第四篇章 行为型设计模式

在设计模式的世界里，23种经典设计模式通常被分为三大类：创建型、结构型和行为型。我们已经探讨了创建型和结构型设计模式，现在我们将开始学习行为型设计模式。正如创建型设计模式关注于对象创建的问题，结构型设计模式关注于类或对象的组合和组装问题，行为型设计模式则主要关注于类或对象之间的**交互问题。**

行为型设计模式的数量较多，共有11种，几乎占据了23种经典设计模式的一半。这些模式分别为：观察者模式、模板模式、策略模式、职责链模式、状态模式、迭代器模式、访问者模式、备忘录模式、命令模式、解释器模式和中介模式。

第一章 观察者模式

一、概述

观察者模式是一种行为设计模式，允许对象间存在**一对多的依赖关系**，当**一个对象的状态发生改变时，所有依赖它的对象都会得到通知并自动更新。**在这种模式中，**发生状态改变的对象被称为“主题”（Subject），依赖它的对象被称为“观察者”（Observer）。**

观察者模式（Observer Design Pattern）也被称为**发布订阅模式**（Publish-Subscribe Design Pattern）。在 GoF 的《设计模式》一书中，它的定义是这样的：

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

翻译成中文就是：在对象之间定义一个一对多的依赖，当一个对象状态改变的时候，所有依赖的对象都会自动收到通知。

一般情况下，被依赖的对象叫作**被观察者**（Observable），依赖的对象叫作**观察者**（Observer）。不过，在实际的项目开发中，这两种对象的称呼是比较灵活的，有各种不同的叫法，比如：Subject-Observer、Publisher-Subscriber、Producer-Consumer等等。不管怎么称呼，只要应用场景符合刚刚给出的定义，都可以看作观察者模式。

让我们通过一个简单的例子来实现观察者模式。假设我们有一个气象站（WeatherStation），需要向许多不同的显示设备（如手机App、网站、电子屏幕等）提供实时天气数据。

首先，我们需要创建一个Subject接口，**表示主题**：

```
public interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}
```

接下来，我们创建一个Observer接口，**表示观察者**：

```
public interface Observer {
    void update(float temperature, float humidity, float pressure);
}
```

现在，我们创建一个**具体的主题**，如WeatherStation，实现Subject接口：

```
public class WeatherStation implements Subject {
    private ArrayList<Observer> observers;
    // 温度
    private float temperature;
    // 湿度
    private float humidity;
    // 大气压
    private float pressure;

    public WeatherStation() {
        observers = new ArrayList<>();
    }

    // 注册一个观察者的方法
    @Override
```

```
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    // 移除一个观察者的方法
    @Override
    public void removeObserver(Observer o) {
        int index = observers.indexOf(o);
        if (index >= 0) {
            observers.remove(index);
        }
    }

    // 通知所有的观察者
    @Override
    public void notifyObservers() {
        // 循环所有的观察者，通知其当前的气象信息
        for (Observer o : observers) {
            o.update(temperature, humidity, pressure);
        }
    }

    // 修改气象内容
    public void measurementsChanged() {
        notifyObservers();
    }

    // 当测量值发生了变化的时候
    public void setMeasurements(float temperature, float humidity, float
pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        // 测量值发生了变化
        measurementsChanged();
    }
}
```

最后，我们创建一个具体的观察者，如PhoneApp，实现Observer接口：

```
public class PhoneApp implements Observer {
    private float temperature;
    private float humidity;
```

```
    private float pressure;
    private Subject weatherStation;

    public PhoneApp(Subject weatherStation) {
        this.weatherStation = weatherStation;
        weatherStation.registerObserver(this);
    }

    @Override
    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        display();
    }

    public void display() {
        System.out.println("PhoneApp: Temperature: " + temperature + "°C,
Humidity: " + humidity + "%, Pressure: " + pressure + " hPa");
    }
}
```

现在我们可以创建一个WeatherStation实例并向其注册PhoneApp观察者。当WeatherStation的数据发生变化时，PhoneApp会收到通知并更新自己的显示。

```
public class Main {
    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();
        PhoneApp phoneApp = new PhoneApp(weatherStation);
        // 模拟气象站数据更新
        weatherStation.setMeasurements(25, 65, 1010);
        weatherStation.setMeasurements(22, 58, 1005);

        // 添加更多观察者  网站上显示-电子大屏
        WebsiteDisplay websiteDisplay = new WebsiteDisplay(weatherStation);
        ElectronicScreen electronicScreen = new ElectronicScreen(weatherStation);

        // 再次模拟气象站数据更新
        weatherStation.setMeasurements(18, 52, 1008);
    }
}
```

在这个例子中，我们**创建了一个WeatherStation实例**，并向其注册了PhoneApp、WebsiteDisplay和ElectronicScreen观察者。当WeatherStation的数据发生变化时，所有观察者都会收到通知并更新自己的显示。
这个例子展示了观察者模式的优点：

1. **观察者和主题之间的解耦**：主题只需要知道观察者实现了Observer接口，而无需了解具体的实现细节。
2. **可以动态添加和删除观察者**：通过调用registerObserver和removeObserver方法，可以在运行时添加和删除观察者。
3. **主题和观察者之间的通信是自动的**：当主题的状态发生变化时，观察者会自动得到通知并更新自己的状态。
   观察者模式广泛应用于各种场景，例如事件处理系统、数据同步和更新通知等。学习并掌握观察者模式对于成为一个优秀的Java程序员非常有帮助。

上面的小例子算是观察者模式的“模板代码”，可以反映该模式大体的设计思路。在真实的软件开发中，并不需要照搬上面的模板代码。观察者模式的实现方法各式各样，函数、类的命名等会根据业务场景的不同有很大的差别，比如 register 函数还可以叫作 attach，remove 函数还可以叫作 detach 等等。不过，万变不离其宗，设计思路都是差不多的。

了解了观察者设计模式的基本使用方式，我们接下来看看他的具体使用场景。

二、使用场景

以下是一些使用观察者设计模式的例子：

1. 股票行情应用：股票行情应用中，当股票价格发生变化时，需要通知订阅了该股票的投资者。这里，股票价格更新可以作为被观察者，投资者可以作为观察者。当股票价格发生变化时，所有订阅了该股票的投资者都会收到通知并更新自己的投资策略。
2. 网络聊天室：在网络聊天室中，当有新消息时，需要通知所有在线的用户。聊天室服务器可以作为被观察者，用户可以作为观察者。当有新消息时，聊天室服务器会通知所有在线用户更新聊天记录。
3. 拍卖系统：在拍卖系统中，当出价发生变化时，需要通知所有关注该拍品的用户。这里，拍卖系统可以作为被观察者，用户可以作为观察者。当出价发生变化时，所有关注该拍品的用户都会收到通知并更新自己的出价策略。
4. 订阅系统：在订阅系统中，当有新的内容发布时，需要通知所有订阅了该内容的用户。这里，内容发布可以作为被观察者，用户可以作为观察者。当有新内容发布时，所有订阅了该内容的用户都会收到通知并获取最新内容。

5. 游戏中的事件系统：在游戏中，当某个事件发生时（如角色升级、道具获得等），可能需要通知多个游戏模块进行相应的处理。这里，游戏事件可以作为被观察者，游戏模块可以作为观察者。当游戏事件发生时，所有关注该事件的游戏模块都会收到通知并执行相应的逻辑。
6. 运动比赛实时更新：在体育比赛中，实时更新比分、技术统计等信息对于球迷和分析师非常重要。在这种场景下，比赛数据更新可以作为被观察者，球迷和分析师可以作为观察者。当比赛数据发生变化时，所有关注比赛的球迷和分析师都会收到通知并更新数据。
7. 物联网传感器系统：在物联网（IoT）系统中，有很多传感器不断地采集数据，当数据发生变化时，需要通知相关联的设备或系统。在这种场景下，传感器可以作为被观察者，关联的设备或系统可以作为观察者。当传感器数据发生变化时，所有关联的设备或系统都会收到通知并执行相应的操作。
8. 电子邮件通知系统：在一个任务管理系统中，当任务的状态发生变化（如：新任务分配、任务完成等）时，需要通知相关的人员。这里，任务状态更新可以作为被观察者，相关人员可以作为观察者。当任务状态发生变化时，所有关注该任务的人员都会收到通知并查看任务详情。
9. 社交网络：在社交网络中，用户关注其他用户以获取实时动态。当被关注的用户发布新动态时，需要通知所有关注者。在这种场景下，被关注的用户可以作为被观察者，关注者可以作为观察者。当被关注的用户发布新动态时，所有关注者都会收到通知并查看动态。

以上示例展示了观察者模式在不同领域的应用。观察者模式有助于实现模块间的松散耦合，提高代码的可维护性和可扩展性。

1、电商系统的应用

在电商系统中，观察者模式可以应用于多种场景，如库存管理、促销通知等。以下是一个**促销通知**的例子：

假设我们有一个电商系统，当某件商品有促销活动时，需要通知所有订阅了该商品的用户。在这个例子中，商品是主题，用户是观察者，其代码逻辑和第一节的例子不能说完全一样，也基本是一模一样。

首先，我们创建一个Subject接口，表示主题：

```
public interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}
```

接下来，我们创建一个Observer接口，表示观察者：

```
public interface Observer {
    void update(String discountInfo);
}
```

现在，我们创建一个具体的主题，如Product，实现Subject接口：

```
public class Product implements Subject {
    private ArrayList<Observer> observers;
    // 折扣消息
    private String discountInfo;

    public Product() {
        observers = new ArrayList<>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int index = observers.indexOf(o);
        if (index >= 0) {
            observers.remove(index);
        }
    }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(discountInfo);
        }
    }

    public void discountChanged() {
        notifyObservers();
    }

    public void setDiscountInfo(String discountInfo) {
        this.discountInfo = discountInfo;
        discountChanged();
    }
}
```

接着，我们创建一个具体的观察者，如User，实现Observer接口：

```
public class User implements Observer {
    private String userName;
    private String discountInfo;
    private Subject product;

    public User(String userName, Subject product) {
        this.userName = userName;
        this.product = product;
        product.registerObserver(this);
    }

    public void update(String discountInfo) {
        this.discountInfo = discountInfo;
        display();
    }

    public void display() {
        System.out.println("用户 " + userName + " 收到促销通知: " + discountInfo);
    }
}
```

现在我们可以创建一个Product实例并向其注册User观察者。当Product的促销信息发生变化时，User会收到通知并显示促销信息。

```
public class Main {
    public static void main(String[] args) {
        Product product = new Product();
        User user1 = new User("张三", product);
        User user2 = new User("李四", product);

        // 模拟商品促销信息更新
        product.setDiscountInfo("本周末满100减50");
        product.setDiscountInfo("双十一全场5折");
    }
}
```

在这个例子中，我们创建了一个Product实例并向其注册了两个User观察者。当Product的促销信息发生变化时，所有观察者都会收到通知并更新自己的显示。

这个例子展示了观察者模式在电商系统中的应用，如何实现商品和用户之间的交互。

小作业：

在电商系统中，当订单状态发生变化时，需要通知买家、卖家以及物流系统。在这个例子中，订单是主题，买家、卖家和物流系统是观察者。

我给出最终目标的main方法如下，请把其他的内容补充完整：

```
public class Main {
    public static void main(String[] args) {
        Order order = new Order();
        // 创建买家
        Buyer buyer = new Buyer("张三", order);
        // 创建卖家
        Seller seller = new Seller("李四", order);
        // 创建物流系统
        LogisticsSystem logisticsSystem = new LogisticsSystem(order);

        // 模拟订单状态更新
        order.setOrderStatus("已付款");
        order.setOrderStatus("已发货");
        order.setOrderStatus("已签收");
    }
}
```

2、erp

在ERP系统中，观察者模式也有很多应用场景，例如库存管理、生产计划等。这里我们举一个库存管理的例子：

假设有一个ERP系统，当某个产品的库存低于安全库存时，需要通知采购部门、销售部门和仓库管理员。在这个例子中，产品库存是主题，采购部门、销售部门和仓库管理员是观察者。

我们可以沿用之前定义的Subject接口和Observer接口。

接下来，我们创建一个具体的主题，如Inventory，实现Subject接口：

```
public class Inventory implements Subject {
    private ArrayList<Observer> observers;
    private int stock;
```

```
    public Inventory() {
        observers = new ArrayList<>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int index = observers.indexOf(o);
        if (index >= 0) {
            observers.remove(index);
        }
    }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(String.valueOf(stock));
        }
    }

    public void stockChanged() {
        notifyObservers();
    }

    public void setStock(int stock) {
        this.stock = stock;
        stockChanged();
    }
}
```

接着，我们创建一个具体的观察者，如PurchaseDepartment、SalesDepartment和WarehouseManager，分别实现Observer接口：

```
public class PurchaseDepartment implements Observer {
    private int stock;
    private Subject inventory;

    public PurchaseDepartment(Subject inventory) {
        this.inventory = inventory;
        inventory.registerObserver(this);
    }
```

```
    public void update(String stock) {
        this.stock = Integer.parseInt(stock);
        display();
    }

    public void display() {
        System.out.println("采购部门收到库存更新: " + stock);
    }
}

public class SalesDepartment implements Observer {
    private int stock;
    private Subject inventory;

    public SalesDepartment(Subject inventory) {
        this.inventory = inventory;
        inventory.registerObserver(this);
    }

    public void update(String stock) {
        this.stock = Integer.parseInt(stock);
        display();
    }

    public void display() {
        System.out.println("销售部门收到库存更新: " + stock);
    }
}

public class WarehouseManager implements Observer {
    private int stock;
    private Subject inventory;

    public WarehouseManager(Subject inventory) {
        this.inventory = inventory;
        inventory.registerObserver(this);
    }

    public void update(String stock) {
        this.stock = Integer.parseInt(stock);
        display();
    }
```

```
    public void display() {
        System.out.println("仓库管理员收到库存更新: " + stock);
    }
}
```

现在我们可以创建一个Inventory实例并向其注册PurchaseDepartment、SalesDepartment和WarehouseManager观察者。当Inventory的库存发生变化时，所有观察者会收到通知并更新自己的显示。

```
public class Main {
    public static void main(String[] args) {
        Inventory inventory = new Inventory();
        PurchaseDepartment purchaseDepartment = new
PurchaseDepartment(inventory);
        SalesDepartment salesDepartment = new SalesDepartment(inventory);
        WarehouseManager warehouseManager = new
WarehouseManager(inventory);

        // 模拟库存变化
        inventory.setStock(500);
        inventory.setStock(300);
        inventory.setStock(100);
    }
}
```

在这个例子中，我们创建了一个Inventory实例并向其注册了PurchaseDepartment、SalesDepartment和WarehouseManager观察者。当Inventory的库存发生变化时，所有观察者都会收到通知并更新自己的显示。
这个例子展示了观察者模式在ERP系统中的一个应用场景，实现了库存信息与相关部门（采购部门、销售部门、仓库管理员）之间的交互。这种模式可以帮助企业在库存发生变化时快速作出相应的决策。

3、金融软件

在金融软件中，观察者模式也有很多应用场景。以股票交易系统为例，当股票价格发生变化时，需要通知投资者和其他关注此股票的系统。在这个例子中，股票是主题，投资者和其他系统是观察者。

我们可以沿用之前定义的Subject接口和Observer接口。

接下来，我们创建一个具体的主题，如Stock，实现Subject接口：

```
// 股票
public class Stock implements Subject {
    private ArrayList<Observer> observers;
    private double stockPrice;

    public Stock() {
        observers = new ArrayList<>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int index = observers.indexOf(o);
        if (index >= 0) {
            observers.remove(index);
        }
    }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(String.valueOf(stockPrice));
        }
    }

    public void stockPriceChanged(double newPrice) {
        this.stockPrice = newPrice;
        notifyObservers();
    }
}
```

接着，我们创建一个具体的观察者，如Investor和TradingSystem，分别实现Observer接口：

```
// 投资者
public class Investor implements Observer {
    private String investorName;
    private double stockPrice;
    private Subject stock;

    public Investor(String investorName, Subject stock) {
        this.investorName = investorName;
```

```
        this.stock = stock;
        stock.registerObserver(this);
    }

    public void update(String stockPrice) {
        this.stockPrice = Double.parseDouble(stockPrice);
        display();
    }

    public void display() {
        System.out.println("投资者 " + investorName + " 收到股票价格更新: " +
stockPrice);
    }
}

// 交易系统
public class TradingSystem implements Observer {
    private double stockPrice;
    private Subject stock;

    public TradingSystem(Subject stock) {
        this.stock = stock;
        stock.registerObserver(this);
    }

    public void update(String stockPrice) {
        this.stockPrice = Double.parseDouble(stockPrice);
        display();
    }

    public void display() {
        System.out.println("交易系统收到股票价格更新: " + stockPrice);
    }
}
```

现在我们可以创建一个Stock实例并向其注册Investor和TradingSystem观察者。当Stock的价格发生变化时，所有观察者会收到通知并更新自己的显示。

```
public class Main {
    public static void main(String[] args) {
        Stock stock = new Stock();
        Investor investor1 = new Investor("小明", stock);
        Investor investor2 = new Investor("小红", stock);
        TradingSystem tradingSystem = new TradingSystem(stock);

        // 模拟股票价格变化
        stock.stockPriceChanged(100);
        stock.stockPriceChanged(110);
        stock.stockPriceChanged(120);
    }
}
```

这个例子展示了观察者模式在金融软件中的一个应用场景，实现了股票价格与相关参与者（投资者、交易系统）之间的交互。这种模式可以帮助金融系统在股票价格发生变化时快速作出响应并通知相关参与者，从而实现实时更新和处理。

