工厂方法模式

题目链接

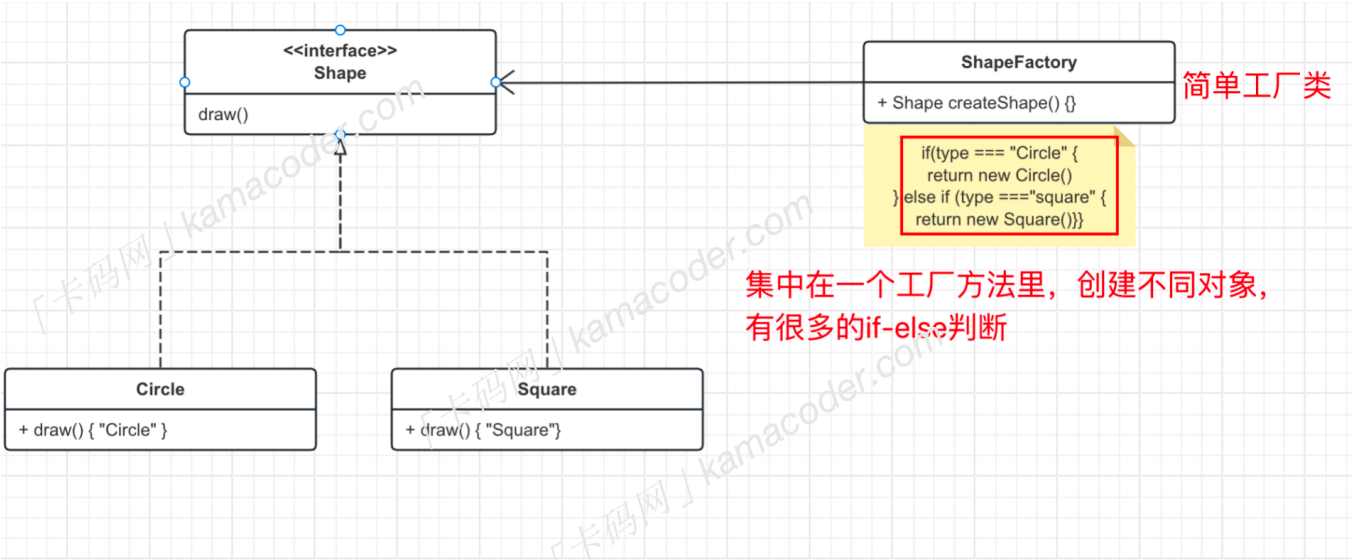
工厂方法模式-积木工厂

简单工厂模式

在了解工厂方法模式之前，有必要对“简单工厂”模式进行一定的了解，简单工厂模式是一种创建型设计模式，但并不属于23种设计模式之一，更多的是一种编程习惯。

简单工厂模式的核心思想是将产品的创建过程封装在一个工厂类中，把创建对象的流程集中在这个工厂类里面。

简单工厂模式包括三个主要角色，工厂类、抽象产品、具体产品，下面的图示则展示了工厂类的基本结构。



- 抽象产品，比如上图中的`Shape`接口，描述产品的通用行为。
- 具体产品：实现抽象产品接口或继承抽象产品类，比如上面的`Circle`类和`Square`类，具体产品通过简单工厂类的`if-else`逻辑来实例化。
- 工厂类：负责创建产品，根据传递的不同参数创建不同的产品示例。

简单工厂类简化了客户端操作，客户端可以调用工厂方法来获取具体产品，而无需直接与具体产品类交互，降低了耦合，但是有一个很大的问题就是不够灵活，如果需要添加新的产品，就需要修改工厂类的代码。

