1、概述
2、简单工厂模式
2.1、介绍
2.2、结构
2.3、具体实现
3、工厂方法模式
3.1、介绍
3.2、结构
3.3、具体实现
4、抽象工厂模式
4.1、介绍
4.2、结构
4.3、具体实现
4.4、适用场景
5、简单工厂模式改进
5.1、介绍
5.2、具体实现
6、JDK中的迭代器

# 1、概述

根据百科的定义，工厂模式是“工厂是用于创建其他对象的对象”。

以咖啡店为例，设计一个咖啡类Coffee，并定义其两个子类（美式咖啡AmericanCoffee和拿铁咖啡LatteCoffee）；再设计一个咖啡店类CoffeeStore，咖啡店具有点咖啡的功能。

```
public class CoffeeStore {

    public Coffee orderCoffee(String type) {
        // 声明Coffee类型的变量
        Coffee coffee = null;
        // 根据不同类型创建不同的coffee子类对象
        if("american".equals(type)) {
            // 类型为美式咖啡，创建AmericanCoffee
            coffee = new AmericanCoffee();
        } else if("latte".equals(type)) {
            // 类型为拿铁咖啡，创建LatteCoffee
            coffee = new LatteCoffee();
        } else {
            throw new RuntimeException("对不起，您所点的咖啡店里没有出售");
        }
        // 附加操作
        coffee.addMilk();
```

```
            coffee.addsugar();

            return coffee;
        }
    }
```

在上面的示例中，我们没有使用任何模式并且应用程序运行良好。但是如果我们考虑未来可能的变化并仔细观察，我们可以预见到当前实现存在以下问题：

- 在当前的应用程序中，无论哪里需要特定的咖啡，它都是使用具体类创建的。将来，如果是**对类名进行修改/提出了不同的具体类**，就必须在整个应用程序中进行更改；
- 目前创建具体的咖啡无论是美式还是拿铁，其构造函数参数列表都为空。如果后续**对构造函数进行了修改**，需要传入一些必要的参数，那么在每一个创建对象的客户端中都需要进行修改；
- 在创建对象时，在上面的代码中客户端知道**具体类**，也知道**对象的创建过程**，我们应该避免将此类对象创建细节和内部类暴露给客户端应用程序。

通过上述问题我们总结出来，在不使用任何模式下程序运行正常，但是却有着**耦合严重**的问题。假如我们要更换/修改对象，所有new对象的地方都需要修改一遍，这显然**违背了软件设计的开闭原则**。如果我们使用工厂来生产对象，我们就只和工厂打交道就可以了，彻底和对象解耦，如果要更换对象，直接在工厂里更换该对象即可，达到了与对象解耦的目的；所以说，工厂模式最大的优点就是：**解耦。**

# 2、简单工厂模式

## 2.1、介绍

简单工厂模式是 `Factory` 最简单形式的类（与工厂方法模式或抽象工厂模式相比）。换句话说，我们可以说：**在简单工厂模式中，我们有一个工厂类，它有一个方法可以根据给定的输入返回不同类型的对象。**简单工厂并不是一种设计模式，反而比较像是一种编程习惯。

## 2.2、结构

简单工厂包含如下角色：

- **抽象产品** ：定义了产品的规范，描述了产品的主要特性和功能。如上述的咖啡类；
- **具体产品** ：实现或者继承抽象产品的子类。如上述的美式咖啡和拿铁咖啡；
- **具体工厂** ：提供了创建产品的方法，调用者通过该方法来获取产品。用于生产需要的咖啡。

## 2.3、具体实现

设立工厂（factory）处理创建对象的细节，一旦有了 `SimpleCoffeeFactory` ，`CoffeeStore` 类中的 `orderCoffee()` 就变成此对象的客户，后期如果需要Coffee对象直接从工厂中获取即可。这样也就**解除了客户端和Coffee实现类的耦合。**

```java
public class SimpleCoffeeFactory {

    public Coffee createCoffee(String type) {
        Coffee coffee = null;
        if("americano".equals(type)) {
            coffee = new AmericanoCoffee();
        } else if("latte".equals(type)) {
            coffee = new LatteCoffee();
        }
        return coffee;
    }
}
```

在开发中也有一部分人将工厂类中的创建对象的功能定义为**静态的**，这个就是**静态工厂模式**，它也不是23种设计模式中的。

```java
public class SimpleCoffeeFactory {

    public static Coffee createCoffee(String type) {
        Coffee coffee = null;
        if("americano".equals(type)) {
            coffee = new AmericanoCoffee();
        } else if("latte".equals(type)) {
            coffee = new LatteCoffee();
        }
        return coffe;
    }
}
```