观察者模式在金融软件中非常重要，因为金融市场的数据实时性和准确性是关键。观察者模式允许金融系统的各个组件之间进行松散耦合，这意味着当市场数据发生变化时，其他相关组件可以更容易地适应这些变化。例如，如果我们想要添加一个新的观察者，如风险管理系统，以便在股票价格波动时采取相应的措施，我们可以轻松地创建一个新的观察者并将其添加到股票主题中，而无需修改股票主题的代码。这使得金融软件的可扩展性和可维护性得到了很大的提高。

此外，观察者模式还可以应用于其他金融软件场景，如汇率变动、利率调整、期货交易等，通过实时通知相关参与者，使得金融市场的运作更加顺畅。

三、发布订阅

发布-订阅模式和**观察者模式**都是用于实现对象间的松耦合通信的设计模式。尽管它们具有相似之处，但它们在实现方式和使用场景上存在一些关键区别。他们在概念上有一定的相似性，都是用于实现**对象间的松耦合通信**。可以将**发布-订阅模式看作是观察者模式的一种变体或扩展。**

我分别解释一下这两种模式。

1、观察者模式

观察者模式定义了一种一对多的依赖关系，当一个对象（被观察者）的状态发生变化时，所有依赖于它的对象（观察者）都会得到通知并自动更新。在这个模式中，被观察者和观察者之间存在直接的关联关系。观察者模式主要包括两类对象：被观察者（Subject）和观察者（Observer）。

Java中的观察者模式示例，我们已经介绍了很多了，这里简单看一下就可以了：

```
interface Observer {
    void update(String message);
}

class ConcreteObserver implements Observer {
    @Override
    public void update(String message) {
        System.out.println("收到通知: " + message);
    }
}

interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

class ConcreteSubject implements Subject {
    private List<Observer> observers = new ArrayList<>();

    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update("状态发生变化");
        }
    }
```

```
}
```

2、发布-订阅模式

发布-订阅模式（生产者和消费者）与观察者模式类似，但它们之间有一个关键区别：发布-订阅模式**引入了一个第三方组件（通常称为消息代理或事件总线）**，该组件负责维护发布者和订阅者之间的关系。这意味着**发布者和订阅者彼此不直接通信**，而是通过消息代理进行通信。这种间接通信**允许发布者和订阅者在运行时动态地添加或删除**，从而提高了系统的灵活性和可扩展性。

Java中的发布-订阅模式示例：

```
interface Subscriber {
    void onEvent(String event);
}

class ConcreteSubscriber implements Subscriber {
    @Override
    public void onEvent(String event) {
        System.out.println("收到事件: " + event);
    }
}

// 创建消息总线
class EventBus {
    // 使用一个map维护，消息类型和该消息的订阅者
    private Map<String, List<Subscriber>> subscribers = new HashMap<>();

    // 订阅一个消息
    public void subscribe(String eventType, Subscriber subscriber) {
        subscribers.computeIfAbsent(eventType, k -> new ArrayList<>
()).add(subscriber);
    }

    // 接触订阅
    public void unsubscribe(String eventType, Subscriber subscriber) {
        List<Subscriber> subs = subscribers.get(eventType);
        if (subs != null) {
            subs.remove(subscriber);
        }
    }
```

```
    // 发布事件
    public void publish(String eventType, String event) {
        List<Subscriber> subs = subscribers.get(eventType);
        if (subs != null) {
            for (Subscriber subscriber : subs) {
                subscriber.onEvent(event);
            }
        }
    }
}

// 使用示例：
public class Main {
    public static void main(String[] args) {
        EventBus eventBus = new EventBus();
        Subscriber subscriber1 = new ConcreteSubscriber();
        Subscriber subscriber2 = new ConcreteSubscriber();
        // 订阅事件
        eventBus.subscribe("eventA", subscriber1);
        eventBus.subscribe("eventA", subscriber2);

        // 发布事件
        eventBus.publish("eventA", "这是事件A的内容");

        // 取消订阅
        eventBus.unsubscribe("eventA", subscriber1);

        // 再次发布事件
        eventBus.publish("eventA", "这是事件A的新内容");
    }
}
```

总结一下两者的区别：

1. 通信方式：观察者模式中，观察者与被观察者之间存在直接的关联关系，而发布-订阅模式中，发布者和订阅者通过一个第三方组件（消息代理或事件总线）进行通信，彼此之间不存在直接关联关系。
2. 系统复杂性：发布-订阅模式引入了一个额外的组件（消息代理或事件总线），增加了系统的复杂性，但同时也提高了系统的灵活性和可扩展性。
3. 使用场景：观察者模式适用于需要将状态变化通知给其他对象的情况，而发布-订阅模式适用于事件驱动的系统，尤其是那些需要跨越多个模块或组件进行通信的场景。

希望这个解释能帮助您理解发布-订阅模式和观察者模式之间的区别。如果您有其他问题，欢迎继续提问。

发布-订阅模式和传统的观察者模式相比，在某些方面具有优势。以下是发布-订阅模式相对于观察者模式的一些优点：

1. 解耦：在发布-订阅模式中，发布者和订阅者之间没有直接关联，它们通过一个中间组件（消息代理或事件总线）进行通信。这种间接通信可以使发布者和订阅者在运行时动态地添加或删除，从而进一步降低了它们之间的耦合度。
2. 可扩展性：发布-订阅模式允许您更容易地向系统中添加新的发布者和订阅者，而无需修改现有的代码。这使得系统在不同组件之间通信时具有更好的可扩展性。
3. 模块化：由于发布者和订阅者之间的通信通过中间组件进行，您可以将系统划分为更小、更独立的模块。这有助于提高代码的可维护性和可读性。
4. 异步通信：发布-订阅模式通常支持异步消息传递，这意味着发布者和订阅者可以在不同的线程或进程中运行。这有助于提高系统的并发性能和响应能力。
5. 消息过滤：在发布-订阅模式中，可以利用中间组件对消息进行过滤，使得订阅者只接收到感兴趣的消息。这可以提高系统的性能，减少不必要的通信开销。

然而，发布-订阅模式也有一些缺点，例如增加了系统的复杂性，因为引入了额外的中间组件。根据具体的应用场景和需求来选择合适的设计模式是很重要的。在某些情况下，观察者模式可能更适合，而在其他情况下，发布-订阅模式可能是更好的选择。

还是以上边的股票交易系统为例。在这个系统中，股票价格的变化会作为事件发布，投资者可以订阅这些股票价格变化事件。当股票价格发生变化时，所有订阅了该股票的投资者都会收到通知。

首先，我们创建一个`Subscriber`接口，用于表示订阅者（投资者）：

```
public interface Subscriber {
    void onStockPriceChanged(String stockSymbol, double newPrice);
}
```

接下来，我们创建一个`EventBus`类，用于管理发布者和订阅者之间的通信：

```
public class EventBus {
    private Map<String, List<Subscriber>> subscribers = new HashMap<>();
```

```
    // 订阅股票价格变化事件
    public void subscribe(String stockSymbol, Subscriber subscriber) {
        subscribers.computeIfAbsent(stockSymbol, k -> new ArrayList<>
()).add(subscriber);
    }

    // 取消订阅股票价格变化事件
    public void unsubscribe(String stockSymbol, Subscriber subscriber) {
        List<Subscriber> subs = subscribers.get(stockSymbol);
        if (subs != null) {
            subs.remove(subscriber);
        }
    }

    // 发布股票价格变化事件
    public void publish(String stockSymbol, double newPrice) {
        List<Subscriber> subs = subscribers.get(stockSymbol);
        if (subs != null) {
            for (Subscriber subscriber : subs) {
                subscriber.onStockPriceChanged(stockSymbol, newPrice);
            }
        }
    }
}
```

然后，我们创建一个具体的订阅者实现，例如`Investor`类：

```
public class Investor implements Subscriber {
    private String name;

    public Investor(String name) {
        this.name = name;
    }

    // 股票代号，新价格
    @Override
    public void onStockPriceChanged(String stockSymbol, double newPrice) {
        System.out.println(name + " 收到股票 " + stockSymbol + " 价格变化通知，新
价格：" + newPrice);
    }
}
```

最后一个定义一个股票类：

```
// 股票
public class Stock{
    private double stockPrice;

    public void stockPriceChanged(double newPrice) {
        this.stockPrice = newPrice;
        eventBus.publish("AAPL", 210.0);
    }
}
```

我们来看一个使用示例：

```
public class Main {
    public static void main(String[] args) {
        EventBus eventBus = new EventBus();
        Investor investor1 = new Investor("投资者A");
        Investor investor2 = new Investor("投资者B");

        // 订阅股票价格变化事件
        eventBus.subscribe("AAPL", investor1);
        eventBus.subscribe("AAPL", investor2);

        Stock stock = new Stock();
        stock.stockPriceChanged(21.5);

        // 取消订阅股票价格变化事件
        eventBus.unsubscribe("AAPL", investor1);

        // 再次发布股票价格变化事件
        stock.stockPriceChanged(21.5);
    }
}
```

这个示例中，`Investor`类实现了`Subscriber`接口，代表投资者。`EventBus`类是负责管理发布者和订阅者之间通信的中间组件。投资者可以通过`EventBus`订阅股票价格

四、源码使用

1、jdk中的观察者

java.util.Observable类实现了主题（Subject）的功能，而java.util.Observer接口则定义了观察者（Observer）的方法。

通过调用Observable对象的notifyObservers()方法，可以通知所有注册的Observer对象，让它们更新自己的状态。

一下是一个使用案例：假设有一个银行账户类，它的余额是可变的。当余额发生变化时，需要通知所有的观察者（比如说银行客户），以便它们更新自己的显示信息。

```
// 银行账户类
public class BankAccount extends Observable {
    private double balance;

    // 构造函数
    public BankAccount(double balance) {
        this.balance = balance;
    }

    // 存款操作
    public void deposit(double amount) {
        balance += amount;
        setChanged(); // 表示状态已经改变
        notifyObservers(); // 通知所有观察者
    }

    // 取款操作
    public void withdraw(double amount) {
        balance -= amount;
        setChanged(); // 表示状态已经改变
        notifyObservers(); // 通知所有观察者
    }

    // 获取当前余额
    public double getBalance() {
        return balance;
    }

    // 主函数
    public static void main(String[] args) {
```

```
        BankAccount account = new BankAccount(1000.0);
        // 创建观察者
        Observer observer1 = new Observer() {
            @Override
            public void update(Observable o, Object arg) {
                System.out.println("客户1: 余额已更新为 " +
((BankAccount)o).getBalance());
            }
        };
        Observer observer2 = new Observer() {
            @Override
            public void update(Observable o, Object arg) {
                System.out.println("客户2: 余额已更新为 " +
((BankAccount)o).getBalance());
            }
        };
        // 注册观察者
        account.addObserver(observer1);
        account.addObserver(observer2);
        // 存款操作，触发观察者更新
        account.deposit(100.0);
        // 取款操作，触发观察者更新
        account.withdraw(50.0);
    }
}
```

这个案例中，BankAccount类继承了java.util.Observable类，表示它是一个主题（Subject）。在存款或取款操作时，它会调用setChanged()方法表示状态已经改变，并调用notifyObservers()方法通知所有观察者（Observer）。

在主函数中，我们创建了两个观察者（observer1和observer2），它们分别实现了Observer接口的update()方法。当观察者收到更新通知时，它们会执行自己的业务逻辑，比如更新显示信息。

这个案例演示了观察者模式在银行系统中的应用，通过观察者模式可以实现银行客户对自己账户余额的实时监控。

2、Guava中的消息总线

Guava 库中的 `EventBus` 类提供了一个简单的消息总线实现，可以帮助您在 Java 应用程序中实现发布-订阅模式。以下是一个简单的示例，演示了如何使用 Guava 的 `EventBus` 来实现一个简单的消息发布和订阅功能。

首先，确保您已将 Guava 添加到项目的依赖项中。如果您使用 Maven，请在 `pom.xml` 文件中添加以下依赖项：

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>30.1-jre</version>
</dependency>
```

接下来，定义一个事件类，例如 `MessageEvent`：

```
public class MessageEvent {
    private String message;

    public MessageEvent(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

现在，创建一个订阅者类，例如 `MessageSubscriber`。在订阅者类中，定义一个方法并使用 `@Subscribe` 注解标记该方法，以便 `EventBus` 能够识别该方法作为事件处理器：

```
public class MessageSubscriber {
    @Subscribe
    public void handleMessageEvent(MessageEvent event) {
        System.out.println("收到消息: " + event.getMessage());
    }
}
```

最后，我们来看一个使用示例：

```
public class Main {
    public static void main(String[] args) {
        // 创建 EventBus 实例
        EventBus eventBus = new EventBus();

        // 创建并注册订阅者
```

```
        MessageSubscriber subscriber = new MessageSubscriber();
        eventBus.register(subscriber);

        // 发布事件
        eventBus.post(new MessageEvent("Hello, EventBus!"));

        // 取消注册订阅者
        eventBus.unregister(subscriber);

        // 再次发布事件（此时订阅者已取消注册，将不会收到消息）
        eventBus.post(new MessageEvent("Another message"));
    }
}
```

在这个示例中，我们创建了一个 `EventBus` 实例，然后创建并注册了一个 `MessageSubscriber` 类型的订阅者。当我们使用 `eventBus.post()` 方法发布一个 `MessageEvent` 事件时，订阅者的 `handleMessageEvent` 方法将被调用，并输出收到的消息。

注意，如果订阅者处理事件的方法抛出异常，`EventBus` 默认情况下不会对异常进行处理。如果需要处理异常，可以在创建 `EventBus` 实例时传入一个自定义的 `SubscriberExceptionHandler`。

五、进阶

观察者模式的应用场景非常广泛，小到代码层面的解耦，大到架构层面的系统解耦，再或者一些产品的设计思路，都有这种模式的影子。

不同的应用场景和需求下，这个模式也有截然不同的实现方式，之前我们所列举的所有的例子都是同步阻塞的实现方式，当然我们的观察者设计模式也有异步非阻塞的实现方式；有进程内的实现方式，也有跨进程的实现方式。

之前讲到的实现方式，是一种同步阻塞的实现方式。观察者和被观察者代码在同一个线程内执行，被观察者一直阻塞，直到所有的观察者代码都执行完成之后，才执行后续的代码。对照上面讲到的用户注册的例子，register() 函数依次调用执行每个观察者的 handleRegSuccess() 函数，等到都执行完成之后，才会返回结果给客户端。

如果注册接口是一个调用**比较频繁的接口**，对性能非常敏感，希望接口的响应时间尽可能短，那我们可以将同步阻塞的实现方式改为**异步非阻塞的实现方式**，以此来**减少响应时间**。

1、异步非阻塞模型

首先，我们需要创建一个通用的观察者接口`Observer`和一个被观察者接口`Observable`。

Observer.java:

```
public interface Observer {
    void update(String message);
}
```

Observable.java:

```
public interface Observable {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String message);
}
```

接下来，我们需要实现一个具体的被观察者类`Subject`和一个具体的观察者类`ConcreteObserver`。

Subject.java:

```
public class Subject implements Observable {
    private List<Observer> observers;
    private ExecutorService executorService;

    public Subject() {
        observers = new ArrayList<>();
        executorService = Executors.newCachedThreadPool();
    }

    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
```

```
    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            executorService.submit(() -> observer.update(message));
        }
    }

    public void setMessage(String message) {
        notifyObservers(message);
    }
}
```

ConcreteObserver.java:

```
public class ConcreteObserver implements Observer {
    private String name;

    public ConcreteObserver(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        System.out.println(name + " received message: " + message);
    }
}
```

最后，我们可以创建一个简单的示例来测试实现的异步非阻塞观察者模式。

Main.java:

```
public class Main {
    public static void main(String[] args) {
        Subject subject = new Subject();
        ConcreteObserver observer1 = new ConcreteObserver("Observer 1");
        ConcreteObserver observer2 = new ConcreteObserver("Observer 2");
        ConcreteObserver observer3 = new ConcreteObserver("Observer 3");

        subject.addObserver(observer1);
        subject.addObserver(observer2);
        subject.addObserver(observer3);

        subject.setMessage("Hello, observers!");
```

```
        // 等待异步任务完成
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

在这个示例中，我们使用了`ExecutorService`的线程池来实现异步非阻塞的通知。每个观察者更新操作都将作为一个任务提交给线程池并异步执行。这将确保性能敏感的场景不会因为观察者的通知而阻塞。

2、跨进程通信

刚刚讲到的两个场景，**不管是同步阻塞实现方式还是异步非阻塞实现方式，都是进程内的实现方式。**如果用户注册成功之后，我们需要**发送用户信息给大数据征信系统**，而大数据征信系统是一个独立的系统，跟它之间的交互是跨不同进程的，那如何实现一个跨进程的观察者模式呢？

如果大数据征信系统提供了发送用户注册信息的 RPC 接口，我们仍然可以沿用之前的实现思路，在 notifyObservers() 函数中调用 RPC 接口来发送数据。但是，我们还有更加优雅、更加常用的一种实现方式，那就是基于消息队列（Message Queue，比如 ActiveMQ）来实现。

当然，这种实现方式也有弊端，那就是需要引入一个新的系统（消息队列），增加了维护成本。不过，它的好处也非常明显。在原来的实现方式中，**观察者需要注册到被观察者中**，被观察者需要依次遍历观察者来发送消息。而**基于消息队列的实现方式，被观察者和观察者解耦更加彻底，两部分的耦合更小。**被观察者完全不感知观察者，同理，观察者也完全不感知被观察者。被观察者只管发送消息到消息队列，观察者只管从消息队列中读取消息来执行相应的逻辑。

第二章 模板模式

今天，我们再学习另外一种行为型设计模式，模板模式。我们多次强调，绝大部分设计模式的原理和实现，都非常简单，难的是掌握应用场景，搞清楚能解决什么问题。模板模式也不例外。模板模式主要是用来解决复用和扩展两个问题。我们今天会结合 Java Servlet、JUnit TestCase、Java InputStream、Java AbstractList 四个例子来具体讲解这两个作用。

一、原理与实现

模板模式，全称是模板方法设计模式，英文是 Template Method Design Pattern。在 GoF 的《设计模式》一书中，它是这么定义的：

> Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

翻译成中文就是：模板方法模式在一个方法中定义一个**算法骨架**，并将某些步骤**推迟到子类中实现**。模板方法模式可以让子类在不改变算法整体结构的情况下，**重新定义算法中的某些步骤**。

这里的“算法”，我们可以理解为广义上的“业务逻辑”，并不特指数据结构和算法中的“算法”。这里的算法骨架就是“模板”，包含算法骨架的方法就是“模板方法”，这也是模板方法模式名字的由来。

原理很简单，代码实现就更加简单，我写了一个示例代码，如下所示。templateMethod() 函数定义为 final，是为了避免子类重写它。method1() 和 method2() 定义为 abstract，是为了强迫子类去实现。不过，这些都不是必须的，在实际的项目开发中，模板模式的代码实现比较灵活，待会儿讲到应用场景的时候，我们会有具体的体现。

下面是一个简单的Java示例，展示了如何使用模板方法设计模式：

1、首先，创建一个抽象类，定义算法的骨架：

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

    // 基本方法，定义算法中不会变化的步骤
    private void step1() {
        System.out.println("Step 1: Prepare the ingredients.");
```

```
    }

    // 抽象方法，定义算法中需要子类实现的步骤
    protected abstract void step2();

    // 基本方法，定义算法中不会变化的步骤
    private void step3() {
        System.out.println("Step 3: Serve the dish.");
    }
}
```

