# 行为型模式（四）

2020年5月25日 11:01

行为型设计模式（四）

**作业要求：**

**1、结合实例，绘制策略模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。**

**2、结合实例，绘制模板方法模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。**

**3、在某数据挖掘工具的数据分类模块中，数据处理流程包括4个步骤，分别是：①读取数据；②转换数据格式；③调用数据分类算法；④显示数据分类结果。对于不同的分类算法而言，第①步，第②步，第④步是相同的，主要区别在于第③步。第③步将调用算法库中已有的分类算法实现。例如朴素的贝叶斯分类（Naive Bayes）算法，决策树（Decision Tree）算法，K最近邻（K-Nearest Neighbor, KNN）算法等。现采用模板方法模式和适配器模式设计该数据分类模块。**

**设计该数据分类模块，绘制对应的类图并编程模拟实现。**

- **作业与笔记github地址**：https://github.com/baobaotql/CCNU_DesignPattern

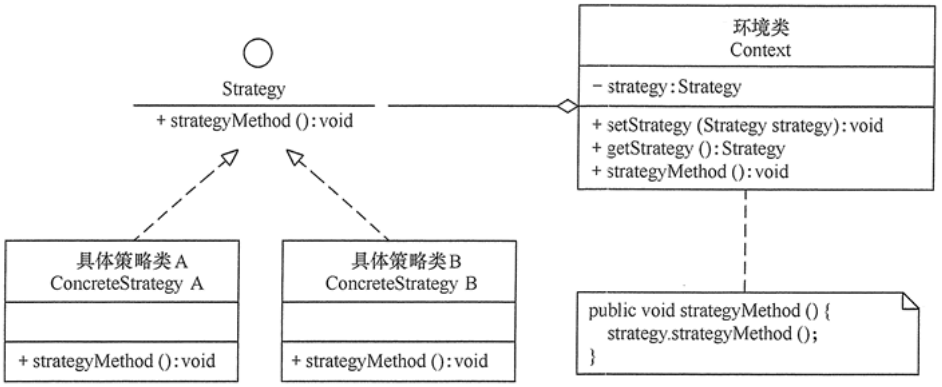
- **Q1:结合实例，绘制策略模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。**

  **A1:**

  - **策略模式的定义**

    该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

  - **优点**
    - 多重条件语句不易维护，而使用策略模式可以避免使用多重条件语句。
    - 策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而避免重复的代码。
    - 策略模式可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同的。
    - 策略模式提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算法。
    - 策略模式把算法的使用放到环境类中，而算法的实现移到具体策略类中，实现了二者的分离。
  - **缺点**
    - 客户端必须理解所有策略算法的区别，以便适时选择恰当的算法类。
    - 策略模式造成很多的策略类。
  - **模式结构**
    - 抽象策略（Strategy）类：定义了一个公共接口，各种不同的算法以不同的方式实现这个接口，环境角色使用这个接口调用不同的算法，一般使用接口或抽象类实现。
    - 具体策略（Concrete Strategy）类：实现了抽象策略定义的接口，提供具体的算法实现。
    - 环境（Context）类：持有一个策略类的引用，最终给客户端调用。
    - **模式结构图**




- **模式分析**

  策略模式是准备一组算法，并将这组算法封装到一系列的策略类里面，作为一个抽象策略类的子类。策略模式的重心不是如何实现算法，而是如何组织这些算法，从而让程序结构更加灵活，具有更好的维护性和扩展性，现在我们来分析其基本结构和实现方法

- 模式代码

```
package strategy;
public class StrategyPattern
{
    public static void main(String[] args)
    {
        Context c=new Context();
        Strategy s=new ConcreteStrategyA();
        c.setStrategy(s);
        c.strategyMethod();
        System.out.println("------------------");
        s=new ConcreteStrategyB();
        c.setStrategy(s);
        c.strategyMethod();
    }
}
//抽象策略类
interface Strategy
{
    public void strategyMethod();     //策略方法
}
//具体策略类A
class ConcreteStrategyA implements Strategy
{
    public void strategyMethod()
    {
        System.out.println("具体策略A的策略方法被访问！");
    }
}
//具体策略类B
class ConcreteStrategyB implements Strategy
{
  public void strategyMethod()
  {
      System.out.println("具体策略B的策略方法被访问！");
  }
}
//环境类
class Context
{
    private Strategy strategy;
```

```
    public Strategy getStrategy()
    {
        return strategy;
    }
    public void setStrategy(Strategy strategy)
    {
        this.strategy=strategy;
    }
    public void strategyMethod()
    {
        strategy.strategyMethod();
    }
}
```

