转载请注明链接：https://blog.csdn.net/feather_wch/article/details/78500923

> 本文以面试题的形式归纳总结，可以直接学习和背诵。
> 鸣谢: 本文基础部分归纳总结自《Head First 设计模式》

如果有帮助，请点个赞，万分感谢！

工厂模式详解

版本：2018/8/24-1(13：00)

介绍

1、工厂模式是什么?

> 1. 工厂(factory)处理创建对象的细节。
> 2. 简单工厂 并不是一种设计模式，而是一种编程习惯。
> 3. 工厂模式分为 工厂方法模式 和 抽象工厂模式 。

2、什么情况下该使用工厂模式?

> 1. 不能预见需要创建哪种类的实例。

定义

3、工厂模式的定义?

> 定义了一个创建对象的接口，由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类。

4、子类决定实例化的类是什么意思?

> 1. 这里的 决定 并不是指子类在运行时决定，
> 2. 而是在编写创建者(Creator)时不需要知道创建哪个产品，而是在选择使用哪个子类时，就决定了实际创建的子类是哪个

5、什么是参数化工厂方法?

> 由参数决定创建哪种产品的工厂，被称为“参数化工厂方法”

设计原则

6、什么是依赖倒置原则?

> 1. 依赖倒置原则(Dependency Inversion Principle)
> 2. 无论是高层组件还是底层组件要依赖抽象，不要依赖具体类
> 3. 核心原则：在于面向接口编程，目的在于“解耦”

7、传统自顶而下的设计思想是什么?

> 1. 高层组件，调用，底层组件。如宠物商店(高层)去获取到狗(底层组件)，然后售卖。
> 2. 缺点：在于如果 底层组件 改变， 高层组件 也一定会被影响。比如狗突变成了龙，宠物商店必然需要改变(太依赖于具体实现)。

8、依赖倒置原则是如何做的?

> 1. 打破传统自顶而下的流程。不再让高层组件依赖底层组件。
> 2. 而是从底部的具体实现去思考，将这些具体的底层组件，去寻找共同点，然后抽象出来，形成一个 抽象组件 。
> 3. 高层组件 、 底层组件 都依赖于 抽象组件 。只要抽象组件不发生变化，高层和底层之间都不会有影响。
> 4. 这就是 倒置依赖

指导方针

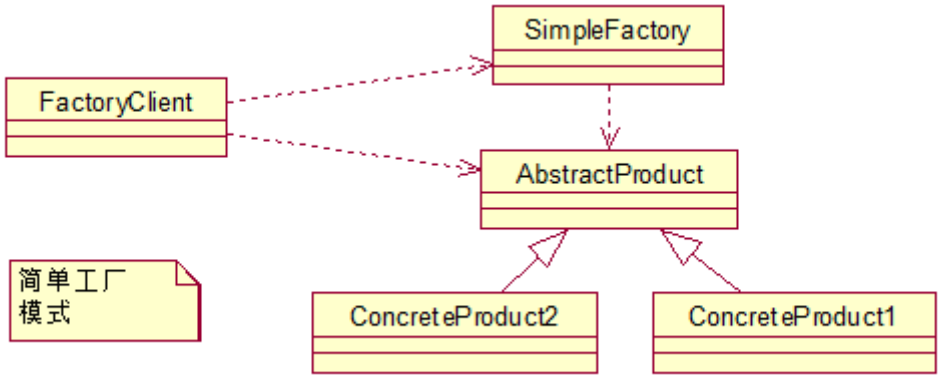
9、遵循原则的指导方针

> 1. 变量不可以持有具体类的引用：使用new就会持有具体类的引用，这就可以使用Factory
> 2. 不要让类派生自具体类：会导致依赖具体类，应该派生自抽象(抽象类和接口)
> 3. 不要覆盖基类中实现的方法：如果需要覆盖就不是适合被继承的基类

分类

简单工厂

10、简单工厂的关系图



11、简单工厂的实现

> 1. 耦合度极高，如果有变动就需要修改工厂。
> 2. 简单工厂: 本身不是一种设计模式，而是编码习惯。
>
> 动物类和具体类:

```
// 动物类和具体类
public abstract class Animal {
    public abstract void cry();
}
public class Cat extends Animal{
    @Override
    public void cry() {
        System.out.println("Miao!");
    }
}
public class Dog extends Animal{
    @Override
    public void cry() {
        System.out.println("Wang!");
    }
}
```

> 简单工厂:

