模板方法模式

题目链接

模板方法模式-咖啡馆

基本概念

模板方法模式（Template Method Pattern）是一种行为型设计模式，它定义了一个算法的骨架，将**一些步骤的实现延迟到子类。**模板方法模式使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。【引用自大话设计第10章】

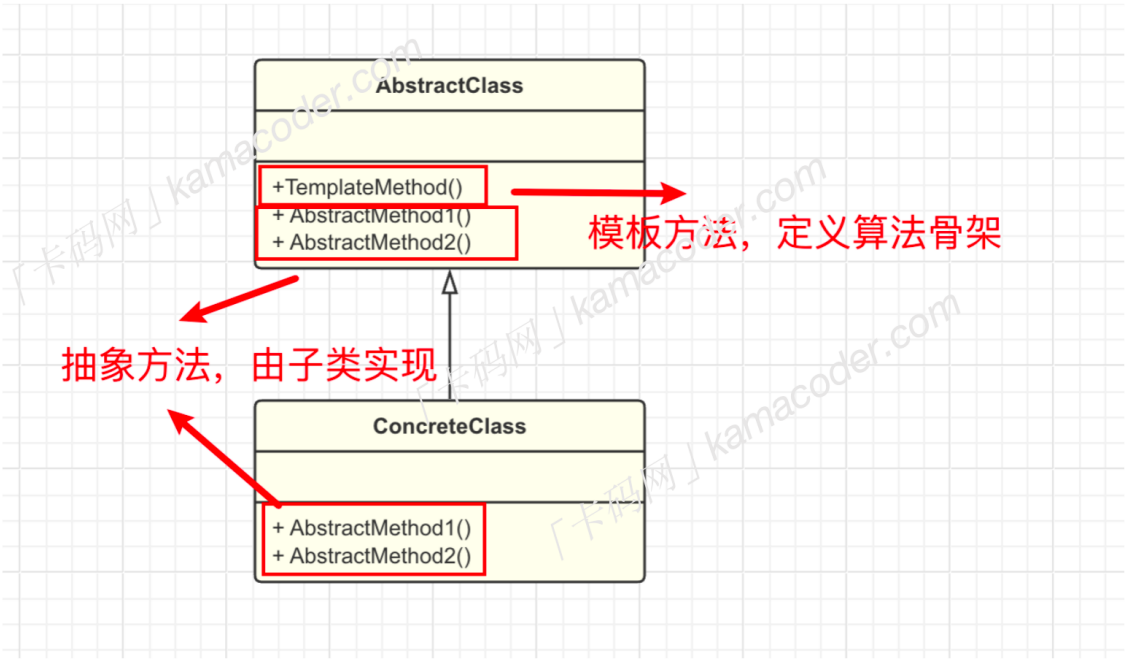
举个简单的例子，做一道菜通常都需要包含至少三步：

- 准备食材
- 亨饪过程
- 上菜

不同菜品的亨饪过程是不一样的，但是我们可以先定义一个”骨架”，包含这三个步骤，亨饪过程的过程放到具体的炒菜类中去实现，这样，无论炒什么菜，都可以沿用相同的炒菜算法，只需在子类中实现具体的炒菜步骤，从而提高了代码的复用性。

基本结构

模板方法模式的基本结构包含以下两个角色：



- 模板类`AbstractClass`：由一个模板方法和若干个基本方法构成，模板方法定义了逻辑的骨架，按照顺序调用包含的基本方法，基本方法通常是一些**抽象方法，这些方法由子类去实现**。基本方法还包含一些具体方法，它们是算法的一部分但已经有默认实现，在具体子类中可以继承或者重写。
- 具体类`ConcreteClass`：继承自模板类，实现了在模板类中定义的抽象方法，以完成算法中特定步骤的具体实现。

简易实现

模板方法模式的简单示例如下：

1. 定义模板类，包含模板方法，定义了算法的骨架，一般都加上`final`关键字，避免子类重写。

```
// 模板类
abstract class AbstractClass {
    // 模板方法，定义了算法的骨架
    public final void templateMethod() {
        step1();
        step2();
        step3();
    }

    // 抽象方法，由子类实现
    protected abstract void step1();
    protected abstract void step2();
    protected abstract void step3();
}
```

2. 定义具体类，实现模板类中的抽象方法

```
// 具体类
class ConcreteClass extends AbstractClass {
    @Override
    protected void step1() {
        System.out.println("Step 1 ");
    }

    @Override
    protected void step2() {
        System.out.println("Step 2 ");
    }

    @Override
```

```
    protected void step3() {
        System.out.println("Step 3");
    }
}
```

