1 原始代码

- 以下是最原始的代码，客户端要订披萨时，直接调用 orderPizza() 方法，传入类型参数，即可得到相应披萨
- **优点：** 客户端调用很方便
- **缺点：** 当要扩展新的披萨类型时，需要修改旧的代码，可知它对修改不闭合

```
    public Pizza orderPizza(String type) {
        Pizza pizza = null;
        switch (type) {
            case "cheese":
                pizza = new CheesePizza();
                break;
            case "greek":
                pizza = new GreekPizza();
                break;
            default:
                throw new IllegalArgumentException("无此类型的Pizza");
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.done();
        return pizza;
    }
    // 客户端代码
    orderPizza("cheese");
```

2 简单工厂

- 将可能会变化的代码做了封装，即封装成了一个创建披萨的简单工厂
- 这样虽然对修改还是开放的，但是修改的代价会低一点（只需修改一处）

```
public class PizzaStoreSimple {
    public Pizza orderPizza(String type) {
        Pizza pizza = SimplePizzaFactory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.done();
        return pizza;
    }
}
// 客户端代码，和原始代码一样
orderPizza("cheese");
```

3 工厂模式

- 工厂模式在简单工厂的基础上做了一层抽象，它让子类决定如何制造披萨
- 和简单工厂的区别：
  - 简单工厂的 PizzaStoreSimple 类就是一个实例工厂，客户端之间调用这个类，就可以返回披萨
  - 工厂模式将实例工厂抽象成了抽象工厂 PizzaStoreFactory，这个抽象工厂可以分化出许多具体的实例工厂，并且不同的实例工厂制造不同类型的披萨
  - 简单来说，工厂模式就是将简单工厂分化成许多小工厂，每个小工厂制作不同的披萨。所有小工厂制作出的披萨总和，就等于简单工厂制作出的披萨种类
- **优点：** 对修改闭合。当要添加新的披萨类型时，不需要修改旧代码，只要新建一个工厂类即可
- **缺点：** 客户端需要自己选定需要的披萨工厂

```
public abstract class PizzaStoreFactory {
    //其实这里就相当于模板模式了
    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.done();
        return pizza;
    }
    //让子类决定如何制造披萨，这里一般是没有类型的，每个工厂对应一种类型
    //当然也可以是多个类型
    abstract Pizza createPizza(String type);
}
// 客户端代码
PizzaStoreFactory pizzaStore = new 客户端需要的工厂();
pizzaStore.orderPizza("这里一般是没有类型的，每个工厂对应一种类型");
```

4 抽象工厂模式

- 抽象工厂模式出现的情况是：一个完整产品是由多个产品家族组合而成。比如披萨由披萨原料组合；车是由轮子、发动机等产品组合而成。 而简单工厂生产的产品就是一个独立的个体。其它的感觉都是一样的
- **优点：** 和简单工厂一样，都是对修改封闭的。只是简单工厂是一个产品，这个是多个产品

- **缺点：** 当扩展产品家族时（添加一种新的原料），新加代码比较繁琐，比较多 。需要在所有的抽象工厂实现类中，实现新加的方法
- 完全可以在抽象工厂的 createPizza() 方法中，分别创建多个产品，再直接组合。而不需要专门定义一个抽象工厂来组合。这样扩展产品族的时候，就好些。这就由沦为了工厂模式了。

```
    public abstract class PizzaStoreAbstractFactory {
        //其实这里就相当于模板模式了
        public PizzaCustom orderPizza(String type) {
            PizzaCustom pizza = createPizza(type);
            pizza.prepareIngredient();
            pizza.prepare();
            pizza.bake();
            pizza.cut();
            pizza.done();
            return pizza;
        }
        //让子类决定如何制造披萨
        abstract PizzaCustom createPizza(String type);
    }
    public class ChinesePizzaFactory extends PizzaStoreAbstractFactory {
        @Override
        PizzaCustom createPizza(String type) {
            return new ChinesePizzaCustom("中国",1,new ChineseIngredientFactory());
        }
    }
    public class ChinesePizzaCustom extends PizzaCustom {
        //这里就是一个抽象工厂，它负责生产一系列的产品家族!!!!
        IIngredientFactory iIngredientFactory;
        public ChinesePizzaCustom(String name, Integer size, IIngredientFactory
iIngredientFactory){
            super(name, size);
            this.iIngredientFactory = iIngredientFactory;
        }
        //名字尺寸都可以自己定义
        public ChinesePizzaCustom() {
            super("中国披萨",2);
        }
        //但是加盟店的原料，应该是总店统一配置的，不然有些加盟店偷工减料，影响声誉
        //这时候就需要一个原料工厂了
        //通过将抽象工厂得到的产品家族组合，最终形成一个完整的产品
        @Override
        void prepareIngredient() {
            System.out.println("开始准备原料...");
            this.cheese = iIngredientFactory.createCheese();
            this.sauce = iIngredientFactory.createSauce();
            System.out.println("原料 "+ cheese + "、"+sauce + " 准备完成！");
        }
    }
    //这个就是产品族的抽象工厂，每个实例工厂都自定义对应工厂生产的产品
    //如 chineseFactory 生产chineseSauce/chineseCheese，JapenPactory生产
japenSauce/japenCheese
    public interface IIngredientFactory {
        Sauce createSauce();
```

```
        Cheese createCheese();
    }
    // 客户端代码
    PizzaStoreAbstractFactory pizzaStore = new ChinesePizzaFactory();
    pizzaStore.orderPizza("");
```