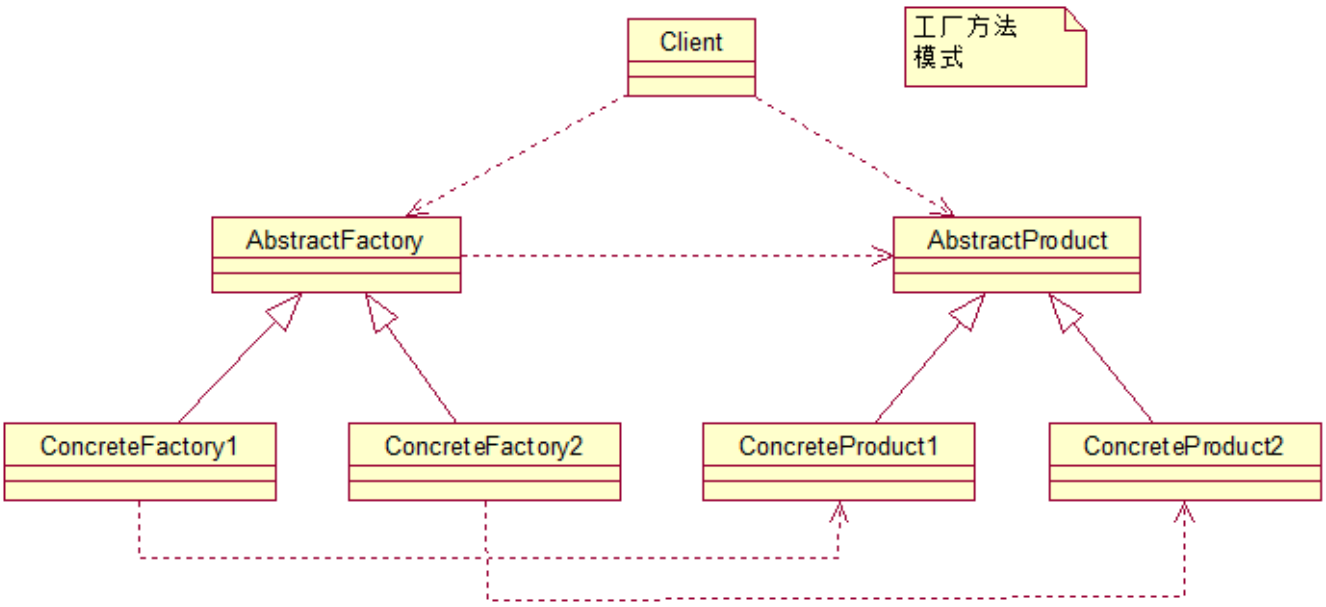
```
// 动物工厂-简单工厂
public class AnimalFactory {
    public static Animal createAnimal(String type){
        if("Dog".equals(type)){
            return new Dog();
        }else if("Cat".equals(type)){
            return new Cat();
        }
        return null;
    }
}
```

> 通过简单工厂：

```
Animal animal = AnimalFactory.createAnimal("Dog");
animal.cry();
```

工厂方法模式

12、工厂方法模式的关系图



13、工厂方法模式的实现

> 1. 将工厂进行抽象，提取出抽象工厂。
> 2. 在继承自抽象工厂的具体工厂中，创建具体的产品类。

```
public abstract class AnimalFactory {
    public abstract Animal createAnimal();
}
// 狗工厂：返回Dog
public class DogFactory extends AnimalFactory{
    @Override
    public Animal createAnimal() {
        return new Dog();
    }
}
// 猫工厂：返回Cat
public class CatFactory extends AnimalFactory{
    @Override
    public Animal createAnimal() {
        return new Cat();
    }
}
```

