第四篇章 行为型设计模式

第五章 状态模式

一、简介概述

从今天起，我们开始学习状态模式。在实际的软件开发中，状态模式并不是很常用，但是在能够用到的场景里，它可以发挥很大的作用。从这一点上来看，它有点像我们之前讲到的**组合模式**。一般不用，特定场景下使用。

状态设计模式是一种行为型设计模式，它允许对象在**其内部状态发生变化时改变其行为**。这种模式可以**消除大量的条件语句**，并将每个状态的行为封装到单独的类中。

状态模式的主要组成部分如下：

1. **上下文（Context）**：上下文通常包含一个**具体状态的引用**，用于维护当前状态。上下文委托给当前状态对象处理状态相关行为。
2. **抽象状态（State）**：定义一个接口，用于封装与上下文的特定状态相关的行为。
3. **具体状态（Concrete State）**：实现抽象状态接口，为具体状态定义行为。每个具体状态类对应一个状态。

现在我们来看一个简单的 Java 示例。假设我们要**模拟一个简易的电视遥控器**，具有开启、关闭和调整音量的功能。

如果我们不使用设计模式，编写出来的代码可能是这个样子的，我们需要针对电视机当前的状态为每一次操作编写判断逻辑：

```
public class TV {
    private boolean isOn;
    private int volume;

    public TV() {
        isOn = false;
        volume = 0;
    }

    public void turnOn() {
        // 如果是开启状态
        if (isOn) {
```

```
            System.out.println("TV is already on.");
            // 否则打开电视
        } else {
            isOn = true;
            System.out.println("Turning on the TV.");
        }
    }

    public void turnOff() {
        if (isOn) {
            isOn = false;
            System.out.println("Turning off the TV.");
        } else {
            System.out.println("TV is already off.");
        }
    }

    public void adjustVolume(int volume) {
        if (isOn) {
            this.volume = volume;
            System.out.println("Adjusting volume to: " + volume);
        } else {
            System.out.println("Cannot adjust volume, TV is off.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        TV tv = new TV();

        tv.turnOn();
        tv.adjustVolume(10);
        tv.turnOff();
    }
}
```

当然在该例子中我们的状态比较少，所以代码看起来也不是很复杂，但是状态如果变多了呢？比如加入换台，快捷键、静音等功能后呢？你会发现条件分支会急速膨胀，所以此时状态设计模式就要登场了：

首先，我们定义抽象状态接口 `TVState`，将每一个修改状态的动作抽象成一个接口：

```
public interface TVState {
    void turnOn();
    void turnOff();
    void adjustVolume(int volume);
}
```

接下来，我们为每个具体状态创建类，实现 `TVState` 接口。例如，我们创建 `TVOnState` 和 `TVOffState` 类：

```
// 在on状态下，去执行以下各种操作
public class TVOnState implements TVState {
    @Override
    public void turnOn() {
        System.out.println("TV is already on.");
    }

    @Override
    public void turnOff() {
        System.out.println("Turning off the TV.");
    }

    @Override
    public void adjustVolume(int volume) {
        System.out.println("Adjusting volume to: " + volume);
    }
}

// 在关机的状态下执行以下的操作
public class TVOffState implements TVState {
    @Override
    public void turnOn() {
        System.out.println("Turning on the TV.");
    }

    @Override
    public void turnOff() {
        System.out.println("TV is already off.");
    }

    @Override
    public void adjustVolume(int volume) {
        System.out.println("Cannot adjust volume, TV is off.");
```

```
    }
}
```

接下来，我们定义上下文类 `TV`：

```
public class TV {
    // 当前状态
    private TVState state;

    public TV() {
        state = new TVOffState();
    }

    public void setState(TVState state) {
        this.state = state;
    }

    public void turnOn() {
        // 打开
        state.turnOn();
        // 设置为开机状态
        setState(new TVOnState());
    }

    public void turnOff() {
        // 关闭
        state.turnOff();
        // 设置为关机状态
        setState(new TVOffState());
    }

    public void adjustVolume(int volume) {
        state.adjustVolume(volume);
    }
}
```

