应用最广泛的模式——工厂方法模式

原创 xyzso1z 最后发布于2019-02-27 23:16:09 阅读数 54 收藏 编辑 展开

1.工厂方法模式介绍

工厂方法模式（Factory Pattern）,是创建型吗设计模式之一。工厂方法模式是一种结构简单的模式，其在我们平时开发中应用广泛，也许你并不知道，但是你已经能够使用了无数次该模式了，如Android中的Activity里的各个生命周期方法，以OnCreate方法为例它就可以看做是一个工厂方法，我们在其中可以构造我们的View并通过setContentView返回给framework处理等。

2.工厂方法模式的定义

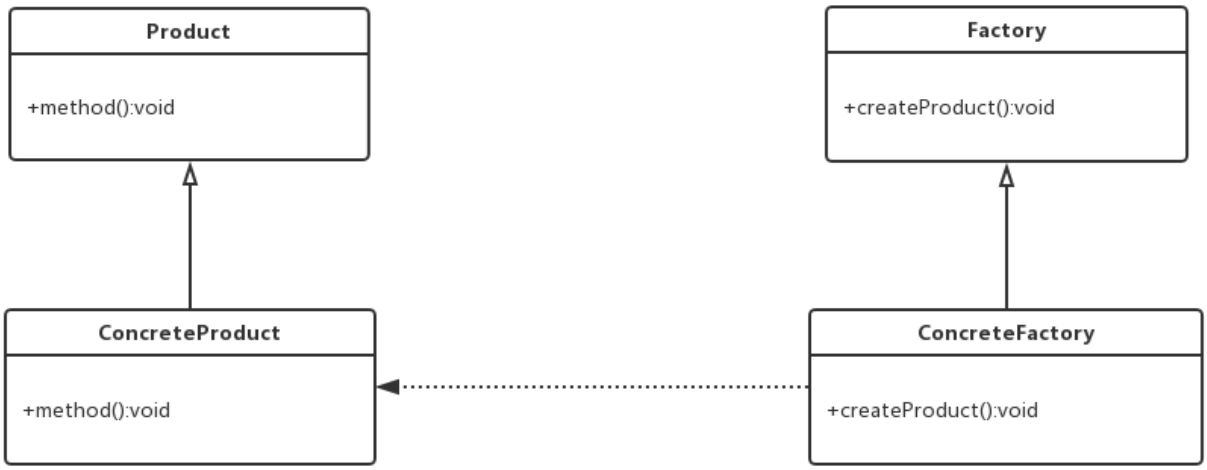
定义一个用于创建对象的接口，让子类决定实例化哪个类。

3.工厂方法模式的使用场景

在任何需要生成复杂对象的地方，都可以使用工厂方法模式。复杂对象适合使用工厂模式，用new就可以完成创建的对象无需使用工厂模式。

4.工厂方法模式的UML类图

UML类图如图：



根据上图我们可以得到一个工厂方法模式的通用模式代码；
抽象产品类：

```
public abstract class Product {
        /*
         * 产品类的抽象方法 由具体的产品类去实现
         */
        public abstract void method();
}
```

具体产品A:

```
public class ConcreteProductA extends Product{

        @Override
        public void method() {
                System.out.println("我是具体的产品A");
        }

}
```

具体产品B:

```


public class ConcreteProductB extends Product{

        @Override
        public void method() {
                System.out.println("我是具体的产品B");
        }

}
```

抽象工厂类:

```
public abstract class Factory {
        /*
         * 抽象工厂方法
         *
         * 具体生产什么由子类去实现
         *
         * @return 具体的产品对象
         */
        public abstract Product createProduct();
}
```

具体工厂类：

```
public class ConcreteFactory extends Factory {

        @Override
        public Product createProduct() {
                return new ConcreteProductA();
        }

}
```

客户端：

```
public class Client {
        public static void main(String[] args) {
          Factory factory=new ConcreteFactory();
          Product product=factory.createProduct();
          product.method();
        }
}
```

这里的几个角色都很简单，主要分为四大模块，一是抽象工厂，其为工厂方法模式的核心；二是具体工厂，其实现了具体的业务逻辑；三是抽象产品，是工厂方法模式所创建的产品父类；四是具体产品，为是实现抽象产品的某个具体产品的对象。
上述的代码中我们在Client 类中构造了一个工厂对象，并通过其生产了一个产品对象，这里我们得到的产品对象是ConcreteProductA的实例，如果想得到ConcreteProductB的实例，更改ConcreteFactory中的逻辑即可：

```
public class ConcreteFactoryB extends Factory {

        @Override
        public Product createProduct() {
                return new ConcreteProductB();
        }

}
```

这种方式比较常见，需要哪种产品生产哪一种产品，有时候也可以利用反射的方式更简洁地来生产具体产品对象，此时，需要在工厂方法的参数列表中传入一个Class类决定是哪一个产品类:

```
public abstract class Factory {
        /*
         * 抽象工厂方法
         *
         * 具体生产什么由子类去实现
         *
         * @return 具体的产品对象
         */
        public abstract <T extends Product> T createProduct(Class<T> clz);
}

```

对于具体的工厂类，则通过反射获取类的实例即可：

```
 public class ConcreteFactory extends Factory {

        @Override
        public <T extends Product> T createProduct(Class<T> clz) {
                Product product = null;
                try {
                        product = (Product) Class.forName(clz.getName()).newInstance();
                } catch (Exception e) {
                        // TODO: handle exception
                }
                return product;
        }

}
```