2、然后，创建具体的子类，实现抽象类中定义的抽象方法：

```
public class ConcreteTemplateA extends AbstractTemplate {
    @Override
    protected void step2() {
        System.out.println("Step 2 (A): Cook the dish using method A.");
    }
}

public class ConcreteTemplateB extends AbstractTemplate {
    @Override
    protected void step2() {
        System.out.println("Step 2 (B): Cook the dish using method B.");
    }
}
```

3、最后，在客户端代码中使用模板方法：

```
public class Main {
    public static void main(String[] args) {
        AbstractTemplate templateA = new ConcreteTemplateA();
        AbstractTemplate templateB = new ConcreteTemplateB();

        System.out.println("Using Template A:");
        templateA.templateMethod();

        System.out.println("\nUsing Template B:");
        templateB.templateMethod();
    }
}
```

运行上面的程序，输出如下：

```
vbnetCopy code
Using Template A:
Step 1: Prepare the ingredients.
Step 2 (A): Cook the dish using method A.
Step 3: Serve the dish.

Using Template B:
Step 1: Prepare the ingredients.
Step 2 (B): Cook the dish using method B.
Step 3: Serve the dish.
```

这个例子中，`AbstractTemplate`是一个抽象类，它定义了一个名为`templateMethod`的模板方法。该方法包含三个步骤：`step1`、`step2`和`step3`。其中，`step1`和`step3`是基本方法，它们的实现在抽象类中定义且不会改变。`step2`是一个抽象方法，需要子类（如`ConcreteTemplateA`和`ConcreteTemplateB`）根据具体需求实现。客户端代码通过创建子类的实例并调用`templateMethod`方法来执行算法。

这种使用场景在源码中非常常见，我们下边开始学习和体会。

二、源码中的作用

1、复用

开篇的时候，我们讲到模板模式有两大作用：复用和扩展。我们先来看它的第一个作用：复用。

模板模式把一个算法中**不变的流程抽象到父类的模板方法 templateMethod() 中**，将可变的部分 **step2()留给子类来实现。**所有的子类都可以复用父类中模板方法定义的流程代码。我们通过两个小例子来更直观地体会一下。

（1）Java InputStream

Java IO 类库中，有很多类的设计用到了模板模式，比如 InputStream、OutputStream、Reader、Writer。我们拿 InputStream 来举例说明一下。

我把 InputStream 部分相关代码贴在了下面。在代码中，read() 函数是一个模板方法，定义了读取数据的整个流程，并且暴露了一个可以由子类来定制的抽象方法。不过这个方法也被命名为了 read()，只是参数跟模板方法不同。

```
public abstract class InputStream implements Closeable {
  //...省略其他代码...
```

```
    public int read(byte b[], int off, int len) throws IOException {
        Objects.checkFromIndexSize(off, len, b.length);
        if (len == 0) {
            return 0;
        }

        int c = read();
        if (c == -1) {
            return -1;
        }
        b[off] = (byte)c;

        int i = 1;
        try {
            for (; i < len ; i++) {
                c = read();
                if (c == -1) {
                    break;
                }
                b[off + i] = (byte)c;
            }
        } catch (IOException ee) {
        }
        return i;
    }

    public abstract int read() throws IOException;
}

// 这里有一个具体的实现类。用于从一个字节缓冲区中读取一个字节。方法的签名和功能如下：
public class ByteArrayInputStream extends InputStream {
    //...省略其他代码...

    @Override
    public synchronized int read() {
        return (pos < count) ? (buf[pos++] & 0xff) : -1;
    }
}
```

（2）Java AbstractList

在 Java AbstractList 类中，addAll() 函数可以看作模板方法，add() 是子类需要重写的方法，尽管没有声明为 abstract 的，但函数实现直接抛出了 UnsupportedOperationException 异常。前提是，如果子类不重写是不能使用的。

```
public boolean addAll(int index, Collection<? extends E> c) {
    rangeCheckForAdd(index);
    boolean modified = false;
    for (E e : c) {
        add(index++, e);
        modified = true;
    }
    return modified;
}

public void add(int index, E element) {
    throw new UnsupportedOperationException();
}
```

其在ArrayList中的实现如下：

```
public void add(int index, E element) {
    rangeCheckForAdd(index);
    checkForComodification();
    root.add(offset + index, element);
    updateSizeAndModCount(1);
}
```

2、扩展

模板模式的第二大作用的是**扩展。**这里所说的扩展，并不是指代码的扩展性，而是指**框架的扩展性**，基于这个作用，模板模式常用在**框架的开发中**，让框架用户可以在不修改框架源码的情况下，定制化框架的功能。我们通过 Junit TestCase、Java Servlet 两个例子来解释一下。

（1）Java Servlet

对于 Java Web 项目开发来说，常用的开发框架是 SpringMVC。利用它，我们只需要关注业务代码的编写，底层的原理几乎不会涉及。但是，如果我们抛开这些高级框架来开发 Web 项目，必然会用到 Servlet。实际上，使用比较底层的 Servlet 来开发 Web 项目也不难。我们只需要定义一个继承 HttpServlet 的类，并且重写其中的 doGet() 或 doPost() 方法，来分别处理 get 和 post 请求。具体的代码示例如下所示：

```
public class HelloServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
    this.doPost(req, resp);
  }

  @Override
  protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
    resp.getWriter().write("Hello World.");
  }
}
```

除此之外，我们还需要在配置文件 web.xml 中做如下配置。Tomcat、Jetty 等 Servlet 容器在启动的时候，会自动加载这个配置文件中的 URL 和 Servlet 之间的映射关系。

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>com.xzg.cd.HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

当我们在浏览器中输入网址（比如，http://127.0.0.1:8080/hello ）的时候，Servlet 容器会接收到相应的请求，并且根据 URL 和 Servlet 之间的映射关系，找到相应的 Servlet（HelloServlet），然后执行它的 service() 方法。service() 方法定义在父类 HttpServlet 中，它会调用 doGet() 或 doPost() 方法，然后输出数据（“Hello world”）到网页。

我们现在来看，HttpServlet 的 service() 函数长什么样子。

```
public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException
{
    HttpServletRequest  request;
    HttpServletResponse response;
    if (!(req instanceof HttpServletRequest &&
          res instanceof HttpServletResponse)) {
        throw new ServletException("non-HTTP request or response");
    }
    request = (HttpServletRequest) req;
    response = (HttpServletResponse) res;
    service(request, response);
}

protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    String method = req.getMethod();
    if (method.equals(METHOD_GET)) {
        long lastModified = getLastModified(req);
        if (lastModified == -1) {
            // servlet doesn't support if-modified-since, no reason
            // to go through further expensive logic
            doGet(req, resp);
        } else {
            long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
            if (ifModifiedSince < lastModified) {
                // If the servlet mod time is later, call doGet()
                // Round down to the nearest second for a proper compare
                // A ifModifiedSince of -1 will always be less
                maybeSetLastModified(resp, lastModified);
                // 子类实现的扩展点
                doGet(req, resp);
            } else {
                resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
            }
        }
    } else if (method.equals(METHOD_HEAD)) {
        long lastModified = getLastModified(req);
        maybeSetLastModified(resp, lastModified);
        doHead(req, resp);
    } else if (method.equals(METHOD_POST)) {
```

```
            // 子类实现的扩展点
            doPost(req, resp);
        } else if (method.equals(METHOD_PUT)) {
            // 子类实现的扩展点
            doPut(req, resp);
        } else if (method.equals(METHOD_DELETE)) {
            // 子类实现的扩展点
            doDelete(req, resp);
        } else if (method.equals(METHOD_OPTIONS)) {
            // 子类实现的扩展点
            doOptions(req,resp);
        } else if (method.equals(METHOD_TRACE)) {
            // 子类实现的扩展点
            doTrace(req,resp);
        } else {
            String errMsg = lStrings.getString("http.method_not_implemented");
            Object[] errArgs = new Object[1];
            errArgs[0] = method;
            errMsg = MessageFormat.format(errMsg, errArgs);
            resp.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED, errMsg);
        }
    }
```

从上面的代码中我们可以看出，HttpServlet 的 service() 方法就是一个模板方法，它实现了整个 HTTP 请求的执行流程，doGet()、doPost() 是模板中可以由子类来定制的部分。实际上，这就相当于 Servlet 框架提供了一个扩展点（doGet()、doPost() 方法），让框架用户在不用修改 Servlet 框架源码的情况下，将业务代码通过扩展点镶嵌到框架中执行。

（2）spring中的核心refresh

spring中存在大量的模板方法，我们列举最核心的refresh方法：

```
@Override
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        StartupStep contextRefresh =
this.applicationStartup.start("spring.context.refresh");

        // Prepare this context for refreshing.
        prepareRefresh();
```

```
        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        // Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);

            StartupStep beanPostProcess =
this.applicationStartup.start("spring.context.beans.post-process");
            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);

            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);
            beanPostProcess.end();

            // Initialize message source for this context.
            initMessageSource();

            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            onRefresh();

            // Check for listener beans and register them.
            registerListeners();

            // Instantiate all remaining (non-lazy-init) singletons.
            finishBeanFactoryInitialization(beanFactory);

            // Last step: publish corresponding event.
            finishRefresh();
        }

        catch (BeansException ex) {
            if (logger.isWarnEnabled()) {
                logger.warn("Exception encountered during context initialization - " +
                        "cancelling refresh attempt: " + ex);
            }
```

```
            // Destroy already created singletons to avoid dangling resources.
            destroyBeans();

            // Reset 'active' flag.
            cancelRefresh(ex);

            // Propagate exception to caller.
            throw ex;
        }

        finally {
            // Reset common introspection caches in Spring's core, since we
            // might not ever need metadata for singleton beans anymore...
            resetCommonCaches();
            contextRefresh.end();
        }
    }
}
```

(3) MyBatis

MyBatis框架中也有运用模板方法设计模式的例子，尽管它们的实现方式可能没有那么明显。以下是一个常见的例子：

1. `BaseExecutor`：在MyBatis中，`BaseExecutor`是一个抽象类，它提供了查询、更新等数据库操作的通用实现。具体的数据库操作是通过它的子类（如`SimpleExecutor`、`ReuseExecutor`和`BatchExecutor`）来实现的。在`BaseExecutor`中，有一个名为`query`的模板方法，它包含了查询操作的通用逻辑。这个方法的部分实现如下：

```
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql)
throws SQLException {
    ErrorContext.instance().resource(ms.getResource()).activity("executing a
query").object(ms.getId());
    if (closed) {
        throw new ExecutorException("Executor was closed.");
    }
    if (queryStack == 0 && ms.isFlushCacheRequired()) {
        clearLocalCache();
```

```
    }
    List<E> list;
    try {
      queryStack++;
      list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key,
boundSql);
    } finally {
      queryStack--;
    }
    if (queryStack == 0) {
      for (DeferredLoad deferredLoad : deferredLoads) {
        deferredLoad.load();
      }
      deferredLoads.clear();
      if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
        clearLocalCache();
      }
    }
    return list;
}


 private <E> List<E> queryFromDatabase(MappedStatement ms, Object
parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key,
BoundSql boundSql) throws SQLException {
    List<E> list;
    localCache.putObject(key, EXECUTION_PLACEHOLDER);
    try {
      list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
    } finally {
      localCache.removeObject(key);
    }
    localCache.putObject(key, list);
    if (ms.getStatementType() == StatementType.CALLABLE) {
      localOutputParameterCache.putObject(key, parameter);
    }
    return list;
  }


protected abstract <E> List<E> doQuery(MappedStatement ms, Object
parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql
boundSql)
```

```
    throws SQLException;
```

在这个`query`方法中，`BaseExecutor`定义了查询操作的通用逻辑，例如异常处理、资源清理等。而具体的查询操作是通过`queryFromDatabase`方法来实现的，这是一个抽象方法，需要由`BaseExecutor`的子类实现。通过这种方式，MyBatis实现了对数据库操作的通用逻辑和具体逻辑的分离，提高了代码的可维护性和可扩展性。

虽然MyBatis中使用模板方法设计模式的例子没有Spring那么明显，但在实际开发过程中，我们可以借鉴这种设计思想，将通用逻辑抽象到模板中，以降低代码复杂性。

三、应用场景

1、电商系统

将通用逻辑和特定逻辑分离，提高代码复用性和可维护性。以下是一些典型的应用场景：

1. **支付流程**：电商系统通常需要支持多种支付方式（如信用卡支付、支付宝支付、微信支付等）。虽然不同的支付方式在实现细节上有所不同，但它们的整体流程是相似的。可以使用模板方法设计模式创建一个支付流程抽象类，定义通用的支付流程骨架，然后通过子类实现各种具体支付方式的逻辑。
2. **订单处理**：电商系统的订单处理流程通常包括一系列步骤，如验证库存、计算价格、生成运单等。这些步骤中，有些是通用的，而有些可能因订单类型、商品类型等因素而异。可以使用模板方法设计模式创建一个订单处理抽象类，定义通用的订单处理流程骨架，然后通过子类实现特定订单类型或商品类型的逻辑。
3. **促销策略**：电商系统中的促销活动通常具有多种策略（如满减、打折、赠品等）。尽管不同策略的具体实现不同，但它们都需要进行一些通用操作（如获取用户信息、验证促销条件等）。可以使用模板方法设计模式创建一个促销策略抽象类，定义通用的促销操作骨架，然后通过子类实现各种具体促销策略的逻辑。
4. **报表生成**：电商系统需要生成各种报表（如销售报表、库存报表、财务报表等）。这些报表在数据查询和报表样式上可能有所不同，但它们的生成流程是类似的（如查询数据、生成报表、导出文件等）。可以使用模板方法设计模式创建一个报表生成抽象类，定义通用的报表生成流程骨架，然后通过子类实现具体报表类型的逻辑。

这些应用场景展示了如何在电商系统中利用模板方法设计模式来简化代码结构和提高可维护性。在实际开发过程中，可以根据业务需求和系统架构选择合适的设计模式。

我们以支付流程为例：

在电商系统中，支付流程是一个典型的应用场景。我们可以使用模板方法设计模式创建一个抽象类`PaymentProcessor`来定义通用的支付流程骨架，然后通过子类实现各种具体支付方式的逻辑。以下是一个简化的代码示例及其中文注释：

首先，我们创建一个抽象类`PaymentProcessor`来定义支付流程：

```
public abstract class PaymentProcessor {

    // 模板方法，定义支付流程骨架
    public final void processPayment(Order order) {
        // 获取支付方式（如信用卡、支付宝、微信等）
        String paymentMethod = getPaymentMethod();

        // 验证订单信息（如订单金额、收货地址等）
        validateOrder(order);

        // 验证支付信息（如支付账号、支付密码等）
        validatePaymentInfo(paymentMethod);

        // 执行支付
        executePayment(paymentMethod, order);

        // 发送支付通知
        sendPaymentNotification(order);
    }

    // 获取支付方式，具体实现由子类提供
    protected abstract String getPaymentMethod();

    // 验证订单信息，通用逻辑
    private void validateOrder(Order order) {
        // 验证订单信息的实现
    }

    // 验证支付信息，具体实现由子类提供
    protected abstract void validatePaymentInfo(String paymentMethod);

    // 执行支付，具体实现由子类提供
    protected abstract void executePayment(String paymentMethod, Order
order);

    // 发送支付通知，通用逻辑
    private void sendPaymentNotification(Order order) {
```

```
        // 发送支付通知的实现
    }
}
```

接下来，我们创建一个具体的支付处理器类`AlipayProcessor`来实现支付宝支付方式：

```
public class AlipayProcessor extends PaymentProcessor {

    @Override
    protected String getPaymentMethod() {
        return "Alipay";
    }

    @Override
    protected void validatePaymentInfo(String paymentMethod) {
        // 验证支付宝支付信息的实现
    }

    @Override
    protected void executePayment(String paymentMethod, Order order) {
        // 执行支付宝支付的实现
    }
}
```