最后，我们可以通过以下方式使用这些类：

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

        tv.turnOn();
        tv.adjustVolume(10);
        tv.turnOff();
    }
}
```

这个例子展示了状态模式的基本结构和用法。通过使用状态模式，我们可以更好地组织和管理与特定状态相关的代码。当状态较多时，这种模式的优势就会凸显出来，同时我们在代码时，因为我们会对每个状态进行独立封装，所以也会简化代码编写。

二、有限状态机

状态模式一般**用来实现状态机**，而**状态机常用在游戏、工作流引擎等系统开发中**。不过，状态机的实现方式有多种，除了状态模式，比较常用的还有分支逻辑法和查表法。今天，我们就详细讲讲这几种实现方式，并且对比一下它们的优劣和应用场景。

话不多说，让我们正式开始今天的学习吧！

有限状态机，英文翻译是 Finite State Machine，缩写为 FSM，简称为状态机，比较官方的说法是：**有限状态机是描述对象在它的生命周期内所经历的状态序列，以及如何响应来自外界的各种事件。**。状态机有 3 个组成部分：状态（State）、事件（Event）、动作（Action）。其中，事件也称为转移条件（Transition Condition）。事件触发状态的转移及动作的执行。不过，动作不是必须的，也可能只转移状态，不执行任何动作。

对于刚刚给出的状态机的定义，我结合一个具体的例子，来进一步解释一下。

“超级马里奥”游戏不知道你玩过没有？在游戏中，马里奥可以变身为三种形态，小马里奥（Small Mario）、大马里奥（Big Mario）和火焰马里奥（Fire Mario）。马里奥可以通过吃蘑菇（Mushroom）、火花（Fire Flower）或被敌人攻击（Enemy Attack）来改变形态。我们将用状态图表示这个马里奥的有限状态机。

这个状态图描述了马里奥角色在游戏中的状态转换。我们可以根据这个有限状态机来实现一个马里奥游戏的简化版本。在实际游戏开发中，通常会使用游戏引擎或编程框架来处理状态转换，而不是手动编写状态机代码。不过，这个简化示例可以帮助我们理解有限状态机在马里奥游戏中的应用。

1、分支法

对于如何实现状态机，我总结了**三种方式**。其中，最简单直接的实现方式是，参照状态转移图，将每一个状态转移，原模原样地直译成代码。这样编写的代码会包含大量的 if-else 或 switch-case 分支判断逻辑，甚至是**嵌套的分支判断逻辑**，所以，我们把这种方法暂且命名为分支法。

按照这个实现思路，我将上面的骨架代码补全一下。补全之后的代码如下所示：

下面是一个使用if-else 语句实现的马里奥形态变化的代码示例：

```
public class Mario {
  private MarioState state;

  public Mario() {
    state = MarioState.SMALL;
  }

  public void handleEvent(Event event) {
    MarioState newState = state;

    // 处理吃蘑菇事件
    if (event == Event.MUSHROOM) {
```

```
            if (state == MarioState.SMALL) {
                newState = MarioState.BIG;
            }
        // 处理吃火花事件
        } else if (event == Event.FIRE_FLOWER) {
            if (state == MarioState.BIG) {
                newState = MarioState.FIRE;
            }
        // 处理遇到小怪事件
        } else if (event == Event.ENEMY_ATTACK) {
            if (state == MarioState.BIG) {
                newState = MarioState.SMALL;
            } else if (state == MarioState.FIRE) {
                newState = MarioState.BIG;
            } else if (state == MarioState.SMALL) {
                newState = MarioState.DEAD;
            }
        // 处理掉坑事件
        } else if (event == Event.FALL_INTO_PIT) {
            newState = MarioState.DEAD;
        }

        System.out.printf("从 %s 变为 %s%n", state, newState);
        state = newState;
    }
}

public class MarioDemo {
    public static void main(String[] args) {
        Mario mario = new Mario();

        mario.handleEvent(Event.MUSHROOM); // 变为大马里奥
        mario.handleEvent(Event.FIRE_FLOWER); // 变为火焰马里奥
        mario.handleEvent(Event.ENEMY_ATTACK); // 变为死亡马里奥
    }
}
```

在这个示例中，我们**使用 if-else 语句来处理状态转换。**在 `handleEvent` 方法中，我们根据事件和当前状态的组合来**确定新状态**，并**更新马里奥的状态。**这种实现方法相较于查表法和面向对象实现更为简单，但可能在状态和事件更多的情况下变得难以维护。选择合适的实现方法取决于实际需求和场景。

2、查表法

实际上，上面这种实现方法有点类似 hard code，对于复杂的状态机来说不适用，而状态机的第二种实现方式查表法，就更加合适了。接下来，我们就一块儿来看下，如何利用查表法来补全骨架代码。

我们可以将马里奥的状态转移方式表示为以下表格：

当前状态/事件	MUSHROOM	FIRE_FLOWER	ENEMY_ATTACK	FALL_INTO_
SMALL	BIG	SMALL	DEAD	DEAD
BIG	BIG	FIRE	SMALL	DEAD
FIRE	FIRE	FIRE	BIG	DEAD
DEAD	DEAD	DEAD	DEAD	DEAD

这个表格显示了马里奥在不同状态下遇到不同事件时将转换为的新状态。从左到右分别表示当前状态（SMALL, BIG, FIRE, DEAD），从上到下表示事件（MUSHROOM, FIRE_FLOWER, ENEMY_ATTACK, FALL_INTO_PIT）。表格中的每个单元格表示对应状态和事件的状态转换结果。

查表法是一种使用查找表来处理状态转换的方法，可以简化状态机的实现。以下是使用查表法实现马里奥形态变化的代码示例：

