设计模式

在软件开发过程中，针对反复出现的问题所总结归纳出的通用解决方案。
常见的设计模式有二十三种

单例模式

即确保这个类只new一个对象，从而避免资源重复消耗。

饿汉式

在类加载时就会创建单例实例，无论之后是否会使用该实例。
缺点：可能造成线程浪费，如果没有被使用的话

```
public class EagerSingleton {
// 在类加载时就创建单例实例
private static final EagerSingleton instance = new EagerSingleton();

// 私有构造函数，防止外部实例化
private EagerSingleton() {}

// 提供全局访问点
public static EagerSingleton getInstance() {
    return instance;
}
}
```

懒汉式

在用的时候才创建对象

非线程安全

如果多个线程同时调用 getInstance() 方法，可能会创建多个实例，就没有保证单例

```
public class LazySingleton {
private static LazySingleton instance;

private LazySingleton() {}

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

线程安全

```
public class ThreadSafeLazySingleton {
// 使用 volatile 关键字保证可见性
private static volatile ThreadSafeLazySingleton instance;

private ThreadSafeLazySingleton() {}

public static ThreadSafeLazySingleton getInstance() {
    if (instance == null) {
        // 第一次检查，提高性能
        synchronized (ThreadSafeLazySingleton.class) {
            if (instance == null) {
                // 第二次检查，确保线程安全
                instance = new ThreadSafeLazySingleton();
            }
        }
    }
    return instance;
}
}
```

工厂模式

让对象的创建和使用分离，不直接使用new关键字创建对象，降低了代码的耦合度
对象创建比较复杂的时候可以使用

简单工厂模式

```
// 产品接口
interface Product {
void operation();
}

// 具体产品类 A
class ConcreteProductA implements Product {
@Override
public void operation() {
    System.out.println("ConcreteProductA operation");
}
}

// 具体产品类 B
class ConcreteProductB implements Product {
@Override
public void operation() {
```

```
    System.out.println("ConcreteProductB operation");
}
}

// 简单工厂类
class SimpleFactory {
public static Product createProduct(String productType) {
    if ("A".equals(productType)) {
        return new ConcreteProductA();
    } else if ("B".equals(productType)) {
        return new ConcreteProductB();
    }
    return null;
}
}

/ 客户端代码
public class SimpleFactoryClient {
public static void main(String[] args) {
//没有用工厂的话
    Product producA=new ConcreteProductA();
//用了工厂模式之后，就是去工厂拿货
    Product productA = SimpleFactory.createProduct("A");
    productA.operation();

    Product productB = SimpleFactory.createProduct("B");
    productB.operation();
}
}
```

抽象工厂模式

与简单工厂模式相比就是由一个工厂生产多种产品，变为一个工厂生产一种产品，有多少产品就要有多少工厂

```
// 产品接口
interface Product {
void operation();
}

// 具体产品类 A
class ConcreteProductA implements Product {
@Override
public void operation() {
    System.out.println("ConcreteProductA operation");
}
}

// 具体产品类 B
class ConcreteProductB implements Product {
@Override
public void operation() {
    System.out.println("ConcreteProductB operation");
}
}

// 抽象工厂类
abstract class Factory {
public abstract Product createProduct();
}

// 具体工厂类 A，负责创建产品 A
class ConcreteFactoryA extends Factory {
@Override
public Product createProduct() {
    return new ConcreteProductA();
}
}

// 具体工厂类 B，负责创建产品 B
class ConcreteFactoryB extends Factory {
@Override
public Product createProduct() {
    return new ConcreteProductB();
}
}

// 客户端代码
public class FactoryMethodClient {
public static void main(String[] args) {
    Factory factoryA = new ConcreteFactoryA();
    Product productA = factoryA.createProduct();
    productA.operation();

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