同样，我们可以创建一个具体的支付处理器类`WechatPayProcessor`来实现微信支付方式：

```
public class WechatPayProcessor extends PaymentProcessor {

    @Override
    protected String getPaymentMethod() {
        return "WechatPay";
    }

    @Override
    protected void validatePaymentInfo(String paymentMethod) {
        // 验证微信支付信息的实现
    }

    @Override
    protected void executePayment(String paymentMethod, Order order) {
```

```
        // 执行微信支付的实现
    }
}
```

在这个例子中，`PaymentProcessor`定义了支付流程的通用骨架，如验证订单信息、发送支付通知等。具体的支付方式（如支付宝、微信支付等）由子类`AlipayProcessor`和`WechatPayProcessor`实现。这样，我们可以轻松地添加新的支付方式，而不需要修改现有的支付流程代码，从而提高代码的可维护性和可扩展性。

现在，我们可以使用`AlipayProcessor`和`WechatPayProcessor`来处理不同的支付方式。例如，当用户选择支付宝支付时，我们可以创建一个`AlipayProcessor`实例来处理支付流程：

```
public class PaymentService {

    public void processPayment(Order order, String paymentType) {
        PaymentProcessor paymentProcessor;

        if ("Alipay".equalsIgnoreCase(paymentType)) {
            paymentProcessor = new AlipayProcessor();
        } else if ("WechatPay".equalsIgnoreCase(paymentType)) {
            paymentProcessor = new WechatPayProcessor();
        } else {
            throw new IllegalArgumentException("Unsupported payment type: " +
paymentType);
        }

        paymentProcessor.processPayment(order);
    }
}
```

在`PaymentService`类中，我们根据用户选择的支付方式创建相应的支付处理器实例，然后调用`processPayment`方法来处理支付流程。这种方式使得支付流程的处理逻辑更加清晰，易于维护和扩展。

需要注意的是，为了更好地支持新的支付方式，我们可以考虑使用工厂模式或策略模式来创建支付处理器实例，进一步提高代码的可维护性和可扩展性。

这个例子展示了如何在电商系统的支付流程中应用模板方法设计模式。通过将通用逻辑和特定逻辑分离，我们可以更轻松地添加新的支付方式，同时保持代码的清晰和易于维护。在实际开发过程中，可以根据业务需求和系统架构灵活地运用模板方法设计模式。

2、在线考试系统

在一个在线考试系统中，我们可以使用模板方法设计模式处理不同类型的题目。在这个场景中，我们可以将通用逻辑（例如展示题目、计算分数等）与特定题目类型的逻辑（例如判断题的判断逻辑、选择题的选择逻辑等）分离。

首先，我们创建一个抽象类`Question`来定义题目处理的通用骨架：

```
public abstract class Question {

    // 模板方法，定义题目处理流程
    public final void processQuestion(String userAnswer) {
        // 展示题目
        displayQuestion();

        // 检查用户答案
        boolean isCorrect = checkAnswer(userAnswer);

        // 计算分数
        int score = calculateScore(isCorrect);

        // 显示结果
        displayResult(isCorrect, score);
    }

    // 展示题目，通用逻辑
    protected void displayQuestion() {
        // 展示题目的实现
    }

    // 检查用户答案，具体实现由子类提供
    protected abstract boolean checkAnswer(String userAnswer);

    // 计算分数，通用逻辑
    protected int calculateScore(boolean isCorrect) {
        // 计算分数的实现
    }

    // 显示结果，通用逻辑
    protected void displayResult(boolean isCorrect, int score) {
        // 显示结果的实现
```

```
    }
}
```

接下来，我们创建一个具体的题目类`MultipleChoiceQuestion`来实现选择题的逻辑：

```
public class MultipleChoiceQuestion extends Question {

    private String correctAnswer;

    public MultipleChoiceQuestion(String correctAnswer) {
        this.correctAnswer = correctAnswer;
    }

    @Override
    protected boolean checkAnswer(String userAnswer) {
        // 检查选择题答案的实现
        return correctAnswer.equalsIgnoreCase(userAnswer);
    }
}
```

同样，我们可以创建一个具体的题目类`TrueOrFalseQuestion`来实现判断题的逻辑：

```
public class TrueOrFalseQuestion extends Question {

    private boolean correctAnswer;

    public TrueOrFalseQuestion(boolean correctAnswer) {
        this.correctAnswer = correctAnswer;
    }

    @Override
    protected boolean checkAnswer(String userAnswer) {
        // 检查判断题答案的实现
        return correctAnswer == Boolean.parseBoolean(userAnswer);
    }
}
```

在这个例子中，`Question`定义了题目处理的通用骨架，如展示题目、计算分数等。具体的题目类型（如选择题、判断题等）由子类`MultipleChoiceQuestion`和`TrueOrFalseQuestion`实现。这样，我们可以轻松地添加新的题目类型，而不需要修改现有的题目处理代码，从而提高代码的可维护性和可扩展性。

现在，我们可以使用`MultipleChoiceQuestion`和`TrueOrFalseQuestion`来处理不同类型的题目。例如，当用户回答一个选择题时，我们可以创建一个`MultipleChoiceQuestion`实例来处理题目：

```
public class ExamService {

    public void processQuestion(Question question,String userAnswer) {
        // 调用processQuestion方法来处理题目
        question.processQuestion(userAnswer);
    }
}
```

在`ExamService`类中，我们调用`processQuestion`方法来处理不同类型的题目。这种方式使得题目处理的逻辑更加清晰，易于维护和扩展。
需要注意的是，为了更好地支持新的题目类型，我们可以考虑使用工厂模式或策略模式来创建题目实例，进一步提高代码的可维护性和可扩展性。
这个例子展示了如何在一个在线考试系统中应用模板方法设计模式。通过将通用逻辑和特定逻辑分离，我们可以更轻松地添加新的题目类型，同时保持代码的清晰和易于维护。在实际开发过程中，可以根据业务需求和系统架构灵活地运用模板方法设计模式。

四、Callback回调

模板模式与Callback回调函数有何区别和联系？

上一节课中，我们学习了模板模式的原理、实现和应用。它常用在框架开发中，通过提供功能扩展点，让框架用户在不修改框架源码的情况下，基于扩展点定制化框架的功能。除此之外，模板模式还可以起到代码复用的作用。

复用和扩展是模板模式的两大作用，实际上，还有另外一个技术概念，也能起到跟模板模式相同的作用，那就是**回调**（Callback）。今天我们今天就来看一下，回调的原理、实现和应用，以及它跟模板模式的区别和联系。

区别：

1. **设计范式**：模板方法设计模式通常用于面向对象编程（OOP），它依赖于继承和多态来实现代码复用。回调函数则通常用于函数式编程，它通过将函数作为参数传递给其他函数来实现代码复用。
2. **实现方式**：模板方法设计模式依赖于抽象类和子类之间的继承关系。在抽象类中定义一个算法的骨架，并将某些步骤延迟到子类中实现。而回调函数通过将一个函数作为参数传递给另一个函数，让调用者可以自定义特定的行为。

联系：

1. **目的**：模板方法设计模式和回调函数都旨在将变化的部分与不变的部分分离，提高代码的复用性和可维护性。
2. **实现相互关系**：在某些情况下，模板方法设计模式可以通过回调函数来实现。例如，在Java中，可以使用匿名内部类或者Lambda表达式作为回调函数，实现模板方法设计模式的目标。类似地，在面向对象的语言中，回调函数也可以通过模板方法设计模式来实现。

总之，模板方法设计模式和回调函数都是解决代码复用和灵活性问题的有效手段。在不同的编程范式和上下文中，可以根据需要选择适当的方法。在实际开发中，我们甚至可以将这两种方法结合使用，以获得更好的灵活性和扩展性。

1、回调的原理解析

相对于普通的函数调用来说，回调是一种双向调用关系。A 类事先注册某个函数 F 到 B 类，A 类在调用 B 类的 P 函数的时候，B 类反过来调用 A 类注册给它的 F 函数。这里的 F 函数就是“回调函数”。A 调用 B，B 反过来又调用 A，这种调用机制就叫作“回调”，如果我们学习过JavaScript这个概念应该十分清楚。

A 类如何将回调函数传递给 B 类呢？不同的编程语言，有不同的实现方法。C 语言可以使用函数指针，Java 则需要使用包裹了回调函数的类对象，我们简称为回调对象。这里我用 Java 语言举例说明一下。代码如下所示：

```
public interface ICallback {
    void methodToCallback();
}
public class BClass {
    public void process(ICallback callback) {
        //...
        callback.methodToCallback();
        //...
    }
}
public class AClass {
```

```
    public static void main(String[] args) {
        BClass b = new BClass();
        b.process(new ICallback() { //回调对象
            @Override
            public void methodToCallback() {
                System.out.println("Call back me.");
            }
        });
    }
}
```

上面就是 Java 语言中回调的典型代码实现。从代码实现中，我们可以看出，回调跟模板模式一样，也具有复用和扩展的功能。除了回调函数之外，BClass 类的 process() 函数中的逻辑都可以复用。如果 ICallback、BClass 类是框架代码，AClass 是使用框架的客户端代码，我们可以通过 ICallback 定制 process() 函数，也就是说，框架因此具有了扩展的能力。

实际上，回调不仅可以应用在代码设计上，在更高层次的架构设计上也比较常用。比如，通过三方支付系统来实现支付功能，用户在发起支付请求之后，一般不会一直阻塞到支付结果返回，而是注册回调接口（类似回调函数，一般是一个回调用的 URL）给三方支付系统，等三方支付系统执行完成之后，将结果通过回调接口返回给用户。

回调可以分为同步回调和异步回调（或者延迟回调）。同步回调指在函数返回之前执行回调函数；异步回调指的是在函数返回之后执行回调函数。上面的代码实际上是同步回调的实现方式，在 process() 函数返回之前，执行完回调函数 methodToCallback()。而上面支付的例子是异步回调的实现方式，发起支付之后不需要等待回调接口被调用就直接返回。从应用场景上来看，同步回调看起来更像模板模式，异步回调看起来更像观察者模式。

2、JdbcTemplate的回调应用

Spring 提供了很多 Template 类，比如，JdbcTemplate、RedisTemplate、RestTemplate。尽管都叫作 xxxTemplate，但它们**并非基于模板模式来实现的**，而是**基于回调**来实现的，确切地说应该是同步回调。而**同步回调从应用场景上很像模板模式**，所以，在命名上，这些类使用 Template（模板）这个单词作为后缀。

这些 Template 类的设计思路都很相近，所以，我们只拿其中的 JdbcTemplate 来举例分析一下。对于其他 Template 类，你可以阅读源码自行分析。

在前面的章节中，我们也多次提到，Java 提供了 JDBC 类库来封装不同类型的数据库操作。不过，直接使用 JDBC 来编写操作数据库的代码，还是有点复杂的。比如，下面这段是使用 JDBC 来查询用户信息的代码。

```
public class JdbcDemo {
    public User queryUser(long id) {
        Connection conn = null;
        Statement stmt = null;
        try {
            //1.加载驱动
            Class.forName("com.mysql.jdbc.Driver");
            conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/ydlclass", "root",
"123");
            //2.创建statement类对象，用来执行SQL语句
            stmt = conn.createStatement();
            //3.ResultSet类，用来存放获取的结果集
            String sql = "select * from user where id=" + id;
            ResultSet resultSet = stmt.executeQuery(sql);
            String eid = null, ename = null, price = null;
            while (resultSet.next()) {
                User user = new User();
                user.setId(resultSet.getLong("id"));
                user.setName(resultSet.getString("name"));
                user.setTelephone(resultSet.getString("telephone"));
                return user;
            }
        } catch (ClassNotFoundException e) {
            // TODO: log...
        } catch (SQLException e) {
            // TODO: log...
        } finally {
            if (conn != null)
                try {
                    conn.close();
                } catch (SQLException e) {
                    // TODO: log...
                }
            if (stmt != null)
                try {
                    stmt.close();
                } catch (SQLException e) {
```

```
                // TODO: log...
            }
        }
        return null;
    }
}
```

queryUser() 函数包含很多流程性质的代码，跟业务无关，比如，加载驱动、创建数据库连接、创建 statement、关闭连接、关闭 statement、处理异常。针对不同的 SQL 执行请求，这些流程性质的代码是相同的、可以复用的，我们不需要每次都重新敲一遍。

针对这个问题，Spring 提供了 JdbcTemplate，对 JDBC 进一步封装，来简化数据库编程。使用 JdbcTemplate 查询用户信息，我们只需要编写跟这个业务有关的代码，其中包括，查询用户的 SQL 语句、查询结果与 User 对象之间的映射关系。其他流程性质的代码都封装在了 JdbcTemplate 类中，不需要我们每次都重新编写。我用 JdbcTemplate 重写了上面的例子，代码简单了很多，如下所示：

```
public class JdbcTemplateDemo {
    @Resource
    private JdbcTemplate jdbcTemplate;
    public User queryUser(long id) {
        String sql = "select * from user where id="+id;
        return jdbcTemplate.query(sql, new UserRowMapper()).get(0);
    }

    class UserRowMapper implements RowMapper<User> {
        public User mapRow(ResultSet rs, int rowNum) throws SQLException {
            User user = new User();
            user.setId(rs.getLong("id"));
            user.setName(rs.getString("name"));
            user.setTelephone(rs.getString("telephone"));
            return user;
        }
    }
}
```

那 JdbcTemplate 底层具体是如何实现的呢？我们来看一下它的源码。因为 JdbcTemplate 代码比较多，我只摘抄了部分相关代码，贴到了下面。其中，JdbcTemplate 通过回调的机制，将不变的执行流程抽离出来，放到模板方法 execute() 中，将可变的部分设计成回调 StatementCallback，由用户来定制。

query() 函数是对 execute() 函数的二次封装，让接口用起来更加方便。

```
@Override
public <T> List<T> query(String sql, RowMapper<T> rowMapper) throws
DataAccessException {
   return query(sql, new RowMapperResultSetExtractor<T>(rowMapper));
}

@Override
public <T> T query(final String sql, final ResultSetExtractor<T> rse) throws
DataAccessException {
   Assert.notNull(sql, "SQL must not be null");
   Assert.notNull(rse, "ResultSetExtractor must not be null");
   if (logger.isDebugEnabled()) {
      logger.debug("Executing SQL query [" + sql + "]");
   }
   class QueryStatementCallback implements StatementCallback<T>, SqlProvider
{
      @Override
      public T doInStatement(Statement stmt) throws SQLException {
         ResultSet rs = null;
         try {
            rs = stmt.executeQuery(sql);
            ResultSet rsToUse = rs;
            if (nativeJdbcExtractor != null) {
               rsToUse = nativeJdbcExtractor.getNativeResultSet(rs);
            }
            return rse.extractData(rsToUse);
         }
         finally {
            JdbcUtils.closeResultSet(rs);
         }
      }
      @Override
      public String getSql() {
         return sql;
      }
   }
   return execute(new QueryStatementCallback());
}

@Override
```

```
public <T> T execute(StatementCallback<T> action) throws DataAccessException
{
   Assert.notNull(action, "Callback object must not be null");
   Connection con = DataSourceUtils.getConnection(getDataSource());
   Statement stmt = null;
   try {
      Connection conToUse = con;
      if (this.nativeJdbcExtractor != null &&

 this.nativeJdbcExtractor.isNativeConnectionNecessaryForNativeStatements()) {
         conToUse = this.nativeJdbcExtractor.getNativeConnection(con);
      }
      stmt = conToUse.createStatement();
      applyStatementSettings(stmt);
      Statement stmtToUse = stmt;
      if (this.nativeJdbcExtractor != null) {
         stmtToUse = this.nativeJdbcExtractor.getNativeStatement(stmt);
      }
      T result = action.doInStatement(stmtToUse);
      handleWarnings(stmt);
      return result;
   }
   catch (SQLException ex) {
      // Release Connection early, to avoid potential connection pool
deadlock
      // in the case when the exception translator hasn't been initialized yet.
      JdbcUtils.closeStatement(stmt);
      stmt = null;
      DataSourceUtils.releaseConnection(con, getDataSource());
      con = null;
      throw getExceptionTranslator().translate("StatementCallback",
getSql(action), ex);
   }
   finally {
      JdbcUtils.closeStatement(stmt);
      DataSourceUtils.releaseConnection(con, getDataSource());
   }
}
```

五、模板模式 VS 回调

回调的原理、实现和应用到此就都讲完了。接下来，我们从应用场景和代码实现两个角度，来对比一下模板模式和回调。

从应用场景上来看，同步回调跟模板模式几乎一致。它们都是在一个大的算法骨架中，自由替换其中的某个步骤，起到代**码复用和扩展**的目的。而**异步回调跟模板模式有较大差别，更像是观察者模式。**

从代码实现上来看，回调和模板模式完全不同。回调基于组合关系来实现，把一个对象传递给另一个对象，是一种对象之间的关系；模板模式基于继承关系来实现，子类重写父类的抽象方法，是一种类之间的关系。

前面我们也讲到，组合优于继承。实际上，这里也不例外。在代码实现上，**回调相对于模板模式会更加灵活**，主要体现在下面几点。

像 Java 这种只支持单继承的语言，基于模板模式编写的子类，已经继承了一个父类，不再具有继承的能力。

回调可以使用匿名类来创建回调对象，可以不用事先定义类；而模板模式针对不同的实现都要定义不同的子类。

如果某个类中定义了多个模板方法，每个方法都有对应的抽象方法，那即便我们只用到其中的一个模板方法，子类也必须实现所有的抽象方法。而回调就更加灵活，我们只需要往用到的模板方法中注入回调对象即可。

还记得上一节课的课堂讨论题目吗？看到这里，相信你应该有了答案了吧？

第三章 策略模式

本章节我们开始学习【策略模式】。该模式最常见的应用场景是，利用它来避免冗长的 if-else 或 switch 分支判断。不过，它的作用还不止如此。它也可以像模板模式那样，提供框架的扩展点等等。