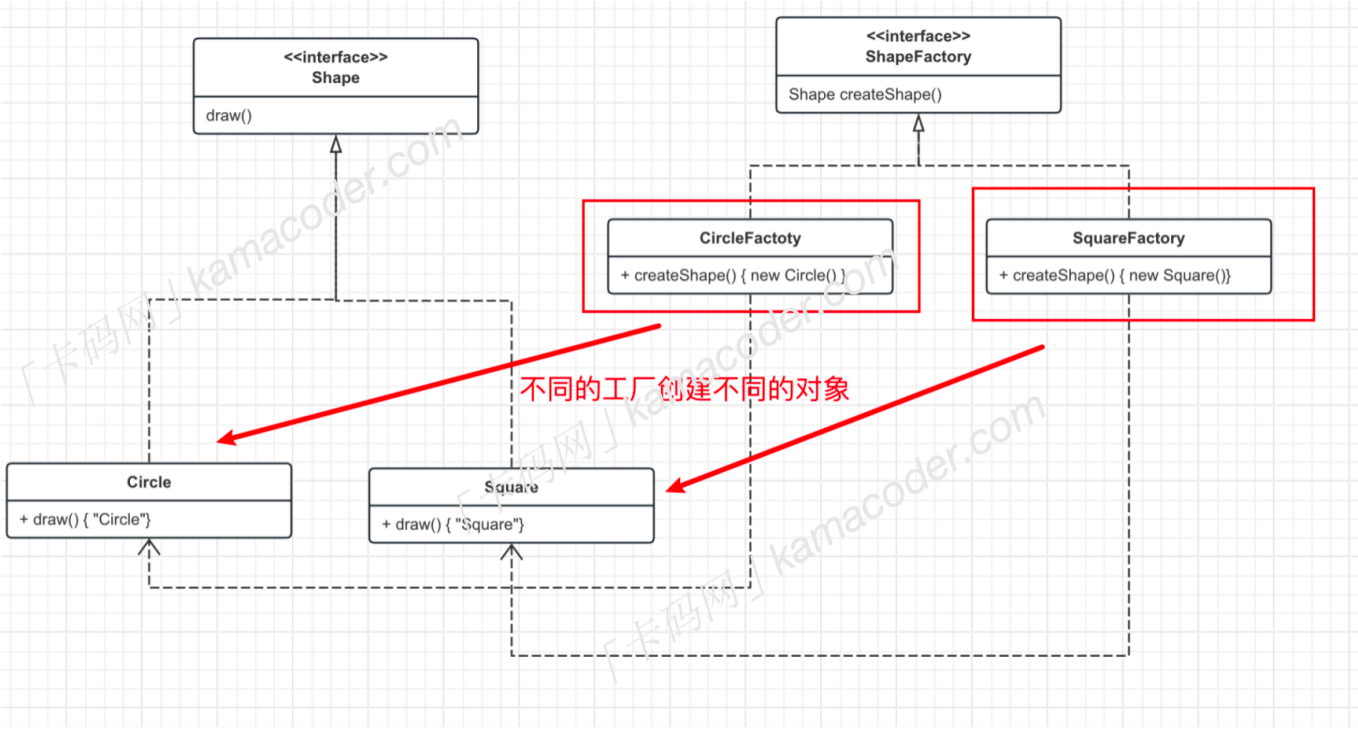
什么是工厂方法模式

工厂方法模式也是一种创建型设计模式，简单工厂模式只有一个工厂类，负责创建所有产品，如果要添加新的产品，通常需要修改工厂类的代码。而工厂方法模式引入了抽象工厂和具体工厂的概念，每个具体工厂只负责创建一个具体产品，添加新的产品只需要添加新的工厂类而无需修改原来的代码，这样就使得产品的生产更加灵活，支持扩展，符合开闭原则。

工厂方法模式分为以下几个角色：

- 抽象工厂：一个接口，包含一个抽象的工厂方法（用于创建产品对象）。
- 具体工厂：实现抽象工厂接口，创建具体的产品。
- 抽象产品：定义产品的接口。
- 具体产品：实现抽象产品接口，是工厂创建的对象。

> *实际上工厂方法模式也很好理解，就拿“手机Phone”这个产品举例，手机是一个抽象产品，小米手机、华为手机、苹果手机是具体的产品实现，而不同品牌的手机在各自的生产厂家生产。*



基本实现

根据上面的类图，我们可以写出工厂方法模式的基本实现。

```
// 抽象产品
interface Shape {
    void draw();
}
```

```java
// 具体产品 - 圆形
class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Circle");
    }
}

// 具体产品 - 正方形
class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Square");
    }
}

// 抽象工厂
interface ShapeFactory {
    Shape createShape();
}

// 具体工厂 - 创建圆形
class CircleFactory implements ShapeFactory {
    @Override
    public Shape createShape() {
        return new Circle();
    }
}

// 具体工厂 - 创建正方形
class SquareFactory implements ShapeFactory {
    @Override
    public Shape createShape() {
        return new Square();
    }
}

// 客户端代码
public class Client {
    public static void main(String[] args) {
        ShapeFactory circleFactory = new CircleFactory();
        Shape circle = circleFactory.createShape();
        circle.draw();  // 输出: Circle

        ShapeFactory squareFactory = new SquareFactory();
```

```
        Shape square = squareFactory.createShape();
        square.draw();  // 输出：Square
    }
}
```

应用场景

工厂方法模式使得每个工厂类的职责单一，每个工厂只负责创建一种产品，当创建对象涉及一系列复杂的初始化逻辑，而这些逻辑在不同的子类中可能有所不同时，可以使用工厂方法模式将这些初始化逻辑封装在子类的工厂中。在现有的工具、库中，工厂方法模式也有广泛的应用，比如：

- Spring 框架中的 Bean 工厂：通过配置文件或注解，Spring 可以根据配置信息动态地创建和管理对象。
- JDBC 中的 Connection 工厂：在 Java 数据库连接中，`DriverManager` 使用工厂方法模式来创建数据库连接。不同的数据库驱动（如 MySQL、PostgreSQL 等）都有对应的工厂来创建连接。

