策略模式

题目链接

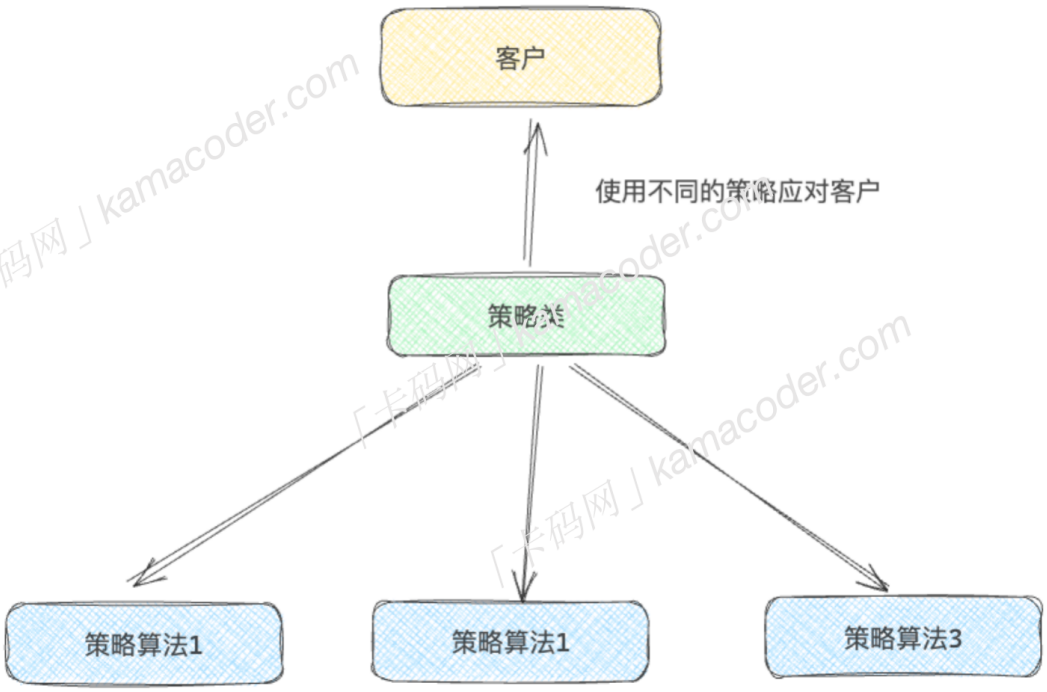
策略模式-超市打折

什么是策略模式

策略模式是一种行为型设计模式，它定义了一系列算法（这些算法完成的是相同的工作，只是实现不同），并将每个算法封装起来，使它们可以相互替换，而且算法的变化不会影响使用算法的客户。

举个例子，电商网站对于商品的折扣策略有不同的算法，比如新用户满减优惠，不同等级会员的打折情况不同，这种情况下会产生大量的`if-else`语句，并且如果优惠政策修改时，还需要修改原来的代码，不符合开闭原则。

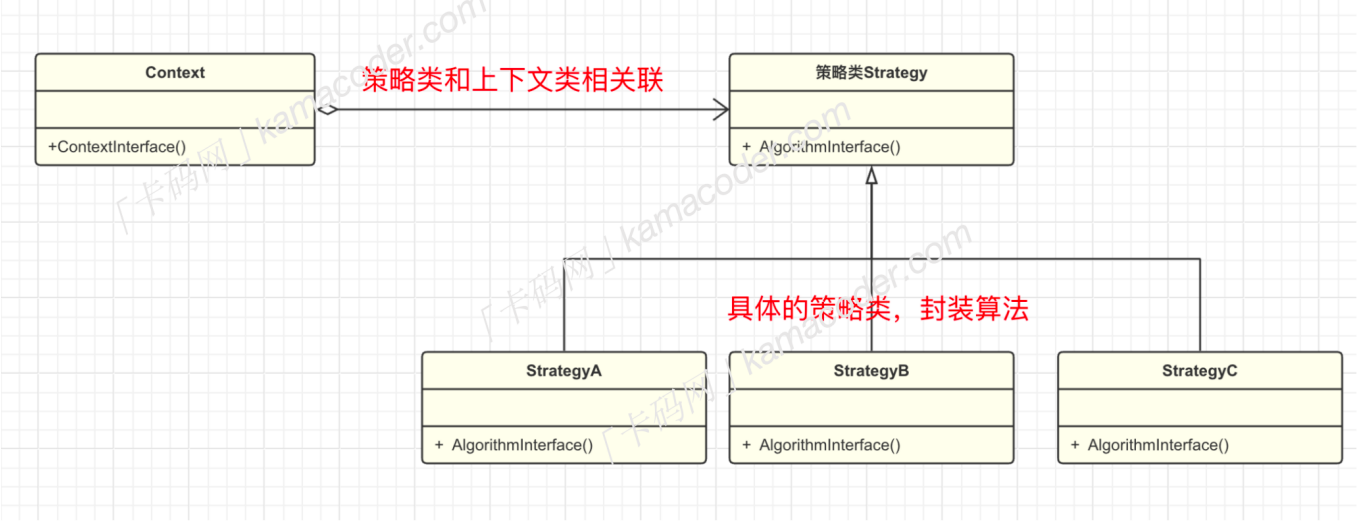
这就可以将不同的优惠算法封装成独立的类来避免大量的条件语句，如果新增优惠算法，可以添加新的策略类来实现，客户端在运行时选择不同的具体策略，而不必修改客户端代码改变优惠策略。



