一 工厂模式介绍

1.1 工厂模式的定义

先来看一下GOF为工厂模式的定义：

> “Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.”（在基类中定义创建对象的一个接口，让子类决定实例化哪个类。工厂方法让一个类的实例化延迟到子类中进行。）

1.2 工厂模式的分类：

（1）简单工厂（Simple Factory）模式，又称静态工厂方法模式（Static Factory Method Pattern）。

（2）工厂方法（Factory Method）模式，又称多态性工厂（Polymorphic Factory）模式或虚拟构造子（Virtual Constructor）模式；

（3）抽象工厂（Abstract Factory）模式，又称工具箱（Kit 或Toolkit）模式。

1.3 在开源框架中的使用

举两个比较常见的例子(我暂时可以准确想到的，当然还有很多很多):

(1)Spring中通过getBean("xxx")获取Bean；

(2) Java消息服务JMS中(下面以消息队列ActiveMQ为例子)

关于消息队列ActiveMQ的使用可以查看：消息队列ActiveMQ的使用详解

```
 // 1、创建一个连接工厂对象，需要指定服务的ip及端口。
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://192.
// 2、使用工厂对象创建一个Connection对象。
Connection connection = connectionFactory.createConnection();
```

1.4 为什么要用工厂模式

(1) **解耦** ： 把对象的创建和使用的过程分开

(2)**降低代码重复:** 如果创建某个对象的过程都很复杂，需要一定的代码量，而且很多地方都要用到，那么就会有很多的重复代码。

(3) **降低维护成本**：由于创建过程都由工厂统一管理，所以发生业务逻辑变化，不需要找到所有需要创建对象B的地方去逐个修正，只需要在工厂里修改即可，降低维护成本。

关于工厂模式的作用，Mark一篇文章：https://blog.csdn.net/lovelion/article/details/7523392

二 简单工厂模式

2.1 介绍

严格的说，简单工厂模式并不是23种常用的设计模式之一，它只算工厂模式的一个特殊实现。简单工厂模式在实际中的应用相对于其他2个工厂模式用的还是相对少得多，因为它只适应很多简单的情况。

最重要的是它违背了我们在概述中说的 **开放-封闭原则** （虽然可以通过反射的机制来避免，后面我们会介绍到）。因为每次你要新添加一个功能，都需要在生switch-case 语句（或者if-else 语句）中去修改代码，添加分支条件。

2.2 适用场景

（1）需要创建的对象较少。

（2）客户端不关心对象的创建过程。

2.3 简单工厂模式角色分配：

1. **工厂(Factory)角色** :简单工厂模式的核心，它负责实现创建所有实例的内部逻辑。工厂类可以被外界直接调用，创建所需的产品对象。
2. **抽象产品(Product)角色** :简单工厂模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。
3. **具体产品(Concrete Product)角色**:简单工厂模式的创建目标，所有创建的对象都是充当这个角色的某个具体类的实例。

2.4 简单工厂实例

创建一个可以绘制不同形状的绘图工具，可以绘制圆形，正方形，三角形，每个图形都会有一个draw()方法用于绘图.

（1）创建Shape接口

```
public interface Shape {
    void draw();
}
```

（2）创建实现该接口的具体图形类

圆形

```
public class Circle implements Shape {
    public Circle() {
        System.out.println("Circle");
    }
    @Override
    public void draw() {
        System.out.println("Draw Circle");
    }
}
```

长方形

```
public class Rectangle implements Shape {
    public Rectangle() {
        System.out.println("Rectangle");
    }
    @Override
    public void draw() {
        System.out.println("Draw Rectangle");
    }
}
```

正方形

```
public class Square implements Shape {
    public Square() {
        System.out.println("Square");
    }

    @Override
    public void draw() {
        System.out.println("Draw Square");
    }
}
```

（3）创建工厂类：

```
public class ShapeFactory {
```

```
    // 使用 getShape 方法获取形状类型的对象
    public static Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
        }
        return null;
    }
}
```