3. 客户端实现

```
public class Main {
    public static void main(String[] args) {
        AbstractClass concreteTemplate = new ConcreteClass();
        // 触发整个算法的执行
        concreteTemplate.templateMethod();
    }
}
```

应用场景

模板方法模式将算法的不变部分被封装在模板方法中，而可变部分算法由子类继承实现，这样做可以很好的提高代码的复用性，但是当算法的框架发生变化时，可能需要修改模板类，这也会影响到所有的子类。

总体来说，当算法的整体步骤很固定，但是个别步骤在更详细的层次上的实现可能不同时，通常考虑模板方法模式来处理。在已有的工具和库中， Spring框架中的`JdbcTemplate`类使用了模板方法模式，其中定义了一些执行数据库操作的模板方法，具体的数据库操作由回调函数提供。而在Java的JDK源码中，`AbstractList` 类也使用了模板方法模式，它提供了一些通用的方法，其中包括一些模板方法。具体的列表操作由子类实现。

本题代码

```
import java.util.Scanner;

// 抽象类
abstract class CoffeeMakerTemplate {
    private String coffeeName; // 添加咖啡名称字段

    // 构造函数，接受咖啡名称参数
    public CoffeeMakerTemplate(String coffeeName) {
        this.coffeeName = coffeeName;
    }

    // 模板方法定义咖啡制作过程
    final void makeCoffee() {
        System.out.println("Making " + coffeeName + ":");
        grindCoffeeBeans();
        brewCoffee();
```

```
        addCondiments();
        System.out.println();
    }

    // 具体步骤的具体实现由子类提供
    abstract void grindCoffeeBeans();
    abstract void brewCoffee();

    // 添加调料的默认实现
    void addCondiments() {
        System.out.println("Adding condiments");
    }
}

// 具体的美式咖啡类
class AmericanCoffeeMaker extends CoffeeMakerTemplate {
    // 构造函数传递咖啡名称
    public AmericanCoffeeMaker() {
        super("American Coffee");
    }

    @Override
    void grindCoffeeBeans() {
        System.out.println("Grinding coffee beans");
    }

    @Override
    void brewCoffee() {
        System.out.println("Brewing coffee");
    }
}

// 具体的拿铁咖啡类
class LatteCoffeeMaker extends CoffeeMakerTemplate {
    // 构造函数传递咖啡名称
    public LatteCoffeeMaker() {
        super("Latte");
    }

    @Override
    void grindCoffeeBeans() {
        System.out.println("Grinding coffee beans");
    }

    @Override
    void brewCoffee() {
```

```
        System.out.println("Brewing coffee");
    }

    // 添加调料的特定实现
    @Override
    void addCondiments() {
        System.out.println("Adding milk");
        System.out.println("Adding condiments");
    }
}

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

        while (scanner.hasNext()) {
            int coffeeType = scanner.nextInt();

            CoffeeMakerTemplate coffeeMaker = null;

            if (coffeeType == 1) {
                coffeeMaker = new AmericanCoffeeMaker();
            } else if (coffeeType == 2) {
                coffeeMaker = new LatteCoffeeMaker();
            } else {
                System.out.println("Invalid coffee type");
                continue;
            }

            // 制作咖啡
            coffeeMaker.makeCoffee();
        }
    }
}
```

