链滴

设计模式 - 工厂模式和抽象工厂模式

作者：zylei21

原文链接：https://ld246.com/article/1608555502740

来源网站：链滴

“ 设计模式详解。”

《Design Patterns: Elements of Reusable Object-Oriented Software》（即《设计模式》一书）由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 合著（Addison-Wesley，1995）。这几位作者常被称为"四人组（Gang of Four）"。

设计模式不是一种规定，也不需要为了设计模式而设计，错误的应用设计模式反而会让代码变得复杂难以维护。相反，正确的应用设计模式，可以让我们的代码更加健壮，扩展性更好，逻辑也更加清晰。

设计模式一书针对不同的应用场景，列出了各自的最佳实践。随着技术的不断发展，可能一些范例已不再适用，但是这本书带给我们思想上的指引，意义是非常重大的。设计模式一书，根据应用场景，致分为建造型、结构型、行为型，总共23种，并提出了6大原则。

本文开始，试着详细介绍每种设计模式，有解释不到位，理解存在偏差的地方，请提出宝贵意见。

建造型-工厂模式

工厂模式 Factory Method ，定义了一个创建对象的接口，由子类决定实现是哪个，工厂方让类实例推迟到子类。工厂模式隐藏创建对象的过程，是创建复杂对象的最佳实践。

应用场景：生产一种产品，这种产品有多种实现。

关键点：实现一个工厂接口

优点：隐藏了创建过程，创建一个产品只需要传入一个名字即可，需要增加一种产品实现，只需要实工厂接口即可，扩展性非常好。

缺点：每次增加一种产品的实现，需要增加一个实现类，并维护工厂实现产品的方法，增加了复杂度。

通常我们创建对象需要用到关键字new，创建一个产品直接new不好吗，为啥非要增加一个工厂呢？不是多此一举吗？下面举一个简单的例子说明一下

这里是一个手机店，手机店里出售小米手机和华为手机。代码如下

非工厂模式

```
public abstract class Phone {
    String name;

    public abstract void init();

    public Phone() {
        init();
    }

    public void test(){
        System.out.println("i was " + name + " phone!");
        System.out.println("testing....");
    }

    public void packingPhone(){
        System.out.println("packing....");
    }
}
```

小米手机

```
public class XiaoMiPhone extends Phone {
    @Override
    public void init() {
        name = "小米";
    }
}
```

华为手机

```
public class HuaWeiPhone extends Phone {
    @Override
    public void init() {
        name = "华为";
    }
}
```

每次顾客来的时候，告诉我手机的名字，我就把手机卖给他，代码如下：

```
public class PhoneStore {

    public Phone order(String name) {
        Phone phone = null;
        switch (name) {
            case "小米":
                phone = new XiaoMiPhone();
                break;
            case "华为":
                phone = new HuaWeiPhone();
                break;
```

```
            default:
                throw new RuntimeException("未知的手机类型");
        }
        phone.test();
        phone.packingPhone();
        return phone;
    }

    public static void main(String[] args) {
        PhoneStore store = new PhoneStore();
        Phone xiaomi = store.order("小米");
        Phone huawei = store.order("华为");
    }
}
```

输出：

```
i was 小米 phone!
testing....
packing....
i was 华为 phone!
testing....
packing....
```

看起来还不错，但是当我们需要增加一个三星手机的时候，我们需要在order方法里面，重新写一个case，用来创建三星手机。当我们不再出售小米手机的时候，我们需要将代码删除掉。

而手机打包，测试这样的业务是不变的，只有创建产品的逻辑在变。如果将一直变的逻辑和不变的逻写在一起，频繁改动，违反了对修改关闭，对扩展开放原则，导致时间越久越难以维护。

简单工厂

现在尝试用简单工厂 **Simple Factory Method** 实现上面的逻辑，代码如下：

1.增加一个工厂接口

```
public interface IFactory {
    Phone create(String name);
}
```

2.手机工厂实现

```
public class PhoneFactory implements IFactory {
    private static final PhoneFactory INSTANCE = new PhoneFactory();

    public static PhoneFactory getInstance() {
        return INSTANCE;
    }

    @Override
    public Phone create(String name) {
        switch (name) {
```

```
            case "小米":
                return new XiaoMiPhone();
            case "华为":
                return new HuaWeiPhone();
            default:
                throw new RuntimeException("未知的手机类型");
        }
    }
}
```

