设计模式七大原则

设计模式的目的

1) 代码重用性 (即：相同功能的代码，不用多次编写)
2) 可读性 (即：编程规范性, 便于其他程序员的阅读和理解)
3) 可扩展性 (即：当需要增加新的功能时，非常的方便，称为可维护)
4) 可靠性 (即：当我们增加新的功能后，对原来的功能没有影响)
5) 使程序呈现高内聚，低耦合的特性

设计模式七大原则

设计模式原则，其实就是程序员在编程时，应当遵守的原则，也是各种设计模 式的基础(即：设计模式为什么这样设计的依据) 设计模式常用的七大原则有:

1) 单一职责原则
2) 接口隔离原则
3) 依赖倒转(倒置)原则
4) 里氏替换原则
5) 开闭原则
6) 迪米特法则
7) 合成复用原则

单一职责原则

理解

让每个类或者每个方法只干一件事,防止出现不符合该类的对象创建或者不符合该方法的对象执行

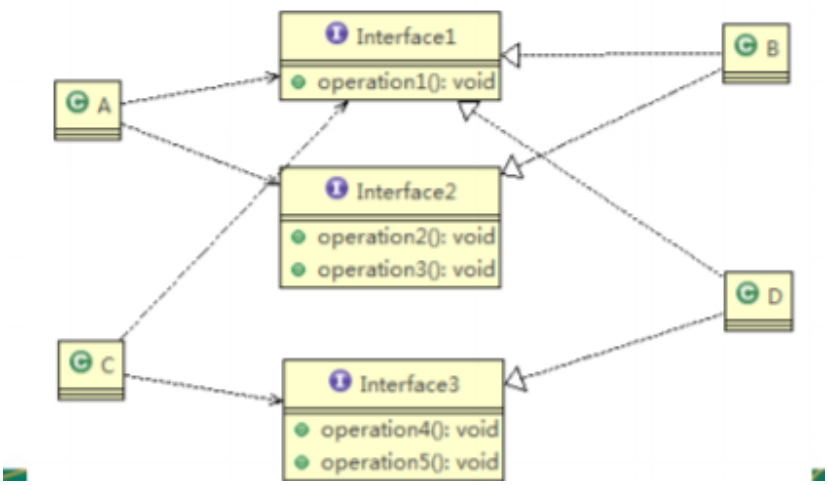
单一职责原则注意事项和细节

1)降低类的复杂度，一个类只负责一项职责。
2) 提高类的可读性，可维护性
3) 降低变更引起的风险
4) 通常情况下，我们应当遵守单一职责原则，只有逻辑足够简单，才可以在代码级违 反单一职责原则；只有类中方法数量足够少，可以在方法级别保持单一职责原则

接口隔离原则

理解

让接口方法分开定义,不要把所有的接口方法创建在同一个接口中,在重写的方法的时候,只需要实现对应的接口就行,防止出现重写不要的方法



依赖倒转原则

1) 高层模块不应该依赖低层模块，二者都应该依赖其抽象

2) 抽象不应该依赖细节，细节应该依赖抽象

3) 依赖倒转(倒置)的中心思想是**面向接口编程**

4) 依赖倒转原则是基于这样的设计理念：相对于细节的多变性，抽象的东西要稳定的 多。以抽象为基础搭建的架构比以 细节为基础的架构要稳定的多。在java中，抽象 指的是接口或抽象类，细节就是具体的实现类

5) 使用接口或抽象类的目的是制定好规范，而不涉及任何具体的操作，把展现细节的 任务交给他们的实现类去完成

依赖倒转原则的注意事项和细节

1) 低层模块尽量都要有抽象类或接口，或者两者都有，程序稳定性更好.

2) 变量的声明类型尽量是抽象类或接口, 这样我们的变量引用和实际对象间，就存在 一个缓冲层，利于程序扩展和优化

3) 继承时遵循里氏替换原则

里氏替换原则

理解

在子类中尽量不要重写父类的方法,两个类A,B向上抽取一个基类,两个类都继承这个基类,如果B想用A中的方法,要在B中实例化一个A类对象,这样就可以调用A类中的方法

基本介绍

在使用继承时，遵循里氏替换原则，在子类中尽量不要重写父类的方法

里氏替换原则告诉我们，继承实际上让两个类耦合性增强了，在适当的情况下，可 以通过聚合，组合，依赖 来解决问题。.

开闭原则

基本介绍

1) 开闭原则（Open Closed Principle）是编程中最基础、最重要的设计原则

2) 一个软件实体如类，模块和函数应该对扩展开放(对提供方)，对修改关闭(对使用 方)。用抽象构建框架，用实现扩展细节。

3) 当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已 有的代码来实现变化。

4) 编程中遵循其它原则，以及使用设计模式的目的就是遵循开闭原则。

理解

对扩展开放(提供方)，对修改关闭(使用方)。即当我们给类增加新功能的时候，尽量不修改代码, 或者尽可能少修改代码,只新增代码,不原来的修改代码

迪米特法则

基本介绍

1) 一个对象应该对其他对象保持最少的了解

2) 类与类关系越密切，耦合度越大

3) 迪米特法则(Demeter Principle)又叫**最少知道原则**，即一个类对自己依赖的类知道的 越少越好。也就是说，对于被依赖的类不管多么复杂，都尽量将逻辑封装在类的内 部。对外除了提供的public方法，不对外泄露任何信息

4) 迪米特法则还有个更简单的定义：只与直接的朋友通信

5) **直接的朋友**：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系， 我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖，关联，组合，聚合 等。其中，我们称出现成员变量，方法参数，方法返回值中的类为直接的朋友，而 出现在局部变量中的类不是直接的朋友。也就是说，陌生的类最好不要以局部变量 的形式出现在类的内部。
我们称出现成员变量，方法参数，方法返回值中的类为直接的朋友，而出现在局部变量中的类不是直接的朋友

迪米特法则注意事项和细节

1) 迪米特法则的核心是降低类之间的耦合

2) 但是注意：由于每个类都减少了不必要的依赖，因此迪米特法则只是要求降低 类间(对象间)耦合关系， 并不是要求完全没有依赖关系

```
//分析 SchoolManager 类的直接朋友类有哪些: Employee(成员变量)、CollegeManager(方法参数)
//CollegeEmployee 不是 直接朋友 而是一个陌生类，它在方法中,是局部变量,这样违背了 迪米特法则
class SchoolManager {
    //返回学校总部的员工
    public List<Employee> getAllEmployee() {
        List<Employee> list = new ArrayList<Employee>();

        for (int i = 0; i < 5; i++) { //这里我们增加了5个员工到 list
            Employee emp = new Employee();
            emp.setId("学校总部员工id= " + i);
            list.add(emp);
        }
        return list;
    }

    //该方法完成输出学校总部和学院员工信息(id)
    void printAllEmployee(CollegeManager sub) {

        //分析问题
        //1. 这里的 CollegeEmployee 不是  SchoolManager的直接朋友,不在 SchoolManager 类的内部
        //2. CollegeEmployee 是以局部变量方式出现在 SchoolManager
        //3. 违反了 迪米特法则

        //获取到学院员工
        List<CollegeEmployee> list1 = sub.getAllEmployee();
        System.out.println("------------学院员工------------");
        for (CollegeEmployee e : list1) {
            System.out.println(e.getId());
        }
        //获取到学校总部员工
        List<Employee> list2 = this.getAllEmployee();
        System.out.println("------------学校总部员工------------");
        for (Employee e : list2) {
            System.out.println(e.getId());
        }
    }
}
```