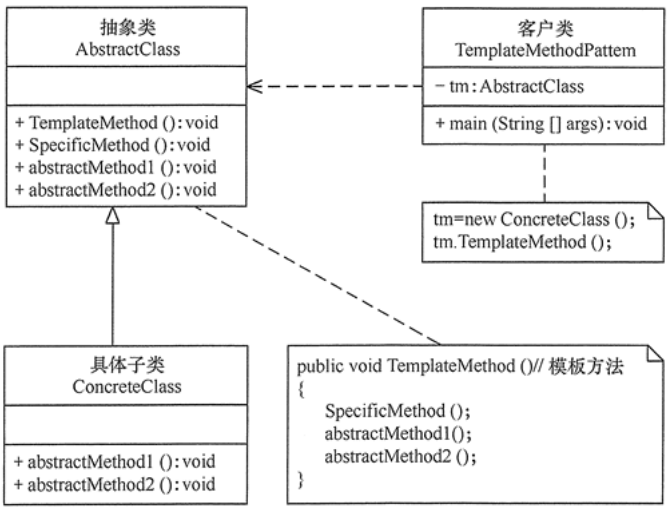
- **Q2:结合实例，绘制模板方法模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。**

  **A2:**

  - **模板方法模式的定义**

    定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。它是一种类行为型模式。

  - **优点**
    - 它封装了不变部分，扩展可变部分。它把认为是不变部分的算法封装到父类中实现，而把可变部分算法由子类继承实现，便于子类继续扩展。
    - 它在父类中提取了公共的部分代码，便于代码复用。
    - 部分方法是由子类实现的，因此子类可以通过扩展方式增加相应的功能，符合开闭原则。
  - **缺点**
    - 对每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象。
    - 父类中的抽象方法由子类实现，子类执行的结果会影响父类的结果，这导致一种反向的控制结构，它提高了代码阅读的难度。
  - **模式结构**
    - 抽象类（Abstract Class）：负责给出一个算法的轮廓和骨架。它由一个模板方法和若干个基本方法构成。这些方法的定义如下。
      - ① 模板方法：定义了算法的骨架，按某种顺序调用其包含的基本方法。
      - ② 基本方法：是整个算法中的一个步骤，包含以下几种类型。
        - 抽象方法：在抽象类中申明，由具体子类实现。
        - 具体方法：在抽象类中已经实现，在具体子类中可以继承或重写它。
        - 钩子方法：在抽象类中已经实现，包括用于判断的逻辑方法和需要子类重写的空方法两种。
    - 具体子类（Concrete Class）：实现抽象类中所定义的抽象方法和钩子方法，它们是一个顶级逻辑的一个组成步骤。
    - **模式结构图**




- **模式分析**

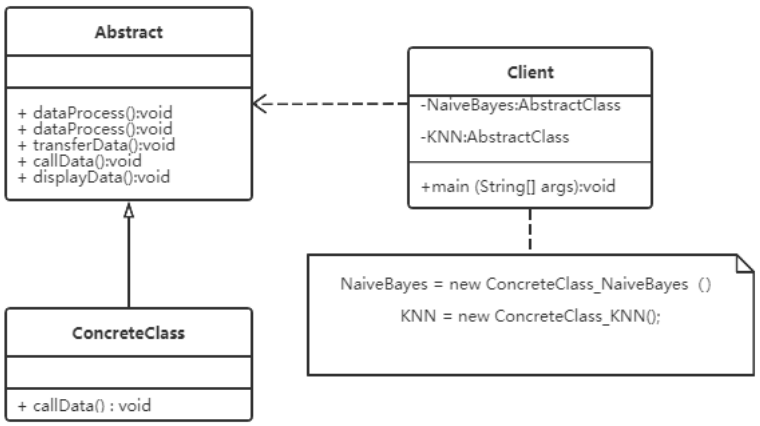
  模板方法模式需要注意抽象类与具体子类之间的协作。它用到了虚函数的多态性技术以及“不用调用我，让我来调用你”的反向控制技术。现在来介绍它们的基本结构。

