链滴

浅析常用设计模式 -- 创建型

作者：Qiyue0726

原文链接：https://ld246.com/article/1566220638992

来源网站：链滴

单例模式

一. 概念

单例模式是一种常见的设计模式。通过单例模式可以确保某个类**有且仅有**一个对象实例，能够自行实化并向整个系统提供了这个实例。确保所有对象访问的都是同一个实例，避免对资源的多重占用，减系统的性能开销。

注： Spring的bean默认为单例模式

二. 实现方法

1. 将构造函数设置为私有，限制外部实例化该对象
2. 提供一个getInstance方法返回内部创建的实例对象

三. 几种基本写法

1. 饿汉式（线程安全）

饿汉式是最简单的一种实现方式。它在类加载的时候就对对象进行实例，并在整个系统的生命周期中直存在，避免了多线程同步的问题。不过即使系统自始至终都没有使用该对象时也会被创建，就会造内存资源的浪费。

```
public class Singleton {

    //私有化构造函数，使该类不会被实例化
    private Singleton(){}
```

```
    //创建Singleton的一个对象实例
    private static final Singleton instance = new Singleton();

    //获取唯一可用的对象实例
    public static Singleton getInstance(){
        return instance;
    }
}
```

2. 懒汉式（线程不安全）

懒汉式是在系统需要时才会去创建，如果实例已经存在了，那么当再次调用获取实例的接口时将不会次创建对象，而是直接返回之前创建的对象。但是这里的懒汉式没有考虑到线程安全的问题，在多线下可能会并发调用它的getInstance()方法，导致创建多个对象实例。该模式适合单线程下使用。

```
public class Singleton {

    private Singleton(){};

    private static Singleton instance = null;

    public static Singleton getInstance(){
        if (instance == null){
            instance = new Singleton();
        }
        return instance;
    }
}
```

3. 懒汉式（线程安全）

为了解决懒汉式带来的线程安全问题，一般有三种解决办法

为getInstance()添加同步锁

```
public class Singleton {

    private Singleton(){};

    private static Singleton instance = null;

    public static synchronized Singleton getInstance(){
        if (instance == null){
            instance = new Singleton();
        }
        return instance;
    }
}
```

缺点： 效率低下，第一次加载会稍慢，虽然解决了线程同步的问题，但是以后每次调用getInstance()法都会进行同步，造成不必要的资源消耗。

双重检查

```
public class Singleton {

    private Singleton(){};

    private static Singleton instance = null;

    public static Singleton getInstance(){
        if (instance == null){
            synchronized (Singleton.class){
                if (instance == null){
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

优点： 资源利用率高，只有第一次调用getInstance()方法才会上锁并创建对象实例，以后每次调用不再需要进行同步操作，减少了系统的开销。

缺点： 第一次加载会稍慢。

这种写法的亮点主要是在getInstance()方法中进行双重的判断，第一层判断主要是为了避免不必要的步，第二层判断则是为了实例为null时才去创建实例。

静态内部类

```
public class Singleton {


    private Singleton(){};

    public static class SingletonHolder {
        private static final Singleton instance = new Singleton();
    }