5 总结

- 不管时工厂模式还是抽象工厂模式，横向扩展很方便（不修改产品属性，增加产品类型），纵向扩展不易（修改产品属性）
- 总得来说，工厂、抽象工厂模式，都是符合开放-封闭原则的（在不修改产品属性的情况下）
- 观察工厂、抽象工厂模式的客户端代码：说明两都是面向接口编程的，应用程序和具体的类之间是解耦的，如果想使用别的产品，只要该一行代码即可。
- 这个披萨的例子可扩展到数据库的例子，不同的数据库由自己的实现，而客户端使用不同的数据库，只要修改一行代码即可

6 通过反射和配置文件优化

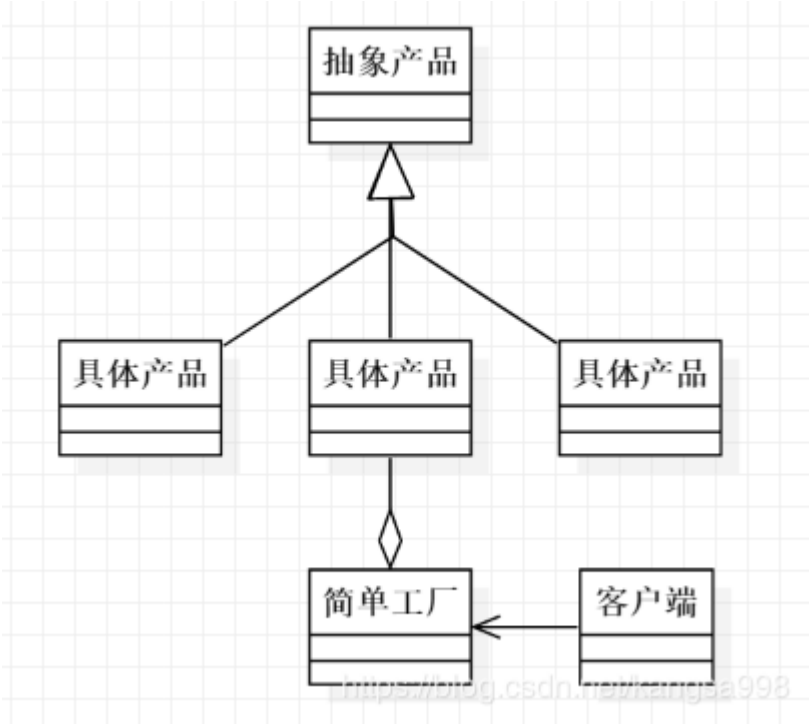
- 通过 SPI 读取 META-INF.services包下的文件名，来决定使用的是那个实例类

```
    //这样客户端的代码就不用变了，只需要修改相应文件夹中的属性即可
    ServiceLoader<PizzaStoreAbstractFactory> serviceLoader =
ServiceLoader.load(PizzaStoreAbstractFactory.class);
    Iterator<PizzaStoreAbstractFactory> iter = serviceLoader.iterator();
    while (iter.hasNext()) {
        PizzaStoreAbstractFactory proxy = iter.next();
        proxy.orderPizza("");
        break;
    }
```