基本结构

策略模式包含下面几个结构:

- 策略类`Strategy`: 定义所有支持的算法的公共接口。
- 具体策略类`ConcreteStrategy`: 实现了策略接口，提供具体的算法实现。
- 上下文类`Context`: 包含一个策略实例，并在需要时调用策略对象的方法。



简单实现

下面是一个简单的策略模式的基本实现:

```
// 1. 抽象策略抽象类
abstract class Strategy {
    // 抽象方法
    public abstract void algorithmInterface();
}

// 2. 具体策略类1
class ConcreteStrategyA extends Strategy {
    @Override
    public void algorithmInterface() {
        System.out.println("Strategy A");
        // 具体的策略1执行逻辑
    }
}

// 3. 具体策略类2
class ConcreteStrategyB extends Strategy {
    @Override
```

```
    public void algorithmInterface() {
        System.out.println("Strategy B");
        // 具体的策略2执行逻辑
    }
}

// 4. 上下文类
class Context {
    private Strategy strategy;

    // 设置具体的策略
     public Context(Strategy strategy) {
        this.strategy = strategy;
     }

    // 执行策略
    public void contextInterface() {
        strategy.algorithmlnterface();
    }
}

// 5. 客户端代码
public class Main{
    public static void main(String[] args) {
        // 创建上下文对象，并设置具体的策略
        Context contextA = new Context(new ConcreteStrategyA());
        // 执行策略
        contextA.contextInterface();

        Context contextB = new Context(new ConcreteStrategyB());
        contextB.contextInterface();u
    }
}
```

使用场景

那什么时候可以考虑使用策略模式呢?

- 当一个系统根据业务场景需要动态地在几种算法中选择一种时，可以使用策略模式。例如，根据用户的行为选择不同的计费策略。
- 当代码中存在大量条件判断，条件判断的区别仅仅在于行为，也可以通过策略模式来消除这些条件语句。

在已有的工具库中，Java 标准库中的 `Comparator` 接口就使用了策略模式，通过实现这个接口，可以创建不同的比较器（指定不同的排序策略）来满足不同的排序需求。

本题代码

```
import java.util.Scanner;

// 抽象购物优惠策略接口
interface DiscountStrategy {
    int applyDiscount(int originalPrice);
}

// 九折优惠策略
class DiscountStrategy1 implements DiscountStrategy {
    @Override
    public int applyDiscount(int originalPrice) {
        return (int) Math.round(originalPrice * 0.9);
    }
}

// 满减优惠策略
class DiscountStrategy2 implements DiscountStrategy {
    private int[] thresholds = {100, 150, 200, 300};
    private int[] discounts = {5, 15, 25, 40};

    @Override
    public int applyDiscount(int originalPrice) {
        for (int i = thresholds.length - 1; i >= 0; i--) {
            if (originalPrice >= thresholds[i]) {
                return originalPrice - discounts[i];
            }
        }
        return originalPrice;
    }
}

// 上下文类
class DiscountContext {
    private DiscountStrategy discountStrategy;

    public void setDiscountStrategy(DiscountStrategy discountStrategy) {
        this.discountStrategy = discountStrategy;
    }

    public int applyDiscount(int originalPrice) {
        return discountStrategy.applyDiscount(originalPrice);
    }
}
```

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

        // 读取需要计算优惠的次数
        int N = Integer.parseInt(scanner.nextLine());

        for (int i = 0; i < N; i++) {
            // 读取商品价格和优惠策略
            String[] input = scanner.nextLine().split(" ");
            int M = Integer.parseInt(input[0]);
            int strategyType = Integer.parseInt(input[1]);

            // 根据优惠策略设置相应的打折策略
            DiscountStrategy discountStrategy;
            switch (strategyType) {
                case 1:
                    discountStrategy = new DiscountStrategy1();
                    break;
                case 2:
                    discountStrategy = new DiscountStrategy2();
                    break;
                default:
                    // 处理未知策略类型
                    System.out.println("Unknown strategy type");
                    return;
            }

            // 设置打折策略
            DiscountContext context = new DiscountContext();
            context.setDiscountStrategy(discountStrategy);