一、原理和实现

策略模式，英文全称是 Strategy Design Pattern。该模式是这样定义的：

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it。

翻译成中文就是：定义一族算法类，将每个算法分别封装起来，让它们可以互相替换。策略模式可以使算法的变化独立于使用它们的客户端（这里的客户端代指使用算法的代码）。

策略模式主要包含以下角色：

1. **策略接口（Strategy）**：定义所有支持的算法的公共接口。客户端使用这个接口与具体策略进行交互。
2. **具体策略（Concrete Strategy）**：实现策略接口的具体策略类。这些类封装了实际的算法逻辑。
3. **上下文（Context）**：持有一个策略对象，用于与客户端进行交互。上下文可以定义一些接口，让客户端不直接与策略接口交互，从而实现策略的封装。

让我们以一个简单的例子来说明策略模式：假设我们要实现一个计算器，支持**加法、减法和乘法运算**。我们可以使用策略模式将**各种运算独立为不同的策略**，并让客户端根据需要选择和使用不同的策略。

首先，我们定义一个策略接口`Operation`：

```
public interface Operation {
    double execute(double num1, double num2);
}
```

接下来，我们创建具体策略类来实现加法、减法和乘法运算：

```
public class Addition implements Operation {
    @Override
    public double execute(double num1, double num2) {
        return num1 + num2;
    }
}

public class Subtraction implements Operation {
    @Override
    public double execute(double num1, double num2) {
        return num1 - num2;
```

```
    }
}

public class Multiplication implements Operation {
    @Override
    public double execute(double num1, double num2) {
        return num1 * num2;
    }
}
```

然后，我们创建一个上下文类`Calculator`，让客户端可以使用这个类来执行不同的运算：

```
public class Calculator {
    private Operation operation;

    public void setOperation(Operation operation) {
        this.operation = operation;
    }

    public double executeOperation(double num1, double num2) {
        return operation.execute(num1, num2);
    }
}
```

现在，客户端可以使用`Calculator`类来执行不同的运算，例如：

```
public class Client {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        calculator.setOperation(new Addition());
        System.out.println("10 + 5 = " + calculator.executeOperation(10, 5));

        calculator.setOperation(new Subtraction());
        System.out.println("10 - 5 = " + calculator.executeOperation(10, 5));

        calculator.setOperation(new Multiplication());
        System.out.println("10 * 5 = " + calculator.executeOperation(10,5));
    }
}
```

在这个例子中，我们使用策略模式将加法、减法和乘法运算独立为不同的策略。客户端可以根据需要选择和使用不同的策略。`Calculator`上下文类持有一个`Operation`策略对象，并通过`setOperation`方法允许客户端设置所需的策略。这种方式使得算法的选择和执行更加灵活，易于扩展和维护。

策略模式的优点包括：

1. 提高代码的可维护性和可扩展性。当需要添加新的算法时，我们只需要实现一个新的具体策略类，而无需修改客户端代码。
2. 符合开闭原则。策略模式允许我们在不修改现有代码的情况下引入新的策略。
3. 避免使用多重条件判断。使用策略模式可以消除一些复杂的条件判断语句，使代码更加清晰和易于理解。

策略模式的缺点包括：

1. 客户端需要了解所有的策略。为了选择合适的策略，客户端需要了解不同策略之间的区别。
2. 增加了类的数量。策略模式会导致程序中具体策略类的数量增加，这可能会导致代码的复杂性增加。

在实际开发中，我们可以根据业务需求和系统架构灵活地运用策略模式。例如，在电商系统中，我们可以使用策略模式处理不同的促销策略；在游戏系统中，我们可以使用策略模式处理不同的角色行为等。

我们抽象出策略设计模式的使用方式：

1、策略的定义

策略类的定义比较简单，包含一个策略接口和一组实现这个接口的策略类。因为所有的策略类都实现相同的接口，所以，客户端代码基于接口而非实现编程，可以灵活地替换不同的策略。示例代码如下所示：

```
public interface Strategy {
    void algorithmInterface();
}
public class ConcreteStrategyA implements Strategy {
    @Override
    public void  algorithmInterface() {
        //具体的算法...
    }
}
public class ConcreteStrategyB implements Strategy {
    @Override
```

```
    public void  algorithmInterface() {
        //具体的算法...
    }
}
```

2、策略的创建

因为策略模式会**包含一组策略**，在使用它们的时候，一般会**通过类型（type）来判断创建哪个策略来使用**。为了封装创建逻辑，我们需要对客户端代码屏蔽创建细节。

事实上我们可以做一定的优化，可以把根据 type 创建策略的逻辑抽离出来，放到工厂类中。示例代码如下所示：

```
public class StrategyFactory {
    private static final Map<String, Strategy> strategies = new HashMap<>();
    static {
        strategies.put("A", new ConcreteStrategyA());
        strategies.put("B", new ConcreteStrategyB());
    }
    public static Strategy getStrategy(String type) {
        if (type == null || type.isEmpty()) {
            throw new IllegalArgumentException("type should not be empty.");
        }
        return strategies.get(type);
    }
}
```

一般来讲，如果**策略类是无状态的，不包含成员变量，只是纯粹的算法实现**，这样的策略对象是可以被共享使用的，不需要在每次调用 getStrategy() 的时候，都创建一个新的策略对象。针对这种情况，我们可以使用上面**这种工厂类的实现方式**，事先创建好每个策略对象，缓存到工厂类中，用的时候直接返回。

相反，如果策略类是有状态的，根据业务场景的需要，我们希望每次从工厂方法中，获得的都是新创建的策略对象，而不是缓存好可共享的策略对象，那我们就需要按照如下方式来实现策略工厂类。

```
public class StrategyFactory {
    public static Strategy getStrategy(String type) {
        if (type == null || type.isEmpty()) {
            throw new IllegalArgumentException("type should not be empty.");
        }
        if (type.equals("A")) {
            return new ConcreteStrategyA();
        } else if (type.equals("B")) {
            return new ConcreteStrategyB();
        }
        return null;
    }
}
```

3、策略的使用

刚刚讲了策略的定义和创建，现在，我们再来看一下，策略的使用。

我们知道，策略模式包含一组可选策略，客户端代码一般如何确定使用哪个策略呢？最常见的是运行时动态确定使用哪种策略，这也是策略模式最典型的应用场景。

这里的“运行时动态”指的是，我们事先并不知道会使用哪个策略，而是在程序运行期间，根据配置、用户输入、计算结果等这些不确定因素，动态决定使用哪种策略。接下来，我们通过一个例子来解释一下。

```
// 策略接口：EvictionStrategy
// 策略类：LruEvictionStrategy、FifoEvictionStrategy、
LfuEvictionStrategy...
// 策略工厂：EvictionStrategyFactory
public class UserCache {
    private Map<String, User> cacheData = new HashMap<>();
    private EvictionStrategy eviction;
    public UserCache(EvictionStrategy eviction) {
        this.eviction = eviction;
    }
    //...
}
// 运行时动态确定，根据配置文件的配置决定使用哪种策略
public class Application {
    public static void main(String[] args) throws Exception {
        EvictionStrategy evictionStrategy = null;
```

```
        Properties props = new Properties();
        props.load(new FileInputStream("./config.properties"));
        String type = props.getProperty("eviction_type");
        evictionStrategy = EvictionStrategyFactory.getEvictionStrategy(type);
        UserCache userCache = new UserCache(evictionStrategy);
        //...
    }
}
// 非运行时动态确定，在代码中指定使用哪种策略
public class Application {
    public static void main(String[] args) {
        //...
        EvictionStrategy evictionStrategy = new LruEvictionStrategy();
        UserCache userCache = new UserCache(evictionStrategy);
        //...
    }
}
```

从上面的代码中，我们也可以看出，“非运行时动态确定”，也就是第二个 Application 中的使用方式，并不能发挥策略模式的优势。在这种应用场景下，策略模式实际上退化成了“面向对象的多态特性”或“基于接口而非实现编程原则”，这其实就是开篇时列举的例子的场景。

二、优化if分支

下边这段代码时我真实遇到过的一段代码，一个解析不同来源报文的方法用了1000多行，是名副其实的屎山，针对这样的代码，我们要如何做优化呢？本章节我们模拟该例子，来使用工厂模式和策略设计模式优化代码。

```
1043                hsmResponse.setRespCode(Resu
1044                hsmResponse.setRespMsg(Resul
1045                return hsmResponse;
1046            case "4B48": //KH
1047                byte[] secretType = new byte
1048                byte[] keyCV = new byte[16];
1049                byte[] secretTag = new byte[
1050                System.arraycopy(bRecv,  srcPo
1051                System.arraycopy(bRecv,  srcPo
1052                System.arraycopy(bRecv,  srcPo
1053                SecretKeyInfoPageResp respDT
1054                respDTO.setSecretLength(Secr
1055                respDTO.setSecretIcv(new Str
1056                respDTO.setSecretType(new St
1057                hsmResponse.setResult(JsonUt
1058                hsmResponse.setRespCode(Resu
1059                hsmResponse.setRespMsg(Resul
1060                return hsmResponse;
1061            case "4538" :
1062                Map<String,String> result =
1063                int publicKyeLength = (int)
1064                int privateKyeLength = bRecv
1065                byte[] publicKey = new byte[
1066                byte[] privateKey = new byte
1067                System.arraycopy(bRecv , src
1068                System.arraycopy(bRecv , src
1069                result.put("publicKey", Byte
1070                result.put("privateKey",Byte
1071                hsmResponse.setResult(JsonUt
1072                hsmResponse.setRespCode(Resu
1073                hsmResponse.setRespMsg(Resul
1074                return hsmResponse;
1075            case "534A" : // SJ"
1076                int resultLength = bRecv.len
1077                byte[] resultBytes = new byt
1078                System.arraycopy(bRecv , src
1079                hsmResponse.setResult(new St
1080                hsmResponse.setRespCode(Resu
1081                hsmResponse.setRespMsg(Resul
1082                return hsmResponse;
1083            default:
1084                log.error("Unknown response:
1085                hsmResponse.setRespCode(Resu
1086                hsmResponse.setRespMsg(Strin
1087                return hsmResponse;
1088        }
1089    } catch (Exception e) {
1090        log.error("密码机返回解析出现异常: {}", E
1091        throw new KmsException(ResultCodes.F
1092    }
1093 }
```

1、基础优化

我们将以解析报文的例子为基础，展示如何使用策略模式优化具有 if 条件分支的代码。

首先，这是一个包含大量 if 分支的报文解析系统：

```
public class MessageParser {
    public void parseMessage(Message message) {
        String messageType = message.getType();

        if ("XML".equalsIgnoreCase(messageType)) {
            // 解析 XML 报文
            System.out.println("解析 XML 报文: " + message.getContent());
        } else if ("JSON".equalsIgnoreCase(messageType)) {
            // 解析 JSON 报文
            System.out.println("解析 JSON 报文: " + message.getContent());
        } else if ("CSV".equalsIgnoreCase(messageType)) {
            // 解析 CSV 报文
            System.out.println("解析 CSV 报文: " + message.getContent());
        } else {
            throw new IllegalArgumentException("未知的报文类型: " + messageType);
        }
    }
}
```

接下来，我们将使用策略模式优化上述代码。

首先，我们定义一个报文解析策略接口 `MessageParserStrategy`：

```
public interface MessageParserStrategy {
    // 解析报文内容的方法，输入一个 Message 对象，无返回值
    void parse(Message message);
}
```

然后，我们实现 XML、JSON 和 CSV 报文解析策略：

```
// XML 报文解析策略
public class XmlMessageParserStrategy implements MessageParserStrategy {
    @Override
    public void parse(Message message) {
        System.out.println("解析 XML 报文: " + message.getContent());
    }
}
```

```
// JSON 报文解析策略
public class JsonMessageParserStrategy implements MessageParserStrategy {
    @Override
    public void parse(Message message) {
        System.out.println("解析 JSON 报文: " + message.getContent());
    }
}

// CSV 报文解析策略
public class CsvMessageParserStrategy implements MessageParserStrategy {
    @Override
    public void parse(Message message) {
        System.out.println("解析 CSV 报文: " + message.getContent());
    }
}
```

接下来，我们创建一个 `MessageParserContext` 类，该类将根据传入的策略解析报文：

```
public class MessageParserContext {
    private MessageParserStrategy strategy;

    // 设置报文解析策略
    public void setStrategy(MessageParserStrategy strategy) {
        this.strategy = strategy;
    }

    // 根据策略解析报文
    public void parseMessage(Message message) {
        strategy.parse(message);
    }
}
```

最后，我们使用策略模式进行报文解析，避免了使用分支判断：

```
public class Main {
    public static void main(String[] args) {
        MessageParserContext parserContext = new MessageParserContext();

        // 使用 XML 报文解析策略
        parserContext.setStrategy(new XmlMessageParserStrategy());
```

```
        parserContext.parseMessage(new Message("XML", "<xml>这是一个 XML 报文</xml>"));

        // 使用 JSON 报文解析策略
        parserContext.setStrategy(new JsonMessageParserStrategy());
        parserContext.parseMessage(new Message("JSON", "{\"message\": \"这是一个 JSON 报文\"}"));

            // 使用 CSV 报文解析策略
        parserContext.setStrategy(new CsvMessageParserStrategy());
        parserContext.parseMessage(new Message("CSV", "这是一个,CSV,报文"));
    }
}
```

2、结合工厂模式

我们可以将策略模式与工厂模式结合，以便根据不同的消息类型自动匹配不同的解析策略。下面是如何实现这个优化的：

首先，我们创建一个 `MessageParserStrategyFactory` 类，用于根据报文类型创建相应的解析策略：

```
public class MessageParserStrategyFactory {
    private static final Map<String, MessageParserStrategy> strategies = new HashMap<>();

    static {
        strategies.put("XML", new XmlMessageParserStrategy());
        strategies.put("JSON", new JsonMessageParserStrategy());
        strategies.put("CSV", new CsvMessageParserStrategy());
    }

    public static MessageParserStrategy getStrategy(String messageType) {
        MessageParserStrategy strategy = strategies.get(messageType.toUpperCase());
        if (strategy == null) {
            throw new IllegalArgumentException("未知的报文类型: " + messageType);
        }
        return strategy;
    }
}
```

接下来，我们修改 `MessageParserContext` 类，使其根据报文类型自动选择解析策略：

```
public class MessageParserContext {
    public void parseMessage(Message message) {
        MessageParserStrategy strategy =
MessageParserStrategyFactory.getStrategy(message.getType());
        strategy.parse(message);
    }
}
```

现在，我们的代码可以根据不同的消息类型自动匹配不同的解析策略，而无需手动设置策略。以下是使用此优化的示例：

```
public class Main {
    public static void main(String[] args) {
        MessageParserContext parserContext = new MessageParserContext();

        // 自动使用 XML 报文解析策略
        parserContext.parseMessage(new Message("XML", "<xml>这是一个 XML 报
文</xml>"));

        // 自动使用 JSON 报文解析策略
        parserContext.parseMessage(new Message("JSON", "{\"message\": \"这是一
个 JSON 报文\"}"));

        // 自动使用 CSV 报文解析策略
        parserContext.parseMessage(new Message("CSV", "这是一个,CSV,报文"));
    }
}
```

通过将策略模式与工厂模式结合，我们优化了报文解析过程。这样的代码更容易扩展和维护，因为我们可以通过在工厂类中添加新的解析策略来轻松地支持新的报文类型。

现在的代码实现就更加优美了。当要添加一个新的报文格式时，我们只需要修改策略工厂类的静态代码段，其他代码都不需要修改，这样就将代码改动最小化、集中化了。你可能会说，这还是需要修改代码，并不完全符合开闭原则。有什么办法让我们完全满足开闭原则呢？

对于 Java 语言来说，我们**可以通过反射来避免对策略工厂类的修改。**

具体是这么做的：我们通过一个**配置文件或者自定义的 annotation 来标注都有哪些策略类**；策略工厂类读取配置文件或者搜索被 annotation 标注的策略类，然后通过反射了动态地加载这些策略类、创建策略对象；当我们新添加一个策略的时候，只需要将这个新添加的策略类添加到配置文件或者用 annotation 标注即可。

三、源码使用

1、ssm框架

1. Spring中的Resource接口：在Spring框架中，`org.springframework.core.io.Resource`接口用于抽象不同类型的资源，例如文件系统资源、类路径资源、URL资源等。Resource接口就像策略模式中的策略接口，而不同类型的资源类（如`ClassPathResource`、`FileSystemResource`等）就像具体策略。客户端可以根据需要选择和使用不同的资源类。
2. Spring中的AOP代理：在Spring AOP中，代理类的创建使用了策略模式。`org.springframework.aop.framework.ProxyFactory`中的`AopProxy`接口定义了创建代理对象的策略接口，而`JdkDynamicAopProxy`和`CglibAopProxy`这两个类分别为基于JDK动态代理和CGLIB动态代理的具体策略。客户端可以根据需要选择使用哪种代理方式。
3. MyBatis中的`Executor`接口：在MyBatis中，`Executor`接口定义了执行SQL语句的策略接口。MyBatis提供了不同的`Executor`实现，例如`SimpleExecutor`、`ReuseExecutor`和`BatchExecutor`等，它们分别表示不同的执行策略。客户端可以通过配置选择使用哪种执行策略。
4. Spring MVC中的`HandlerMapping`接口：在Spring MVC框架中，`HandlerMapping`接口定义了映射请求到处理器的策略接口。Spring MVC提供了多种`HandlerMapping`实现，例如`BeanNameUrlHandlerMapping`、`RequestMappingHandlerMapping`等，分别表示不同的映射策略。客户端可以通过配置选择使用哪种映射策略。

这些例子展示了策略模式在SSM框架中的应用。策略模式通过将算法和客户端分离，使得系统更加灵活和可扩展。在实际开发中，我们可以参考这些例子，根据业务需求和系统架构灵活地运用策略模式。