    public static Singleton getInstance(){
        return SingletonHolder.instance;
    }
}
```

这是最推荐的一种书写方法，可以同时保证线程安全和懒汉式的延迟加载。

工厂模式

工厂模式是在Java开发中最常用的一种实例化对象的设计模式，它大致可分为三类：简单工厂模式、厂方法模式、抽象工厂模式

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指新创建的对象。

优点：

1. 解耦。将对象的创建和使用分开，用户只需要知道工厂的名称就能得到具体的产品，无需知道产品具体实现过程。

2. 扩展性高。当需要增加新产品时，只需要添加一个具体的产品类和对应的工厂类即可，无需对原工进行任何修改，满足开闭原则。

缺点： 每增加一个新产品就要添加一个具体的产品类和对应的工厂类，增加系统的复杂度。

在面向对象编程领域中，**开闭原则**规定“软件中的对象（类，模块，函数等等）应该**对于扩展是开放，但是对于修改是封闭的**”

一. 简单工厂模式

1. 概念

简单工厂模式又称为静态工厂方法模式。它只有一个工厂类和一个产品抽象类，当新增一个产品时，需要对工厂类进行修改。

一个工厂可以生产多种产品。客户需要知道传入工厂类的参数，不关心产品如何生产。

简单工厂模式并不是23种设计模式之一，只能算是工厂模式的一个特殊实现。因为它违背了**开闭原则**。

2. 角色类

- 抽象产品类。抽象一个产品基类（或接口），定义产品的某些特征。
- 具体产品类。继承抽象产品类，创建具体需要的产品的实例。
- 简单工厂类。简单工厂模式的核心，根据需求创建（调用）具体的产品对象。

3.实例

抽象产品类

```
package SimpleFactory;

public abstract class Product {

    //描述产品的特征
    public abstract void getInformation();
}
```

具体产品类A、B

```
package SimpleFactory;

public class ProductA extends Product {

    public void getInformation() {
        System.out.println("产品A");
    }
}
```

```
package SimpleFactory;

public class ProductB extends Product {

    public void getInformation() {
        System.out.println("产品B");
    }
}
```

简单工厂类

```
package SimpleFactory;

public class Factory {

    public static final int TYPE_A = 1;
    public static final int TYPE_B = 2;

    public static Product createProduct(int type) {
        switch (type){
            case TYPE_A:
                return new ProductA();
            case TYPE_B:
                return new ProductB();
            default:
                System.out.println("没有该产品！");
                return null;
        }
    }
}
```

测试

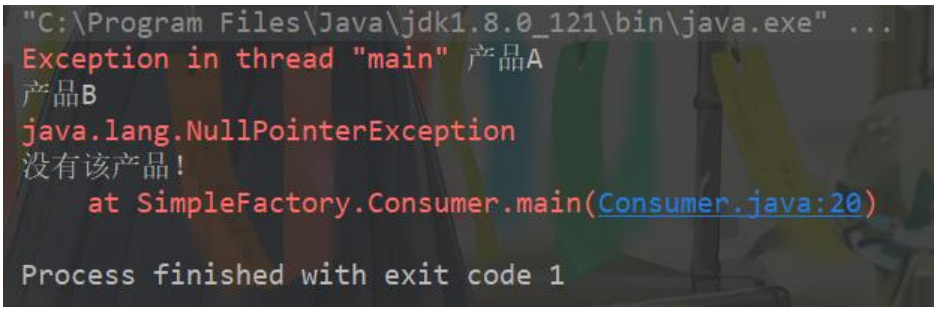
```
package SimpleFactory;

public class Consumer {

    public static void main (String args[]) {
        Product productA = Factory.createProduct('A');
        productA.getInformation();

        Product productB = Factory.createProduct('B');
        productB.getInformation();

        //不存在C产品,会空指针报错
        Product productC = Factory.createProduct('C');
        productC.getInformation();
    }
}
```



二. 工厂方法模式

1. 概念

工厂方法模式又称为多态性工厂模式。它有多个工厂类和一个产品抽象类。完全实现了开闭原则。

一个工厂只生产一个产品。客户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，具体产品对象由具体工厂类创建。

在工厂方法模式中，核心的工厂类提取成为一个抽象工厂类，由对应产品的子类工厂负责创建对应的品类对象。抽象工厂类只负责定义子类工厂需要实现的接口，不涉及具体产品类实例化的细节。

2. 角色类

- 抽象工厂类。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个接口。定义工厂类的方法，用来创建产品类。在Java中由抽象类或接口实现。
- 具体工厂类。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。用于创建对应的品对象。
- 抽象产品类。定义产品公有的属性、方法。在Java中由抽象类或接口实现。
- 具体产品类。实现对应产品类的属性、方法。

3. 实例

抽象产品类