合成复用原则

基本介绍

```
原则是尽量使用合成/聚合的方式，而不是使用继承
```

设计原则核心思想

```
1）找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代 码混在一起。

2）针对接口编程，而不是针对实现编程。

3）为了交互对象之间的松耦合设计而努力
```

设计模式类型

设计模式分为三种类型，共23种

1) 创建型模式：**单例模式**、抽象工厂模式、原型模式、建造者模式、**工厂模式。**

2) 结构型模式：适配器模式、桥接模式、**装饰模式**、组合模式、外观模式、享元模式、**代理模式。**

3) 行为型模式：模版方法模式、命令模式、访问者模式、迭代器模式、**观察者模式**、中介者模式、备忘录模式、解释器模式（Interpreter模式）、状态模 式、策略模式、职责链模式(责任链模式)。

单例模式

单例设计模式介绍

所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法(静态方法)。

单例设计模式八种方式

1) 饿汉式(静态常量)
2) 饿汉式（静态代码块）
3) 懒汉式(线程不安全)
4) 懒汉式(线程安全，同步方法)
5) 懒汉式(线程安全，同步代码块)
6) 双重检查
7) 静态内部类
8) 枚举

1.饿汉式(静态常量)

步骤如下：

1) 构造器私有化 (防止 new)
2) 类的内部创建对象
3) 向外暴露一个静态的公共方法。getInstance
4) 代码实现

```
//饿汉式(静态变量)
class Singleton {
    //1. 构造器私有化, 外部能new
    private Singleton() {
    }

    //2.本类内部创建对象实例
```

```
    private final static Singleton instance = new Singleton();

    //3. 提供一个公有的静态方法，返回实例对象
    public static Singleton getInstance() {
        return instance;
    }
}
```

优缺点说明：

1) 优点：这种写法比较简单，就是在类装载的时候就完成实例化。**避免了线程同步问题。**

2) 缺点：在类装载的时候就完成实例化，**没有达到Lazy Loading的效果。**如果从始 至终从未使用过这个实例，则会造成 **内存的浪费**

3) 这种方式基于classloder机制避免了多线程的同步问题，不过，instance在类装载时就实例化，在单例模式中大多数都是调用getInstance方法，但是导致类装载的原因有很多种，因此不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化instance就没有达到lazy loading的效果

4) 结论：这种单例模式**可用**，可能造成内存浪费

2.饿汉式（静态代码块）

```
//饿汉式(静态变量)
class Singleton {
    //1. 构造器私有化, 外部能new
    private Singleton() {
    }
    //2.本类内部创建对象实例
    private  static Singleton instance;

    static { // 在静态代码块中，创建单例对象
        instance = new Singleton();
    }

    //3. 提供一个公有的静态方法，返回实例对象
    public static Singleton getInstance() {
        return instance;
    }
}
```

优缺点说明：

1) 这种方式和上面的方式其实类似，只不过将类实例化的过程放在了静态代码块中，也是在类装载的时候，就执行静态 代码块中的代码，初始化类的实例。优缺点和上面是一样的。

2) 结论：这种单例模式**可用**，但是可能造成内存浪费

3.懒汉式(线程不安全)

```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

    //提供一个静态的公有方法，当使用到该方法时，才去创建 instance
    //即懒汉式
    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
```

```
        }
        return instance;
    }
}
```

优缺点说明：

1) 起到了**Lazy Loading**的效果，但是只能在单线程下使用。

2) 如果在多线程下，一个线程进入了if (singleton == null)判断语句块，还未来得及 往下执行，另一个线程也通过了这个 判断语句，这时便会产生多个实例。所以 在多线程环境下不可使用这种方式

3) 结论：在实际开发中，**不要使用**这种方式.

4.懒汉式(线程安全，同步方法)

```
// 懒汉式(线程安全，同步方法)
class Singleton {
    private static Singleton instance;
    private Singleton() {}

    //提供一个静态的公有方法，加入同步处理的代码，解决线程安全问题
    //即懒汉式
    public static synchronized Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

优缺点说明：

1) 解决了线程不安全问题

2) 效率太低了，每个线程在想获得类的实例时候，执行getInstance()方法都要进行 同步。而其实这个方法只执行一次实 例化代码就够了，后面的想获得该类实例， 直接return就行了。方法进行同步效率太低

3) 结论：在实际开发中，**不推荐**使用这种方式

5.懒汉式(同步代码块)

优缺点说明：

1) 这种方式，本意是想对第四种实现方式的改进，因为前面同步方法效率太低， 改为同步产生实例化的的代码块

2) 但是这种同步并不能起到线程同步的作用。跟第3种实现方式遇到的情形一 致，假如一个线程进入了if (singleton == null)判断语句块，还未来得及往下执行， 另一个线程也通过了这个判断语句，这时便会产生多个实例

3) 结论：在实际开发中，**不能使用**这种方式

6.双重检查

```
class Singleton {
    private static volatile Singleton instance;
    private Singleton() {}

    //提供一个静态的公有方法，加入双重检查代码，解决线程安全问题， 同时解决懒加载问题
    //同时保证了效率， 推荐使用
    public static synchronized Singleton getInstance() {
```

```
        if(instance == null) {
            synchronized (Singleton.class) {
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

优缺点说明：

1) Double-Check概念是多线程开发中常使用到的，如代码中所示，我们进行了两次if (singleton == null)检查，这样就 可以保证线程安全了。

2) 这样，实例化代码只用执行一次，后面再次访问时，判断if (singleton == null)， 直接return实例化对象，也避免的反 复进行方法同步.

3) 线程安全；延迟加载；效率较高

4) 结论：在实际开发中，**推荐使用**这种单例设计模式

7.静态内部类

```
// 静态内部类完成， 推荐使用
class Singleton {
    //构造器私有化
    private Singleton() {}

    //写一个静态内部类,该类中有一个静态属性 Singleton
    private static class SingletonInstance {
        private static final Singleton INSTANCE = new Singleton();
    }

    //提供一个静态的公有方法，直接返回SingletonInstance.INSTANCE
    public static synchronized Singleton getInstance() {
        return SingletonInstance.INSTANCE;
    }
}
```

优缺点说明：

1) 这种方式采用了类装载的机制来保证初始化实例时只有一个线程。

2) 静态内部类方式在Singleton类被装载时并不会立即实例化，而是在需要实例化 时，调用getInstance方法，才会装载 SingletonInstance类，从而完成Singleton的 实例化。

3) 类的静态属性只会在第一次加载类的时候初始化，所以在这里，JVM帮助我们 保证了线程的安全性，在类进行初始化 时，别的线程是无法进入的。

4) 优点：避免了线程不安全，利用静态内部类特点实现延迟加载，效率高

5) 结论：**推荐使用.**

8.枚举

```java
//使用枚举，可以实现单例，推荐
enum Singleton {
    INSTANCE; //属性
}

// 测试类
public class SingletonTest08 {
    public static void main(String[] args) {
        Singleton instance = Singleton.INSTANCE;
        Singleton instance2 = Singleton.INSTANCE;
        System.out.println(instance == instance2);
    }
}
```