（4）测试方法：

```
public class Test {

    public static void main(String[] args) {

        // 获取 Circle 的对象，并调用它的 draw 方法
        Shape circle = ShapeFactory.getShape("CIRCLE");
        circle.draw();

        // 获取 Rectangle 的对象，并调用它的 draw 方法
        Shape rectangle = ShapeFactory.getShape("RECTANGLE");
        rectangle.draw();

        // 获取 Square 的对象，并调用它的 draw 方法
        Shape square = ShapeFactory.getShape("SQUARE");
        square.draw();
    }
}
```

输出结果：

```
Circle
Draw Circle
Rectangle
Draw Rectangle
Square
Draw Square
```

这样的实现有个问题，如果我们新增产品类的话，就需要修改工厂类中的getShape（）方法，这很明显不符合 **开放-封闭原则** 。

2.5 使用反射机制改善简单工厂

将工厂类改为下面的形式：

```
package factory_pattern;

/**
 * 利用反射解决简单工厂每次增加新了产品类都要修改产品工厂的弊端
 *
 * @author Administrator
 *
 */
public class ShapeFactory2 {
    public static Object getClass(Class<? extends Shape> clazz) {
        Object obj = null;

        try {
            obj = Class.forName(clazz.getName()).newInstance();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }

        return obj;
    }
```

测试方法：

```
package factory_pattern;

public class Test2 {
    public static void main(String[] args) {

        Circle circle = (Circle) ShapeFactory2.getClass(factory_pattern.Circle.class);
        circle.draw();

        Rectangle rectangle = (Rectangle) ShapeFactory2.getClass(factory_pattern.Recta
        rectangle.draw();

        Square square = (Square) ShapeFactory2.getClass(factory_pattern.Square.class);
        square.draw();
```

```
    }

}
```

这种方式的虽然符合了 **开放-关闭原则** ，但是每一次传入的都是产品类的全部路径，这样比较麻烦。如果需要改善的话可以通过 **反射+配置文件** 的形式来改善，这种方式使用的也是比较多的。

3 工厂方法模式

3.1 介绍

工厂方法模式应该是在工厂模式家族中是用的最多模式，一般项目中存在最多的就是这个模式。

工厂方法模式是简单工厂的仅一步深化， 在工厂方法模式中，我们不再提供一个统一的工厂类来创建所有的对象，而是针对不同的对象提供不同的工厂。也就是说 **每个对象都有一个与之对应的工厂** 。

3.2 适用场景

- 一个类不知道它所需要的对象的类：在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类。
- 一个类通过其子类来指定创建哪个对象：在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏
- 将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无需关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

3.3 工厂方法模式角色分配：

1. **抽象工厂(Abstract Factory)角色**：是工厂方法模式的核心，与应用程序无关。任何在模式中创建的对象的工厂类必须实现这个接口。
2. **具体工厂(Concrete Factory)角色** ： 这是实现抽象工厂接口的具体工厂类，包含与应用程序密切相关的逻辑，并且受到应用程序调用以创建某一种产品对象。
3. **抽象产品(AbstractProduct)角色** ： 工厂方法模式所创建的对象的超类型，也就是产品对象的共同父类或共同拥有的接口。
4. **具体产品(Concrete Product)角色** ： 这个角色实现了抽象产品角色所定义的接口。某具体产品有专门的具体工厂创建，它们之间往往一一对应

3.4 工厂方法模式实例

上面简单工厂例子中的图形接口以及相关图像实现类不变。我们只需要增加一个工厂接口以及实现这个接口的工厂类即可。

(1)增加一个工厂接口：

```
public interface Factory {
    public Shape getShape();
}
```

（2）增加相关工厂类:

圆形工厂类

```
public class CircleFactory implements Factory {

    @Override
    public Shape getShape() {
        // TODO Auto-generated method stub
        return new Circle();
    }

}
```

长方形工厂类

```
public class RectangleFactory implements Factory{

    @Override
    public Shape getShape() {
        // TODO Auto-generated method stub
        return new Rectangle();
    }

}
```

圆形工厂类

```
public class SquareFactory implements Factory{

    @Override
    public Shape getShape() {
        // TODO Auto-generated method stub
        return new Square();
    }

}
```

（3）测试:

```
public class Test {

    public static void main(String[] args) {
        Factory circlefactory = new CircleFactory();
        Shape circle = circlefactory.getShape();
        circle.draw();
    }

}
```