```
package FactoryMethod;

public interface Product {

    void getInformation();

}
```

具体产品类A、B

```
package FactoryMethod;

public class ProductA implements Product{

    public void getInformation() {
        System.out.println("产品A");
    }
```

```
}
```

```
package FactoryMethod;

public class ProductB implements Product{

    public void getInformation() {
        System.out.println("产品B");
    }
}
```

抽象工厂类

```
package FactoryMethod;

import SimpleFactory.Product;

public interface Factory {

    Product createProduct();

}
```

具体工厂类A、B

```
package FactoryMethod;

public class FactoryA implements Factory {

    public Product createProduct() {
        return new ProductA();
    }
}
```

```
package FactoryMethod;

public class FactoryB implements Factory {

    public Product createProduct() {
        return new ProductB();
    }
}
```

测试

```
package FactoryMethod;

public class Consumer {

    public static void main(String[] args){

        Factory factoryA = new FactoryA();
        Product productA = factoryA.createProduct();
        productA.getInformation();
```

```
        Factory factoryB = new FactoryB();
        Product productB = factoryB.createProduct();
        productB.getInformation();

    }
}
```

三. 抽象工厂模式

1. 概念

抽象工厂模式又称为工具箱模式。它有多个工厂类和多个产品类。不符合开闭原则。

一个工厂可以生产一组产品。提供一个创建一系列或相互依赖的对象的接口，而无需指定它们具体的。

缺点： 添加新的产品对像时，难以扩展抽象工厂以便生产新种类的产品。

区别： 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的（同属于同一个产品族的产品是在一起使用的），而工厂方法中的工厂是生产单一产品的工厂。

2. 角色类

● 抽象工厂类。抽象工厂模式的核心，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个接口。在Java中由抽象类或接口实现。

● 具体工厂类。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。用于创建对应的品对象。

● 抽象产品类。定义产品公有的属性、方法。在Java中由抽象类或接口实现。

● 具体产品类。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。

3.实例

抽象产品类A、B

```
package AbstractFactory;

public interface ProductA {

}


package AbstractFactory;
```

```
public interface ProductB {

}
```

具体产品类A1、A2、B1、B2

```
package AbstractFactory;

public class ProductA1 implements ProductA {

    public ProductA1(){
        System.out.println("产品A1");
    }
}

package AbstractFactory;

public class ProductA2 implements ProductA {

    public ProductA2(){
        System.out.println("产品A2");
    }
}

package AbstractFactory;

public class ProductB1 implements ProductB {

    public ProductB1(){
        System.out.println("产品B1");
    }
}

package AbstractFactory;

public class ProductB2 implements ProductB {

    public ProductB2(){
        System.out.println("产品B2");
    }
}
```

抽象工厂类

```
package AbstractFactory;

public interface Factory {

    ProductA createProductA();

    ProductB createProductB();
}
```

具体工厂类1、2

```
package AbstractFactory;

/**
 * @ClassName: FactoryA
 * @Description: 生成产品A1、B1
 */
public class Factory1 implements Factory {
    public ProductA createProductA() {
        return new ProductA1();
    }

    public ProductB createProductB() {
        return new ProductB1();
    }
}


package AbstractFactory;

/**
 * @ClassName: Factory2
 * @Description: 生成产品A2、B2
 */
public class Factory2 implements Factory {
    public ProductA createProductA() {
        return new ProductA2();
    }

    public ProductB createProductB() {
        return new ProductB2();
    }
}
```

测试

```
package AbstractFactory;

public class Consumer {

    public static void main(String [] args){
        Factory1 factory1 = new Factory1();
        factory1.createProductA();
        factory1.createProductB();

        Factory2 factory2 = new Factory2();
        factory2.createProductA();
        factory2.createProductB();
    }
}
```

文章代码已上传至Github