> 测试：

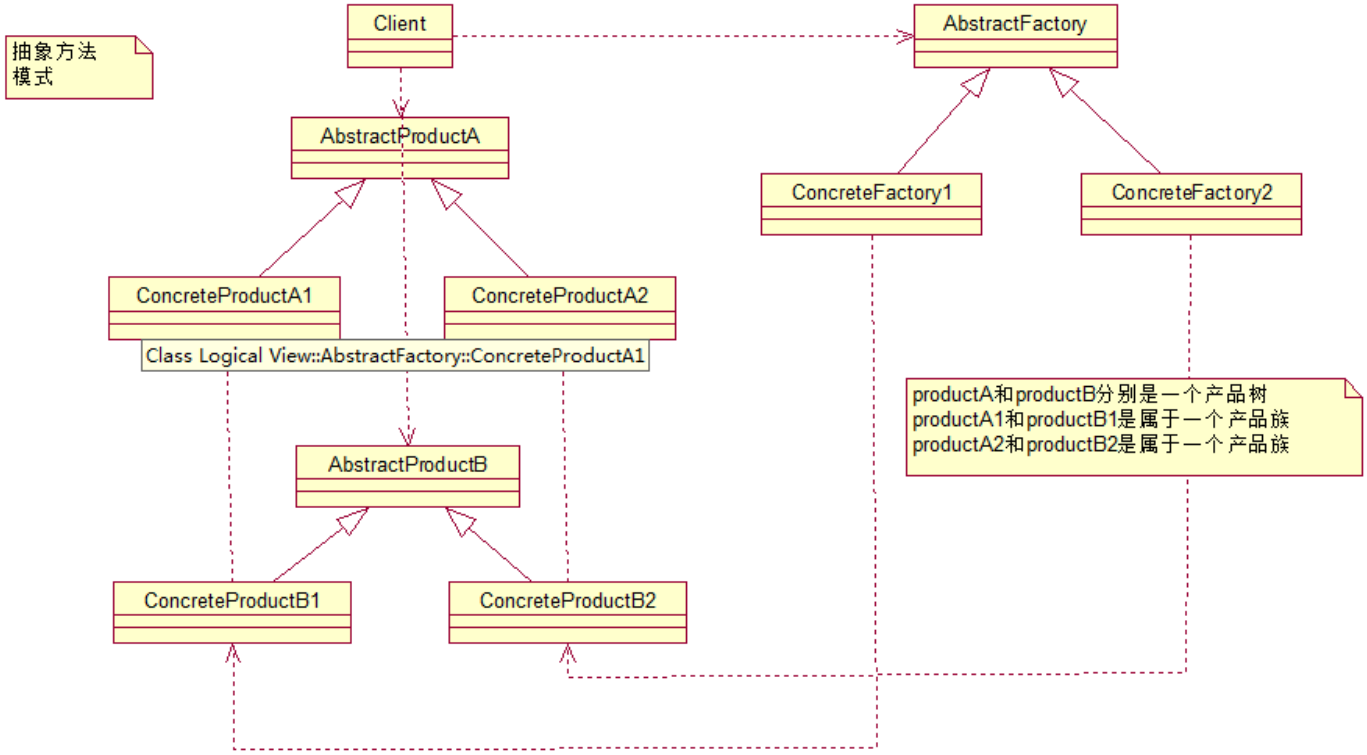
```
// 获得Dog
AnimalFactory factory = new DogFactory();
Animal animal = factory.createAnimal();
animal.cry();
// 获取Cat
factory = new CatFactory();
animal = factory.createAnimal();
animal.cry();
```

抽象工厂模式

14、抽象工厂模式是什么?

> 1. 提供了一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类。
> 2. 抽象工厂提供的一组产品的接口，每个接口的具体实现都是使用工厂方法，可以说工厂模式是在抽象工厂内部的。
> 3. 何时使用：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。
> 4. 如何解决：在一个产品族里面，定义多个产品。

15、抽象工厂模式的关系图



16、抽象工厂模式的实现

> 1. 本例：包子、华为手机、中国KFC属于产品树-都是中国的；包子和塔可(都是食物)属于产品族
> 2. SuperFactory + 两个具体工厂

```
// 超级工厂和具体工厂
public abstract class SuperFactory {
    public abstract Food createFood();
    public abstract Phone createPhone();
    public abstract KFC createKFC();
}
// 中国工厂：生产一个产品族
public class ChineseFactory extends SuperFactory {
    @Override
    public Food createFood() {
        return new BaoZi();
    }

    @Override
    public Phone createPhone() {
        return new Huawei();
    }

    @Override
    public KFC createKFC() {
        return new ChineseKFC();
    }
}
// 美国工厂：生产一个产品族
public class AmericanFactory extends SuperFactory {
    @Override
    public Food createFood() {
        return new Taco();
    }
    @Override
    public Phone createPhone() {
        return new Apple();
    }
    @Override
    public KFC createKFC() {
        return new AmericanKFC();
    }
}
```

```java
// 产品树：食物
public abstract class Food {
    public abstract void takeOut();
}
public class BaoZi extends Food {
    @Override
    public void takeOut() {
        System.out.println("包子");
    }
}
public class Taco extends Food {
    @Override
    public void takeOut() {
        System.out.println("塔可");
    }
}


// 产品树：FKC
public abstract class KFC {
    public abstract void openDoor();
}
public class ChineseKFC extends KFC{
    @Override
    public void openDoor() {
        System.out.println("中国版KFC");
    }
}
public class AmericanKFC extends KFC{
    @Override
    public void openDoor() {
        System.out.println("美国版KFC");
    }
}


// 产品树：手机
public abstract class Phone {
    public abstract void call();
}
public class Huawei extends Phone{
    @Override
    public void call() {
        System.out.println("华为手机");
    }
}
public class Apple extends Phone{
    @Override
    public void call() {
        System.out.println("苹果手机");
    }
}
```