```
enum MarioState {
    SMALL, BIG, FIRE, DEAD
}

enum Event {
    MUSHROOM, FIRE_FLOWER, ENEMY_ATTACK, FALL_INTO_PIT
}

public class Mario {
    private MarioState state;

    // 使用二维数组定义状态转换表
    private static final MarioState[][] TRANSITION_TABLE = {
            // SMALL, BIG, FIRE, DEAD
```

```
            {MarioState.BIG, MarioState.SMALL, MarioState.SMALL,
MarioState.DEAD}, // MUSHROOM
            {MarioState.SMALL, MarioState.FIRE, MarioState.FIRE, MarioState.DEAD},
// FIRE_FLOWER
            {MarioState.DEAD, MarioState.SMALL, MarioState.BIG, MarioState.DEAD},
// ENEMY_ATTACK
            {MarioState.DEAD, MarioState.DEAD, MarioState.DEAD, MarioState.DEAD}
 // FALL_INTO_PIT
    };

    public Mario() {
        state = MarioState.SMALL;
    }

    public void handleEvent(Event event) {
        // 使用查表法获取状态转换后的新状态
        MarioState newState = TRANSITION_TABLE[event.ordinal()][state.ordinal()];

        // 打印状态转换信息
        System.out.printf("从 %s 变为 %s%n", state, newState);

        // 更新状态
        state = newState;
    }
}

public class MarioDemo {
    public static void main(String[] args) {
        Mario mario = new Mario();

        mario.handleEvent(Event.MUSHROOM); // 变为大马里奥
        mario.handleEvent(Event.FIRE_FLOWER); // 变为火焰马里奥
        mario.handleEvent(Event.FALL_INTO_PIT); // 变为死亡马里奥
    }
}
```

在这个示例中，我们使用了一个二维数组 `TRANSITION_TABLE` 来表示状态转换表。数组的行表示事件，列表示马里奥的当前状态，数组的元素表示新状态。通过查找表，我们可以直接获取状态转换后的新状态，从而简化状态机的实现。

`handleEvent` 方法中，我们根据事件和当前状态的序数来查找新状态，并更新马里奥的状态。这个查表法实现的有限状态机相比之前的面向对象实现更为简洁，但可能不适用于需要处理复杂事件或动作的场景。根据实际需求选择合适的实现方法是很重要的。

相对于分支逻辑的实现方式，查表法的代码实现更加清晰，可读性和可维护性更好。当修改状态机时，我们只需要修改 transitionTable 和 actionTable 两个二维数组即可。实际上，如果我们把这两个二维数组存储在配置文件中，当需要修改状态机时，我们甚至可以不修改任何代码，只需要修改配置文件就可以了

3、状态模式

在查表法的代码实现中，事件触发的动作只是简单的状态或者数值，所以，我们用一个 MarioState类型的二维数组 TRANSITION_TABLE 就能表示，二维数组中的值表示出发事件后的新状态。但是，如果要执行的动作并非这么简单，而是一系列复杂的逻辑操作（比如加减分数、处理位置信息等等），我们就没法用如此简单的二维数组来表示了。这也就是说，查表法的实现方式**有一定局限性。**

虽然分支逻辑的实现方式不存在这个问题，但它又存在前面讲到的其他问题，比如分支判断逻辑较多，导致代码可读性和可维护性不好等。实际上，针对分支逻辑法存在的问题，我们可以**使用状态模式来解决。**

状态模式通过将事件触发的状态转移和动作执行，拆分到不同的状态类中，来避免分支判断逻辑。我们还是结合代码来理解这句话。

利用状态模式，我们来补全 MarioStateMachine 类，补全后的代码如下所示。

以下是一个使用 Java 实现的简化版马里奥形态变化的案例代码，我为代码添加了中文注释以便理解：

```
// 定义事件枚举类型
enum Event {
    // 吃蘑菇，吃火花，遇到小怪，调入深坑
    MUSHROOM, FIRE_FLOWER, ENEMY_ATTACK, FALL_INTO_PIT
}