优缺点说明：

1) 这借助JDK1.5中添加的枚举来实现单例模式。不仅能避免多线程同步问题，而 且还能防止反序列化重新创建新的对象。

2) 这种方式是Effective Java作者Josh Bloch 提倡的方式

3) 结论：**推荐使用**

单例模式注意事项和细节说明

单例模式注意事项和细节说明

1) 单例模式保证了 系统内存中该类只存在一个对象，节省了系统资源，对于一些需要频繁创建销毁的对象，使用单例 模式可以提高系统性能

2) 当想实例化一个单例类的时候，必须要记住使用相应的获取对象的方法，而不是使用new

3) 单例模式使用的场景：需要频繁的进行创建和销毁的对象、创建对象时耗时过多或耗费资源过多(即：重量级对 象)，但又经常用到的对象、工具类对象、频繁访问数据库或文件的对象(比如数据源、session工厂等)

工厂设计模式

简单工厂模式

具体的需求

一个披萨的项目：要便于披萨种类的扩展，要便于维护

1) 披萨的种类很多(比如 GreekPizz、CheesePizz 等)

2) 披萨的制作有 prepare，bake, cut, box

3) 完成披萨店订购功能。

基本介绍

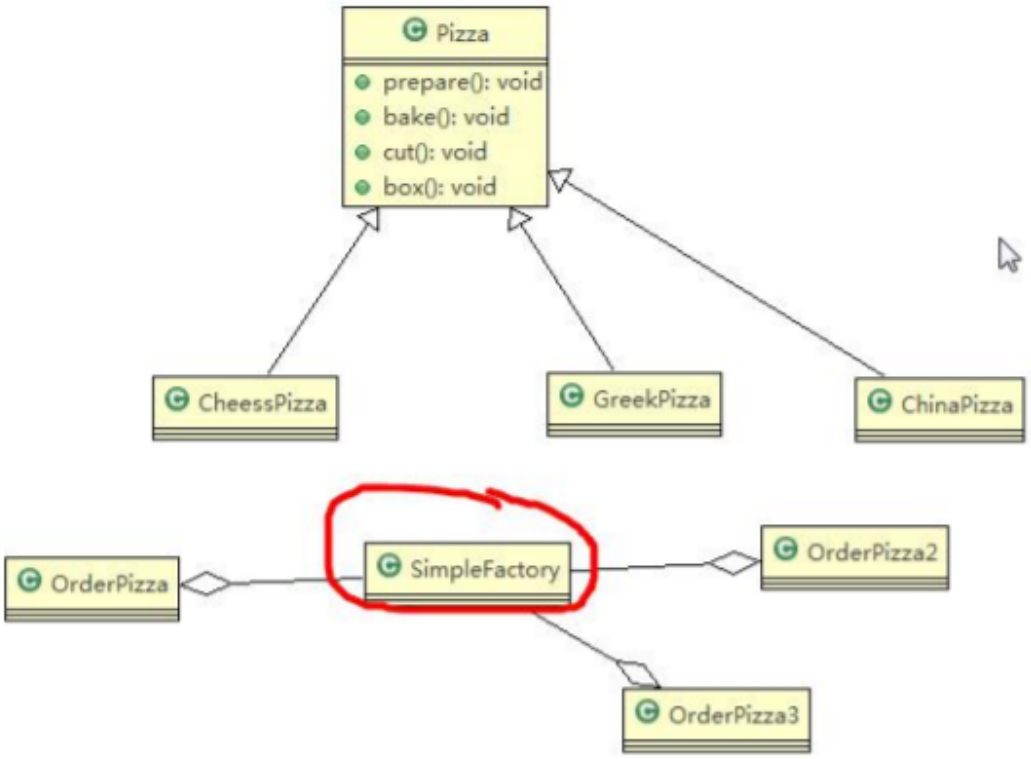
1) 简单工厂模式是属于创建型模式，是工厂模式的一种。简单工厂模式是由一 个工厂对象决定创建出哪一种产品类的实 例。简单工厂模式是工厂模式家族中最简单实用的模式

2) 简单工厂模式：定义了一个创建对象的类，由这个类来封装实例化对象的行为(代码)

3) 在软件开发中，当我们会用到大量的创建某种、某类或者某批对象时，就会 使用到工厂模式.

理解

把对象创建的一些通用的方法,抽取为一个工厂,如果新增对象,或者修改对象行为,只需要对工厂类进行修改,不需要大范围修改每个对象.




```java
//简单工厂类
public class SimpleFactory {

    //根据orderType 返回对应的Pizza 对象
    public Pizza createPizza(String orderType) {

        Pizza pizza = null;

        System.out.println("使用简单工厂模式");
        if (orderType.equals("greek")) {
            pizza = new GreekPizza();
            pizza.setName(" 希腊披萨 ");
        } else if (orderType.equals("cheese")) {
            pizza = new CheesePizza();
            pizza.setName(" 奶酪披萨 ");
        } else if (orderType.equals("pepper")) {
            pizza = new PepperPizza();
            pizza.setName("胡椒披萨");
        }
        return pizza;
    }

    //简单工厂模式 也叫 静态工厂模式

    public static Pizza createPizza2(String orderType) {

        Pizza pizza = null;

        System.out.println("使用简单工厂模式2");
```

```
30          if (orderType.equals("greek")) {
31              pizza = new GreekPizza();
32              pizza.setName(" 希腊披萨 ");
33          } else if (orderType.equals("cheese")) {
34              pizza = new CheesePizza();
35              pizza.setName(" 奶酪披萨 ");
36          } else if (orderType.equals("pepper")) {
37              pizza = new PepperPizza();
38              pizza.setName("胡椒披萨");
39          }
40
41          return pizza;
42      }
43
44  }
```

工厂方法模式

新的需求

披萨项目新的需求：(多一种类型)

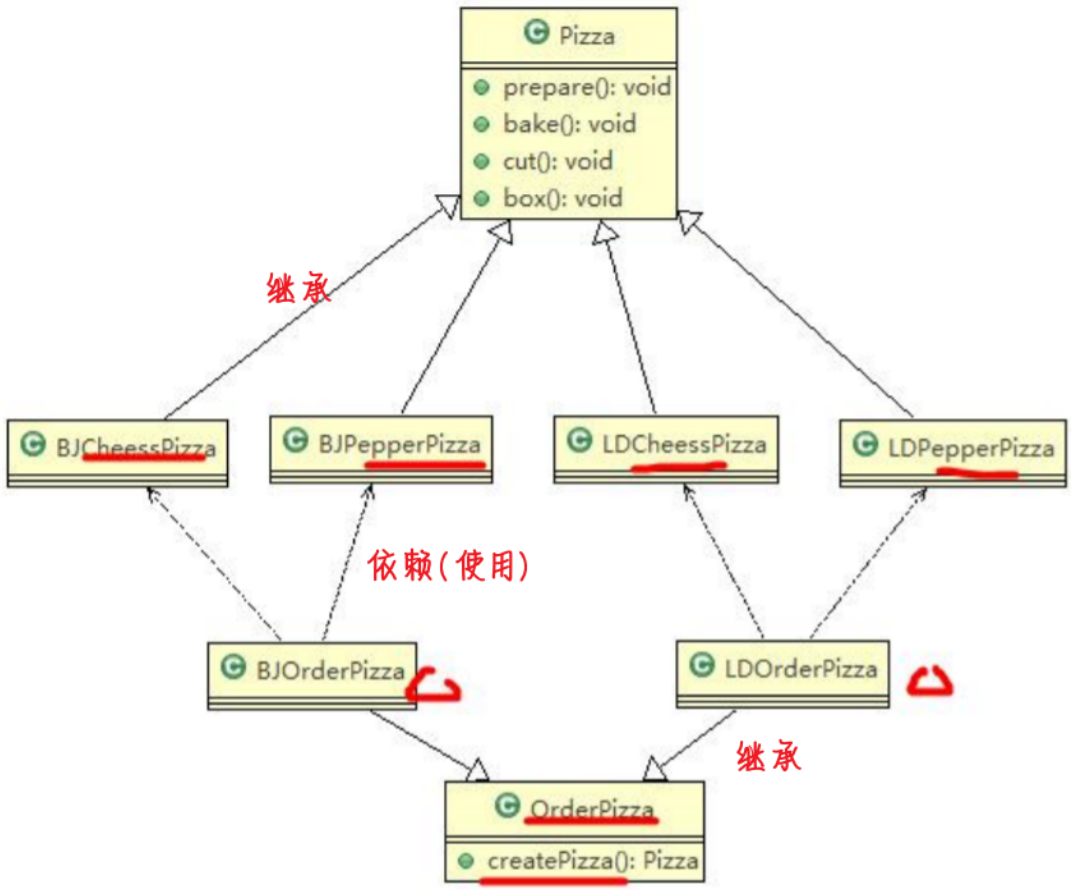
客户在点披萨时，可以点不同口味的披萨，比如 北京的奶酪pizza、 北京的胡椒pizza 或者是伦敦的奶酪pizza、伦敦的胡椒pizza。

