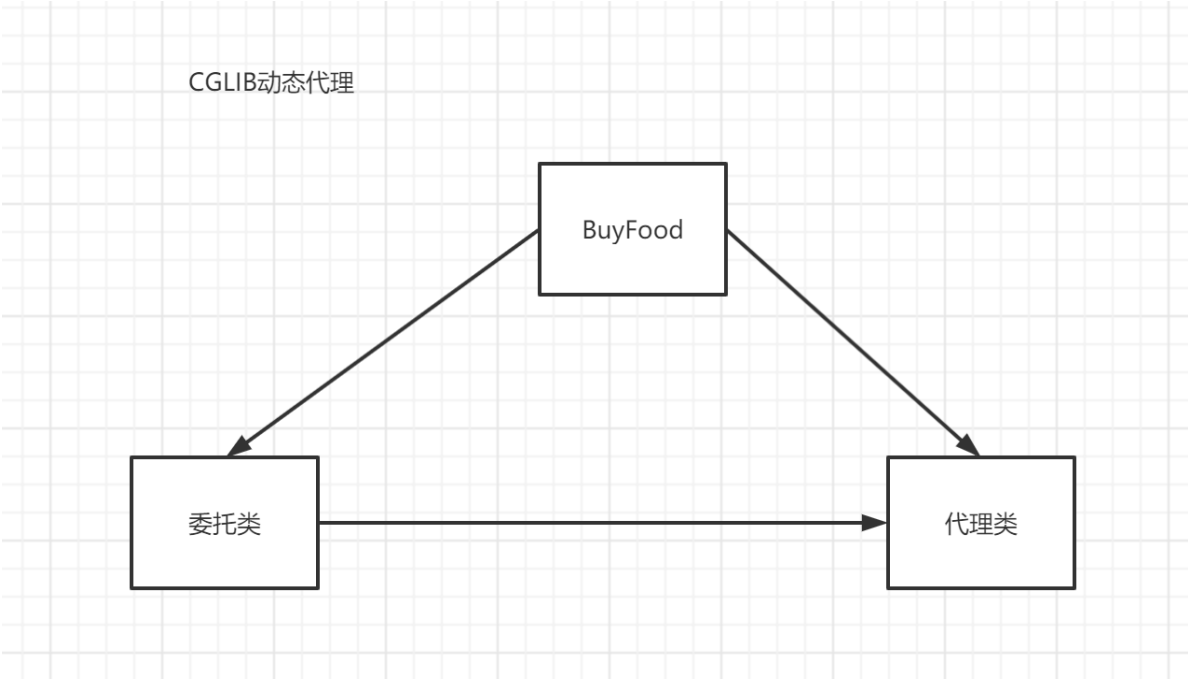

```
package _26design_pattern.com.cskaoyan._04proxy.dynamic_proxy.jdk_proxy;

import _26design_pattern.com.cskaoyan._04proxy.bean.JingTian;

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

/**
 * @description:
 * @author: 景天
 * @date: 2022/8/2 17:19
 **/

public class JingTianInvocationHandler implements InvocationHandler {
    JingTian delegate = new JingTian();
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        // proxy代理对象
        // method是执行的方法
        // args:方法参数
        System.out.println("买个鸡蛋");
        Object invoke = method.invoke(delegate, args);
        System.out.println("买个豆浆");
        return invoke;
    }
}
```

```
package _26design_pattern.com.cskaoyan._04proxy.dynamic_proxy.jdk_proxy;
```

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

/**
 * @description:
 * @author: 景天
 * @date: 2022/8/2 17:31
 **/

public class UserTest {
    public static void main(String[] args) {
        UserImpl user = new UserImpl();

        // 没使用代理
        //System.out.println("verify");
        //user.insert();
        //System.out.println("logging");
        //System.out.println("verify");
        //
        //user.delete();
        //System.out.println("logging");
        //System.out.println("verify");
        //
        //user.update();
        //System.out.println("logging");
        //System.out.println("verify");
        //
        //user.query();
        //System.out.println("logging");