输出结果：

```
Circle
Draw Circle
```

4 抽象工厂模式

4.1 介绍

在工厂方法模式中，其实我们有一个潜在意识的意识。那就是我们生产的都是同一类产品。抽象工厂模式是工厂方法的仅一步深化，在这个模式中的工厂类不单单可以创建一种产品，而是可以创建一组产品。

抽象工厂应该是比较最难理解的一个工厂模式了。

4.2 适用场景

- 和工厂方法一样客户端不需要知道它所创建的对象的类。
- 需要一组对象共同完成某种功能时，并且可能存在多组对象完成不同功能的情况。（同属于同一个产品族的产品）
- 系统结构稳定，不会频繁的增加对象。（因为一旦增加就需要修改原有代码，不符合开闭原则）

4.3 抽象工厂方法模式角色分配：

1. **抽象工厂（AbstractFactory）角色**：是工厂方法模式的核心，与应用程序无关。任何在模式中创建的对象的工厂类必须实现这个接口。
2. **具体工厂类（ConreteFactory）角色**：这是实现抽象工厂接口的具体工厂类，包含与应用程序密切相关的逻辑，并且受到应用程序调用以创建某一种产品对象。
3. **抽象产品（Abstract Product）角色**：工厂方法模式所创建的对象的超类型，也就是产品对象的共同父类或共同拥有的接口。
4. **具体产品（Concrete Product）角色**：抽象工厂模式所创建的任何产品对象都是某一个具体产品类的实例。在抽象工厂中创建的产品属于同一产品族，这不同于工厂模式中的工厂只创建单一产品，我后面也会详解介绍到。

。

4.4 抽象工厂的工厂和工厂方法中的工厂有什么区别呢?

抽象工厂是生产一整套有产品的（至少要生产两个产品），这些产品必须相互是有关系或有依赖的，而工厂方法中的工厂是生产单一产品的工厂。

4.5 抽象工厂模式实例

不知道大家玩过穿越火线或者吃鸡这类游戏了吗，游戏中存在各种枪。我们假设现在存在AK、M4A1两类枪，每一种枪对应一种子弹。我们现在这样考虑生产AK的工厂可以顺便生产AK使用的子弹，生产M4A1的工厂可以顺便生产M4A1使用的子弹。（AK工厂生产AK系列产品包括子弹啊，AK枪的类型啊这些，M4A1工厂同理）

（1）创建相关接口：

枪

```
public interface Gun {
    public void shooting();
}
```

子弹

```
public interface Bullet {
    public void load();
}
```

（2）创建接口对应实现类：

AK类

```
public class AK implements Gun{

    @Override
    public void shooting() {
        System.out.println("shooting with AK");

    }

}
```

M4A1类

```
public class M4A1 implements Gun {

    @Override
    public void shooting() {
        System.out.println("shooting with M4A1");

    }

}
```

AK子弹类

```
public class AK_Bullet implements Bullet {

    @Override
    public void load() {
        System.out.println("Load bullets with AK");
    }

}
```

M4A1子弹类

```
public class M4A1
_Bullet implements Bullet {

    @Override
    public void load() {
        System.out.println("Load bullets with M4A1");
    }

}
```

（3）创建工厂接口

```
public interface Factory {
    public Gun produceGun();
    public Bullet produceBullet();
}
```

（4）创建具体工厂

生产AK和AK子弹的工厂

```
public class AK_Factory implements Factory{

    @Override
    public Gun produceGun() {
        return new AK();
    }

    @Override
    public Bullet produceBullet() {
        return new AK_Bullet();
    }

}
```

生产M4A1和M4A1子弹的工厂

```
public class M4A1_Factory implements Factory{

    @Override
    public Gun produceGun() {
        return new M4A1();
    }

    @Override
    public Bullet produceBullet() {
        return new M4A1_Bullet();
    }

}
```

（5）测试

```
public class Test {

    public static void main(String[] args) {

     Factory factory;
     Gun gun;
     Bullet bullet;

     factory =new AK_Factory();
     bullet=factory.produceBullet();
     bullet.load();
     gun=factory.produceGun();
```

```
     gun.shooting();

    }

}
```

输出结果：

```
Load bullets with AK
shooting with AK
```