7 简单工厂 UML 关系图

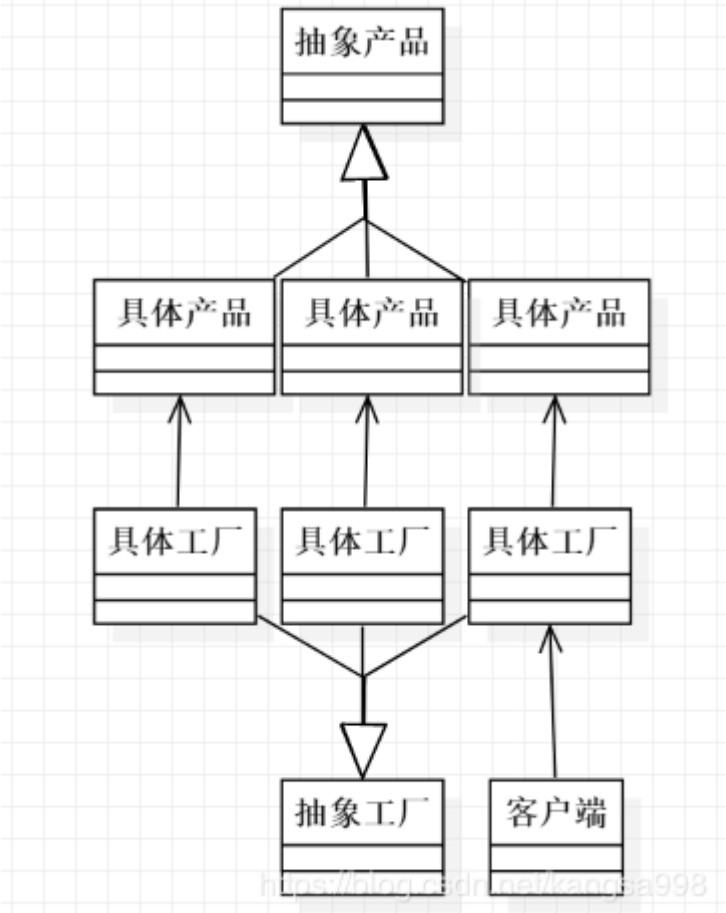
- 开发人员只能通过修改简单工厂来扩展
- 客户端可以关联一个简单工厂，或者直接调用静态的简单工厂

- 一个简单工厂聚合多个具体产品



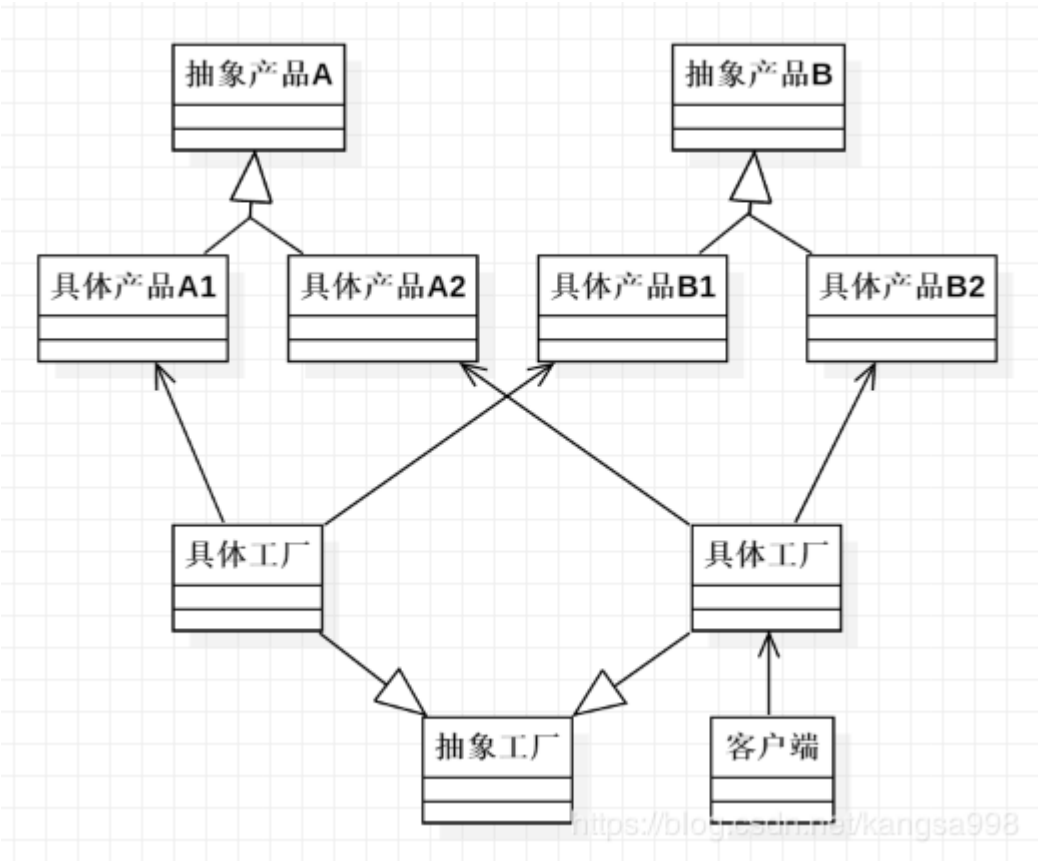
8 工厂模式 UML 关系图

- 一般具体工厂只对应一个具体产品
- 这样开发人员就可以直接增加新的工厂来达到扩展作用
- 客户端关联 /依赖某一个具体的工厂
- 和简单工厂的差别就是将一个总的简单工厂分化成许多小的具体工厂



9 抽象工厂 UML 关系图

- 抽象工厂主要是针对产品家族的说法
- 一个具体的工厂，生成一个完整产品的多个部件
- 客户端依赖/关联某个具体工厂，来获得一个产品的所有部件
- 和工厂模式相比就是扩展了产品种类



参考

大话设计模式 Head First 设计模式 github 源码地址