细心的小伙伴可能会发现了，在设立工程解除客户端与具体实现类之间耦合的同时，又**产生了新的耦合**，即 `CoffeeStore` 对象和 `SimpleCoffeeFactory` 工厂对象的耦合，**工厂对象和商品对象的耦合。**后期如果再加新品种的咖啡，我们势必要需求修改 `SimpleCoffeeFactory` 的代码，**违反了开闭原则。**

但是如果我们考虑未来可能会出现的各种变化并仔细观察，简单工厂模式确实相对于没有使用任何模式的代码**拓展性和可维护性更好。**工厂类的客户端可能有很多，比如说我可能不只是这个咖啡店需要制作咖啡，奶茶店、早餐店可能都会生产咖啡，甚至是会生产不同的咖啡，这时我们新增对应的咖啡实现类之后，**只需要修改工厂类的代码**，省去其他的修改操作，这种操作可以**使得整个业务逻辑是符合开闭原则的。**

# 3、工厂方法模式

## 3.1、介绍

**工厂方法模式**（英语：**Factory method pattern**）属于创建模式类别。在GoF中对工厂方法模式的定义为：**“定义用于创建对象的接口，但让子类决定要实例化哪个类。工厂方法让类将实例化推迟到子类”。**这种模式能够解决简单工厂模式中存在的耦合问题，**整体完全遵循开闭原则。**

## 3.2、结构

工厂方法模式相对于简单工厂模式则是多了一个抽象工厂的角色：

- **抽象工厂**（Abstract Factory）：提供了创建产品的接口，调用者通过它访问具体工厂的工厂方法来创建产品。
- **具体工厂**（Concrete Factory）：主要是实现抽象工厂中的抽象方法，完成具体产品的创建。
- **抽象产品**（Product）：定义了产品的规范，描述了产品的主要特性和功能。
- **具体产品**（Concrete Product）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应。

## 3.3、具体实现

依旧以咖啡为例，在简单工厂模式中是以一个工厂来生产所有咖啡的，而在工厂方法模式中，则是**将生产咖啡的工厂定义成抽象工厂**，当我们需要生产美式咖啡和拿铁咖啡时，**分别建立两个新工厂去实现抽象工厂**，分别生产对应的咖啡。

工厂的代码实现如下：

```
// 抽象工厂
public interface CoffeeFactory {

    Coffee createCoffee();
}

==========================

// 具体工厂
public class LatteCoffeeFactory implements CoffeeFactory {

    public Coffee createCoffee() {
        return new LatteCoffee();
    }
}

public class AmericanCoffeeFactory implements CoffeeFactory {

    public Coffee createCoffee() {
        return new AmericanCoffee();
    }
}
```

咖啡店即客户端的代码实现如下：

```
public class CoffeeStore {

    private CoffeeFactory factory;

    // 通过传递具体工厂的实现类生产对应的咖啡
    public CoffeeStore(CoffeeFactory factory) {
        this.factory = factory;
    }

    // 下单生产咖啡
    public Coffee orderCoffee(String type) {
        Coffee coffee = factory.createCoffee();
        coffee.addMilk();
        coffee.addsugar();
        return coffee;
    }
}
```

从以上的编写的代码可以看到，要增加产品类时也要相应地增加工厂类，不需要修改工厂类的代码了，这样就解决了简单工厂模式的缺点。在工厂方法模式中使用了多态的特性，在保持简单工厂模式特点的同时，还解决了遗留下来的问题：

- 用户只需要知道具体工厂的名称就可得到所要的产品，**无须知道产品的具体创建过程**；
- 在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，**符合开闭原则。**

> 🌈
>
> 但利与弊同在，每增加一个产品时就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度，容易引起类爆炸。

# 4、抽象工厂模式

## 4.1、介绍

**抽象工厂模式**（英语：**Abstract factory pattern**）是一种软件开发**设计模式。**抽象工厂模式的实质是**“提供接口，创建一系列相关或独立的对象，而不指定这些对象的具体类。”**在正常使用中，客户端程序需要创建抽象工厂的具体实现，然后使用抽象工厂作为**接口**来创建这一主题的具体对象。客户端程序不需要知道（或关心）它从这些内部的工厂方法中获得对象的具体类型，因为客户端程序仅使用这些对象的通用接口。抽象工厂模式将一组对象的实现细节与他们的一般使用分离开来。