            // 应用打折策略并输出优惠后的价格
            int discountedPrice = context.applyDiscount(M);
            System.out.println(discountedPrice);
        }
    }
}
```

其他语言版本

Java

使用策略枚举类实现

```
import java.util.Scanner;

interface Strategy {
    void preferentialMethod(int price);
}

//策略枚举类
enum DiscountStrategy implements Strategy {
    STRATEGY1 {
        @Override
        public void preferentialMethod(int price) {
            double discountedPrice = 0.9 * price;
            System.out.println((int) discountedPrice);
        }
    },
    STRATEGY2 {
        @Override
        public void preferentialMethod(int price) {
            int[][] discountRules = {
                {300, 40},
                {200, 25},
                {150, 15},
                {100, 5}
            };

            for (int[] rule : discountRules) {
                if (price >= rule[0]) {
                    price -= rule[1];
                    break;
                }
            }
            System.out.println(price);
        }
    };

    public static DiscountStrategy fromType(int type) {
        switch (type) {
            case 1:
                return STRATEGY1;
            case 2:
                return STRATEGY2;
            default:
```

```
                throw new IllegalArgumentException("无效选择，请输入1或2");
            }
        }
    }

class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public void executeStrategy(int price) {
        strategy.preferentialMethod(price);
    }
}

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

            for (int i = 0; i < num; i++) {
                try {
                    String input = scanner.nextLine();
                    String[] parts = input.split(" ");

                    if (parts.length != 2) {
                        System.out.println("输入错误！");
                        continue;
                    }

                    int price = Integer.parseInt(parts[0]);
                    int type = Integer.parseInt(parts[1]);

                    DiscountStrategy strategy =
DiscountStrategy.fromType(type);
                    Context context = new Context(strategy);
                    context.executeStrategy(price);
                } catch (NumberFormatException e) {
                    System.out.println("输入格式错误，请输入有效的价格和类型！");
                } catch (IllegalArgumentException e) {
                    System.out.println(e.getMessage());
                }
```

```
            }
        } catch (Exception e) {
            System.out.println("An error occurred: " + e.getMessage());
        } finally {
            scanner.close();
        }
    }
}
```

C++

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

// 抽象购物优惠策略接口
class DiscountStrategy {
public:
    virtual int applyDiscount(int originalPrice) = 0;
    virtual ~DiscountStrategy() = default; // 添加虚析构函数
};

// 九折优惠策略
class DiscountStrategy1 : public DiscountStrategy {
public:
    int applyDiscount(int originalPrice) override {
        return static_cast<int>(std::round(originalPrice * 0.9));
    }
};

// 满减优惠策略
class DiscountStrategy2 : public DiscountStrategy {
private:
    int thresholds[4] = {100, 150, 200, 300};
    int discounts[4] = {5, 15, 25, 40};

public:
    int applyDiscount(int originalPrice) override {
        for (int i = sizeof(thresholds) / sizeof(thresholds[0]) - 1; i >=
0; i--) {
            if (originalPrice >= thresholds[i]) {
                return originalPrice - discounts[i];
            }
```

```
        }
        return originalPrice;
    }
};

// 上下文类
class DiscountContext {
private:
    DiscountStrategy* discountStrategy;

public:
    void setDiscountStrategy(DiscountStrategy* discountStrategy) {
        this->discountStrategy = discountStrategy;
    }

    int applyDiscount(int originalPrice) {
        return discountStrategy->applyDiscount(originalPrice);
    }
};

int main() {
    // 读取需要计算优惠的次数
    int N;
    std::cin >> N;
    std::cin.ignore(); // 忽略换行符

    for (int i = 0; i < N; i++) {
        // 读取商品价格和优惠策略
        int M, strategyType;
        std::cin >> M >> strategyType;

        // 根据优惠策略设置相应的打折策略
        DiscountStrategy* discountStrategy;
        switch (strategyType) {
            case 1:
                discountStrategy = new DiscountStrategy1();
                break;
            case 2:
                discountStrategy = new DiscountStrategy2();
                break;
            default:
                // 处理未知策略类型
                std::cout << "Unknown strategy type" << std::endl;
                return 1;
        }
```

```
        // 设置打折策略
        DiscountContext context;
        context.setDiscountStrategy(discountStrategy);

        // 应用打折策略并输出优惠后的价格
        int discountedPrice = context.applyDiscount(M);
        std::cout << discountedPrice << std::endl;

        // 释放动态分配的打折策略对象
        delete discountStrategy;
    }

    return 0;
}
```