调整后的手机店

```
public class PhoneStore {
    private final IFactory factory = PhoneFactory.getInstance();

    public Phone order(String name) {
        Phone phone = factory.create(name);
        phone.test();
        phone.packingPhone();
        return phone;
    }

    public static void main(String[] args) {
        PhoneStore store = new PhoneStore();
        Phone xiaomi = store.order("小米");
        Phone huawei = store.order("华为");
    }
}
```

输出：

```
i was 小米 phone!
testing....
packing....
i was 华为 phone!
testing....
packing....
```

可以看到调整后的代码，输出结果没有变化，现在手机店只需要将顾客购买的手机名字传入即可，不需要负责生产具体手机的逻辑了，这样只要生产手机的步骤不变，那么这里就一直不会修改。

我们用一个专门用来生产手机的工厂，这里实现生产手机的具体业务。

我们不需要将生产手机具体逻辑暴露出去，只需要提供一个接口方法即可。如果我们修改出售手机的牌，只改动这里，其它不需要修改，扩展的时候也只需要新增一个实现即可，做到了对修改关闭，对展开放。

假设我们的手机店，每个品牌的手机都需要自己的专卖店，那上面的代码就不能满足要求了。

下面修改一下工厂的实现：

现在我们将手机店改为抽象类，将生产手机的方法改为抽象方法。

工厂模式

```
public abstract class PhoneStore {

    abstract Phone create(String name);

    public Phone order(String name) {
        Phone phone = create(name);
        phone.test();
        phone.packingPhone();
        return phone;
    }

    public static void main(String[] args) {
        PhoneStore xiaomi =new XiaoMiPhoneStore();
        PhoneStore huawei =new HuaWeiPhoneStore();

        xiaomi.order("老人机");
        xiaomi.order("学生机");
        huawei.order("学生机");
        huawei.order("老人机");
    }
}
```

小米的手机店：

```
public class XiaoMiPhoneStore extends PhoneStore {

    @Override
    public Phone create(String name) {
        switch (name) {
            case "老人机":
                return new XiaoMiOldPhone();
            case "学生机":
                return new XiaoMiStudentPhone();
            default:
                throw new RuntimeException("未知的手机类型");
        }
    }
}
```

小米手机店出售的老人机

```
public class XiaoMiOldPhone extends Phone {
    @Override
    public void init() {
        name = "小米老人机";
    }
}
```

小米手机店出售的学生机：

```
public class XiaoMiStudentPhone extends Phone {
    @Override
```

```
    public void init() {
        name = "小米学生机";
    }
}
```

华为手机店：

```
public class HuaWeiPhoneStore extends PhoneStore {

    @Override
    public Phone create(String name) {
        switch (name) {
            case "老人机":
                return new HuaWeiOldPhone();
            case "学生机":
                return new HuaWeiStudentPhone();
            default:
                throw new RuntimeException("未知的手机类型");
        }
    }
}
```

华为老人机：

```
public class HuaWeiOldPhone extends Phone {
    @Override
    public void init() {
        name = "华为老人机";
    }
}
```

华为学生机

```
public class HuaWeiStudentPhone extends Phone {
    @Override
    public void init() {
        name = "华为学生机";
    }
}
```

执行测试代码输出：

```
i was 小米老人机!
testing....
packing....
i was 小米学生机!
testing....
packing....
i was 华为学生机!
testing....
packing....
i was 华为老人机!
testing....
packing....
```

可以看到，现在我们有两个专卖店，小米和华为，可以出售自己品牌的老人机和学生机，而我们不需改动抽象的手机店，只要去指定的专卖店即可。

**

**

建造型-抽象工厂模式

抽象工厂模式 Abstract Factory ，工厂模式隐藏创建对象的过程，只能生产一种产品当需要一个生产一族产品的时候，就需要用到抽象工厂模式。

应用场景：生产一族产品

关键点：用一个抽象工厂，实现生产一族产品，不指定具体产品实现。

优点：具体的工厂中，可以有自己的实现，抽象工厂实现了，相同的生产不同产品，可以有自己的逻。

缺点：产品族扩展困难。

上面的小米手机店和华为手机店可以生产自己的老人机和学生机，由于工厂越来越大，我们无法约束己手机店用料，为了让我们的框架更加强大，我们现在加入原料工厂，我们每个品牌生产的手机，必使用指定的原材料。