        // 更改需求 要求做操作前
        // 身份验证  verify

        // 操作完后,记录日志  logging

        // 使用代理
        // 创建代理对象
        User proxy = (User)
Proxy.newProxyInstance(UserImpl.class.getClassLoader(),
                UserImpl.class.getInterfaces(),
                new InvocationHandler() {
                    @Override
                    public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                        System.out.println("verify");
                        Object invoke = method.invoke(user, args);
                        System.out.println("logging");

                        return invoke;
                    }
                });
        // 执行代理对象的方法
```

```
        proxy.insert();
        proxy.delete();
        proxy.update();
        proxy.query();

    }
}
```

cglib动态代理

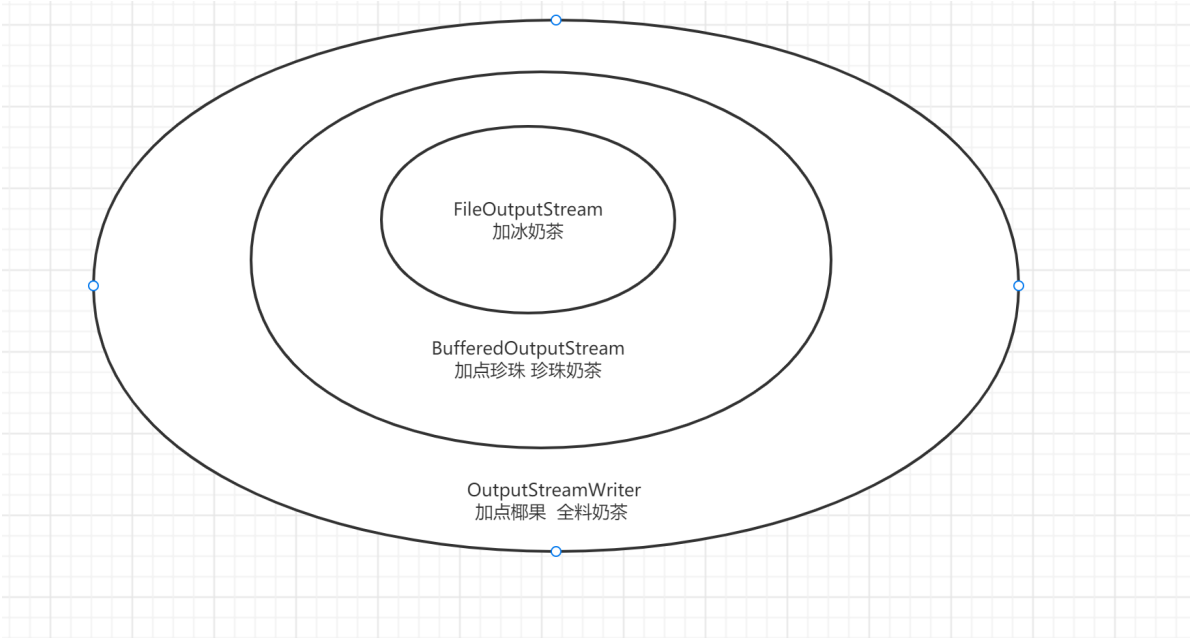


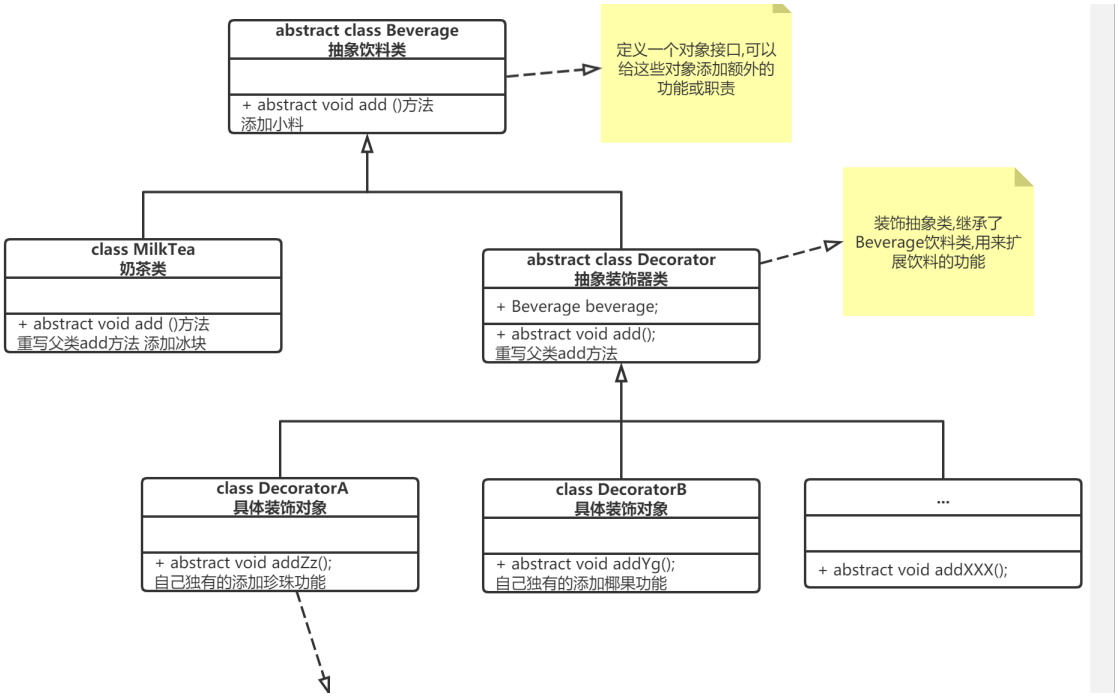

