链滴

设计模式 | 04 工厂方法模式

作者：qq692310342
原文链接：https://ld246.com/article/1567560116614
来源网站：链滴

开头说几句

博主的博客地址：https://www.jeffcc.top/

推荐入门学习设计模式java版本的数据Head First 《设计模式》

什么是工厂方法模式

专业定义：工厂方法模式定义了一个创建对象的接口，但是由子类决定实例化的类是哪一个。工厂方让实例化延迟到了子类。

博主的理解：工厂方法其实也是使用了把不变的部分和变化的部分分开封装起来的设计原则，将变化部分提供一个抽象方法来交给子类来实现，不变的部分由工厂完成封装。

这种设计可以将对象的创建封装起来，以便得到更加松耦合，更有弹性的设计。

设计原则

1. 依赖倒置原则，要依赖抽象，不要依赖具体类，具体做法是需要高层组件（工厂）和底层组件（实类）之间不要有太多的依赖关系，而是可以通过一个共同的抽象类（工厂产生的对象）来实现依赖倒。

2. 多用组合，少用继承。

3. 针对接口编程，而不是针对实现编程。

4. 为交互对象之间的松耦合设计而努力。

5. 类应该对外开放拓展，对修改关闭。

设计要点

1. 所有的工厂都是用来封装对象的创建的。

2. 工厂方法使用继承，把对象的创建委托给子类，子类实现工厂方法来创建对象。

3. 工厂方法允许类的实例化延迟到子类。

设计实现

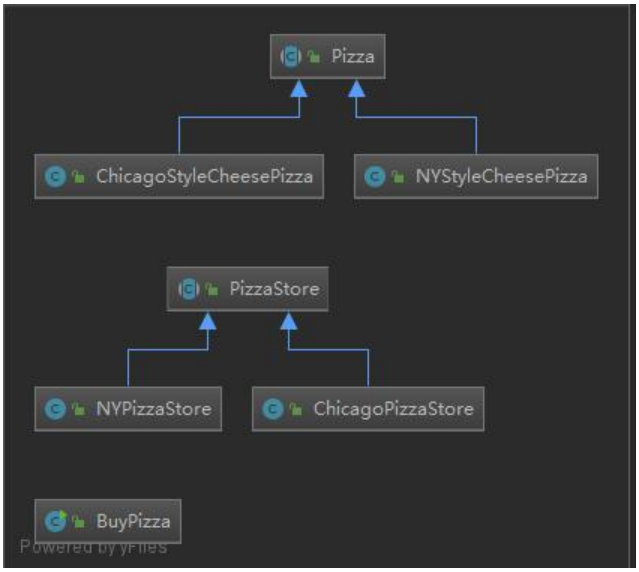
设计背景

一家披萨店需要拓展加盟业务，届时会有许多不同地方口味的披萨出现，而我们需要做的就是设计一工厂来让加盟商使用，具体的风味的披萨由加盟商自己决定，工厂只负责包装等内容。

代码实现

项目结构

项目类图



披萨工厂

```
package factory;

import pizza.Pizza;

/**
 * 披萨工厂
 */
public abstract class PizzaStore {

    public Pizza orderPizza(String type){
        Pizza pizza;
//        进行创建pizza
        pizza = createPizza(type);
//        下面进行的是共同的东西，所以封装起来到工厂中
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;

    }

    /**
     * 抽取出一个抽象的制作pizza的方法来交给子类具体实现
     * 工厂模式的关键所在
     * @param type  什么风味的pizza
     * @return
     */
    public abstract Pizza createPizza(String type);

}
```

ChicagoPizzaStore加盟店

```
package store;

import factory.PizzaStore;
import pizza.Pizza;
import pizza.impl.ChicagoStyleCheesePizza;

/**
 * ChicagoPizzaStore加盟店
 */
public class ChicagoPizzaStore extends PizzaStore {

    /**
     * 模拟测试生产pizza  太多类型了就写一个 理论可以无限拓展
     * @param type 什么风味的pizza
     * @return
     */
    @Override
    public Pizza createPizza(String type) {
```

```
        if ("cheese".equals(type)) {
            return new ChicagoStyleCheesePizza();
        } else {
            return null;
        }
    }
}
```

NYpizza加盟店

```
package store;

import factory.PizzaStore;
import pizza.Pizza;
import pizza.impl.NYStyleCheesePizza;

/**
 * 纽约pizza加盟店
 */
public class NYPizzaStore extends PizzaStore {

    /**
     * 模拟测试生产pizza  太多类型了就写一个 理论可以无限拓展
     * @param type 什么风味的pizza
     * @return
     */
    @Override
    public Pizza createPizza(String type) {
        if ("cheese".equals(type)) {
            return new NYStyleCheesePizza();
        } else {
            return null;
        }
    }
}
```