最后看看Client中的实现：

```
public class Client {
        public static void main(String[] args) {
          Factory factory=new ConcreteFactory();
          Product product=factory.createProduct(ConcreteProductA.class);
          product.method();

        }
}

```

需要哪一个类的对象就传入哪一个类的类型即可，这种方法比较简洁、动态，如果你不喜欢这种方法，也可以尝试为每一个产品都定义一个具体的工厂，各司其职。

```
public class ConcreteFactoryA extends Factory {

        @Override
        public Product createProduct() {
                return new ConcreteProductA();
```

```
        }

}

public class ConcreteFactoryB extends Factory {

        @Override
        public Product createProduct() {
                return new ConcreteProductB();
        }

}

public class Client {
        public static void main(String[] args) {
          Factory factoryA=new ConcreteFactoryA();
          Product productA=factoryA.createProduct();
          productA.method();

          Factory factoryB=new ConcreteFactoryB();
          Product productB=factoryB.createProduct();
          productB.method();
        }
}
```

像这样拥有多个工厂的方式我们称为多工厂方法模式，同样地，回到我们最初的那个工厂方法模式，当我们的工厂只有一个的时候，我们还是为工厂提供一个抽象类，那么，我们是否可以将其简化掉呢?如果确实你的工厂类只有一个，那么简化掉抽象类肯定没有问题的，我们只需要将对应的工厂方法静态方法即可：

```
public class  Factory {

        @Override
        public static Product createProduct() {
                return new ConcreteProductA();
        }

}

```

像这种方式又称为简单工厂模式或静态工厂模式，它是工厂模式的一个弱化版本。
工厂方法模式是完全符合设计原则的，起降低了对象之间的耦合，而且，工厂方法模式依赖于抽象的结构，其将实例化的任务交由子类去完成，有非常好的扩展性。

5.模式的简单实现

工厂方法模式对于大家来说是非常好理解的一个模式，这里以一个生活中的小例子来说明。小明是一家汽车厂的厂长，对他来说，组装汽车没有什么好神秘的，无非就是将一些进口的核心部件，比如发动机和一些国内的零部件组装起来，小明的汽车厂主要就是组装某款SUV车型，比如Q3、Q5、Q7，对于这类车型来说，内部结构差异并不是很大，因此，对小明来说，一条生产线足以应付这3种车型，对于该类生产线小明提供了一个抽象类定义：

```
//工厂基础类
public abstract class AudiFactory {
     /*
     *某车型的工厂方法
     *
     *@param clz 具体的SUV型号类型
     *@return 具体型号的SUV车对象
     */
        public abstract <T extends AudiCar> T createAudiCar(Class<T> clz);
}

```

那么有没有必要为每一种车型都提供一条生产线呢？在这里，小明厂里所生产的3种SUV车型可能在主结构上并没有什么差异，因此，对于小明来说没有必要为每一种车型都提供一条不同的生产线，一条生产线即可：

```
//具体工厂类
public class AudiCarFactory extends AudiFactory {

        @Override
        public <T extends AudiCar> T createAudiCar(Class<T> clz) {
                AudiCar car=null;
                try {
                        car=(AudiCar)Class.forName(clz.getName()).newInstance();
                } catch (InstantiationException e) {
                        e.printStackTrace();
                } catch (IllegalAccessException e) {
                        e.printStackTrace();
                } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                }
                return (T)car;
        }

}

```

对于这3种车型，除了一些车都有的基本共性外，还提供了自动巡航功能，类似于无人驾驶，这些功能小明都使用一个抽象的基类来声明：

```


//汽车抽象类
public abstract class AudiCar {

        /*
         * 汽车的抽象产品类
         *
         * 定义汽车的行为方法 车可以启动开走
         */
        public abstract void drive();
        /*
         * 汽车抽象产品类
         *
         * 定义汽车的一个行为方法 车可以自动巡航
         */
```

```
        public abstract void selfNavigation();
}
```

接下来就是生产每一种具体的车型：

```
public class AudiQ3 extends AudiCar {

        @Override
        public void drive() {
                System.out.println("Q3启动了！");
        }

        @Override
        public void selfNavigation() {
                System.out.println("Q3巡航了！");
        }

}

public class AudiQ5 extends AudiCar {

        @Override
        public void drive() {
                System.out.println("Q5启动了！");
        }

        @Override
        public void selfNavigation() {
                System.out.println("Q5巡航了！");
        }

}

public class AudiQ7 extends AudiCar {

        @Override
        public void drive() {
                System.out.println("Q7启动了！");
        }

        @Override
        public void selfNavigation() {
                System.out.println("Q7巡航了！");
        }

}
```

最后我们将各个类组装起来形成一条完整的流水线：

```
public class Client {

        public static void main(String[] args) {

```

```
                AudiFactory factory=new AudiCarFactory();

                AudiCar audiCarQ3=factory.createAudiCar(AudiQ3.class);
                audiCarQ3.drive();
                audiCarQ3.selfNavigation();

                AudiCar audiCarQ5=factory.createAudiCar(AudiQ5.class);
                audiCarQ5.drive();
                audiCarQ5.selfNavigation();

                AudiCar audiCarQ7=factory.createAudiCar(AudiQ7.class);
                audiCarQ7.drive();
                audiCarQ7.selfNavigation();
        }

}

```

输出：

```
Q3启动了！
Q3巡航了！
Q5启动了！
Q5巡航了！
Q7启动了！
Q7巡航了！
```

xyzso1z
原创文章 80　获赞 40　访问量 2万+