> 测试：

```
// 多于一个的产品族，但是只用到了美国产品族
SuperFactory factory = new AmericanFactory();

factory.createFood().takeOut();
factory.createKFC().openDoor();
factory.createPhone().call();
```

17、抽象工厂模式如何增加产品树?

> 1. 比如：新增汽车的产品树
> 2. 增加Car、ChineseCar、AmericanCar这个产品树
> 3. 在SuperFactory中增加createCar()，在美国、中国工厂中去返回对应的产品。这就是产品族的修改。

18、抽象工厂模式如何增加一个工厂?(产品族)

> 1. 给产品树中增加德国产品：德国汽车、德国手机、德国KFC、德国食物
> 2. 新增德国工厂：每个方法返回对应的德国产品。

工厂方法和抽象工厂的异同

19、工厂方法和抽象工厂的相同处

> 都是负责将客户从具体类型中 解耦

20、工厂方法和抽象工厂的不同处?

> 1. 工厂方法使用“继承”，抽象工厂利用“组合”
> 2. 工厂方法通过子类创建对象，客户只需要知道所需的抽象类型，由子类负责决定具体类型。
> 3. 抽象工厂提供用来创建一个产品家族的抽象类型，该类型子类定义了产品产生的方法。使用该工厂，需要先实例化该工厂，再将其传入针对抽象类型所写的代码中。

补充实例(生产披萨)

21、实例解读

> 1. 披萨连锁店的生产过程，在不同地区有着当地特色风味的披萨店，不同披萨店的披萨原料也有所不同，然而却想要掌控披萨的生产和原料添加的过程，但是也要保留一定的区域特色。披萨店就可以作为工厂，披萨就是产品。而披萨原料的选择，也是一种工厂生产特色披萨的过程。
> 2. 该实例使用了工厂模式+抽象工厂。
> 3. 工厂方法模式： 工厂 = PizzaStore，产品 = Pizza，用 继承实现 。

4. 抽象工厂模式： 工厂 = 原料工厂，产品 = Pizza， 用 组合 的方式组合原料，实现了地区风味的披萨。

披萨商店（工厂）Pizzastore：

```
public abstract class PizzaStore {
        public PizzaStore() {
        }
//下订单，将披萨的制作通过工厂的`create`去实现，只进行披萨的准备和装箱
        public Pizza orderPizza(String type) {
                Pizza pizza = createPizza(type);
                pizza.preapare();
                pizza.box();
                return pizza;
        }
//交付给子类去实现具体生产的过程，这就是`工厂方法`的体现
        protected abstract Pizza createPizza(String type);
}
```

北京披萨商店（具体工厂）PizzaStoreBeijing：

```
public class PizzaStoreBeijing extends PizzaStore{
        public PizzaStoreBeijing() {
        }
        @Override
        protected Pizza createPizza(String type) {
                Pizza pizza;
                //原料工厂，用于创造出不同配料组成的披萨
                PizzaIngredientFactory factory = new IngredientFactoryBei();
                if(type.equals("one")) {
                        pizza = new PizzaBeiOne(factory);//北京披萨一号
                }else if(type.equals("two")) {
                        pizza = new PizzaBeiTwo(factory););//北京披萨二号

                }else {
                        pizza = null;
                }
                return pizza;
        }
}
```

南京披萨商店（具体工厂）PizzaStoreNanjing：

```
public class PizzaStoreNanjing extends PizzaStore{
        public PizzaStoreNanjing() {
        }
        @Override
        protected Pizza createPizza(String type) {
                Pizza pizza;
                PizzaIngredientFactory factory = new IngredientFactoryNan();
                if(type.equals("one")) {
                        pizza = new PizzaNanOne(factory);
                }else if(type.equals("two")) {
                        pizza = new PizzaNanTwo(factory);
                }else {
                        pizza = null;
                }
                return pizza;
        }
}
```