工厂方法模式：

定义了一个创建对象的抽象方法，由子类决定要实例化的类。工厂方 法模式将对象的实例化推迟到子类。

理解

有一个总的工厂和一系列子工厂,父类工厂创建一个共有的抽象的方法,工厂子类重写该方法执行类调用执行各个工厂子 类,父类工厂只负责共有的方法和行为,子类工厂进行具体操作.



```java
public abstract class OrderPizza {

    //定义一个抽象方法，createPizza ，让各个工厂子类自己实现
    abstract Pizza createPizza(String orderType);

    // 构造器
    public OrderPizza() {
        Pizza pizza = null;
        String orderType; // 订购披萨的类型
        do {
            orderType = getType();
            pizza = createPizza(orderType); //抽象方法，由工厂子类完成
            //输出pizza 制作过程
            pizza.prepare();
            pizza.bake();
            pizza.cut();
            pizza.box();

        } while (true);
    }

    // 写一个方法，可以获取客户希望订购的披萨种类
    private String getType() {
        try {
            BufferedReader strin = new BufferedReader(new
InputStreamReader(System.in));
            System.out.println("input pizza 种类:");
            String str = strin.readLine();
            return str;
        } catch (IOException e) {
            e.printStackTrace();
            return "";
        }
    }

}
```

抽象工厂模式

基本介绍

1) 抽象工厂模式：定义了一个interface用于创建相关或有依赖关系的对象簇，而无需 指明具体的类

2) 抽象工厂模式可以将简单工厂模式和工厂方法模式进行整合。

3) 从设计层面看，抽象工厂模式就是对简单工厂模式的改进(或者称为进一步的抽象)。

4) 将工厂抽象成两层，**AbsFactory(抽象工厂)** 和 **具体实现的工厂子类。**程序员可以 根据创建对象类型使用对应的工厂 子类。这样将单个的简单工厂类变成了工厂簇， 更利于代码的维护和扩展。

理解

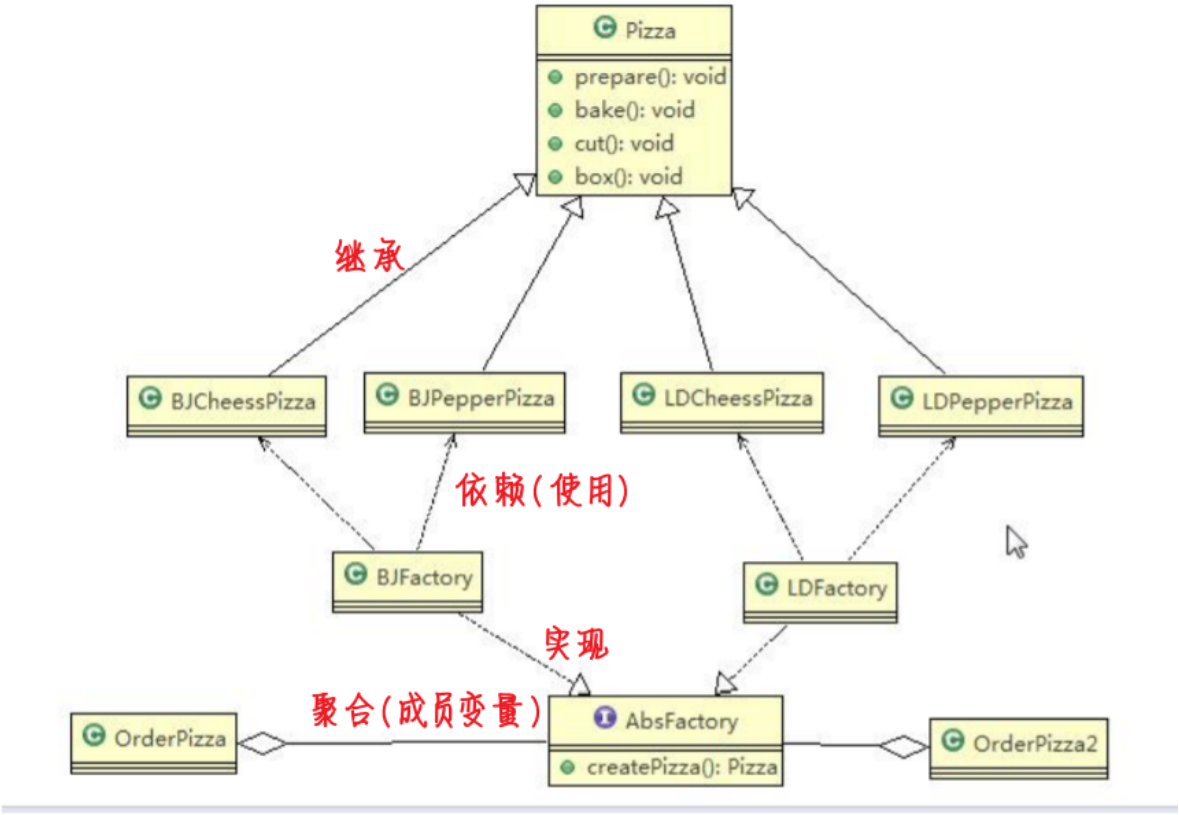
简单工厂:把对象创建的一些通用的方法,抽取为一个工厂,如果新增对象,或者修改对象行为,只需要对工厂类进行修改,不需 要大范围修改每个对象.

工厂方法:有一个总的工厂和一系列子工厂,父类工厂创建一个共有的抽象的方法,工厂子类重写该方法执行类调用执行各个 工厂子类,父类工厂只负责共有的方法和行为,子类工厂进行具体操作.