其他语言版本

Java

添加钩子函数，也可由用户来确定是否添加牛奶或其它配料。

```
import java.util.Scanner;

// 抽象类，定义咖啡制作的基本步骤
```

```
abstract class CoffeeModel {
    private String coffeeName;

    // 构造函数，接受咖啡名称参数
    public CoffeeModel(String coffeeName) {
        this.coffeeName = coffeeName;
    }

    protected abstract void grind();
    protected abstract void brew();
    protected abstract void addCondiments();

    // 添加其他调料可使用该类
    public void addThings(){};

    // 模板方法，定义咖啡制作的流程
    public final void createCoffeeTemplate() {
        System.out.println("Making " + coffeeName + ":");
        grind();
        brew();
        //根据情况，是否调用添加更多调料
        if (isAddThings()) {
            addThings();
        }
        addCondiments();
        System.out.println();
    }

    // 默认不添加其他调料。如牛奶等
    public boolean isAddThings() {
        return false;
    }
}

//美式咖啡类实现
class CreateAmericanCoffee extends CoffeeModel {
    public CreateAmericanCoffee() {
        super("American Coffee");
    }

    @Override
    protected void grind() {
        System.out.println("Grinding coffee beans");
    }

    @Override
```

```
    protected void brew() {
        System.out.println("Brewing coffee");
    }

    @Override
    protected void addCondiments() {
        System.out.println("Adding condiments");
    }

    // 美式咖啡默认不添加其他调料，如牛奶等
    @Override
    public boolean isAddThings() {
        return false;
    }
}

//拿铁类实现
class CreateLatte extends CoffeeModel {
    private boolean addThingsFlag = true;

    public CreateLatte() {
        super("Latte");
    }

    @Override
    protected void grind() {
        System.out.println("Grinding coffee beans");
    }

    @Override
    protected void brew() {
        System.out.println("Brewing coffee");
    }

    @Override
    protected void addCondiments() {
        System.out.println("Adding condiments");
    }

    //需要添加调料，牛奶
    @Override
    public void addThings(){
        System.out.println("Adding milk");
    }

    // 拿铁默认添加牛奶
```

```
    @Override
    public boolean isAddThings() {
        return this.addThingsFlag;
    }

    // 外部调用以改变是否添加牛奶的状态，钩子函数
    public void setAddThingsFlag(boolean flag) {
        this.addThingsFlag = flag;
    }
}

//客户端
public class Main {
    public static void main(String[] args) {
        try (Scanner scanner = new Scanner(System.in)) {
            while (scanner.hasNextInt()) {
                int input = scanner.nextInt();
                CoffeeModel coffee;
                switch (input) {
                    case 1:
                        coffee = new CreateAmericanCoffee();
                        break;
                    case 2:
                        coffee = new CreateLatte();
                        break;
                    default:
                        System.out.println("无效选择，请输入1或2");
                        continue;
                }
                coffee.createCoffeeTemplate();
            }
        }
    }
}
```

C++

```
#include <iostream>
#include <string>
#include <memory>

// 抽象类
class CoffeeMakerTemplate {
private:
    std::string coffeeName;
```

```
public:
    // 构造函数，接受咖啡名称参数
    CoffeeMakerTemplate(const std::string& coffeeName) :
coffeeName(coffeeName) {}

    // 模板方法定义咖啡制作过程
    virtual void makeCoffee() {
        std::cout << "Making " << coffeeName << ":\n";
        grindCoffeeBeans();
        brewCoffee();
        addCondiments();
        std::cout << '\n';
    }

    // 具体步骤的具体实现由子类提供
    virtual void grindCoffeeBeans() = 0;
    virtual void brewCoffee() = 0;

    // 添加调料的默认实现
    virtual void addCondiments() {
        std::cout << "Adding condiments\n";
    }
};

// 具体的美式咖啡类
class AmericanCoffeeMaker : public CoffeeMakerTemplate {
public:
    // 构造函数传递咖啡名称
    AmericanCoffeeMaker() : CoffeeMakerTemplate("American Coffee") {}

    void grindCoffeeBeans() override {
        std::cout << "Grinding coffee beans\n";
    }

    void brewCoffee() override {
        std::cout << "Brewing coffee\n";
    }
};

// 具体的拿铁咖啡类
class LatteCoffeeMaker : public CoffeeMakerTemplate {
public:
    // 构造函数传递咖啡名称
    LatteCoffeeMaker() : CoffeeMakerTemplate("Latte") {}

    void grindCoffeeBeans() override {
```

```
        std::cout << "Grinding coffee beans\n";
    }

    void brewCoffee() override {
        std::cout << "Brewing coffee\n";
    }

    // 添加调料的特定实现
    void addCondiments() override {
        std::cout << "Adding milk\n";
        std::cout << "Adding condiments\n";
    }
};

int main() {
    std::unique_ptr<CoffeeMakerTemplate> coffeeMaker;

    int coffeeType;
    while (std::cin >> coffeeType) {
        if (coffeeType == 1) {
            coffeeMaker = std::make_unique<AmericanCoffeeMaker>();
        } else if (coffeeType == 2) {
            coffeeMaker = std::make_unique<LatteCoffeeMaker>();
        } else {
            std::cout << "Invalid coffee type\n";
            continue;
        }

        // 制作咖啡
        coffeeMaker->makeCoffee();
    }

    return 0;
}
```