Python

```
class DiscountStrategy:
    def apply_discount(self, original_price):
        pass

class DiscountStrategy1(DiscountStrategy):
    def apply_discount(self, original_price):
        return round(original_price * 0.9)

class DiscountStrategy2(DiscountStrategy):
    def __init__(self):
        self.thresholds = [100, 150, 200, 300]
        self.discounts = [5, 15, 25, 40]

    def apply_discount(self, original_price):
        for threshold, discount in zip(reversed(self.thresholds),
reversed(self.discounts)):
            if original_price >= threshold:
                return original_price - discount
        return original_price

class DiscountContext:
    def __init__(self):
        self.discount_strategy = None

    def set_discount_strategy(self, discount_strategy):
        self.discount_strategy = discount_strategy

    def apply_discount(self, original_price):
        return self.discount_strategy.apply_discount(original_price)
```

```python
if __name__ == "__main__":
    # 读取需要计算优惠的次数
    N = int(input())

    for _ in range(N):
        # 读取商品价格和优惠策略
        input_data = input().split(" ")
        M = int(input_data[0])
        strategy_type = int(input_data[1])

        # 根据优惠策略设置相应的打折策略
        if strategy_type == 1:
            discount_strategy = DiscountStrategy1()
        elif strategy_type == 2:
            discount_strategy = DiscountStrategy2()
        else:
            # 处理未知策略类型
            print("Unknown strategy type")
            break

        # 设置打折策略
        context = DiscountContext()
        context.set_discount_strategy(discount_strategy)

        # 应用打折策略并输出优惠后的价格
        discounted_price = context.apply_discount(M)
        print(discounted_price)
```

Go

```go
package main

import "fmt"

// 抽象购物优惠策略接口
type DiscountStrategy interface {
    applyDiscount(originalPrice int) int
}

// 九折优惠策略
type DiscountStrategy1 struct{}

func (d *DiscountStrategy1) applyDiscount(originalPrice int) int {
    return int(float64(originalPrice) * 0.9)
}
```

```
// 满减优惠策略
type DiscountStrategy2 struct {
    thresholds []int
    discounts  []int
}

func (d *DiscountStrategy2) applyDiscount(originalPrice int) int {
    for i := len(d.thresholds) - 1; i >= 0; i-- {
        if originalPrice >= d.thresholds[i] {
            return originalPrice - d.discounts[i]
        }
    }
    return originalPrice
}

// 上下文类
type DiscountContext struct {
    discountStrategy DiscountStrategy
}

func (d *DiscountContext) setDiscountStrategy(discountStrategy
DiscountStrategy) {
    d.discountStrategy = discountStrategy
}

func (d *DiscountContext) applyDiscount(originalPrice int) int {
    return d.discountStrategy.applyDiscount(originalPrice)
}

func main() {
    // 读取需要计算优惠的次数
    var N int
    fmt.Scan(&N)

    for i := 0; i < N; i++ {
        // 读取商品价格和优惠策略
        var M, strategyType int
        fmt.Scan(&M, &strategyType)

        // 根据优惠策略设置相应的打折策略
        var discountStrategy DiscountStrategy
        switch strategyType {
        case 1:
            discountStrategy = &DiscountStrategy1{}
        case 2:
```

```
            discountStrategy = &DiscountStrategy2{
                thresholds: []int{100, 150, 200, 300},
                discounts:  []int{5, 15, 25, 40},
            }
        default:
            // 处理未知策略类型
            fmt.Println("Unknown strategy type")
            return
        }

        // 设置打折策略
        context := &DiscountContext{}
        context.setDiscountStrategy(discountStrategy)

        // 应用打折策略并输出优惠后的价格
        discountedPrice := context.applyDiscount(M)
        fmt.Println(discountedPrice)
    }
}
```

Typescript

```
interface Strategy {
  algorithm(price: number): void;
}

class NinePercentDiscount implements Strategy {
  algorithm(price) {
    console.log(price * 0.9);
  }
}

class FullDiscount implements Strategy {
  private priceMap = {
    100: 5,
    150: 15,
    200: 25,
    300: 40,
  };

  algorithm(price) {
    console.log(price - this.priceMap[price]);
  }
}
```

```
class StrategyContext {
  private discountStrategy: Strategy;

  constructor(way) {
    switch (way) {
      case 1:
        this.discountStrategy = new NinePercentDiscount();
        break;
      case 2:
        this.discountStrategy = new FullDiscount();
        break;
      default:
        throw "1 or 2";
    }
  }

  discount(price) {
    this.discountStrategy.algorithm(price);
  }
}

// @ts-ignore
entry(4, (...args) => {
  args.forEach(([price, way]) => {
    new StrategyContext(way).discount(price);
  });
})([100, 1])([200, 2])([300, 1])([300, 2]);


export function entry(count: number, fn: (...args: any) => void) {
  function dfs(...args) {
    if (args.length < count) {
      return (arg) => dfs(...args, arg);
    }

    return fn(...args);
  }
  return dfs;
}
```