抽象工厂模式

题目链接

抽象工厂模式-家具工厂

什么是抽象工厂模式

抽象工厂模式也是一种创建型设计模式，提供了一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类【引用自大话设计模式第15章】

这样的描述似乎理解起来很困难，我们可以把它与【工厂方法模式】联系起来看。

之前我们已经介绍了“工厂方法模式”，那为什么还有要抽象工厂模式呢?

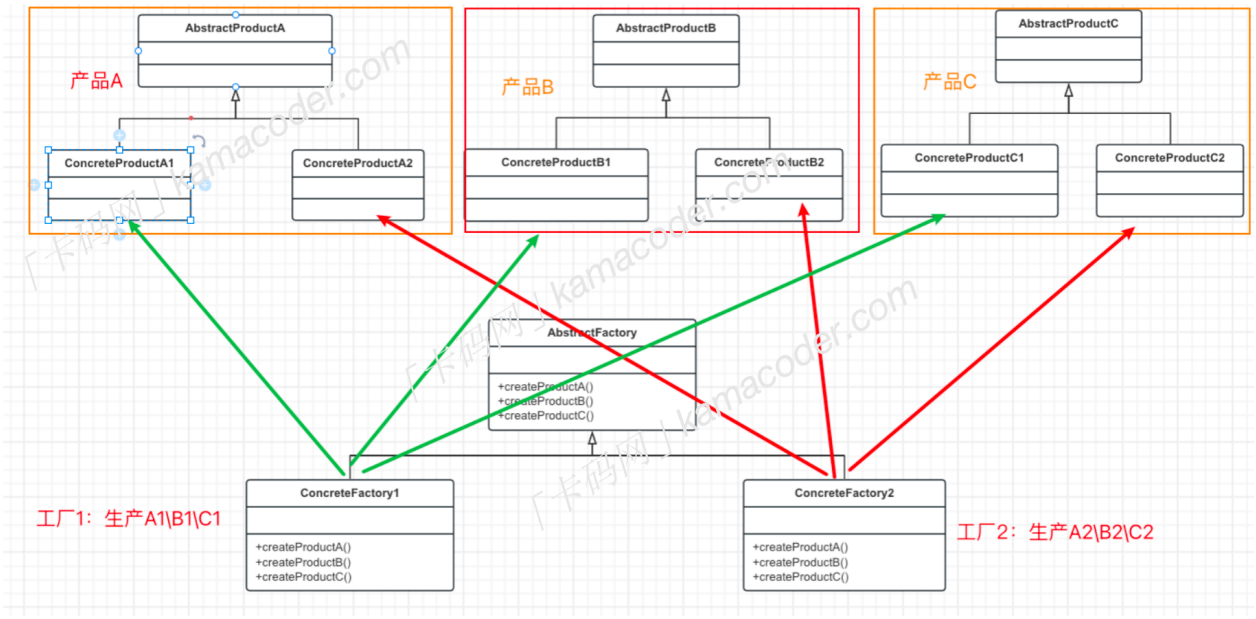
这就涉及到创建“多类”对象了，在工厂方法模式中，每个具体工厂只负责创建单一的产品。但是如果有多类产品呢，比如说“手机”，一个品牌的手机有高端机、中低端机之分，这些具体的产品都需要建立一个单独的工厂类，但是它们都是相互关联的，都共同属于同一个品牌，这就可以使用到【抽象工厂模式】。

抽象工厂模式可以确保一系列相关的产品被一起创建，这些产品能够相互配合使用，再举个例子，有一些家具，比如沙发、茶几、椅子，都具有古典风格的和现代风格的，抽象工厂模式可以将生产现代风格的家具放在一个工厂类中，将生产古典风格的家具放在另一个工厂类中，这样每个工厂类就可以生产一系列的家具。