- **模式代码**

```
package templateMethod;

public class TemplateMethodPattern
{
    public static void main(String[] args)
    {
        AbstractClass tm=new ConcreteClass();
        tm.TemplateMethod();
    }
}
//抽象类
abstract class AbstractClass
{
    public void TemplateMethod() //模板方法
    {
        SpecificMethod();
        abstractMethod1();
         abstractMethod2();
    }
    public void SpecificMethod() //具体方法
    {
        System.out.println("抽象类中的具体方法被调用...");
    }
    public abstract void abstractMethod1(); //抽象方法1
    public abstract void abstractMethod2(); //抽象方法2
}
//具体子类
class ConcreteClass extends AbstractClass
{
    public void abstractMethod1()
    {
        System.out.println("抽象方法1的实现被调用...");
    }
    public void abstractMethod2()
    {
        System.out.println("抽象方法2的实现被调用...");
    }
}
```

- Q3:**在某数据挖掘工具的数据分类模块中,数据处理流程包括4个步骤,分别是:**
  **①读取数据;**
  **②转换数据格式;**
  **③调用数据分类算法;**
  **④显示数据分类结果。**
  **对于不同的分类算法而言，第①步，第②步，第④步是相同的，主要区别在于第③步。第③步将调用算法库中已有的分类算法实现。例如朴素的贝叶斯分类（Naive Bayes）算法,决策树（Decision Tree）算法，K最近邻（K-Nearest Neighbor, KNN）算法等。现采用模板方法模式和适配器模式设计该数据分类模块。**
  **设计该数据分类模块，绘制对应的类图并编程模拟实现。**
  **A3:**
  - 模式结构图




    - 创建抽象模板结构

```
public abstract class Abstract Class {
//模板方法
//申明为final，不希望子类覆盖这个方法，防止更改流程的执行顺序
        final void dataProcess(){
        //第一步：读数据
        this.readData();
        //第二步：转换数据
        this.transferData();
        //第三步：调用数据分类算法
        this.callData();
        //第四步：显示数据分类结果
        this.displayData();
    }

void dataProcess(){
        System.out.println("读取数据");
    }

    void transferData(){
        System.out.println("转换数据格式");
    }

//声明为抽象方法，具体由子类实现
    abstract void callData();
```

```
    void displayData(){
        System.out.println("显示数据分类结果");
    }
}
```

- 创建具体模板

```
//适配者1
public class ConcreteClass_NaiveBayes extend  Abstract Class{
    @Override
    public void callData(){
        System.out.println("调用Naive Bayes朴素贝叶斯算法");
    }
}
  //适配者2
  public class ConcreteClass_DecisionTree extend  Abstract Class{
    @Override
    public void callData(){
        System.out.println("调用Decision Tree决策树算法");
    }
  //适配者3
   public class ConcreteClass_KNN extend  Abstract Class{
    @Override
   public void callData(){
        System.out.println("调用KNN K最邻近算法");
    }
}
```

- 客户端调用

```
public class Template Method{
  public static void main(String[] args){

//Naive Bayes
    ConcreteClass_NaiveBayes NaiveBayes = new ConcreteClass_NaiveBayes();
    NaiveBayes.dataProcess();

//KNN
  ConcreteClass_ KNN = new ConcreteClass_KNN();
    KNN.dataProcess();
    }

}
```

  - 输出结构

```
读取数据
转换数据格式
调用Naive Bayes朴素贝叶斯算法
显示数据分类结果

读取数据
转换数据格式
调用KNN K最邻近算法
显示数据分类结果
```