抽象工厂:总的工厂定义为接口,接口中定义共有的方法,子工厂进行实现该接口,重写方法.(工厂方法)

定义一个工厂类(类似),将抽象父工厂定义为成员变量,在该类的行为方法中调用接口方法,(简单工厂)

执行类需要实例化工厂类,参数为子工厂(父类引用指向子类对象),



工厂模式小结

1) 工厂模式的意义将实例化对象的代码提取出来，放到一个类中统一管理和维护，达到和主项目的 依赖关系的解耦。从而提　　高项目的扩展和维护性。

2) 三种工厂模式 (简单工厂模式、工厂方法模式、抽象工厂模式)

3) 设计模式的依赖抽象原则

*创建对象实例时，不要直接 **new** 类, 而是把这个new 类的动作放在一个工厂的方法中，并返回。有的书上说，变量不要直　　接持有具体类的引用。

*不要让类继承具体类，而是继承抽象类或者是实现interface(接口)

*不要覆盖基类中已经实现的方法。

原型模式

克隆羊问题

现在有一只羊tom，姓名为: tom, 年龄为：1，颜色为：白色，请编写程序创建和tom 羊 属性完全相同的10只羊。

理解

原型模式:用于克隆,创建属性值相同的对象,对象中的成员变量含有引用数据类型,

如果克隆的对象的该成员变量的哈希值相同,则为浅拷贝(克隆对象和原对象的成员变量指向同一个)

如果原对象的引用数据类型的成员变量修改,克隆对象 也会 被修改

如果克隆的对象的该成员变量的哈希值不同,则为深拷贝(克隆对象也克隆了一个成员变量)

如果原对象的引用数据类型的成员变量修改,克隆对象 不会 被修改

扩展性强

浅拷贝:

对象实现 Cloneable 接口,重写 clone()方法,修改 clone() 方法内的返回值

```
public class Client {

    public static void main(String[] args) {
        System.out.println("原型模式完成对象的创建");
        // TODO Auto-generated method stub
        Sheep sheep = new Sheep("tom", 1, "白色");
        sheep.friend = new Sheep("jack", 2, "黑色");

        Sheep sheep2 = (Sheep) sheep.clone(); //克隆
        Sheep sheep3 = (Sheep) sheep.clone(); //克隆
        Sheep sheep4 = (Sheep) sheep.clone(); //克隆
        Sheep sheep5 = (Sheep) sheep.clone(); //克隆

    }

}


public class Sheep implements Cloneable {
    private String name;
    private int age;
    private String color;
    private String address = "蒙古羊";
    public Sheep friend; //是对象, 克隆是会如何处理

    // 省略 setter,getter,toString方法

    //克隆该实例，使用默认的clone方法来完成
    @Override
    protected Object clone()  {

        Sheep sheep = null;
        try {
            sheep = (Sheep)super.clone();
        } catch (Exception e) {
            // TODO: handle exception
            System.out.println(e.getMessage());
        }
        // TODO Auto-generated method stub
        return sheep;
    }

}

```

深拷贝:

对象实现 Cloneable , Serializable 接口

一:代码简单,但是需要对所有的引用数据类型的成员变量一一处理,扩展性差

1.按照浅拷贝方式重写 clone() 方法,此时除了引用数据类型都可以正常克隆了

2.在 clone() 中,单独处理引用数据类型的成员变量,

对象.变量名=(变量类型)变量名.clone();

二:代码复杂,但是新增一个引用数据类型的成员变量时,也不需要重新新增代码

1.创建一个用于深拷贝的方法,自定义

2.创建流对象,ByteArrayOutputStream,ObjectOutputStream,ByteArrayInputStream,ObjectInputStream

3.进行序列化,用 ObjectOutputStream 对对象进行序列化,oos.writeObject(this)

4.反序列化,用 ObjectInputStream 对对象进行反序列化,返回该对象,即为克隆对象

类型 对象名 = (类型)ois.readObject();

```
public class DeepProtoType implements Serializable, Cloneable{

    public String name; //String 属性
    public DeepCloneableTarget deepCloneableTarget;// 引用类型
    public DeepProtoType() {
        super();
    }


    //深拷贝 - 方式 1 使用clone 方法
    @Override
    protected Object clone() throws CloneNotSupportedException {

        DeepProtoType deep = null;
        //这里完成对基本数据类型(属性)和String的克隆
        deep = (DeepProtoType) super.clone();
        //对引用类型的属性，进行单独处理
        deep.deepCloneableTarget  =
(DeepCloneableTarget)deepCloneableTarget.clone();

        // TODO Auto-generated method stub
        return deep;
    }

    //深拷贝 - 方式2 通过对象的序列化实现 (推荐)

    public Object deepClone() {
        //创建流对象
        ByteArrayOutputStream bos = null;
        ObjectOutputStream oos = null;
        ByteArrayInputStream bis = null;
        ObjectInputStream ois = null;

        try {
            //序列化
            bos = new ByteArrayOutputStream();
            oos = new ObjectOutputStream(bos);
            oos.writeObject(this); //当前这个对象以对象流的方式输出

            //反序列化
            bis = new ByteArrayInputStream(bos.toByteArray());
            ois = new ObjectInputStream(bis);
            DeepProtoType copyObj = (DeepProtoType)ois.readObject();

            return copyObj;

        } catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
```

```
            return null;
        } finally {
            //关闭流
            try {
                bos.close();
                oos.close();
                bis.close();
                ois.close();
            } catch (Exception e2) {
                // TODO: handle exception
                System.out.println(e2.getMessage());
            }
        }
    }
}

public class Client {

    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        DeepProtoType p = new DeepProtoType();
        p.name = "宋江";
        p.deepCloneableTarget = new DeepCloneableTarget("大牛", "小牛");

        //方式1 完成深拷贝
//      DeepProtoType p2 = (DeepProtoType) p.clone();


        //方式2 完成深拷贝
        DeepProtoType p2 = (DeepProtoType) p.deepClone();

        System.out.println("p.name=" + p.name + ",p.hashCode=" + p.hashCode() +
",p.deepCloneableTarget=" + p.deepCloneableTarget.hashCode());
        System.out.println("p2.name=" + p.name + ",p2.hashCode=" +
p2.hashCode() +",p2.deepCloneableTarget=" + p2.deepCloneableTarget.hashCode());

    }

}
```

原型模式的注意事项和细节

1) 创建新的对象比较复杂时，可以利用原型模式简化对象的创建过程，同时也能够提 高效率

2) 不用重新初始化对象，而是动态地获得对象运行时的状态

3) 如果原始对象发生变化(增加或者减少属性)，其它克隆对象的也会发生相应的变化， 无需修改代码

4) 在实现深克隆的时候可能需要比较复杂的代码

5) 缺点：需要为每一个类配备一个克隆方法，这对全新的类来说不是很难，但对已有 的类进行改造时，需要修改其源代 码，违背了ocp原则，这点请同学们注意

建造者模式

适配器模式

基本介绍

1) 适配器模式(Adapter Pattern)将某个类的接口转换成客户端期望的另一个接口表 示，主的目的是兼容性，让原本因接口 不匹配不能一起工作的两个类可以协同 工作。其别名为包装器(Wrapper)