基本结构

抽象工厂模式包含多个抽象产品接口，多个具体产品类，一个抽象工厂接口和多个具体工厂，每个具体工厂负责创建一组相关的产品。

- 抽象产品接口`AbstractProduct`：定义产品的接口，可以定义多个抽象产品接口，比如说沙发、椅子、茶几都是抽象产品。
- 具体产品类`ConcreteProduct`: 实现抽象产品接口，产品的具体实现，古典风格和沙发和现代风格的沙发都是具体产品。
- 抽象工厂接口`AbstractFactory`：声明一组用于创建产品的方法，每个方法对应一个产品。
- 具体工厂类`ConcreteFactory`： 实现抽象工厂接口，负责创建一组具体产品的对象，在本例中，生产古典风格的工厂和生产现代风格的工厂都是具体实例。



在上面的图示中：`AbstractProductA/B/C` 就是抽象产品，`ConcreteProductA2/A2/B1/B2/C1/C2`就是抽象产品的实现，`AbstractFactory`定义了抽象工厂接口，接口里的方法用于创建具体的产品，而`ConcreteFactory`就是具体工厂类，可以创建一组相关的产品。

基本实现

想要实现抽象工厂模式，需要遵循以下步骤：

- 定义抽象产品接口（可以有多个），接口中声明产品的公共方法。
- 实现具体产品类，在类中实现抽象产品接口中的方法。
- 定义抽象工厂接口，声明一组用于创建产品的方法。
- 实现具体工厂类，分别实现抽象工厂接口中的方法，每个方法负责创建一组相关的产品。
- 在客户端中使用抽象工厂和抽象产品，而不直接使用具体产品的类名。

```
// 1. 定义抽象产品
// 抽象产品A
interface ProductA {
    void display();
}

// 抽象产品B
interface ProductB {
    void show();
}
```

```java
// 2. 实现具体产品类
// 具体产品A1
class ConcreteProductA1 implements ProductA {
    @Override
    public void display() {
        System.out.println("Concrete Product A1");
    }
}

// 具体产品A2
class ConcreteProductA2 implements ProductA {
    @Override
    public void display() {
        System.out.println("Concrete Product A2");
    }
}

// 具体产品B1
class ConcreteProductB1 implements ProductB {
    @Override
    public void show() {
        System.out.println("Concrete Product B1");
    }
}

// 具体产品B2
class ConcreteProductB2 implements ProductB {
    @Override
    public void show() {
        System.out.println("Concrete Product B2");
    }
}

// 3. 定义抽象工厂接口
interface AbstractFactory {
    ProductA createProductA();
    ProductB createProductB();
}
// 4. 实现具体工厂类
// 具体工厂1，生产产品A1和B1
class ConcreteFactory1 implements AbstractFactory {
    @Override
    public ProductA createProductA() {
        return new ConcreteProductA1();
    }
```

```
    @Override
    public ProductB createProductB() {
        return new ConcreteProductB1();
    }
}

// 具体工厂2,生产产品A2和B2
class ConcreteFactory2 implements AbstractFactory {
    @Override
    public ProductA createProductA() {
        return new ConcreteProductA2();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB2();
    }
}

// 客户端代码
public class AbstractFactoryExample {
    public static void main(String[] args) {
        // 使用工厂1创建产品A1和产品B1
        AbstractFactory factory1 = new ConcreteFactory1();
        ProductA productA1 = factory1.createProductA();
        ProductB productB1 = factory1.createProductB();
        productA1.display();
        productB1.show();

        // 使用工厂2创建产品A2和产品B2
        AbstractFactory factory2 = new ConcreteFactory2();
        ProductA productA2 = factory2.createProductA();
        ProductB productB2 = factory2.createProductB();
        productA2.display();
        productB2.show();
    }
}
```

应用场景

抽象工厂模式能够保证一系列相关的产品一起使用，并且在不修改客户端代码的情况下，可以方便地替换整个产品系列。但是当需要增加新的产品类时，除了要增加新的具体产品类，还需要修改抽象工厂接口及其所有的具体工厂类，扩展性相对较差。因此抽象工厂模式特别适用于一系列相关或相互依赖的产品被一起创建的情况，典型的应用场景是**使用抽象工厂模式来创建与不同数据库的连接对象。**