```
package _26design_pattern.com.cskaoyan._04proxy.dynamic_proxy.cglib_proxy;

import _26design_pattern.com.cskaoyan._04proxy.bean.JingTian;
import net.sf.cglib.core.DebuggingClassWriter;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.InvocationHandler;

import java.lang.reflect.Method;

/**
 * @description:
 * @author: 景天
 * @date: 2022/8/2 17:40
 **/

public class CGLIBTest {
    public static void main(String[] args) {
        System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY,
                "D:\\workspace2\\java44th");
        // 创建代理对象
```

```
        JingTian proxy = (JingTian) Enhancer.create(JingTian.class,
JingTian.class.getInterfaces(),
                new InvocationHandler() {
                    @Override
                    public Object invoke(Object o, Method method, Object[]
objects) throws Throwable {
                        JingTian delegate = new JingTian();
                        System.out.println("买个油条");
                        Object invoke = method.invoke(delegate, objects);
                        System.out.println("买个粥");
                        return invoke;
                    }
                });
        // superClass 委托类
        // interfaces 委托类的接口
        // callback invocationHandler
        // 执行代理对象的方法
        proxy.buyBreakFast();
    }
}
```

装饰器设计模式

动态的给对一个对象添加一些额外的职责或功能,IO中缓冲流就用到了这种模式






```
package _26design_pattern.com.cskaoyan._05decorator;

/**
 * @description:
 * @author: 景天
 * @date: 2022/8/2 17:59
 **/

abstract public class Beverage {
    // 加小料
    abstract void add();
}
class MilkTea extends Beverage{

    @Override
    void add() {
        System.out.println("老板,来杯奶茶,加冰!");
    }
}

// 定义装饰器类
abstract class Decorator extends Beverage{
    Beverage beverage;

    public void setBeverage(Beverage beverage) {
        this.beverage = beverage;
    }

    @Override
    void add() {
        beverage.add();
    }
}

// 具体小料类
class DecoratorA extends Decorator {
```

```
    @Override
    void add() {
        super.add();
        addZz();
    }

    private void addZz() {
        System.out.println("再加点珍珠");
    }
}

class DecoratorB extends Decorator {
    @Override
    void add() {
        super.add();
        addYg();
    }

    private void addYg() {
        System.out.println("再加点椰果");
    }
}
```