本题代码

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

// 抽象积木接口
interface Block {
    void produce();
}

// 具体圆形积木实现
class CircleBlock implements Block {
    @Override
    public void produce() {
        System.out.println("Circle Block");
    }
}

// 具体方形积木实现
class SquareBlock implements Block {
    @Override
    public void produce() {
        System.out.println("Square Block");
    }
}

// 抽象积木工厂接口
```

```
interface BlockFactory {
    Block createBlock();
}

// 具体圆形积木工厂实现
class CircleBlockFactory implements BlockFactory {
    @Override
    public Block createBlock() {
        return new CircleBlock();
    }
}

// 具体方形积木工厂实现
class SquareBlockFactory implements BlockFactory {
    @Override
    public Block createBlock() {
        return new SquareBlock();
    }
}

// 积木工厂系统
class BlockFactorySystem {
    private List<Block> blocks = new ArrayList<>();

    public void produceBlocks(BlockFactory factory, int quantity) {
        for (int i = 0; i < quantity; i++) {
            Block block = factory.createBlock();
            blocks.add(block);
            block.produce();
        }
    }

    public List<Block> getBlocks() {
        return blocks;
    }
}

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

        // 创建积木工厂系统
        BlockFactorySystem factorySystem = new BlockFactorySystem();

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

```
            scanner.nextLine();

            // 读取每次生产的积木类型和数量
            for (int i = 0; i < productionCount; i++) {
                String[] productionInfo = scanner.nextLine().split(" ");
                String blockType = productionInfo[0];
                int quantity = Integer.parseInt(productionInfo[1]);

                if (blockType.equals("Circle")) {
                    factorySystem.produceBlocks(new CircleBlockFactory(),
quantity);
                } else if (blockType.equals("Square")) {
                    factorySystem.produceBlocks(new SquareBlockFactory(),
quantity);
                }
            }
        }
}
```

其他语言版本

Java

利用反射实现工厂方法

```
import java.lang.reflect.Constructor;
import java.util.Scanner;

// 抽象工厂类
abstract class BlocksFactory {
    public abstract blocks createBlocks(Class<blocks> c, String str, int
num);
}

//具体工厂实现类
class BlocksFactoryImpl extends BlocksFactory {

    @Override
    public blocks createBlocks(Class<blocks> c, String str, int num) {
        try {
        //反射获取blocks类中带有两个参数的构造器
            Constructor<blocks> constructor =
c.getConstructor(String.class, int.class);
            return constructor.newInstance(str, num);
        } catch (Exception e) {
```

```
            e.printStackTrace();
            return null;
        }
    }
}

//定义积木接口
interface blocks {
    void blockPrint(String str, int num);
}

//圆形积木
class CircleBlocks implements blocks{
    private String str;
    private int num;

    public CircleBlocks(String str, int num) {
        this.str = str;
        this.num = num;
    }

    public void blockPrint(String str,int num){
        for(int i=0;i<num;i++){
            System.out.println(str+" Block");
        }
    }
}

//方形积木
class SquareBlocks implements blocks{

    private String str;
    private int num;

    public SquareBlocks(String str, int num) {
        this.str = str;
        this.num = num;
    }

    public void blockPrint(String str,int num){
        for(int i=0;i<num;i++){
            System.out.println(str+" Block");
        }
    }
}
```

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

        //获取工厂实例
        BlocksFactory factory = new BlocksFactoryImpl();

        for (int n = 0; n < N; n++) {
            String type = scanner.next();//用户输入类型（Circle 或 Square）
            int num = scanner.nextInt();// 用户输入数量

            //构建正确的类名，并第一个字母大写
            String className = type.substring(0, 1).toUpperCase() +
type.substring(1) + "Blocks"; // 构造正确的类名
            try {
        //使用工厂方法创建对象
                blocks block = factory.createBlocks((Class<blocks>)
Class.forName(className), type, num);
                if (block != null) {
                    block.blockPrint(type, num);
                }
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }

        scanner.close();
    }
}
```

Cpp

```
#include <iostream>
#include <vector>

// 抽象积木接口
class Block {
public:
    virtual void produce() = 0;
};

// 具体圆形积木实现
class CircleBlock : public Block {
public:
    void produce() override {
```

```
        std::cout << "Circle Block" << std::endl;
    }
};

// 具体方形积木实现
class SquareBlock : public Block {
public:
    void produce() override {
        std::cout << "Square Block" << std::endl;
    }
};

// 抽象积木工厂接口
class BlockFactory {
public:
    virtual Block* createBlock() = 0;
};

// 具体圆形积木工厂实现
class CircleBlockFactory : public BlockFactory {
public:
    Block* createBlock() override {
        return new CircleBlock();
    }
};

