##简单/静态工厂模式

> 创建一个工厂类，对实现同一接口的各种类进行对象的创建，而使用该对象的类则不负责对象的创建。提供了最佳的创建对象方式. **这里的接口，对于产品类来说实现时使用的抽象类，因为是定义实体类；而工厂则更多使用接口类，因为只提供一个创建对象的方法** 如：

```
//接口outer
abstract class Outer{
    public out();
}

//实现Outer接口
class Printer extends Outer{
    //实现方法
    public out(){.........}
}

//使用类
class Computer{
    //设计为私有的，封装
    private Outer out;
    //使用类不负责创建对象，接受工厂创建的对象
    public Computer(Outer out){
        this.out = out;
    }
    public void print(){
        out.out();
    }
}

//工厂类
class OutputFactory{
    //**返回的是接口类型！**
    public Outer getOutput(){
        return new Printer();
    }
}

//测试
public class test{
    public static void main(String[] args){
        //先创建工厂
        OutputFactory s = new OutputFactory();
        //创建Computer类的实例,使用工厂传入创建的类，也是一次封装，以后不用修改这个代码
        Computer o = new Computer(s.getOutput);
        o.out;
    }
}
```

> 由上述代码可看出，首先接口和实现接口的类，然后在使用类中调用接口的引用都是常见使用多态的思考方式；但不同的是，要求使用类的构造器为带接口引用参数的，用来初始化接口引用，创建对象的工作交给了工厂类。而在创建使用类的对象时，使用工厂的方法，这个方法实际上又是一次封装。前面用多态，在使用类定义方法时封装了**接口方法的调用**，后者是在创建使用类对象中，**封装了接口的对象的创建**。所以**关于使用类的方法定义、创建该类对象的代码都不需要再修改**。以后只需要修改工厂类中的代码就可以了。 最基础的多态是在使用类定义中直接使用父类（接口）的引用，实际创建时传入子类引用，这样如果子类需要替换，则需要**修改所有创建子类对象的代码**，而简单工厂代码，则解决了这个问题。 缺点：不符合开-闭原则；一旦添加新产品会修改代码逻辑。 扩展

```
//一个工厂类创建所有对象，使用一些逻辑（if-else/switch）
//是上面的创建对象方法的扩展
public static Animal createAnimal(String type) {
    if ("dog".equals(type)) {
        return new Dog();
    } else if ("cat".equals(type)) {
        return new Cat();
    } else {
        return null;
    }
}
```

##工厂方法模式

> 工厂也有一个抽象接口，具体工厂生产一类具体的实例对象； 概述：**一个抽象产品类，可以派生出多个具体产品类。一个抽象工厂类，可以派生出多个具体工厂类**（即两个都有接口）。每个具体工厂类只能创建一个具体产品类的实例。

```
//宠物工厂接口
public interface AnimalFactory {
    // 可以获取任何的宠物
    Animal createAnimal();
}

//猫工厂
public class CatFactory implements AnimalFactory {
    @Override
    // 创建猫
    public Animal createAnimal() {
        return new Cat();
    }

}

//狗工厂
public class DogFactory implements AnimalFactory {
    // 创建狗
    @Override
    public Animal createAnimal() {
        return new Dog();
```

```
    }

}
```

> 优点：
>
> 1. 客户端不需要在负责对象的创建,明确了各个类的职责
> 2. 如果有新的对象增加,只需要增加一个具体的类和具体的工厂类即可
> 3. 不会影响已有的代码,后期维护容易,增强系统的扩展性

##抽象工厂模式

> 解决的问题：每个工厂只能创建一类产品（工厂方法模式） **抽象工厂模式与工厂方法模式最大的区别：抽象工厂中每个工厂可以创建多种类的产品；而工厂方法每个工厂只能创建一类 主要对象概念** 抽象工厂：描述具体工厂的公共接口 具体工厂：描述具体工厂，创建产品的实例，供外界调用 抽象产品族：描述抽象产品的公共接口 抽象产品：描述具体产品的公共接口 具体产品：具体产品 产品族：一类产品。 产品等级：产品的继承结构