// 定义马里奥状态接口
interface MarioState {
    void handleEvent(Event event);
}
```

```java
// 实现死亡马里奥状态
class DeadMario implements MarioState {
    private Mario mario;

    public DeadMario(Mario mario) {
        this.mario = mario;
    }

    @Override
    public void handleEvent(Event event) {
        System.out.println("马里奥已死亡，无法处理事件");
    }
}

// 实现小马里奥状态
class SmallMario implements MarioState {
    private Mario mario;

    public SmallMario(Mario mario) {
        this.mario = mario;
    }

    @Override
    public void handleEvent(Event event) {
        switch (event) {
            case MUSHROOM:
                System.out.println("变为大马里奥");
                mario.setState(new BigMario(mario));
                break;
            case FIRE_FLOWER:
                System.out.println("小马里奥不能直接变为火焰马里奥");
                break;
            case ENEMY_ATTACK:
                System.out.println("小玛丽奥去世了");
                mario.setState(new DeadMario(mario));
                break;
            case FALL_INTO_PIT:
                System.out.println("小玛丽奥去世了");
                mario.setState(new DeadMario(mario));
                break;
        }
    }
}
```

```
// 实现大马里奥状态
class BigMario implements MarioState {
    private Mario mario;

    public BigMario(Mario mario) {
        this.mario = mario;
    }

    @Override
    public void handleEvent(Event event) {
        switch (event) {
            case MUSHROOM:
                System.out.println("保持大马里奥");
                break;
            case FIRE_FLOWER:
                System.out.println("变为火焰马里奥");
                mario.setState(new FireMario(mario));
                break;
            case ENEMY_ATTACK:
                System.out.println("变为小马里奥");
                mario.setState(new SmallMario(mario));
                break;
            case FALL_INTO_PIT:
                System.out.println("马里奥去世了");
                mario.setState(new DeadMario(mario));
                break;
        }
    }
}

// 实现火焰马里奥状态
class FireMario implements MarioState {
    private Mario mario;

    public FireMario(Mario mario) {
        this.mario = mario;
    }

    @Override
    public void handleEvent(Event event) {
        switch (event) {
            case MUSHROOM:
```

```
                System.out.println("保持火焰马里奥");
                break;
            case FIRE_FLOWER:
                System.out.println("保持火焰马里奥");
                break;
            case ENEMY_ATTACK:
                System.out.println("变为大马里奥");
                mario.setState(new BigMario(mario));
                break;
            case FALL_INTO_PIT:
                System.out.println("马里奥去世了");
                mario.setState(new DeadMario(mario));
                break;
        }
    }
}

// 定义马里奥类，作为状态的上下文
class Mario {
    private MarioState state;

    public Mario() {
        state = new SmallMario(this);
    }

    public void setState(MarioState state) {
        this.state = state;
    }

    public void handleEvent(Event event) {
        state.handleEvent(event);
    }
}

// 测试类
public class MarioDemo {
    public static void main(String[] args) {
        Mario mario = new Mario();

        mario.handleEvent(Event.MUSHROOM); // 变为大马里奥
        mario.handleEvent(Event.FIRE_FLOWER); // 变为火焰马里奥
        mario.handleEvent(Event.ENEMY_ATTACK); // 变为大马里奥
    }
```