简单工厂、工厂方法、抽象工厂的区别

- 简单工厂模式：一个工厂方法创建所有具体产品
- 工厂方法模式：一个工厂方法创建一个具体产品
- 抽象工厂模式：一个工厂方法可以创建一类具体产品

本题代码

```
import java.util.Scanner;

// 抽象椅子接口
interface Chair {
    void showInfo();
}

// 具体现代风格椅子
class ModernChair implements Chair {
    @Override
    public void showInfo() {
        System.out.println("modern chair");
    }
}

// 具体古典风格椅子
class ClassicalChair implements Chair {
    @Override
    public void showInfo() {
        System.out.println("classical chair");
    }
}

// 抽象沙发接口
interface Sofa {
    void displayInfo();
}

// 具体现代风格沙发
class ModernSofa implements Sofa {
    @Override
    public void displayInfo() {
        System.out.println("modern sofa");
    }
}

// 具体古典风格沙发
```

```
class ClassicalSofa implements Sofa {
    @Override
    public void displayInfo() {
        System.out.println("classical sofa");
    }
}

// 抽象家居工厂接口
interface FurnitureFactory {
    Chair createChair();
    Sofa createSofa();
}

// 具体现代风格家居工厂
class ModernFurnitureFactory implements FurnitureFactory {
    @Override
    public Chair createChair() {
        return new ModernChair();
    }

    @Override
    public Sofa createSofa() {
        return new ModernSofa();
    }
}

// 具体古典风格家居工厂
class ClassicalFurnitureFactory implements FurnitureFactory {
    @Override
    public Chair createChair() {
        return new ClassicalChair();
    }

    @Override
    public Sofa createSofa() {
        return new ClassicalSofa();
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取订单数量
        int N = scanner.nextInt();
```

```
        // 处理每个订单
        for (int i = 0; i < N; i++) {
            // 读取家具类型
            String furnitureType = scanner.next();

            // 创建相应风格的家居装饰品工厂
            FurnitureFactory factory = null;
            if (furnitureType.equals("modern")) {
                factory = new ModernFurnitureFactory();
            } else if (furnitureType.equals("classical")) {
                factory = new ClassicalFurnitureFactory();
            }

            // 根据工厂生产椅子和沙发
            Chair chair = factory.createChair();
            Sofa sofa = factory.createSofa();

            // 输出家具信息
            chair.showInfo();
            sofa.displayInfo();
        }
    }
}
```

其他语言代码

Java

三元运算符简化工厂选择的逻辑。

```
import java.util.Scanner;

// 定义一个家具工厂接口，可以创建椅子和沙发
interface FurnitureFactory{
    Chair createChair();
    Sofa createSofa();
}

// 实现古典家具工厂，生产古典风格的椅子和沙发
class ClassicalFurnitureFactory implements FurnitureFactory{
    @Override
    public Chair createChair(){
        return new ClassicalChair();
    }
```

```
    @Override
    public Sofa createSofa(){
        return new ClassicalSofa();
    }
}

// 实现现代家具工厂，生产现代风格的椅子和沙发
class ModernFurnitureFactory implements FurnitureFactory{
    @Override
    public Chair createChair(){
        return new ModernChair();
    }

    @Override
    public Sofa createSofa(){
        return new ModernSofa();
    }
}

// 定义椅子接口，包含生产椅子的方法
interface Chair{
    void produceChair();
}

// 实现古典椅子类，继承椅子接口
class ClassicalChair implements Chair{
    @Override
    public void produceChair(){
        System.out.println("classical chair");
    }
}

// 实现现代椅子类，继承椅子接口
class ModernChair implements Chair{
    @Override
    public void produceChair(){
        System.out.println("modern chair");
    }
}

// 定义沙发接口，包含生产沙发的方法
interface Sofa{
    void produceSofa();
}

// 实现古典沙发类，继承沙发接口
```

```
class ClassicalSofa implements Sofa{
    @Override
    public void produceSofa(){
        System.out.println("classical sofa");
    }
}

// 实现现代沙发类，继承沙发接口
class ModernSofa implements Sofa{
    @Override
    public void produceSofa(){
        System.out.println("modern sofa");
    }
}

//主程序类
public class Main{
    public static void main (String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 创建古典和现代家具工厂实例
        FurnitureFactory classicalFurnitureFactory = new
ClassicalFurnitureFactory();
        FurnitureFactory modernFurnitureFactory = new
ModernFurnitureFactory();

        // 读取生产次数
        int productionCount = scanner.nextInt();
        scanner.nextLine();

        for (int i = 0; i < productionCount; i++) {
            String type = scanner.nextLine();

             // 根据用户输入选择相应的家具工厂，并创建家具
            FurnitureFactory factory = (type.equals("modern")) ?
modernFurnitureFactory : classicalFurnitureFactory;
            Chair chair = factory.createChair();
            chair.produceChair();
            Sofa sofa = factory.createSofa();
            sofa.produceSofa();
        }
        scanner.close();
    }
}
```