披萨抽象类（关键）

```
package pizza;

import java.util.ArrayList;

/**
 * pizza的抽象类
 * 也就是之前说的依赖倒置所依赖的类!!!
 * 需要有pizza的操作的方法
 */
public abstract class Pizza {

    public String name;//pizza名称
    public String dough;//pizza面团类型
```

```
    public String suace;//pizza酱料类型
    public ArrayList<String> toppings = new ArrayList<>(); //佐料


    /**
     * 提供默认的基本做法 进行烘烤、切片、装盒 如果需要可以进行重写
     */
    public void prepare() {
        System.out.println("Preparing" + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding toppings:");
        toppings.forEach(topping -> {
            System.out.println("   " + topping);
        });

    }

    /**
     * 烘烤
     */
    public void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    /**
     * 切片
     */
    public void cut() {
        System.out.println("Cutting the pizza into diagonal slices!");
    }

    /**
     * 装盒
     */
    public void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

    public String getName() {
        return name;
    }
}
```

ChicagoStyleCheesePizza

```
package pizza.impl;

import pizza.Pizza;

/**
 * 实现不同的风味的pizza ChicagoStyleCheesePizza
 */
public class ChicagoStyleCheesePizza extends Pizza {
```

```
    /**
     * 设置风味
     */
    public ChicagoStyleCheesePizza() {
        name = "Chicago Style Deep Dish Cheese Pizza";
        dough = "Extra Thick Curst Dough";
        suace = "Plum Tomato Suace";
        toppings.add("Shredded Mozzarella Cheese");
    }


    /**
     * 模拟需要有不同的切片方法
     */
    @Override
    public void cut() {
        System.out.println("Cutting the pizza into square slices");
    }
}
```

NYStyleCheesePizza

```
package pizza.impl;

import pizza.Pizza;

/**
 * 实现不同的风味的pizza 纽约风味
 */
public class NYStyleCheesePizza extends Pizza {

    /**
     * 设置风味
     */
    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese Pizza";
        dough="Thin Curst Dough";
        suace="Marinara Suace";
        toppings.add("Grated Reggiano Cheese");
    }
}
```

测试买pizza

```
package test;

import factory.PizzaStore;
import pizza.Pizza;
import store.ChicagoPizzaStore;
```

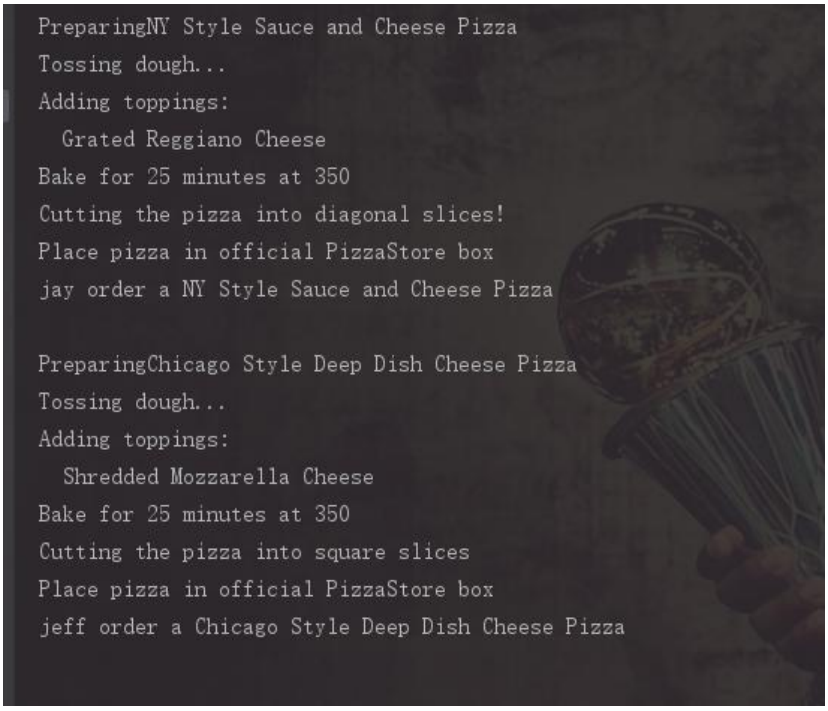
```
import store.NYPizzaStore;

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

//        找到NY店
        PizzaStore nyPizzaStore = new NYPizzaStore();
        Pizza pizza1 = nyPizzaStore.orderPizza("cheese");
        System.out.println("jay order a "+pizza1.getName()+"\n");

//        Chicago店
        PizzaStore chicagoPizzaStore = new ChicagoPizzaStore();
        Pizza pizza2 = chicagoPizzaStore.orderPizza("cheese");
        System.out.println("jeff order a "+pizza2.getName()+"\n");
    }
}
```

输出结果



最后说两句

工厂方法让子类决定要实例化的类是哪一个，这句话的理解并不是指模式允许子类本身在运行时做出定，而是要在编写创建者类的时候，不需要知道实际所创建的产品是哪一个，选择了使用哪一个子类也就“决定”了实际创建的产品是什么。

接下来会出一篇关于抽象工厂模式的文章 和工厂方法模式有一定的相通性。

END

2019年9月4日09:20:13