这里我们以MyBatis中的`Executor`接口为例，展示策略模式在MyBatis中的应用。

首先，`Executor`接口是策略接口，定义了执行SQL语句的公共方法。以下是简化后的`Executor`接口：

```
public interface Executor {

  <E> List<E> query(MappedStatement ms, Object parameter) throws
SQLException;

  int update(MappedStatement ms, Object parameter) throws SQLException;

  // ... 其他方法
}
```

接下来，我们来看MyBatis提供的不同`Executor`实现：

(1) `SimpleExecutor`：简单执行器，每次执行SQL都会创建一个新的预处理语句(`PreparedStatement`)。

```
public class SimpleExecutor extends BaseExecutor {

  @Override
  public int doUpdate(MappedStatement ms, Object parameter) throws
SQLException {
    // ... 省略具体实现
  }

  @Override
  public <E> List<E> doQuery(MappedStatement ms, Object parameter) throws
SQLException {
    // ... 省略具体实现
  }

  // ... 其他方法
}
```

(2) `ReuseExecutor`：重用执行器，会尽量重用预处理语句(`PreparedStatement`)，以减少创建和销毁预处理语句的开销。

```
public class ReuseExecutor extends BaseExecutor {

    @Override
    public int doUpdate(MappedStatement ms, Object parameter) throws
SQLException {
      // ... 省略具体实现
    }
```

```
    @Override
    public <E> List<E> doQuery(MappedStatement ms, Object parameter) throws
SQLException {
        // ... 省略具体实现
    }

    // ... 其他方法
}
```

（3）`BatchExecutor`：批处理执行器，可以将多个SQL语句一起发送到数据库服务器，减少网络开销。

```
public class BatchExecutor extends BaseExecutor {

    @Override
    public int doUpdate(MappedStatement ms, Object parameter) throws
SQLException {
        // ... 省略具体实现
    }

    @Override
    public <E> List<E> doQuery(MappedStatement ms, Object parameter) throws
SQLException {
        // ... 省略具体实现
    }

    // ... 其他方法
}
```

客户端可以通过配置选择使用哪种执行策略。在MyBatis配置文件（`mybatis-config.xml`）中，我们可以设置`<setting>`标签的`defaultExecutorType`属性来指定执行器类型：

```
<settings>
  <setting name="defaultExecutorType" value="SIMPLE" />
  <!-- 可选值：SIMPLE, REUSE, BATCH -->
</settings>
```

在这个例子中，`Executor`接口就像策略模式中的策略接口，而`SimpleExecutor`、`ReuseExecutor`和`BatchExecutor`这三个类就像具体策略。客户端可以根据需要选择和使用不同的执行器类型。这种方式使得SQL执行策略的选择和实现更加灵活和可扩展。

2、jdk源码

下面我们以`java.util.Comparator`接口为例，展示策略模式在JDK中的应用。

假设我们有一个`Student`类，表示学生。我们需要对一个`Student`对象的列表进行排序。根据不同的需求，我们可能需要按照学生的姓名、年龄或成绩进行排序。这时，我们可以使用策略模式，通过实现`Comparator`接口，为不同的排序需求提供不同的比较策略。

首先，定义`Student`类：

```
public class Student {
    private String name;
    private int age;
    private double score;

    // 构造方法、getter和setter方法省略
}
```

然后，实现`Comparator`接口，定义不同的比较策略：

```
public class NameComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.getName().compareTo(s2.getName());
    }
}

// 根据学生的年龄进行排序
public class AgeComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return Integer.compare(s1.getAge(), s2.getAge());
    }
}
```

```
// 根据学生的成绩进行排序
public class ScoreComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return Double.compare(s1.getScore(), s2.getScore());
    }
}
```

最后，在客户端代码中，根据需要选择和使用不同的比较策略：

```
public class Client {
    public static void main(String[] args) {
        // 创建一个Student对象的列表
        List<Student> students = new ArrayList<>();
        students.add(new Student("Alice", 20, 90.0));
        students.add(new Student("Bob", 18, 85.0));
        students.add(new Student("Charlie", 22, 88.0));

        // 使用姓名比较策略进行排序
        Collections.sort(students, new NameComparator());
        System.out.println("按姓名排序: " + students);

        // 使用年龄比较策略进行排序
        Collections.sort(students, new AgeComparator());
        System.out.println("按年龄排序: " + students);

        // 使用成绩比较策略进行排序
        Collections.sort(students, new ScoreComparator());
        System.out.println("按成绩排序: " + students);
    }
}
```

在这个例子中，我们使用策略模式将不同的排序策略独立为不同的类。客户端可以根据需要选择和使用不同的排序策略，而无需修改代码。这种方式使得排序策略的选择和实现更加灵活和可扩展。在实际开发过程中，可以根据业务需求和系统架构灵活地运用策略模式。

四、使用场景

策略模式在实际工作场景中有很多应用，以下是一些常见的使用场景：

1. 支付系统：在电商或其他在线支付场景中，我们可能需要支持多种支付方式（如信用卡、PayPal、微信支付、支付宝等）。我们可以使用策略模式定义一个支付接口，并为每种支付方式提供一个具体的实现。客户端可以根据用户的选择使用不同的支付策略。
2. 促销策略：在商城系统中，我们可能需要根据不同的促销活动（如满减、打折、买一送一等）提供不同的折扣策略。我们可以使用策略模式定义一个折扣接口，并为每种促销活动提供一个具体的实现。客户端可以根据不同的促销活动选择合适的折扣策略。
3. 日志记录：在实际项目中，我们可能需要将日志记录到不同的存储介质（如控制台、文件、数据库等）。我们可以使用策略模式定义一个日志记录接口，并为每种存储介质提供一个具体的实现。客户端可以根据需要选择和使用不同的日志记录策略。
4. 数据压缩：在处理大量数据时，我们可能需要对数据进行压缩，以节省存储空间和网络传输时间。我们可以使用策略模式定义一个数据压缩接口，并为不同的压缩算法（如ZIP、GZIP、LZ77等）提供具体的实现。客户端可以根据需要选择和使用不同的压缩策略。
5. 路由选择：在网络通信或分布式系统中，我们可能需要根据不同的情况（如网络状况、负载均衡等）选择不同的路由策略。我们可以使用策略模式定义一个路由选择接口，并为不同的路由选择算法提供具体的实现。客户端可以根据实际情况选择合适的路由策略。
6. 机器学习算法：在机器学习领域，我们可能需要根据不同的问题和数据类型选择不同的学习算法（如线性回归、支持向量机、神经网络等）。我们可以使用策略模式定义一个学习算法接口，并为不同的学习算法提供具体的实现。客户端可以根据实际问题选择合适的学习算法。
7. 密码加密：在安全领域，我们可能需要对用户密码进行加密，以保护用户数据的安全。我们可以使用策略模式定义一个加密接口，并为不同的加密算法（如MD5、SHA-1、SHA-256等）提供具体的实现。客户端可以根据需要选择和使用不同的加密策略。
8. 认证策略：在Web应用中，我们可能需要根据不同的场景选择不同的认证策略（如基于用户名/密码的认证、OAuth认证、单点登录等）。我们可以使用策略模式定义一个认证接口，并为不同的认证方式提供具体的实现。客户端可以根据实际需求选择合适的认证策略。
9. 图像处理：在图像处理领域，我们可能需要对图像进行不同的处理操作（如缩放、旋转、滤镜等）。我们可以使用策略模式定义一个图像处理接口，并为不同的处理操作提供具体的实现。客户端可以根据需要选择和使用不同的图像处理策略。

10. 任务调度：在分布式计算或并行计算中，我们可能需要根据不同的场景选择不同的任务调度策略（如FIFO、优先级队列、轮询等）。我们可以使用策略模式定义一个任务调度接口，并为不同的调度算法提供具体的实现。客户端可以根据实际情况选择合适的任务调度策略。
11. 语言翻译：在开发多语言支持的应用程序时，我们可能需要根据用户的语言选择不同的翻译策略。我们可以使用策略模式定义一个翻译接口，并为不同的语言提供具体的实现。客户端可以根据用户的语言选择合适的翻译策略。
12. 数据库访问：在开发支持多种数据库的应用程序时，我们可能需要根据不同的数据库类型选择不同的数据访问策略。我们可以使用策略模式定义一个数据库访问接口，并为不同的数据库（如MySQL、PostgreSQL、Oracle等）提供具体的实现。客户端可以根据实际的数据库类型选择合适的数据访问策略。
13. 验证码生成：在开发Web应用程序时，我们可能需要为用户提供不同类型的验证码（如数字验证码、字母验证码、图像验证码等）。我们可以使用策略模式定义一个验证码生成接口，并为不同类型的验证码提供具体的实现。客户端可以根据需要选择和使用不同的验证码生成策略。
14. 通信协议：在开发网络通信应用时，我们可能需要支持多种通信协议（如HTTP、FTP、SMTP等）。我们可以使用策略模式定义一个通信协议接口，并为不同的协议提供具体的实现。客户端可以根据需要选择和使用不同的通信协议策略。
15. 地图导航：在开发地图导航应用时，我们可能需要根据用户的需求提供不同的路径规划策略（如最短路径、最快路径、避免拥堵等）。我们可以使用策略模式定义一个路径规划接口，并为不同的路径规划需求提供具体的实现。客户端可以根据用户的需求选择合适的路径规划策略。

1、支付系统

下面是一个使用策略模式的支付系统的简化示例。我们将定义一个支付策略接口`PaymentStrategy`，并为不同的支付方式（如信用卡支付和支付宝支付）提供具体的实现。客户端可以根据用户选择的支付方式来调用相应的支付策略。

首先，我们定义一个支付策略接口`PaymentStrategy`：

```
public interface PaymentStrategy {
    // 支付操作
    void pay(double amount);
}
```

接着，我们为信用卡支付方式提供一个具体实现`CreditCardPaymentStrategy`：、

```
```

```
public class CreditCardPaymentStrategy implements PaymentStrategy {
    private String name;
    private String cardNumber;
    private String cvv;
    private String expirationDate;

    public CreditCardPaymentStrategy(String name, String cardNumber, String
cvv, String expirationDate) {
        this.name = name;
        this.cardNumber = cardNumber;
        this.cvv = cvv;
        this.expirationDate = expirationDate;
    }

    @Override
    public void pay(double amount) {
        System.out.println("使用信用卡支付： " + amount + "元");
    }
}
```

接下来，我们为支付宝支付方式提供一个具体实现`AlipayPaymentStrategy`：

```
public class AlipayPaymentStrategy implements PaymentStrategy {
    private String email;
    private String password;

    public AlipayPaymentStrategy(String email, String password) {
        this.email = email;
        this.password = password;
    }

    @Override
    public void pay(double amount) {
        System.out.println("使用支付宝支付： " + amount + "元");
    }
}
```

现在，我们可以在客户端代码中根据用户选择的支付方式来调用相应的支付策略：

```
public class PaymentClient {
    public static void main(String[] args) {
        // 创建信用卡支付策略实例
```

```
        PaymentStrategy creditCardStrategy = new CreditCardPaymentStrategy("张三", "1234567890123456", "123", "12/23");

        // 创建支付宝支付策略实例
        PaymentStrategy alipayStrategy = new AlipayPaymentStrategy("zhangsan@example.com", "mypassword");

        // 根据用户选择的支付方式进行支付
        double paymentAmount = 100.0;
        System.out.println("用户选择信用卡支付：");
        creditCardStrategy.pay(paymentAmount);

        System.out.println("用户选择支付宝支付：");
        alipayStrategy.pay(paymentAmount);
    }
}
```

在这个示例中，我们定义了一个支付策略接口`PaymentStrategy`，并为不同的支付方式提供了具体实现。客户端代码可以根据用户选择的支付方式来调用相应的支付策略。这样，当需要添加新的支付方式时，我们只需提供新的支付策略实现，而无需修改客户端代码，从而提高了系统的可扩展性。

2、优惠策略

下面是一个使用策略模式的促销系统的简化示例。我们将定义一个促销策略接口`PromotionStrategy`，并为不同的促销方式（如满减、打折和无优惠）提供具体的实现。客户端可以根据用户选择的促销方式来调用相应的促销策略。

首先，我们定义一个促销策略接口`PromotionStrategy`：

```
public interface PromotionStrategy {
    // 计算优惠后的价格
    double calculateDiscount(double originalPrice);
}
```

接着，我们为满减优惠方式提供一个具体实现`FullReductionStrategy`：

```
public class FullReductionStrategy implements PromotionStrategy {
    private double threshold;
    private double reduction;
```

```
    public FullReductionStrategy(double threshold, double reduction) {
        this.threshold = threshold;
        this.reduction = reduction;
    }

    @Override
    public double calculateDiscount(double originalPrice) {
        return originalPrice >= threshold ? originalPrice - reduction : originalPrice;
    }
}
```

接下来，我们为打折优惠方式提供一个具体实现`DiscountStrategy`：

```
public class DiscountStrategy implements PromotionStrategy {
    private double discount;

    public DiscountStrategy(double discount) {
        this.discount = discount;
    }

    @Override
    public double calculateDiscount(double originalPrice) {
        return originalPrice * discount;
    }
}
```

最后，我们为无优惠方式提供一个具体实现`NoDiscountStrategy`：

```
public class NoDiscountStrategy implements PromotionStrategy {
    @Override
    public double calculateDiscount(double originalPrice) {
        return originalPrice;
    }
}
```

现在，我们可以在客户端代码中根据用户选择的促销方式来调用相应的促销策略：

```
public class PromotionClient {
    public static void main(String[] args) {
        // 创建满减策略实例
```

```
        PromotionStrategy fullReductionStrategy = new FullReductionStrategy(300, 100);

        // 创建打折策略实例
        PromotionStrategy discountStrategy = new DiscountStrategy(0.8);

        // 创建无优惠策略实例
        PromotionStrategy noDiscountStrategy = new NoDiscountStrategy();

        // 根据用户选择的促销方式计算优惠后的价格
        double originalPrice = 350.0;
        System.out.println("用户选择满减优惠：");
        System.out.println("优惠后价格：" + fullReductionStrategy.calculateDiscount(originalPrice));

        System.out.println("用户选择打折优惠：");
        System.out.println("优惠后价格：" + discountStrategy.calculateDiscount(originalPrice));

        System.out.println("用户选择无优惠：");
        System.out.println("优惠后价格：" + noDiscountStrategy.calculateDiscount(originalPrice));
    }
}
```

在这个示例中，我们定义了一个促销策略接口`PromotionStrategy`，并为不同的促销方式提供了具体实现。客户端代码可以根据用户选择的促销方式来调用相应的促销策略。这样，当需要添加新的促销方式时，我们只需提供新的促销策略实现，而无需修改客户端代码，从而提高了系统的可扩展性。

例如，假设我们现在需要添加一个新的促销方式：购买指定商品后可以获得免费赠品。我们可以为这种促销方式提供一个具体实现`GiftStrategy`：

```
public class GiftStrategy implements PromotionStrategy {
    private double targetPrice;
    private String giftName;

    public GiftStrategy(double targetPrice, String giftName) {
        this.targetPrice = targetPrice;
        this.giftName = giftName;
    }

    @Override
```

```
    public double calculateDiscount(double originalPrice) {
        if (originalPrice >= targetPrice) {
            System.out.println("获得赠品：" + giftName);
        }
        return originalPrice;
    }
}
```

然后，我们可以在客户端代码中添加一个新的促销方式选择，让用户可以选择这种新的促销方式：

```
public class PromotionClient {
    public static void main(String[] args) {
        // ...
        // 创建赠品策略实例
        PromotionStrategy giftStrategy = new GiftStrategy(500, "免费赠品");

        // 根据用户选择的促销方式计算优惠后的价格
        double originalPrice = 550.0;
        // ...
        System.out.println("用户选择赠品优惠：");
        System.out.println("优惠后价格：" +
giftStrategy.calculateDiscount(originalPrice));
    }
}
```

这个示例表明，通过使用策略模式，我们可以轻松地添加新的促销方式，而无需修改客户端代码。策略模式使我们的系统变得更加灵活，便于维护和扩展。

3、地图导航

下面是一个使用策略模式的地图导航系统的简化示例。我们将定义一个路径规划策略接口`RoutePlanningStrategy`，并为不同的路径规划需求（如最短路径、最快路径和避免拥堵）提供具体的实现。客户端可以根据用户的需求选择合适的路径规划策略。

首先，我们定义一个路径规划策略接口`RoutePlanningStrategy`：

```
public interface RoutePlanningStrategy {
    // 计算路径
    void calculateRoute(String start, String end);
}
```

接着，我们为最短路径规划需求提供一个具体实现`ShortestRouteStrategy`：

```
public class ShortestRouteStrategy implements RoutePlanningStrategy {
    @Override
    public void calculateRoute(String start, String end) {
        System.out.println("计算最短路径：" + start + " -> " + end);
    }
}
```

接下来，我们为最快路径规划需求提供一个具体实现`FastestRouteStrategy`：

```
public class FastestRouteStrategy implements RoutePlanningStrategy {
    @Override
    public void calculateRoute(String start, String end) {
        System.out.println("计算最快路径：" + start + " -> " + end);
    }
}
```

最后，我们为避免拥堵路径规划需求提供一个具体实现`AvoidCongestionRouteStrategy`：、

```
public class AvoidCongestionRouteStrategy implements RoutePlanningStrategy {
    @Override
    public void calculateRoute(String start, String end) {
        System.out.println("计算避免拥堵路径：" + start + " -> " + end);
    }
}
```