Cpp

```cpp
#include <iostream>
#include <string>

// 抽象椅子接口
class Chair {
public:
    virtual void showInfo() = 0;
};

// 具体现代风格椅子
class ModernChair : public Chair {
public:
    void showInfo() override {
        std::cout << "modern chair" << std::endl;
    }
};

// 具体古典风格椅子
class ClassicalChair : public Chair {
public:
    void showInfo() override {
        std::cout << "classical chair" << std::endl;
    }
};

// 抽象沙发接口
class Sofa {
public:
    virtual void displayInfo() = 0;
};

// 具体现代风格沙发
class ModernSofa : public Sofa {
public:
    void displayInfo() override {
        std::cout << "modern sofa" << std::endl;
    }
};

// 具体古典风格沙发
class ClassicalSofa : public Sofa {
public:
    void displayInfo() override {
        std::cout << "classical sofa" << std::endl;
```

```
    }
};

// 抽象家居工厂接口
class FurnitureFactory {
public:
    virtual Chair* createChair() = 0;
    virtual Sofa* createSofa() = 0;
};

// 具体现代风格家居工厂
class ModernFurnitureFactory : public FurnitureFactory {
public:
    Chair* createChair() override {
        return new ModernChair();
    }

    Sofa* createSofa() override {
        return new ModernSofa();
    }
};

// 具体古典风格家居工厂
class ClassicalFurnitureFactory : public FurnitureFactory {
public:
    Chair* createChair() override {
        return new ClassicalChair();
    }

    Sofa* createSofa() override {
        return new ClassicalSofa();
    }
};

int main() {
    // 读取订单数量
    int N;
    std::cin >> N;

    // 处理每个订单
    for (int i = 0; i < N; i++) {
        // 读取家具类型
        std::string furnitureType;
        std::cin >> furnitureType;

        // 创建相应风格的家居装饰品工厂
```

```
        FurnitureFactory* factory = nullptr;
        if (furnitureType == "modern") {
            factory = new ModernFurnitureFactory();
        } else if (furnitureType == "classical") {
            factory = new ClassicalFurnitureFactory();
        }

        // 根据工厂生产椅子和沙发
        Chair* chair = factory->createChair();
        Sofa* sofa = factory->createSofa();

        // 输出家具信息
        chair->showInfo();
        sofa->displayInfo();

        // 释放动态分配的对象
        delete chair;
        delete sofa;
        delete factory;
    }

    return 0;
}
```