Python

```
from abc import ABC, abstractmethod

# 抽象类
class CoffeeMakerTemplate(ABC):
    # 构造函数，接受咖啡名称参数
    def __init__(self, coffee_name):
        self.coffee_name = coffee_name

    # 模板方法定义咖啡制作过程
```

```
    def make_coffee(self):
        print(f"Making {self.coffee_name}:")
        self.grind_coffee_beans()
        self.brew_coffee()
        self.add_condiments()
        print()

    # 具体步骤的具体实现由子类提供
    @abstractmethod
    def grind_coffee_beans(self):
        pass

    @abstractmethod
    def brew_coffee(self):
        pass

    # 添加调料的默认实现
    def add_condiments(self):
        print("Adding condiments")


# 具体的美式咖啡类
class AmericanCoffeeMaker(CoffeeMakerTemplate):
    # 构造函数传递咖啡名称
    def __init__(self):
        super().__init__("American Coffee")

    def grind_coffee_beans(self):
        print("Grinding coffee beans")

    def brew_coffee(self):
        print("Brewing coffee")


# 具体的拿铁咖啡类
class LatteCoffeeMaker(CoffeeMakerTemplate):
    # 构造函数传递咖啡名称
    def __init__(self):
        super().__init__("Latte")

    def grind_coffee_beans(self):
        print("Grinding coffee beans")

    def brew_coffee(self):
        print("Brewing coffee")
```

```
    # 添加调料的特定实现
    def add_condiments(self):
        print("Adding milk")
        print("Adding condiments")


# 客户端代码
if __name__ == "__main__":
    while True:
        try:
            coffee_type = int(input())

            coffee_maker = None

            if coffee_type == 1:
                coffee_maker = AmericanCoffeeMaker()
            elif coffee_type == 2:
                coffee_maker = LatteCoffeeMaker()
            else:
                print("Invalid coffee type")
                continue

            # 制作咖啡
            coffee_maker.make_coffee()

        except EOFError:
            break
```

Go

```
package main

import (
    "fmt"
    "os"
)

// 抽象类接口
type CoffeeMakerTemplate interface {
    MakeCoffee()
    GrindCoffeeBeans()
    BrewCoffee()
    AddCondiments()
}

// 具体的美式咖啡类
```

```go
type AmericanCoffeeMaker struct {
    coffeeName string
}

// 构造函数传递咖啡名称
func NewAmericanCoffeeMaker() *AmericanCoffeeMaker {
    return &AmericanCoffeeMaker{coffeeName: "American Coffee"}
}

// 实现接口
func (a *AmericanCoffeeMaker) MakeCoffee() {
    fmt.Printf("Making %s:\n", a.coffeeName)
    a.GrindCoffeeBeans()
    a.BrewCoffee()
    a.AddCondiments()
    fmt.Println()
}

func (a *AmericanCoffeeMaker) GrindCoffeeBeans() {
    fmt.Println("Grinding coffee beans")
}

func (a *AmericanCoffeeMaker) BrewCoffee() {
    fmt.Println("Brewing coffee")
}

func (a *AmericanCoffeeMaker) AddCondiments() {
    fmt.Println("Adding condiments")
}

// 具体的拿铁咖啡类
type LatteCoffeeMaker struct {
    coffeeName string
}

// 构造函数传递咖啡名称
func NewLatteCoffeeMaker() *LatteCoffeeMaker {
    return &LatteCoffeeMaker{coffeeName: "Latte"}
}

// 实现接口
func (l *LatteCoffeeMaker) MakeCoffee() {
    fmt.Printf("Making %s:\n", l.coffeeName)
    l.GrindCoffeeBeans()
    l.BrewCoffee()
    l.AddCondiments()
```

```
    fmt.Println()
}

func (l *LatteCoffeeMaker) GrindCoffeeBeans() {
    fmt.Println("Grinding coffee beans")
}

func (l *LatteCoffeeMaker) BrewCoffee() {
    fmt.Println("Brewing coffee")
}

func (l *LatteCoffeeMaker) AddCondiments() {
    fmt.Println("Adding milk")
    fmt.Println("Adding condiments")
}

func main() {
    for {
        var coffeeType int
        if _, err := fmt.Scan(&coffeeType); err != nil {
            if err.Error() == "expected integer" || err.Error() == "EOF" {
                break
            }
            fmt.Println(err)
            os.Exit(1)
        }

        var coffeeMaker CoffeeMakerTemplate

        switch coffeeType {
        case 1:
            coffeeMaker = NewAmericanCoffeeMaker()
        case 2:
            coffeeMaker = NewLatteCoffeeMaker()
        default:
            fmt.Println("Invalid coffee type")
            continue
        }

        // 制作咖啡
        coffeeMaker.MakeCoffee()
    }
}
```