现在，我们可以在客户端代码中根据用户选择的路径规划需求来选择相应的路径规划策略：

```
public class RoutePlanningClient {
    public static void main(String[] args) {
        // 创建最短路径规划策略实例
        RoutePlanningStrategy shortestRouteStrategy = new
ShortestRouteStrategy();
```

```
        // 创建最快路径规划策略实例
        RoutePlanningStrategy fastestRouteStrategy = new FastestRouteStrategy();

        // 创建避免拥堵路径规划策略实例
        RoutePlanningStrategy avoidCongestionRouteStrategy = new
AvoidCongestionRouteStrategy();

        // 根据用户选择的路径规划需求进行路径规划
        String start = "北京";
        String end = "上海";
        System.out.println("用户选择最短路径规划：");
        shortestRouteStrategy.calculateRoute(start, end);

        System.out.println("用户选择最快路径规划：");
        fastestRouteStrategy.calculateRoute(start, end);

        System.out.println("用户选择避免拥堵路径规划：");
        avoidCongestionRouteStrategy.calculateRoute(start, end);
    }
}
```

在这个示例中，我们定义了一个路径规划策略接口`RoutePlanningStrategy`，并为不同的路径规划需求提供了具体实现。客户端代码可以根据用户选择的路径规划需求来调用相应的路径规划策略。这样，当需要添加新的路径规划需求时，我们只需提供新的路径规划策略实现，而无需修改客户端代码，从而提高了系统的可扩展性。

例如，假设我们现在需要添加一个新的路径规划需求：优先选择高速公路。我们可以为这种路径规划需求提供一个具体实现`HighwayFirstRouteStrategy`：

```
public class HighwayFirstRouteStrategy implements RoutePlanningStrategy {
    @Override
    public void calculateRoute(String start, String end) {
        System.out.println("计算优先高速公路路径：" + start + " -> " + end);
    }
}
```

然后，我们可以在客户端代码中添加一个新的路径规划需求选择，让用户可以选择这种新的路径规划需求：

```
public class RoutePlanningClient {
    public static void main(String[] args) {
        // ...
        // 创建优先高速公路路径规划策略实例
```

```
        RoutePlanningStrategy highwayFirstRouteStrategy = new
HighwayFirstRouteStrategy();

        // 根据用户选择的路径规划需求进行路径规划
        String start = "北京";
        String end = "上海";
        // ...
        System.out.println("用户选择优先高速公路路径规划：");
        highwayFirstRouteStrategy.calculateRoute(start, end);
    }
}
```

这个示例表明，通过使用策略模式，我们可以轻松地添加新的路径规划需求，而无需修改客户端代码。策略模式使我们的系统变得更加灵活，便于维护和扩展。

五、重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

一提到 if-else 分支判断，有人就觉得它是烂代码。如果 if-else 分支判断不复杂、代码不多，这并没有任何问题，毕竟 if-else 分支判断几乎是所有编程语言都会提供的语法，存在即有理由。遵循 KISS 原则，怎么简单怎么来，就是最好的设计。非得用策略模式，搞出 n 多类，反倒是一种过度设计。

一提到策略模式，有人就觉得，它的作用是避免 if-else 分支判断逻辑。实际上，这种认识是很片面的。策略模式主要的作用还是解耦策略的定义、创建和使用，控制代码的复杂度，让每个部分都不至于过于复杂、代码量过多。除此之外，对于复杂代码来说，策略模式还能让其满足开闭原则，添加新策略的时候，最小化、集中化代码改动，减少引入 bug 的风险。

实际上，设计原则和思想比设计模式更加普适和重要。掌握了代码的设计原则和思想，我们能更清楚的了解，为什么要用某种设计模式，就能更恰到好处地应用设计模式。

策略模式定义一族算法类，将每个算法分别封装起来，让它们可以互相替换。策略模式可以使算法的变化独立于使用它们的客户端（这里的客户端代指使用算法的代码）。

策略模式用来解耦策略的定义、创建、使用。实际上，一个完整的策略模式就是由这三个部分组成的。

- 策略类的定义比较简单，包含一个策略接口和一组实现这个接口的策略类。

- 策略的创建由工厂类来完成，封装策略创建的细节。
- 策略模式包含一组策略可选，客户端代码如何选择使用哪个策略，有两种确定方法：编译时静态确定和运行时动态确定。其中，“运行时动态确定”才是策略模式最典型的应用场景。

除此之外，我们还可以通过策略模式来移除 if-else 分支判断。实际上，这得益于策略工厂类，更本质上点讲，是借助“查表法”，根据 type 查表替代根据 type 分支判断。

思考，在什么情况下，我们才有必要去掉代码中的 if-else 或者 switch-case 分支逻辑呢？

第四章 职责链模式

本章节我们学习职责链模式。不管是之前学习的模板模式、策略模式，还是今天学习的职责链模式他们具有相同的作用：**复用和扩展**，在实际的项目开发中比较常用。在框架开发中，我们也可以利用它们来提供框架的扩展点，能够让框架的使用者在不修改框架源码的情况下，基于扩展点定制化框架的功能。

一、原理和实现

职责链模式的英文翻译是 Chain Of Responsibility Design Pattern。在 GoF 的《设计模式》中，它是这么定义的：

> Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

翻译成中文就是：将请求的发送和接收解耦，让多个接收对象都有机会处理这个请求。将这些接收对象串成一条链，并沿着这条链传递这个请求，**直到链上的某个接收对象能够处理它为止**，实时上，在常见的使用场景中，我们的责任链并不是和概念中的完全一样。

- 原始概念中，是直到链上的某个接收对象能够处理它为止。
- 实际使用中，链上的所有对象都可以对请求进行特殊处理。

二、实现方式

1、使用链表实现

第一种实现方式如下所示。其中，Handler 是所有处理器类的抽象父类，handle() 是抽象方法。每个具体的处理器类（HandlerA、HandlerB）的 handle() 函数的代码结构类似，如果它能处理该请求，就不继续往下传递；如果不能处理，则交由后面的处理器来处理（也就是调用 successor.handle()）。HandlerChain 是处理器链，从数据结构的角度来看，它就是一个记录了链头、链尾的链表。其中，记录链尾是为了方便添加处理器。

```
public abstract class Handler {
  protected Handler successor = null;
  public void setSuccessor(Handler successor) {
    this.successor = successor;
  }
  public abstract void handle();
}

public class HandlerA extends Handler {
  @Override
  public boolean handle() {
    boolean handled = false;
    //...
    if (!handled && successor != null) {
      successor.handle();
    }
  }
}

public class HandlerB extends Handler {
  @Override
  public void handle() {
    boolean handled = false;
    //...
    if (!handled && successor != null) {
      successor.handle();
    }
  }
}

public class HandlerChain {
```

```
    private Handler head = null;
    private Handler tail = null;
    public void addHandler(Handler handler) {
        handler.setSuccessor(null);
        if (head == null) {
            head = handler;
            tail = handler;
            return;
        }
        tail.setSuccessor(handler);
        tail = handler;
    }
    public void handle() {
        if (head != null) {
            head.handle();
        }
    }
}

// 使用举例
public class Application {
    public static void main(String[] args) {
        HandlerChain chain = new HandlerChain();
        chain.addHandler(new HandlerA());
        chain.addHandler(new HandlerB());
        chain.handle();
    }
}
```

实际上，上面的代码实现不够优雅。处理器类的 handle() 函数，不仅包含自己的业务逻辑，还包含对下一个处理器的调用，也就是代码中的 successor.handle()。一个不熟悉这种代码结构的程序员，在添加新的处理器类的时候，很有可能忘记在 handle() 函数中调用 successor.handle()，这就会导致代码出现 bug。

针对这个问题，我们对代码进行重构，利用模板模式，将调用 successor.handle() 的逻辑从具体的处理器类中剥离出来，放到抽象父类中。这样具体的处理器类只需要实现自己的业务逻辑就可以了。重构之后的代码如下所示：

```
public abstract class Handler {
    protected Handler successor = null;
    public void setSuccessor(Handler successor) {
        this.successor = successor;
    }
```

```
    public final void handle() {
        boolean handled = doHandle();
        if (successor != null && !handled) {
            successor.handle();
        }
    }
    protected abstract boolean doHandle();
}

public class HandlerA extends Handler {
    @Override
    protected boolean doHandle() {
        boolean handled = false;
        //...
        return handled;
    }
}

public class HandlerB extends Handler {
    @Override
    protected boolean doHandle() {
        boolean handled = false;
        //...
        return handled;
    }
}
// HandlerChain和Application代码不变
```

2、使用数组实现

我们再来看第二种实现方式，代码如下所示。这种实现方式更加简单。HandlerChain 类用数组而非链表来保存所有的处理器，并且需要在 HandlerChain 的 handle() 函数中，依次调用每个处理器的 handle() 函数。

```
public interface IHandler {
    boolean handle();
}

public class HandlerA implements IHandler {
    @Override
    public boolean handle() {
```

```
        boolean handled = false;
        //...
        return handled;
    }
}
public class HandlerB implements IHandler {
    @Override
    public boolean handle() {
        boolean handled = false;
        //...
        return handled;
    }
}

public class HandlerChain {
    private List<IHandler> handlers = new ArrayList<>();
    public void addHandler(IHandler handler) {
        this.handlers.add(handler);
    }
    public void handle() {
        for (IHandler handler : handlers) {
            boolean handled = handler.handle();
            if (handled) {
                break;
            }
        }
    }
}
// 使用举例
public class Application {
    public static void main(String[] args) {
        HandlerChain chain = new HandlerChain();
        chain.addHandler(new HandlerA());
        chain.addHandler(new HandlerB());
        chain.handle();
    }
}
```

3、扩展

在 GoF 给出的定义中，**如果处理器链上的某个处理器能够处理这个请求，那就不会继续往下传递请求。实际上，职责链模式还有一种变体，那就是请求会被所有的处理器都处理一遍，不存在中途终止的情况。**这种变体也有两种实现方式：用链表存储处理器和用数组存储处理器，跟上面的两种实现方式类似，只需要稍微修改即可。

我这里只给出其中一种实现方式，如下所示。另外一种实现方式你对照着上面的实现自行修改。

```
public abstract class Handler {
  protected Handler successor = null;
  public void setSuccessor(Handler successor) {
    this.successor = successor;
  }
  public final void handle() {
    doHandle();
    if (successor != null) {
      successor.handle();
    }
  }
  protected abstract void doHandle();
}


public class HandlerA extends Handler {
  @Override
  protected void doHandle() {
    //...
  }
}

public class HandlerB extends Handler {
  @Override
  protected void doHandle() {
    //...
  }
}

public class HandlerChain {
  private Handler head = null;
  private Handler tail = null;
  public void addHandler(Handler handler) {
```

```
        handler.setSuccessor(null);
        if (head == null) {
            head = handler;
            tail = handler;
            return;
        }
        tail.setSuccessor(handler);
        tail = handler;
    }
    public void handle() {
        if (head != null) {
            head.handle();
        }
    }
}

// 使用举例
public class Application {
    public static void main(String[] args) {
        HandlerChain chain = new HandlerChain();
        chain.addHandler(new HandlerA());
        chain.addHandler(new HandlerB());
        chain.handle();
    }
}
```

三、源码实现

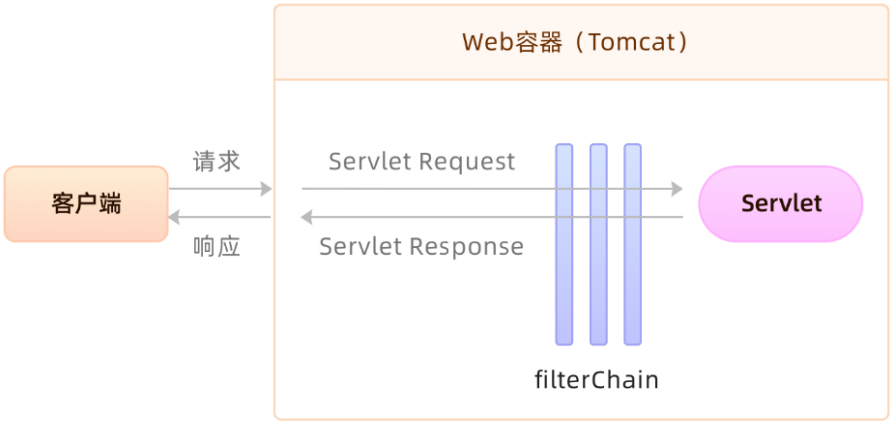
职责链模式常用在框架的开发中，为框架提供扩展点，让框架的使用者在不修改框架源码的情况下，基于扩展点添加新的功能。实际上，更具体点来说，职责链模式最常用来开发框架的过滤器和拦截器。今天，我们就通过 Servlet Filter、Spring Interceptor一级mybatis的plugin 这三个 Java 开发中常用的组件，来具体讲讲它在框架开发中的应用。

话不多说，让我们正式开始今天的学习吧！

一、Servlet Filter

1、使用

Servlet Filter 是 Java Servlet **规范中定义的组件**，翻译成中文就是过滤器，它可以实现对 HTTP 请求的过滤功能，比如**鉴权、限流、记录日志、验证参数**等等。因为它是 Servlet 规范的一部分，所以，只要是支持 Servlet 的 Web 容器（比如，Tomcat、Jetty 等），都支持过滤器功能。为了帮助你理解，我画了一张示意图阐述它的工作原理，如下所示。



以下是一个简单的 Servlet Filter 示例，用于记录 HTTP 请求访问的耗时：

首先，创建一个名为 `RequestTimingFilter` 的 Java 类，实现 `javax.servlet.Filter` 接口。

```
public class RequestTimingFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) {
        // 可以在这里进行 Filter 的初始化操作
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
        // 在请求到达时记录开始时间
        long startTime = System.currentTimeMillis();

        // 将请求传递给责任链上的下一个 Filter 或目标 Servlet
        chain.doFilter(request, response);
```

```
        // 在请求处理完成后记录结束时间，并计算耗时
        long endTime = System.currentTimeMillis();
        long timeElapsed = endTime - startTime;

        HttpServletRequest httpServletRequest = (HttpServletRequest) request;
        String requestURI = httpServletRequest.getRequestURI();

        System.out.println("请求 " + requestURI + " 的访问耗时：" + timeElapsed + "
毫秒");
    }

    @Override
    public void destroy() {
        // 可以在这里进行 Filter 的销毁操作
    }
}
```

接下来，在你的 web 应用的 `web.xml` 配置文件中，注册并配置这个 Filter。将以下 XML 代码片段添加到 `web.xml` 文件中的 `<web-app>` 标签内：

```
<filter>
    <filter-name>requestTimingFilter</filter-name>
    <filter-class>RequestTimingFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>requestTimingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

这样，当你的 web 应用收到 HTTP 请求时，`RequestTimingFilter` 将会拦截这个请求，并在请求处理前后记录时间。通过计算请求开始和结束的时间差，我们可以得到请求访问的耗时。

2、源码

从刚刚的示例代码中，我们发现，添加过滤器非常方便，不需要修改任何代码，定义一个实现 javax.servlet.Filter 的类，再改改配置就搞定了，完全符合开闭原则。那 Servlet Filter 是如何做到如此好的扩展性的呢？我想你应该已经猜到了，它利用的就是职责链模式。现在，我们通过剖析它的源码，详细地看看它底层是如何实现的。

在上一节课中，我们讲到，职责链模式的实现包含处理器接口（IHandler）或抽象类（Handler），以及处理器链（HandlerChain）。对应到 Servlet Filter，javax.servlet.Filter 就是处理器接口，FilterChain 就是处理器链。接下来，我们重点来看 FilterChain 是如何实现的。

不过，我们前面也讲过，Servlet 只是一个规范，并不包含具体的实现，所以，Servlet 中的 FilterChain 只是一个接口定义。具体的实现类由遵从 Servlet 规范的 Web 容器来提供，比如，ApplicationFilterChain 类就是 Tomcat 提供的 FilterChain 的实现类，源码如下所示。

为了让代码更易读懂，我对代码进行了简化，只保留了跟设计思路相关的代码片段。完整的代码你可以自行去 Tomcat 中查看。

```
// 一个过滤器链包含了多个filter和一个servlet
public final class ApplicationFilterChain implements FilterChain {
  private int pos = 0; //当前执行到了哪个filter
  private int n; //filter的个数
  private ApplicationFilterConfig[] filters;
  private Servlet servlet;

  @Override
  public void doFilter(ServletRequest request, ServletResponse response) {
    if (pos < n) {
      ApplicationFilterConfig filterConfig = filters[pos++];
      Filter filter = filterConfig.getFilter();
      // 调用该方法是，应为内部会调用chain.doFilter(request, response);
      // 所以会形成递归调用
      filter.doFilter(request, response, this);
    } else {
      // filter都处理完毕后，执行servlet
      servlet.service(request, response);
    }
  }

  public void addFilter(ApplicationFilterConfig filterConfig) {
    for (ApplicationFilterConfig filter:filters)
      if (filter==filterConfig)
        return;
    if (n == filters.length) {
      //扩容INCREMENT个位置
      ApplicationFilterConfig[] newFilters = new ApplicationFilterConfig[n +
INCREMENT];
      System.arraycopy(filters, 0, newFilters, 0, n);
```