## 4.2、结构

抽象工厂模式的主要角色同工厂方法模式相同，只是一些具体含义有所差异：

- **抽象产品**（Abstract Product）：定义了产品的规范，描述了产品的主要特性和功能，抽象工厂模式有多个抽象产品。
- **具体产品**（Concrete Product）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它 同具体工厂之间是多对一的关系。
- **抽象工厂**（Abstract Factory）：提供了创建产品的接口，它包含多个创建产品的方法，可以创建多个不同等级的产品。
- **具体工厂**（Concrete Factory）：主要是实现抽象工厂中的多个抽象方法，完成具体产品的创建。

## 4.3、具体实现

这里的例子就不引用前面的咖啡了，个人觉得咖啡的例子用在这个模式中有点不太好理解，因此以Windows和Mac两个系统的不同样式为例，这两个系统都存在按钮和边框这两种元素，当我们需要切换两种不同系统的样式时，可以直接通过Windows工厂或Mac工厂进行统一切换。当我们添加另外一个系统的样式时，如Linux系统的样式，只需要新增一个Linux工厂去实现抽象工厂即可，无需去修改原有代码。

- 定义产品抽象类，即上方说的抽象产品：

```
public interface Button {}
public interface Border {}
```

- 实现抽象类，即上方说的具体产品：

```
public class MacButton implements Button {}
public class MacBorder implements Border {}

public class WinButton implements Button {}
public class WinBorder implements Border {}
```

- 定义产品抽象工厂：

```
public interface AbstractFactory {

    Button createButton();

    Border createBorder();
}
```

- 实现抽象工厂，定义具体工厂：

```
public class MacFactory implements AbstractFactory {
    public Button createButton() {
        return new MacButton();
```

```
    }
    public Border createBorder() {
        return new MacBorder();
    }
}

---

public class WinFactory implements AbstractFactory {
    public Button createButton() {
        return new WinButton();
    }
    public Border createBorder() {
        return new WinBorder();
    }
}
```

### 4.4、适用场景

在以下情况可以考虑使用抽象工厂模式：

- 一个系统要独立于它的产品的创建、组合和表示时。
- 一个系统要由多个产品系列中的一个来配置时。
- 需要强调一系列相关的产品对象的设计以便进行联合使用时。
- 提供一个产品类库，而只想显示它们的接口而不是实现时。

## 5、简单工厂模式改进

### 5.1、介绍

通常我们在使用简单工厂模式的时候会由创建方法create通过传入的参数来判断要实例化哪个对象，就像下面这样

```
public class SimpleCoffeeFactory {

    public Coffee createCoffee(String type) {
        Coffee coffee = null;
        if("americano".equals(type)) {
            coffee = new AmericanoCoffee();
        } else if("latte".equals(type)) {
            coffee = new LatteCoffee();
        }
        return coffee;
    }
}
```

这里面依旧是两种咖啡，通过type来判断需要生产哪种咖啡，即决定实例化哪个子类。现在遇到这么一个问题，如果**新增一个子类**的话，那就**必须要修改工厂类代码**了，这也是简单工厂模式的痛点。可能你会说“改一下又何妨?”，虽不说影响不影响什么开闭设计原则，但是有个情况你可曾想到，你这个类要打包发布给别人用呢？别人在没有源码的情况下如何扩展呢？这里就需要我们动态的**通过配置文件**来加载实现类了。

## 5.2、具体实现

实现的基本思路为：通过读取本地的 `.properties` 文件来获取我们需要实例化的类，然后通过**反射**来生成对象，这样当你把发布出去的时候，使用者只用更改配置文件就可以让工厂去实例化自己后来才写的实现类。

在Resource目录下创建配置文件 `concrete-bean.properties` ，并在其中指明具体产品的全类名：

```
american=top.xbaoziplus.design-pattern.factory.config_factory.AmericanCoffee
latte=top.xbaoziplus.design-pattern.factory.config_factory.LatteCoffee
```