> 举例：“假设”有各类的自动售卖机（抽象工厂），可以出售各类食品（**抽象产品族**）。有饮料、零食（**抽象产品，两个产品族**），比如常见的零食售卖机（具体工厂），出售矿泉水与面包（**具体产品**）。 **个人理解**：抽象工厂会对产品进行更多一层的抽象（具体表现为**抽象类**）；最顶层的抽象（食品），代表了工厂生产的是相关类型的产品；第二层抽象（饮料、零食），是指工厂生产多种类型产品。所以**抽象工厂模式是创建一组相关或相互依赖的对象（第一层抽象）** 实体类：

```
/**
 * @ Product.java
 * 抽象产品族 (食品)**第一层抽象**
 */
abstract class Product {
    //产品介绍
    abstract void intro();
}

/**
 * @ ProductA.java
 * 抽象产品  (饮料)**第二层抽象**
 */
abstract class ProductA extends Product{
    @Override
    abstract void intro();
}

/**
 * @ ProductB.java
 * 抽象产品  (零食)
 */
abstract class ProductB extends Product{
    @Override
    abstract void intro();
}
```

```
/**
 * @ ProductAa.java
 * 具体产品  (矿泉水)
 */
public  class ProductAa extends ProductA{
    @Override
    void intro() {
        System.out.println("矿泉水");
    }
}

/**
 * @ ProductBb.java
 * 抽象产品  (面包)
 */
public class ProductBb extends ProductB{
    @Override
    void intro() {
        System.out.println("面包");
    }
}
```

工厂：

```
/**
 * @ Factory.java
 * 抽象工厂
 */
abstract class Factory {
    //生产饮料
    abstract Product getProductA();
    //生产零食
    abstract Product getProductB();
}

/**
 * @ FactoryA.java
 * 具体工厂A
 * 负责具体的A类产品生产
 */
public class FactoryA extends Factory{
    @Override
    Product getProductA() {
        //生产矿泉水
        return new ProductAa();
    }
    @Override
    Product getProductB() {
        //生产面包
        return new ProductBb();
```

```
    }
}
```

> 由上面代码可见，抽象工厂的工厂接口是规划好生产**所有产品**的。 优点：降低耦合； 符合开-闭原则：因为不会改变代码结构（如if-else/switch这种），只是添加代码量 符合单一职责原则、不使用静态工厂方法，可以形成基于继承的等级结构。 缺点：难以扩展新种类产品

对比：

> 工厂方法模式解决了简单工厂模式的"开放 - 关闭原则 抽象工厂模式解决了工厂方法模式一个具体工厂只能创建一类产品 参考：

1. https://www.jianshu.com/p/d951ac56136e Java 设计模式——工厂模式
2. https://mp.weixin.qq.com/s?__biz=Mzl4Njg5MDA5NA==&mid=2247484243&idx=1&sn=972cbe6cdb578256e4d4771e7ca25de3&chksm=ebd74252dca0cb44419903758e8ca52d9ab287562f80be9365e305d6dcc2deaa45b40f9fd2e9&scene=21#wechat_redirect

命令模式

> 处理的需求为：要求某个方法对输入参数数据的处理可以根据实际调用的不同而发生改变。 基本解决思路：**处理行为即为方法**，要求处理行为发生变化则把**方法作为参数**传入方法中；先定一个接口，接口中只有一个抽象处理方法，其次在处理类中定义一个方法，其中某一参数为该接口，具体实例中则传入实现该接口的实例。 **实现处理类与处理行为的分离** 如：

```
//封装处理行为的接口
public interface  Command{
    public String s(Sting[] ag);
}

//处理类
public class Process{
    //传入接口对象
    public void s2(String[] ag,Command c){
        c.s(ag);
    }
}
```