```
            filters = newFilters;
        }
        filters[n++] = filterConfig;
    }
}
```

ApplicationFilterChain 中的 doFilter() 函数的代码实现比较有技巧，**实际上是一个递归调用。**

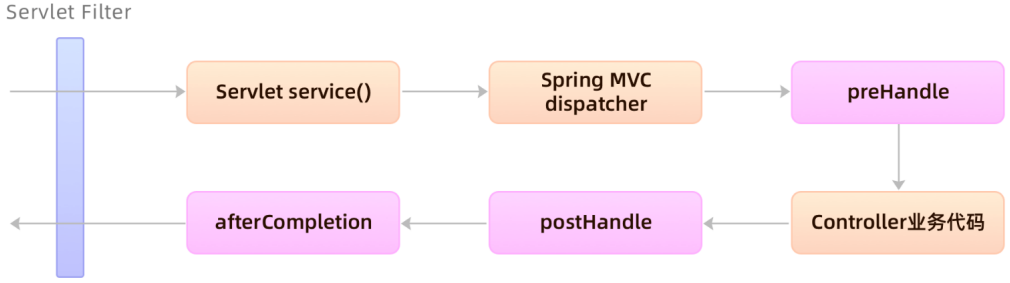
这样实现主要是为了在一个 doFilter() 方法中，**支持双向拦截，既能拦截客户端发送来的请求，也能拦截发送给客户端的响应**，你可以结合着 我们上边的那个例子，以及对比待会要讲到的 Spring Interceptor，来自己理解一下。而我们上一节课给出的两种实现方式，都没法做到在业务逻辑执行的前后，同时添加处理代码。

二、Spring Interceptor

1、使用

刚刚讲了 Servlet Filter，现在我们来讲一个功能上跟它非常类似的东西，Spring Interceptor，翻译成中文就是**拦截器。**尽管英文单词和中文翻译都不同，但这两者基本上可以看作一个概念，都用来实现对 HTTP 请求进行拦截处理。

它们不同之处在于，**Servlet Filter 是 Servlet 规范的一部分，实现依赖于 Web 容器。Spring Interceptor 是 Spring MVC 框架的一部分，由 Spring MVC 框架来提供实现。**客户端发送的请求，会先经过 Servlet Filter，然后再经过 Spring Interceptor，最后到达具体的业务代码中。我画了一张图来阐述一个请求的处理流程，具体如下所示。



在项目中，我们该如何使用 Spring Interceptor 呢？我们将上边的例子转化为spring的拦截器：

以下是一个简单的 Spring Interceptor 示例，用于记录 HTTP 请求访问的耗时：

首先，创建一个名为 `RequestTimingInterceptor` 的 Java 类，实现 HandlerInterceptor 接口。

```
public class RequestTimingInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
        // 在请求到达时记录开始时间
        long startTime = System.currentTimeMillis();
        request.setAttribute("startTime", startTime);

        // 返回 true 以将请求传递给责任链上的下一个 Interceptor 或目标 Controller
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) {
        // 请求处理完成后，在这里我们什么都不做
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {
        // 在请求处理完成后记录结束时间，并计算耗时
        long endTime = System.currentTimeMillis();
        long startTime = (Long) request.getAttribute("startTime");
        long timeElapsed = endTime - startTime;

        String requestURI = request.getRequestURI();
        System.out.println("请求 " + requestURI + " 的访问耗时：" + timeElapsed + " 毫秒");
    }
}
```

接下来，在你的 Spring 配置类中，注册并配置这个 Interceptor。将以下代码添加到一个配置类中（通常是一个使用 `@Configuration` 注解的 Java 类）：

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        // 将 RequestTimingInterceptor 添加到拦截器注册表中
        registry.addInterceptor(new RequestTimingInterceptor());
    }
}
```

这样，当你的 Spring 应用收到 HTTP 请求时，`RequestTimingInterceptor` 将会拦截这个请求，并在请求处理前后记录时间。通过计算请求开始和结束的时间差，我们可以得到请求访问的耗时。

2、源码

同样，我们还是来剖析一下，Spring Interceptor 底层是如何实现的。当然，它也是基于职责链模式实现的。其中，HandlerExecutionChain 类是职责链模式中的处理器链。它的实现相较于 Tomcat 中的 ApplicationFilterChain 来说，逻辑更加清晰，不需要使用递归来实现，主要是因为它将**请求和响应的拦截工作，拆分到了两个函数中实现**。HandlerExecutionChain 的源码如下所示，同样，我对代码也进行了一些简化，只保留了关键代码。

```
public class HandlerExecutionChain {
    // HandlerExecutionChain 包含一个拦截器链和一个处理器
    private final Object handler;
    private HandlerInterceptor[] interceptors;

    // 向拦截器链中添加一个拦截器
    public void addInterceptor(HandlerInterceptor interceptor) {
        initInterceptorList().add(interceptor);
    }

    // 在处理器执行之前，应用拦截器链中的所有拦截器的 preHandle 方法
    boolean applyPreHandle(HttpServletRequest request, HttpServletResponse
response) throws Exception {
        HandlerInterceptor[] interceptors = getInterceptors();
        if (!ObjectUtils.isEmpty(interceptors)) {
            for (int i = 0; i < interceptors.length; i++) {
                HandlerInterceptor interceptor = interceptors[i];
```

```
            // 如果 preHandle 方法返回 false，责任链将被中断，后续拦截器不会被执行
            if (!interceptor.preHandle(request, response, this.handler)) {
                triggerAfterCompletion(request, response, null);
                return false;
            }
        }
    }
    return true;
}

// 在处理器执行之后，应用拦截器链中的所有拦截器的 postHandle 方法
void applyPostHandle(HttpServletRequest request, HttpServletResponse response, ModelAndView mv) throws Exception {
    HandlerInterceptor[] interceptors = getInterceptors();
    if (!ObjectUtils.isEmpty(interceptors)) {
        // 从后向前执行拦截器链中的 postHandle 方法
        for (int i = interceptors.length - 1; i >= 0; i--) {
            HandlerInterceptor interceptor = interceptors[i];
            interceptor.postHandle(request, response, this.handler, mv);
        }
    }
}

// 当处理器完成后，触发拦截器链中所有拦截器的 afterCompletion 方法
void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, Exception ex)
    throws Exception {
    HandlerInterceptor[] interceptors = getInterceptors();
    if (!ObjectUtils.isEmpty(interceptors)) {
        for (int i = this.interceptorIndex; i >= 0; i--) {
            HandlerInterceptor interceptor = interceptors[i];
            try {
                interceptor.afterCompletion(request, response, this.handler, ex);
            } catch (Throwable ex2) {
                logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);
            }
        }
    }
}
}
```

在 Spring 框架中，DispatcherServlet 的 doDispatch() 方法来分发请求，它在真正的业务逻辑执行前后，执行 HandlerExecutionChain 中的 applyPreHandle() 和 applyPostHandle() 函数，用来实现拦截的功能。具体的代码实现很简单，我们截取了关键的几行代码如下。

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {


    // 拦截器之前
    if (!mappedHandler.applyPreHandle(processedRequest, response)) {
        return;
    }

    // 调用处理器
    mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
    if (asyncManager.isConcurrentHandlingStarted()) {
        return;
    }

    // 拦截器之后
    mappedHandler.applyPostHandle(processedRequest, response, mv);

    // 其他内容均省略

}
```

`HandlerExecutionChain` 类主要包含以下方法：

1. `applyPreHandle()`：在处理器执行之前，应用拦截器链中的所有拦截器的 `preHandle` 方法。如果某个拦截器的 `preHandle` 方法返回 `false`，责任链将被中断，后续拦截器不会被执行。
2. `applyPostHandle()`：在处理器执行之后，应用拦截器链中的所有拦截器的 `postHandle` 方法。这些方法是从后向前执行的，即先执行最后一个拦截器的 `postHandle` 方法，再执行倒数第二个拦截器的 `postHandle` 方法，依次类推。
3. `triggerAfterCompletion()`：当处理器完成后，触发拦截器链中所有拦截器的 `afterCompletion`方法。这些方法同样是从后向前执行的，即先执行最后一个拦截器的`afterCompletion`方法，再执行倒数第二个拦截器的 `afterCompletion` 方法，依次类推。

上述方法保证了在 Spring 框架中，拦截器链在处理器的执行前、执行中和执行后可以对请求进行处理。这种机制使得 Spring 框架的拦截器功能非常灵活和易于扩展。

三、mybtias的插件

1、使用

MyBatis 在实现插件系统时采用了责任链模式。当执行 SQL 语句时，MyBatis 会将这个操作交给责任链上的一系列插件进行处理。每个插件都有机会在操作执行前和执行后进行一些处理。

以下是一个简单的 MyBatis 插件示例，用于记录 SQL 执行耗时，具体的知识需要大家自行学习mybaits的插件：

首先，创建一个名为 `SqlTimingInterceptor` 的 Java 类，实现 `Interceptor` 接口。

```
@Intercepts({
    @Signature(type = Executor.class,method = "update",args =
{MappedStatement.class, Object.class}),
    @Signature(type = Executor.class,method = "query",args =
{MappedStatement.class, Object.class, RowBounds.class, ResultHandler.class})})
public class SqlTimingInterceptor implements Interceptor {

    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        // 在 SQL 执行前记录开始时间
        long startTime = System.currentTimeMillis();

        // 执行目标方法（SQL）
        Object result = invocation.proceed();

        // 在 SQL 执行后记录结束时间，并计算耗时
        long endTime = System.currentTimeMillis();
        long timeElapsed = endTime - startTime;

        MappedStatement mappedStatement = (MappedStatement)
invocation.getArgs()[0];
        String sqlId = mappedStatement.getId();
```

```
        System.out.println("执行 SQL [" + sqlId + "] 的耗时：" + timeElapsed + " 毫秒");

        return result;
    }

    @Override
    public Object plugin(Object target) {
        return Plugin.wrap(target, this);
    }

    @Override
    public void setProperties(Properties properties) {
        // 在这里配置插件的属性，如果需要的话
    }
}
```

接下来，在你的 MyBatis 配置文件（例如：`mybatis-config.xml`）中，注册并配置这个插件。将以下 XML 代码片段添加到 `<configuration>` 标签内：

```
<plugins>
    <plugin interceptor="SqlTimingInterceptor"/>
</plugins>
```

这样，当你的 MyBatis 应用执行 SQL 语句时，`SqlTimingInterceptor` 将会拦截这个操作，并在执行前后记录时间。通过计算执行开始和结束的时间差，我们可以得到 SQL 执行的耗时。

2、源码

mybatis的plugin使用的是动态代理实现，针对每一个插件都包一次代理，实现了类似的功能，从这个我们也发现实现某一种设计模式的方法并非是固定的。

首先，MyBatis 定义了一个 `Interceptor` 接口，所有的插件都需要实现这个接口。

```
public interface Interceptor {
    Object intercept(Invocation invocation) throws Throwable;
    Object plugin(Object target);
    void setProperties(Properties properties);
}
```

接下来，我们来看一下 `Invocation` 类。`Invocation` 封装了一个方法调用，它包含了要调用的方法、方法所属的对象以及方法的参数。`Invocation` 是责任链上的一个节点。

```
public class Invocation {
    private final Object target; // 被调用方法所属的对象
    private final Method method; // 要调用的方法
    private final Object[] args; // 方法的参数

    // ... 构造函数和其他方法
}
```

当一个插件处理 `Invocation` 时，它可以选择调用 `proceed()` 方法来继续执行责任链上的下一个插件。如果一个插件没有调用 `proceed()`，责任链将被中断，后续的插件将不会被执行。

```
public Object proceed() throws InvocationTargetException,
IllegalAccessException {
    return method.invoke(target, args);
}
```

现在让我们看一下责任链是如何构建的。在 MyBatis 中，责任链的创建是通过 `Plugin` 类实现的。`Plugin` 类实现了 JDK 的 `InvocationHandler` 接口，它允许我们通过代理的方式拦截方法调用。在 `plugin()` 方法中，`Plugin` 类使用 Java 的动态代理机制创建了一个代理对象，并将当前插件（Interceptor）设置为代理对象的处理器。

```
public class Plugin implements InvocationHandler {
    private final Object target;
    private final Interceptor interceptor;
    private final Map<Class<?>, Set<Method>> signatureMap;

    // ... 构造函数和其他方法

    public static Object wrap(Object target, Interceptor interceptor) {
        Map<Class<?>, Set<Method>> signatureMap =
getSignatureMap(interceptor);
        Class<?> type = target.getClass();
        Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
        if (interfaces.length > 0) {
            return Proxy.newProxyInstance(
                type.getClassLoader(),
```

```
                interfaces,
                new Plugin(target, interceptor, signatureMap));
        }
        return target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        try {
            Set<Method> methods = signatureMap.get(method.getDeclaringClass());
            if (methods != null && methods.contains(method)) {
                return interceptor.intercept(new Invocation(target, method, args));
            }
            return method.invoke(target, args);
        } catch (Exception e) {
            throw ExceptionUtil.unwrapThrowable(e);
        }
    }
}
```

在这个实现中，当一个插件拦截到一个方法调用时，`invoke()` 方法将被执行。在这个方法中，我们首先检查当前方法是否在插件的签名映射中（`signatureMap`）。如果在映射中，说明插件需要处理这个方法，于是我们调用插件的 `intercept()` 方法。否则，我们直接调用原始方法。

这样，当 MyBatis 执行 SQL 语句时，这个操作会被传递给责任链上的插件。每个插件都有机会在操作执行前和执行后进行处理。插件可以修改操作的参数、改变操作的结果，甚至中断操作的执行。这种架构使得 MyBatis 的插件系统非常灵活和易于扩展。

四、应用场景

责任链模式在工作中的应用广泛，以下列举了一些常见的使用场景，大家可以自行学习：

1. 日志记录器：在应用程序中，我们可能需要将日志记录到不同的位置，如控制台、文件、数据库等。我们可以创建一个日志记录器链，每个记录器处理特定级别的日志，然后将请求传递给下一个记录器。这样，可以根据日志级别灵活地记录日志信息。

2. Web 应用中的过滤器和拦截器：在 Web 应用程序中，我们经常需要对请求进行预处理和后处理，如身份验证、授权、编码转换、请求日志记录等。过滤器和拦截器就是典型的使用责任链模式的场景，请求和响应在过滤器或拦截器链中依次传递，每个过滤器或拦截器执行特定的任务。
3. 工作流引擎：在一个工作流引擎中，一个请求可能需要经过多个处理步骤，这些步骤可以看作是一个责任链。每个处理器处理请求的一个部分，然后将请求传递给下一个处理器，直到请求被完全处理。
4. 软件审批流程：在企业软件开发过程中，代码审查、需求审批、文档审查等流程可能需要多个审批者按顺序审批。这种场景下，责任链模式能够确保每个审批者只关注自己的审批职责，并将审批请求传递给下一个审批者。
5. 电子邮件处理：在一个电子邮件处理系统中，可能需要对不同类型的邮件进行不同的处理，如**垃圾邮件过滤、自动回复、邮件归类**等。在这种情况下，可以使用责任链模式来创建一个邮件处理链，每个处理器负责处理特定类型的邮件，然后将邮件传递给下一个处理器。
6. 事件处理系统：在一个事件驱动的系统中，可能需要对不同类型的事件进行不同的处理。责任链模式可以用来创建一个事件处理器链，每个处理器负责处理特定类型的事件，并将事件传递给下一个处理器。这样可以确保系统的可扩展性和灵活性。
7. 规则引擎：在某些业务场景下，可能需要按照一定的规则对数据进行处理。规则引擎是典型的使用责任链模式的场景。每个规则可以看作是一个处理器，对数据进行特定的处理，然后将数据传递给下一个规则，直至所有规则都被执行。
8. 任务调度系统：在任务调度系统中，根据任务的优先级、类型和资源需求，可能需要将任务分配给不同的执行器。责任链模式可以确保每个执行器只关注自己可以处理的任务，并将其他任务传递给下一个执行器。

除了以上的使用场景，我们再给大家列举一个使用场景，如下：

对于支持 UGC（User Generated Content，用户生成内容）的应用（比如论坛）来说，用户生成的内容（比如，在论坛中发表的帖子）可能会包含一些敏感词（比如涉黄、广告、反动等词汇）。针对这个应用场景，我们就可以利用职责链模式来过滤这些敏感词。

在这个应用场景中，我们可以创建一个过滤器链来过滤用户生成的内容。每个过滤器负责处理一种类型的敏感词，然后将内容传递给下一个过滤器。以下是一个简单的实现示例：

首先，我们定义一个过滤器接口：

```
public interface ContentFilter {
    String filter(String content);
}
```

然后，我们实现不同类型的过滤器，例如涉黄过滤器、广告过滤器和反动过滤器：

```
// 涉黄过滤器
public class PornographyFilter implements ContentFilter {
    @Override
    public String filter(String content) {
        // 这里用简单的字符串替换来表示过滤操作，实际应用中需要更复杂的过滤逻辑
        return content.replaceAll("涉黄词汇", "***");
    }
}

// 广告过滤器
public class AdvertisementFilter implements ContentFilter {
    @Override
    public String filter(String content) {
        return content.replaceAll("广告词汇", "***");
    }
}

// 反动过滤器
public class ReactionaryFilter implements ContentFilter {
    @Override
    public String filter(String content) {
        return content.replaceAll("反动词汇", "***");
    }
}
```

接下来，我们创建一个过滤器链：

```
public class FilterChain {
    private List<ContentFilter> filters = new ArrayList<>();

    public FilterChain addFilter(ContentFilter filter) {
        filters.add(filter);
        return this;
    }

    public String doFilter(String content) {
        for (ContentFilter filter : filters) {
            content = filter.filter(content);
        }
        return content;
    }
```

```
}
```

最后，我们可以在应用中使用过滤器链来处理用户生成的内容：

```
public class Main {
    public static void main(String[] args) {
        // 创建一个过滤器链
        FilterChain filterChain = new FilterChain();
        filterChain.addFilter(new PornographyFilter())
                .addFilter(new AdvertisementFilter())
                .addFilter(new ReactionaryFilter());

        // 用户生成的内容
        String userContent = "这里有一些涉黄词汇，这里有一些广告词汇，这里有一些反动词汇。";

        // 使用过滤器链处理内容
        String filteredContent = filterChain.doFilter(userContent);
        System.out.println(filteredContent);
    }
}
```

代码中的注释已经用中文解释了每个部分的作用。通过使用责任链模式，我们可以灵活地添加、删除或修改过滤器，使得过滤用户生成内容的逻辑更加易于维护和扩展。