> 披萨（产品的抽象类）Pizza：

```
public abstract class Pizza {

        String description = "NULL PIZZA";
        Dough dough = null;
        Cheeze cheese = null;

        public abstract void preapare();//进行准备：用相应原料制作出未烘焙好的披萨饼
        public void box(){}//装箱
        public String getDescription() {
                return description+(dough==null?"":",dough")
                                +(cheese==null?"":",cheese");
        }
}
```

> 北京披萨，两种：

```
// 1、有奶酪和面团
public class PizzaBeiOne extends Pizza{

        PizzaIngredientFactory factory;
        public PizzaBeiOne(PizzaIngredientFactory factory) {
                this.description = "BeiOne";
                this.factory = factory;
        }

        @Override
        public void preapare() {
                // TODO Auto-generated method stub
                dough = factory.createDough();
                cheese = factory.createCheese();
        }

}
// 2、单纯的面团，没有奶酪
public class PizzaBeiTwo extends Pizza{
        PizzaIngredientFactory factory;
        public PizzaBeiTwo(PizzaIngredientFactory factory) {
                // TODO Auto-generated constructor stub
                this.description = "BeiTwo";
                this.factory = factory;
        }
        @Override
        public void preapare() {
                // TODO Auto-generated method stub
                dough = factory.createDough();
                //cheese = factory.createCheese();
        }
}
```

> 南京披萨，两种：PizzaNanOne和PizzaNanTwo同理。
>
> 1. 以上就是 工厂方法模式 的体现，利用工厂 `Store` 和产品 `Pizza` 就生产出了不同地区的披萨。通过 继承 将客户和产品解耦。
> 2. 以下就是 抽象工厂模式 的体现，因为不同地区的同种披萨，例如奶酪披萨里面芝士等原料的比例和成分都是不一样的，这里就通过原料工厂 `Ingredient Factory` 来完成这一需求。每种原料都是一种接口 `interface` ，实际原料实现这一接口，并通过工厂去配成相应披萨。
> 披萨原料工厂PizzaIngredientFactory，和南京、北京具体的原料工厂：

```
// 1、原料工厂
public interface PizzaIngredientFactory {
        public Cheeze createCheese();
        public Dough createDough();
}

// 2、北京原料工厂
public class IngredientFactoryBei implements PizzaIngredientFactory{
        public IngredientFactoryBei() {
        }
        @Override
        public Cheeze createCheese() {
                // 生产北京风味原料
                return new CheezeBei();
        }
        @Override
        public Dough createDough() {
                return new DoughBei();
        }
}
// 3、南京运料工厂
public class IngredientFactoryNan implements PizzaIngredientFactory{
        public IngredientFactoryNan() {
        }
        @Override
        public Cheeze createCheese() {
                Cheeze cheese = new CheezeNan();
                return cheese;
        }
        @Override
        public Dough createDough() {
                return new DoughNan();
        }
}
```

> 原料：Dough 和Cheese

```
// 1、面团
public interface Dough {
}
public class DoughBei implements Dough{

        public DoughBei() {
        }
}
public class DoughNan implements Dough{

        public DoughNan() {
        }
}

// 2、奶酪
public interface Cheeze {

}
public class CheezeBei implements Cheeze{

        public CheezeBei() {
        }
}
public class CheezeNan implements Cheeze{

        public CheezeNan() {
        }
}
```

> 测试：下订单

```
public class OrderPizza {
        public OrderPizza() {
        }
        public static void main(String[] args) {
                // 北京披萨店开业啦~!
                PizzaStore pizzaStore = new PizzaStoreBeijing();
                //第一个订单：北京一号至尊披萨
                Pizza pizza = pizzaStore.orderPizza("one");
                System.out.println(pizza.getDescription());
                //第二个订单：北京一号华贵披萨
                pizza = pizzaStore.orderPizza("two");
                System.out.println(pizza.getDescription());

                //南京披萨店也开业了！
                pizzaStore = new PizzaStoreNanjing();
                pizza = pizzaStore.orderPizza("one");
                System.out.println(pizza.getDescription());
                pizza = pizzaStore.orderPizza("two");
                System.out.println(pizza.getDescription());
        }

}
```

建造者模式

1、工厂模式和建造者模式的区别?

> 1. 工厂模式： 定义了一个创建对象的接口，由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类。
> 2. 建造者模式：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

参考资料

1. 简单工厂、工厂方法、抽象工厂、策略模式、策略与工厂的区别
2. 抽象工厂的讲解