改进工厂类，不再是显式实例化具体的对象，而是在获取配置文件中的全类名之后，通过反射对产品进行实例化：

```
public class CoffeeFactory {
    public static Coffee createCoffee(String type) {
        Coffee coffee;

        // 实例化Properties对象，用于读取.properties
        Properties properties = new Properties();
        InputStream is = null;
        try {
            is = CoffeeClient.class.getResourceAsStream("concrete-bean.properties");
            // 装载concrete-bean.properties文件
            properties.load(is);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                is.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        try {
            // 根据key获取value,value即为将要实例化的类的全类名，再通过forName获取其字节码对象
```

```
            Class<?> clazz =
Class.forName(properties.getProperty(type));
            // 使用反射进行实例化
            coffee = (Coffee) clazz.newInstance();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        return coffee;
    }
}
```

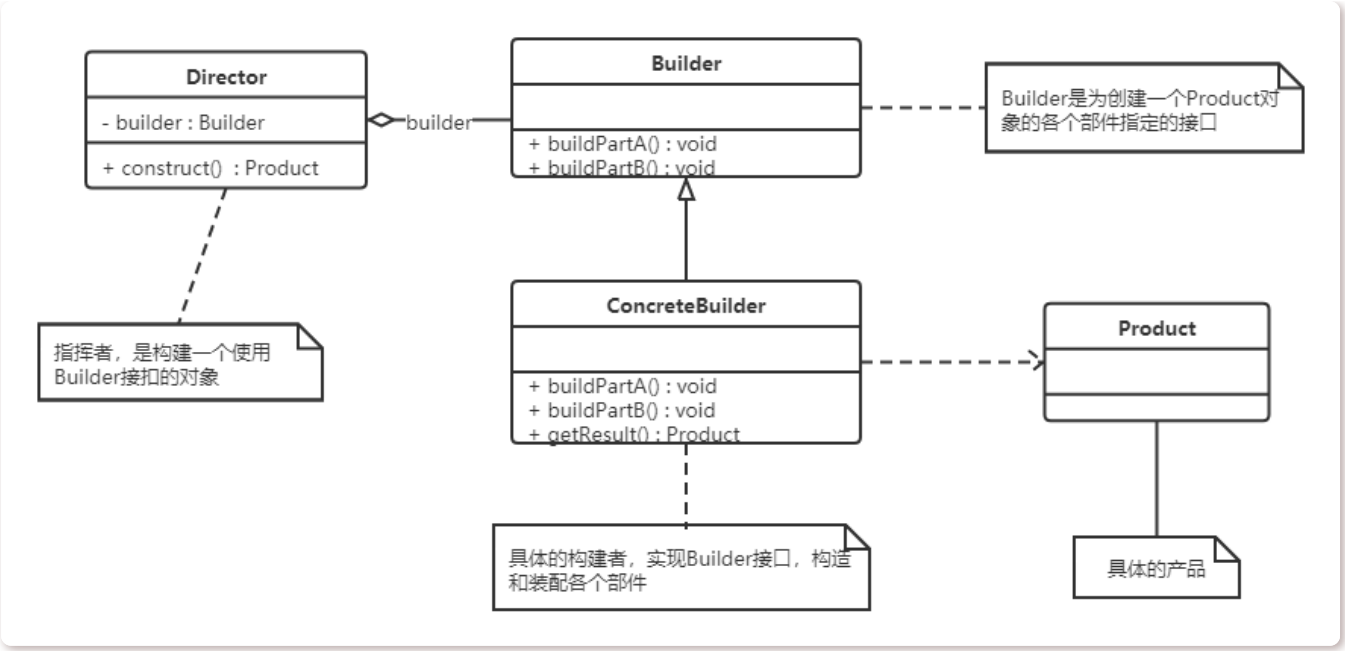
# 6、JDK中的迭代器

对下面的代码大家应该很熟，使用迭代器遍历集合，获取集合中的元素。而单列集合获取迭代器的方法就使用到了工厂方法模式。

```
public class Demo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("令狐冲");
        list.add("风清扬");
        list.add("任我行");

        //获取迭代器对象
        Iterator<String> it = list.iterator();
        //使用迭代器遍历
        while(it.hasNext()) {
            String ele = it.next();
            System.out.println(ele);
        }
    }
}
```

我们看通过类图看看结构：




`Collection`接口是**抽象工厂类**，`ArrayList`是**具体的工厂类**；`Iterator`接口是**抽象商品类**，`ArrayList`类中的`Iter内部类`是**具体的商品类。**在具体的工厂类中`iterator()`方法创建具体的商品类的对象。同时在JDK中还有许多地方也用到了工厂模式，如：

- `DateForamt`类中的`getInstance()`方法使用的是工厂模式；
- `Calendar`类中的`getInstance()`方法使用的是工厂模式；