2) 适配器模式属于结构型模式

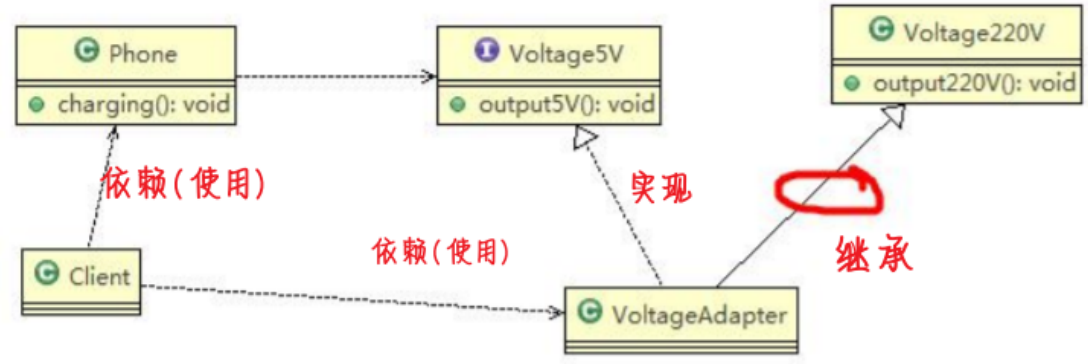
3) 主要分为三类：类适配器模式、对象适配器模式、接口适配器模式

类适配器模式注意事项和细节

1) Java是单继承机制，所以类适配器需要**继承**src类这一点算是一个缺点, 因为这要求dst必须是接口，有一定局限性;

2) src类的方法在Adapter中都会暴露出来，也增加了使用的成本。

3) 由于其继承了src类，所以它可以根据需求重写src类的方法，使得Adapter的灵 活性增强了。



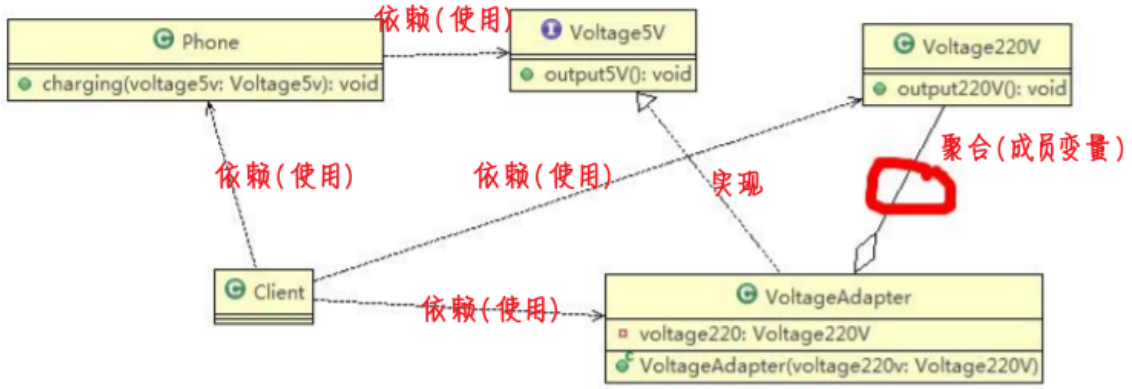

```java
/**
 *
 * 类适配器:分别创建被适配的类(整理前方法)和适配接口(整理后方法),适配器进行继承被适配的类,实现适配接口,
 *              在适配器内部进行重写,操作、整理、适配,适配出需要的形式。测试类直接调用适配器即可,
 *              不用管被适配的类和适配接口
 *
 */

//适配器类
public class VoltageAdapter extends Voltage220V implements IVoltage5V {

    @Override
    public int output5V() {
        // TODO Auto-generated method stub
        //获取到220V电压
        int srcV = output220V();
        int dstV = srcV / 44 ; //转成 5v
        return dstV;
    }

}

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println(" === 类适配器模式 ====");
        Phone phone = new Phone();
        phone.charging(new VoltageAdapter());
    }
```

```
31
32  }
33
```

对象适配器模式注意事项和细节

1) 对象适配器和类适配器其实算是同一种思想，只不过实现方式不同。 根**据合成复用原则，使用组合替代继承**， 所以它解 决了类适配器必须继承src的 局限性问题，也不再要求dst必须是接口。

2) 使用成本更低，更灵活。




```
1   /**
2    *
3    * 对象适配器:分别创建被适配的类(整理前方法)和适配接口(整理后方法),适配器只需要实现适配接口即可,
4                   在类内引用被适配类的对象,适配器和被适配类使用聚合关系代替继承关系(合成复用原则)
5                   通过构造器，传入一个被适配器类的实例,然后重写方法,
6                   测试类直接调用适配器,需要传入被适配类对选对象
7    */
8
9   //适配器类
10  public class VoltageAdapter  implements IVoltage5V {
11
12      private Voltage220V voltage220V; // 关联关系-聚合
13
14
15      //通过构造器，传入一个 Voltage220V 实例
16      public VoltageAdapter(Voltage220V voltage220v) {
17          this.voltage220V = voltage220v;
18      }
19
20      @Override
21      public int output5V() {
22
23          int dst = 0;
24          if(null != voltage220V) {
25              int src = voltage220V.output220V();//获取220V 电压
26              System.out.println("使用对象适配器，进行适配~~");
27              dst = src / 44;
28              System.out.println("适配完成，输出的电压为=" + dst);
29          }
30          return dst;
31      }
```

```
}


public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println(" === 对象适配器模式 ====");
        Phone phone = new Phone();
        phone.charging(new VoltageAdapter(new Voltage220V()));
    }

}
```

接口适配器模式介绍

1) 一些书籍称为：适配器模式(Default Adapter Pattern)或缺省适配器模式。

2) 当不需要全部实现接口提供的方法时，可先设计一个抽象类实现接口，并为该接 口中每个方法提供一个默认实现（空方 法），那么该抽象类的子类可有选择地覆 盖父类的某些方法来实现需求

3) 适用于一个接口不想使用其所有的方法的情况。

```
/**
 *
 * 接口适配器:缺省适配器模式。定义一个接口,定义一个抽象类当适配器,实现接口,重新所有方法,方法空实现.
                测试类只需要创建一个适配器对象,重写自己需要的接口即可
 *
 */

//在AbsAdapter 我们将 Interface4 的方法进行默认实现
public abstract class AbsAdapter implements Interface4 {

    //默认实现
    public void m1() {

    }

    public void m2() {

    }

}

public class Client {
    public static void main(String[] args) {

        AbsAdapter absAdapter = new AbsAdapter() {
            //只需要去覆盖我们 需要使用 接口方法
            @Override
            public void m1() {
                // TODO Auto-generated method stub
                System.out.println("使用了m1的方法");
            }
        };
    absAdapter.m1();
    }
```

```
35 }
```

适配器模式的注意事项和细节

1) 三种命名方式，是根据 src是以怎样的形式给到Adapter（在Adapter里的形式）来 命名的。

2) 类适配器：以类给到，在Adapter里，就是将src当做类，继承 对象适配器：以对象给到，在Adapter里，将src作为一个 对象，持有 接口适配器：以接口给到，在Adapter里，将src作为一个接口，实现

3) Adapter模式最大的作用还是将原本不兼容的接口融合在一起工作。

4) 实际开发中，实现起来不拘泥于我们讲解的三种经典形式

桥接模式

装饰者设计模式

星巴克咖啡订单项目（咖啡馆）：

1) 咖啡种类/单品咖啡：Espresso(意大利浓咖啡)、ShortBlack、LongBlack(美式 咖啡)、Decaf(无因咖啡)