Python

```
from abc import ABC, abstractmethod

# 抽象椅子接口
class Chair(ABC):
    @abstractmethod
    def show_info(self):
        pass

# 具体现代风格椅子
class ModernChair(Chair):
    def show_info(self):
        print("modern chair")

# 具体古典风格椅子
class ClassicalChair(Chair):
    def show_info(self):
        print("classical chair")

# 抽象沙发接口
class Sofa(ABC):
```

```
    @abstractmethod
    def display_info(self):
        pass

# 具体现代风格沙发
class ModernSofa(Sofa):
    def display_info(self):
        print("modern sofa")

# 具体古典风格沙发
class ClassicalSofa(Sofa):
    def display_info(self):
        print("classical sofa")

# 抽象家居工厂接口
class FurnitureFactory(ABC):
    @abstractmethod
    def create_chair(self):
        pass

    @abstractmethod
    def create_sofa(self):
        pass

# 具体现代风格家居工厂
class ModernFurnitureFactory(FurnitureFactory):
    def create_chair(self):
        return ModernChair()

    def create_sofa(self):
        return ModernSofa()

# 具体古典风格家居工厂
class ClassicalFurnitureFactory(FurnitureFactory):
    def create_chair(self):
        return ClassicalChair()

    def create_sofa(self):
        return ClassicalSofa()

def main():
    # 读取订单数量
    N = int(input())

    # 处理每个订单
    for _ in range(N):
```

```
        # 读取家具类型
        furniture_type = input()

        # 创建相应风格的家居装饰品工厂
        factory = None
        if furniture_type == "modern":
            factory = ModernFurnitureFactory()
        elif furniture_type == "classical":
            factory = ClassicalFurnitureFactory()

        # 根据工厂生产椅子和沙发
        chair = factory.create_chair()
        sofa = factory.create_sofa()

        # 输出家具信息
        chair.show_info()
        sofa.display_info()

if __name__ == "__main__":
    main()
```

Go

```
package main

import "fmt"

// 抽象椅子接口
type Chair interface {
    showInfo()
}

// 具体现代风格椅子
type ModernChair struct{}

func (mc *ModernChair) showInfo() {
    fmt.Println("modern chair")
}

// 具体古典风格椅子
type ClassicalChair struct{}

func (cc *ClassicalChair) showInfo() {
    fmt.Println("classical chair")
}
```

```go
// 抽象沙发接口
type Sofa interface {
    displayInfo()
}

// 具体现代风格沙发
type ModernSofa struct{}

func (ms *ModernSofa) displayInfo() {
    fmt.Println("modern sofa")
}

// 具体古典风格沙发
type ClassicalSofa struct{}

func (cs *ClassicalSofa) displayInfo() {
    fmt.Println("classical sofa")
}

// 抽象家居工厂接口
type FurnitureFactory interface {
    createChair() Chair
    createSofa() Sofa
}

// 具体现代风格家居工厂
type ModernFurnitureFactory struct{}

func (mf *ModernFurnitureFactory) createChair() Chair {
    return &ModernChair{}
}

func (mf *ModernFurnitureFactory) createSofa() Sofa {
    return &ModernSofa{}
}

// 具体古典风格家居工厂
type ClassicalFurnitureFactory struct{}

func (cf *ClassicalFurnitureFactory) createChair() Chair {
    return &ClassicalChair{}
}

func (cf *ClassicalFurnitureFactory) createSofa() Sofa {
    return &ClassicalSofa{}
}
```

```
func main() {
    // 读取订单数量
    var N int
    fmt.Scan(&N)

    // 处理每个订单
    for i := 0; i < N; i++ {
        // 读取家具类型
        var furnitureType string
        fmt.Scan(&furnitureType)

        // 创建相应风格的家居装饰品工厂
        var factory FurnitureFactory
        if furnitureType == "modern" {
            factory = &ModernFurnitureFactory{}
        } else if furnitureType == "classical" {
            factory = &ClassicalFurnitureFactory{}
        }

        // 根据工厂生产椅子和沙发
        chair := factory.createChair()
        sofa := factory.createSofa()

        // 输出家具信息
        chair.showInfo()
        sofa.displayInfo()
    }
}
```