// 具体方形积木工厂实现
class SquareBlockFactory : public BlockFactory {
public:
    Block* createBlock() override {
        return new SquareBlock();
    }
};

// 积木工厂系统
class BlockFactorySystem {
private:
    std::vector<Block*> blocks;

public:
    void produceBlocks(BlockFactory* factory, int quantity) {
        for (int i = 0; i < quantity; i++) {
            Block* block = factory->createBlock();
            blocks.push_back(block);
            block->produce();
        }
```

```
    }

    const std::vector<Block*>& getBlocks() const {
        return blocks;
    }

    ~BlockFactorySystem() {
        // 释放所有动态分配的积木对象
        for (Block* block : blocks) {
            delete block;
        }
    }
};

int main() {
    // 创建积木工厂系统
    BlockFactorySystem factorySystem;

    // 读取生产次数
    int productionCount;
    std::cin >> productionCount;

    // 读取每次生产的积木类型和数量
    for (int i = 0; i < productionCount; i++) {
        std::string blockType;
        int quantity;
        std::cin >> blockType >> quantity;

        if (blockType == "Circle") {
            factorySystem.produceBlocks(new CircleBlockFactory(),
quantity);
        } else if (blockType == "Square") {
            factorySystem.produceBlocks(new SquareBlockFactory(),
quantity);
        }
    }

    return 0;
}
```

Python

```
from abc import ABC, abstractmethod

# 抽象积木接口
class Block(ABC):
```

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

# 具体圆形积木实现
class CircleBlock(Block):
    def produce(self):
        print("Circle Block")

# 具体方形积木实现
class SquareBlock(Block):
    def produce(self):
        print("Square Block")

# 抽象积木工厂接口
class BlockFactory(ABC):
    @abstractmethod
    def create_block(self):
        pass

# 具体圆形积木工厂实现
class CircleBlockFactory(BlockFactory):
    def create_block(self):
        return CircleBlock()

# 具体方形积木工厂实现
class SquareBlockFactory(BlockFactory):
    def create_block(self):
        return SquareBlock()

# 积木工厂系统
class BlockFactorySystem:
    def __init__(self):
        self.blocks = []

    def produce_blocks(self, factory, quantity):
        for _ in range(quantity):
            block = factory.create_block()
            self.blocks.append(block)
            block.produce()

    def get_blocks(self):
        return self.blocks

# 主函数
def main():
```

```
    # 创建积木工厂系统
    factory_system = BlockFactorySystem()

    # 读取生产次数
    production_count = int(input())

    # 读取每次生产的积木类型和数量
    for _ in range(production_count):
        block_type, quantity = input().split()
        quantity = int(quantity)

        if block_type == "Circle":
            factory_system.produce_blocks(CircleBlockFactory(), quantity)
        elif block_type == "Square":
            factory_system.produce_blocks(SquareBlockFactory(), quantity)

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

Go

```
package main

import (
    "fmt"
)

// 抽象积木接口
type Block interface {
    produce()
}

// 具体圆形积木实现
type CircleBlock struct{}

func (c *CircleBlock) produce() {
    fmt.Println("Circle Block")
}

// 具体方形积木实现
type SquareBlock struct{}

func (s *SquareBlock) produce() {
    fmt.Println("Square Block")
}
```

```go
// 抽象积木工厂接口
type BlockFactory interface {
    createBlock() Block
}

// 具体圆形积木工厂实现
type CircleBlockFactory struct{}

func (cf *CircleBlockFactory) createBlock() Block {
    return &CircleBlock{}
}

// 具体方形积木工厂实现
type SquareBlockFactory struct{}

func (sf *SquareBlockFactory) createBlock() Block {
    return &SquareBlock{}
}

// 积木工厂系统
type BlockFactorySystem struct {
    blocks []Block
}

func (bfs *BlockFactorySystem) produceBlocks(factory BlockFactory, quantity
int) {
    for i := 0; i < quantity; i++ {
        block := factory.createBlock()
        bfs.blocks = append(bfs.blocks, block)
        block.produce()
    }
}

func (bfs *BlockFactorySystem) getBlocks() []Block {
    return bfs.blocks
}

func main() {
    // 创建积木工厂系统
    factorySystem := &BlockFactorySystem{}

    // 读取生产次数
    var productionCount int
    fmt.Scan(&productionCount)

    // 读取每次生产的积木类型和数量
```

```
    for i := 0; i < productionCount; i++ {
        var blockType string
        var quantity int
        fmt.Scan(&blockType, &quantity)

        if blockType == "Circle" {
            factorySystem.produceBlocks(&CircleBlockFactory{}, quantity)
        } else if blockType == "Square" {
            factorySystem.produceBlocks(&SquareBlockFactory{}, quantity)
        }
    }
}
```