2) 调料：Milk、Soy(豆浆)、Chocolate

3) 要求在扩展新的咖啡种类时，具有良好的扩展性、改动方便、维护方便

4) 使用OO的来计算不同种类咖啡的费用: 客户可以点单品咖啡，也可以单品咖 啡+调料组合。

理解

装饰者： 有一个抽象基类,定义属性和行为,一级父类

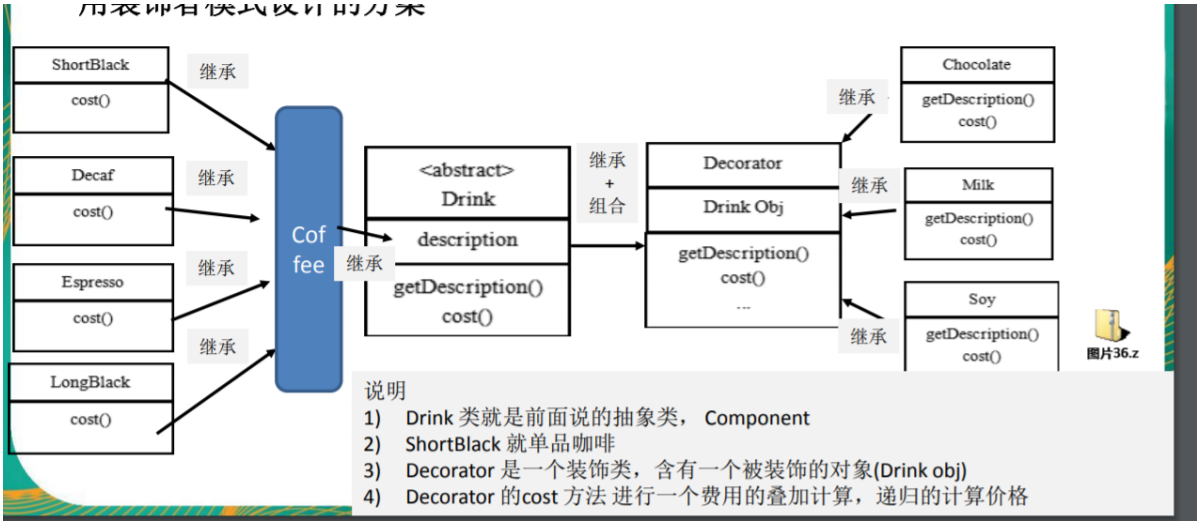
一个单体对象类,继承基类,二级父类

一个装饰类,继承基类,并且在装饰类创建基类对象,有参构造的参数为基类,二级父类

三级子类分别继承单体对象类和装饰类

测试类新建一个单体对象的三级子类,父类引用指向子类对象,将该对象当成装饰子类的构造参数

Milk m = new Milk(new LongBlack());




```
1 /**
2  *
3  * 装饰者：    有一个抽象基类,定义属性和行为,一级父类
```

```
            一个单体对象类,继承基类,二级父类
            一个装饰类,继承基类,并且在装饰类创建基类对象,有参构造的参数为基类,二级父类
            三级子类分别继承单体对象类和装饰类
            测试类新建一个单体对象的三级子类,父类引用指向子类对象,将该对象当成装饰子类的构造参数
                Milk m = new Milk(new LongBlack());
 *
 */

public class Decorator extends Drink {
    private Drink obj;

    public Decorator(Drink obj) { //组合
        // TODO Auto-generated constructor stub
        this.obj = obj;
    }

    @Override
    public float cost() {
        // TODO Auto-generated method stub
        // getPrice 自己价格
        return super.getPrice() + obj.cost();
    }

    @Override
    public String getDes() {
        // TODO Auto-generated method stub
        // obj.getDes() 输出被装饰者的信息
        return des + " " + getPrice() + " && " + obj.getDes();
    }



}


public class CoffeeBar {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // 装饰者模式下的订单：2份巧克力+一份牛奶的LongBlack

        // 1. 点一份 LongBlack
        Drink order = new LongBlack();
        System.out.println("费用1=" + order.cost());
        System.out.println("描述=" + order.getDes());

        // 2. order 加入一份牛奶
        order = new Milk(order);

        System.out.println("order 加入一份牛奶 费用 =" + order.cost());
        System.out.println("order 加入一份牛奶 描述 = " + order.getDes());


        // 3. order 加入一份巧克力

        order = new Chocolate(order);

```

```
        System.out.println("order 加入一份牛奶 加入一份巧克力  费用 =" + order.cost());
        System.out.println("order 加入一份牛奶 加入一份巧克力 描述 = " + order.getDes());

        // 3. order 加入一份巧克力

        order = new Chocolate(order);

        System.out.println("order 加入一份牛奶 加入2份巧克力   费用 =" + order.cost());
        System.out.println("order 加入一份牛奶 加入2份巧克力 描述 = " + order.getDes());

        System.out.println("============================");

        Drink order2 = new DeCaf();

        System.out.println("order2 无因咖啡  费用 =" + order2.cost());
        System.out.println("order2 无因咖啡 描述 = " + order2.getDes());

        order2 = new Milk(order2);

        System.out.println("order2 无因咖啡 加入一份牛奶  费用 =" + order2.cost());
        System.out.println("order2 无因咖啡 加入一份牛奶 描述 = " + order2.getDes());


    }

}
```

装饰者模式定义

装饰者模式:动态的将新功能附加到对象上。在对象功能扩展方面，它比继承更有弹性，装饰者模式也体现了开闭原则(ocp)

组合模式

外观模式

享元模式

代理模式

代理模式的基本介绍

1) 代理模式：为一个对象提供一个替身，以控制对这个对象的访问。即通过代理 对象访问目标对象.这样做的好处是:可以在 目标对象实现的基础上,增强额外的 功能操作,即**扩展目标对象的功能。**

2) 被代理的对象可以是远程对象、创建开销大的对象或需要安全控制的对象

3) 代理模式有不同的形式, 主要有三种 静态代理、动态代理 (JDK代理、接口代理)和 Cglib代理 (可以在内存动态的创建对 象，而不需要实现接口， 他是属于 动态代理的范畴) 。

4) 代理模式示意图

静态代理模式:

静态代理在使用时,需要定义接口或者父类,被代理对象(即目标对象)与代理对象一 起实现相同的接口或者是继承相同父类

理解

1.创建一个接口

2.创建一个类,实现接口,重写接口中的方法,当目标对象(被代理对象)

3.创建一个类实现接口,当代理对象,创建接口的成员对象,重写方法

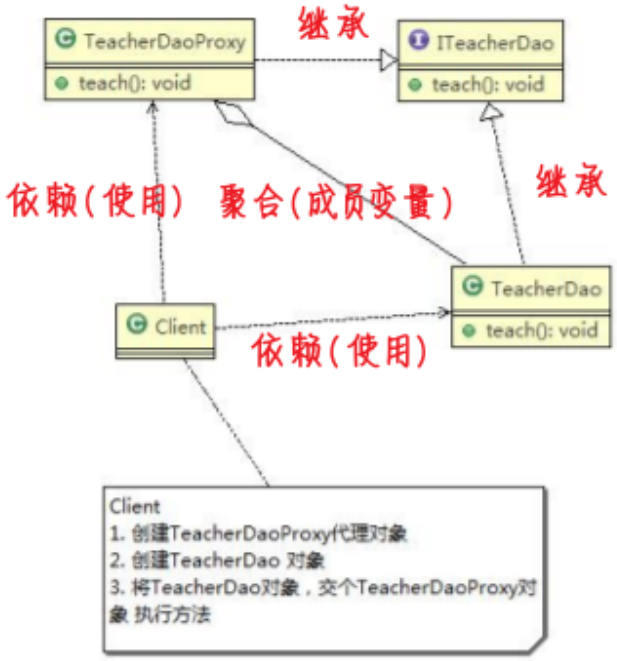
4.测试类创建被代理对象、代理对象,将被代理对象传递给代理对象

静态代理优缺点

1) 优点：在不修改目标对象的功能前提下, 能通过代理对象对目标功能扩展

2) 缺点：因为代理对象需要与目标对象实现一样的接口,所以会有很多代理类

3) 一旦接口增加方法,目标对象与代理对象都要维护




```java
//代理对象,静态代理
public class TeacherDaoProxy implements ITeacherDao{

    private ITeacherDao target; // 目标对象，通过接口来聚合

    //构造器
    public TeacherDaoProxy(ITeacherDao target) {
        this.target = target;
    }

    @Override
    public void teach() {
        // TODO Auto-generated method stub
        System.out.println("开始代理  完成某些操作。。。。。 ");//方法
        target.teach();
        System.out.println("提交。。。。。");//方法
    }
}