示例代码如下：

手机类增加电池和屏幕

```
public abstract class Phone {
    String name;
    Battery battery;
    Screen screen;

    public abstract void init();

    public void test() {
        System.out.println("i was " + name + "!");
        System.out.println("battery is " + battery.name);
        System.out.println("screen is " + screen.name);
        System.out.println("testing....");
    }

    public void packingPhone() {
        System.out.println("packing....");
    }
}
```

对于上面的电池和屏幕分别加入实现：

```
public abstract class Battery {
    String name;

    public Battery() {
        init();
```

```
    }

    abstract void init();
}

public class PinShengBattery extends Battery {
    @Override
    void init() {
        name ="PinSheng Battery";
    }
}

public class SamsungBattery extends Battery {
    @Override
    void init() {
        name="Samsung Battery";
    }
}

public abstract class Screen {
    String name;

    public Screen() {
        init();
    }

    abstract void init();
}

public class JDFScreen extends Screen {
    @Override
    void init() {
        name ="JDF Screen";
    }
}

public class SamsungScreen extends Screen {
    @Override
    void init() {
        name ="Samsung Screen";
    }
}
```

原料工厂接口类及实现类

```
public interface PhoneMaterialFactory {
    Screen createScreen();
    Battery createBattery();
}

public class XiaoMiPhoneMaterialFactory implements PhoneMaterialFactory {
    @Override
    public Screen createScreen() {
        return new SamsungScreen();
    }
```

```
    @Override
    public Battery createBattery() {
        return new PinShengBattery();
    }
}

public class HuaWeiPhoneMaterialFactory implements PhoneMaterialFactory{
    @Override
    public Screen createScreen() {
        return new JDFScreen();
    }

    @Override
    public Battery createBattery() {
        return new SamsungBattery();
    }
}
```

抽象手机店类及实现修改如下：

```
public abstract class PhoneStore {

    abstract Phone create(String name);

    public Phone order(String name) {
        Phone phone = create(name);
        phone.init();
        phone.test();
        phone.packingPhone();
        return phone;
    }

    public static void main(String[] args) {
        PhoneStore xiaomi =new XiaoMiPhoneStore();
        PhoneStore huawei =new HuaWeiPhoneStore();

        xiaomi.order("老人机");
        xiaomi.order("学生机");
        huawei.order("学生机");
        huawei.order("老人机");
    }
}

public class XiaoMiPhoneStore extends PhoneStore {

    @Override
    public Phone create(String name) {
        PhoneMaterialFactory phoneMaterialFactory =new XiaoMiPhoneMaterialFactory();
        switch (name) {
            case "老人机":
                return new XiaoMiOldPhone(phoneMaterialFactory);
            case "学生机":
                return new XiaoMiStudentPhone(phoneMaterialFactory);
```

```
            default:
                throw new RuntimeException("未知的手机类型");
        }
    }
}

public class HuaWeiPhoneStore extends PhoneStore {

    @Override
    public Phone create(String name) {
        PhoneMaterialFactory phoneMaterialFactory = new HuaWeiPhoneMaterialFactory();
        switch (name) {
            case "老人机":
                return new HuaWeiOldPhone(phoneMaterialFactory);
            case "学生机":
                return new HuaWeiStudentPhone(phoneMaterialFactory);
            default:
                throw new RuntimeException("未知的手机类型");
        }
    }
}
```

测试代码输出如下：

```
i was 小米老人机!
battery is PinSheng Battery
screen is Samsung Screen
testing....
packing....
i was 小米学生机!
battery is PinSheng Battery
screen is Samsung Screen
testing....
packing....
i was 华为学生机!
battery is Samsung Battery
screen is JDF Screen
testing....
packing....
i was 华为老人机!
battery is Samsung Battery
screen is JDF Screen
testing....
packing....
```

现在的手机店可以按着我们指定的原材料进行生产手机，当我们需要增加原材料的实现时，只需要实对应的接口即可。

以上就是抽象工厂的示例代码，抽象工厂用于定义一族产品的生产制作过程。

总结：

工厂方法和抽象工厂方法是很常用的设计模式，在一些框架中如jdk，spring中大量使用了工厂模式工厂模式使用灵活，扩展方便，结构清晰，非常值得我们学习。

代码比较粗糙，有问题欢迎指正，共同学习。

设计模式不是为了生搬硬套，应该根据实际情况，灵活的运用，后面会介绍更多的设计模式和设计原，欢迎大家一起讨论学习。

欢迎关注我们