```

```java
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //创建目标对象(被代理对象)
        TeacherDao teacherDao = new TeacherDao();

        //创建代理对象，同时将被代理对象传递给代理对象
        TeacherDaoProxy teacherDaoProxy = new TeacherDaoProxy(teacherDao);

        //通过代理对象，调用到被代理对象的方法
        //即：执行的是代理对象的方法，代理对象再去调用目标对象的方法
        teacherDaoProxy.teach();
    }

}
```

动态代理

动态代理模式的基本介绍

1) 代理对象,不需要实现接口，但是目标对象要实现接口，否则不能用动态代理
2) 代理对象的生成，是利用JDK的API，动态的在内存中构建代理对象
3) 动态代理也叫做：JDK代理、接口代理

理解

1.创建一个接口
2.创建一个类,实现接口,重写接口中的方法,当目标对象(被代理对象)
3.创建一个类,创建一个 Object 对象,创建一个方法,返回
public static Object newProxyInstance(ClassLoader loader,
Class<?>[] interfaces,
InvocationHandler h)
① ClassLoader loader:指定当前目标对象使用的类加载器, 获取加载器的方法固定
② Class<?>[] interfaces:目标对象实现的接口类型，使用泛型方法确认类型
③ InvocationHandler h:事情处理，执行目标对象的方法时，会触发事情处理器方法, 会把当前执行的目标对象方法作为参数传 入; 用来增强内容
重写public Object invoke(Object proxy, Method method, Object[] args)
4.测试类创建被代理对象,调用代理对象的方法创建代理对象

```java
public class ProxyFactory {

    //维护一个目标对象, Object
    private Object target;

    //构造器,对 target 进行初始化
    public ProxyFactory(Object target) {

        this.target = target;
    }

    //给目标对象 生成一个代理对象
    public Object getProxyInstance() {

        //说明
        /*
         *  public static Object newProxyInstance(ClassLoader loader,
                                                  Class<?>[] interfaces,
```

```
                                          InvocationHandler h)

            //1. ClassLoader loader ： 指定当前目标对象使用的类加载器，获取加载器的方法固定
            //2. Class<?>[] interfaces: 目标对象实现的接口类型，使用泛型方法确认类型
            //3. InvocationHandler h : 事情处理，执行目标对象的方法时，会触发事情处理器方法，会把当前执行的目标对象方                                                法作为参数传入
         */
        return Proxy.newProxyInstance(
                target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),
                new InvocationHandler() {

                    @Override
                    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                        // TODO Auto-generated method stub
                        System.out.println("JDK代理开始~~");
                        //反射机制调用目标对象的方法,方法返回值
                        Object returnVal = method.invoke(target, args);
                        System.out.println("JDK代理提交");
                        System.out.println("returnVal="+returnVal);
                        return returnVal;
                    }
                });
    }

}

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //创建目标对象
        ITeacherDao target = new TeacherDao();

        //给目标对象，创建代理对象，可以转成 ITeacherDao
        ITeacherDao proxyInstance = (ITeacherDao)new ProxyFactory(target).getProxyInstance();

        // proxyInstance=class com.sun.proxy.$Proxy0 内存中动态生成了代理对象
        System.out.println("proxyInstance=" + proxyInstance.getClass());

        //通过代理对象，调用目标对象的方法
        proxyInstance.teach();

        System.out.println("================");

        proxyInstance.sayHello(" tom ");
    }

}
```

```
JDK代理开始~~
老师授课中....
JDK代理提交
returnVal=null
```

```
JDK代理开始~~
hello  tom
JDK代理提交
returnVal=sayHello
```

Cglib代理

Cglib代理模式的基本介绍

1) 静态代理和JDK代理模式都要求目标对象是实现一个接口,但是有时候目标对象只是一个单独的对象,并没有实现任何 的接口,这个时候可使用目标对象子类来实现代理----这就是Cglib代理

2) Cglib代理也叫作子类代理,它是在内存中构建一个子类对象从而实现对目标对象功 能扩展, 有些书也将Cglib代理归属 到动态代理。

3) 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，实现方法拦截

4) 在AOP编程中如何选择代理模式： 1. 目标对象需要实现接口，用JDK代理 2. 目标对象不需要实现接口，用Cglib代理

5) Cglib包的底层是通过使用字节码处理框架ASM来转换字节码并生成新的类

理解

1.创建一个目标对象

2.创建一个代理对象,实现 MethodInterceptor 接口

创建获取代理对象方法,

public Object getProxyInstance() {
//1. 创建一个工具类
Enhancer enhancer = new Enhancer();
//2. 设置父类
enhancer.setSuperclass(target.getClass());
//3. 设置回调函数
enhancer.setCallback(this);
//4. 创建子类对象，即代理对象
return enhancer.create();
}

重写 intercept() 方法,增强代理

Object intercept(Object var1, Method var2, Object[] var3, MethodProxy var4) throws Throwable;

```
1
2  public class ProxyFactory implements MethodInterceptor {
3
4      //维护一个目标对象
5      private Object target;
6
7      //构造器，传入一个被代理的对象
8      public ProxyFactory(Object target) {
9          this.target = target;
10     }
11
12     //返回一个代理对象:  是 target 对象的代理对象
13     public Object getProxyInstance() {
14         //1. 创建一个工具类
15         Enhancer enhancer = new Enhancer();
16         //2. 设置父类
17         enhancer.setSuperclass(target.getClass());
18         //3. 设置回调函数
```

```
        enhancer.setCallback(this);
        //4. 创建子类对象，即代理对象
        return enhancer.create();

    }


    //重写  intercept 方法，会调用目标对象的方法
    @Override
    public Object intercept(Object arg0, Method method, Object[] args,
MethodProxy arg3) throws Throwable {
        // TODO Auto-generated method stub
        System.out.println("Cglib代理模式 ~~ 开始");
        Object returnVal = method.invoke(target, args);
        System.out.println("Cglib代理模式 ~~ 提交");
        return returnVal;
    }

}


public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //创建目标对象
        TeacherDao target = new TeacherDao();
        //获取到代理对象，并且将目标对象传递给代理对象
        TeacherDao proxyInstance = (TeacherDao)new
ProxyFactory(target).getProxyInstance();

        //执行代理对象的方法，触发intecept 方法，从而实现  对目标对象的调用
        String res = proxyInstance.teach();
        System.out.println("res=" + res);
    }

}

```