```
}
```

在这个简化示例中，我们定义了 `MarioState` 接口以及实现了`DeadMario`、`SmallMario`、`BigMario` 和 `FireMario` 类，分别表示马里奥的四种形态。每个形态类实现了 `handleEvent` 方法，用于处理不同的游戏事件并根据有限状态机规则进行状态转换。

`Mario` 类作为状态的上下文，用于管理和切换马里奥的状态。它有一个 `setState` 方法，用于更新当前状态。`handleEvent` 方法将事件传递给当前状态，以便根据事件执行相应的状态转换。

在 `MarioDemo` 测试类中，我们创建了一个 `Mario` 实例，并通过调用 `handleEvent` 方法模拟游戏中的事件。通过运行这个测试类，你可以观察到马里奥根据有限状态机的规则在不同形态之间切换。

这个简化示例展示了如何使用有限状态机来实现马里奥角色的形态变化。在实际游戏开发中，你可能需要考虑更多的事件和状态，以及与游戏引擎或框架集成的方式。不过，这个示例可以帮助你理解有限状态机在游戏中的应用。

三、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。

今天我们讲解了状态模式。虽然网上有各种状态模式的定义，但是你只要记住状态模式是状态机的一种实现方式即可。状态机又叫有限状态机，它有 3 个部分组成：状态、事件、动作。其中，事件也称为转移条件。事件触发状态的转移及动作的执行。不过，动作不是必须的，也可能只转移状态，不执行任何动作。

针对状态机，今天我们总结了三种实现方式。

第一种实现方式叫分支逻辑法。利用 if-else 或者 switch-case 分支逻辑，参照状态转移图，将每一个状态转移原模原样地直译成代码。对于简单的状态机来说，这种实现方式最简单、最直接，是首选。

第二种实现方式叫查表法。对于状态很多、状态转移比较复杂的状态机来说，查表法比较合适。通过二维数组来表示状态转移图，能极大地提高代码的可读性和可维护性。

第三种实现方式叫状态模式。对于状态并不多、状态转移也比较简单，但事件触发执行的动作包含的业务逻辑可能比较复杂的状态机来说，我们首选这种实现方式。

状态模式的代码实现还存在一些问题，比如，状态接口中定义了所有的事件函数，这就导致，即便某个状态类并不需要支持其中的某个或者某些事件，但也要实现所有的事件函数。不仅如此，添加一个事件到状态接口，所有的状态类都要做相应的修改。针对这些问题，你有什么解决方法吗？

第六章 迭代器模式

迭代器模式。它用来**遍历集合对象**。不过，很多编程语言都将迭代器作为一个基础的类库，直接提供出来了。在平时开发中，特别是业务开发，我们直接使用即可，**很少会自己去实现一个迭代器**。不过，知其然知其所以然，弄懂原理能帮助我们更好的使用这些工具类，所以，我觉得还是有必要学习一下这个模式。

我们知道，大部分编程语言都提供了多种遍历集合的方式，比如 for 循环、foreach 循环、迭代器等。所以，今天我们除了讲解迭代器的原理和实现之外，还会重点讲一下，相对于其他遍历方式，利用迭代器来遍历集合的优势。

话不多说，让我们正式开始今天的学习吧！

一、原理和实现

迭代器模式（Iterator Design Pattern），也叫作**游标模式**（Cursor Design Pattern）。

今天我们将学习迭代器（Iterator）设计模式。迭代器模式是一种**行为型设计模式**，它用于遍历集合对象，而无需**暴露该对象的底层表示**。这种模式非常适合在**处理大型集合**时使用，因为它提供了一种**更抽象的方式来访问集合的元素**。

让我们先来看一下迭代器模式的几个关键组件：

1. Iterator接口：定义了遍历集合所需的操作，例如next()、hasNext()等。
2. ConcreteIterator类：实现Iterator接口，具体实现遍历的细节。
3. Aggregate接口：定义创建Iterator对象的方法。
4. ConcreteAggregate类：实现Aggregate接口，具体实现创建ConcreteIterator的方法。

现在，让我们创建一个**基于迭代器模式的整数数组遍历示例**。首先，我们需要创建相应的接口和类。在这个例子中，我们将创建一个`IntArray`类，它实现了`Aggregate`接口，并使用`IntArrayIterator`类作为具体迭代器。

1、Iterator接口（与之前的示例相同）：

```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

2、创建具体的迭代器类（IntArrayIterator）：

```
public class IntArrayIterator implements Iterator {
    private IntArray intArray;
    private int index;

    public IntArrayIterator(IntArray intArray) {
        this.intArray = intArray;
        this.index = 0;
    }

    @Override
    public boolean hasNext() {
        return index < intArray.getLength();
    }

    @Override
    public Object next() {
        int value = intArray.getValueAt(index);
        index++;
        return value;
    }
}
```

3、Aggregate接口（与之前的示例相同）：

```
public interface Aggregate {
    Iterator iterator();
}
```

4、创建具体的聚合类（IntArray）：

```
public class IntArray implements Aggregate {
    private int[] values;

    public IntArray(int[] values) {
        this.values = values;
```

```
    }

    public int getValueAt(int index) {
        return values[index];
    }

    public int getLength() {
        return values.length;
    }

    @Override
    public Iterator iterator() {
        return new IntArrayIterator(this);
    }
}
```

现在我们可以使用迭代器模式来遍历一个整数数组：

```
public class Main {
    public static void main(String[] args) {
        int[] values = {1, 2, 3, 4, 5};
        IntArray intArray = new IntArray(values);

        Iterator iterator = intArray.iterator();
        while (iterator.hasNext()) {
            int value = (int) iterator.next();
            System.out.println(value);
        }
    }
}
```

运行这个程序，你将看到数组中的整数按顺序输出。这就是一个简单的基于迭代器模式的整数数组遍历示例。

二、jdk中的迭代器

1、使用

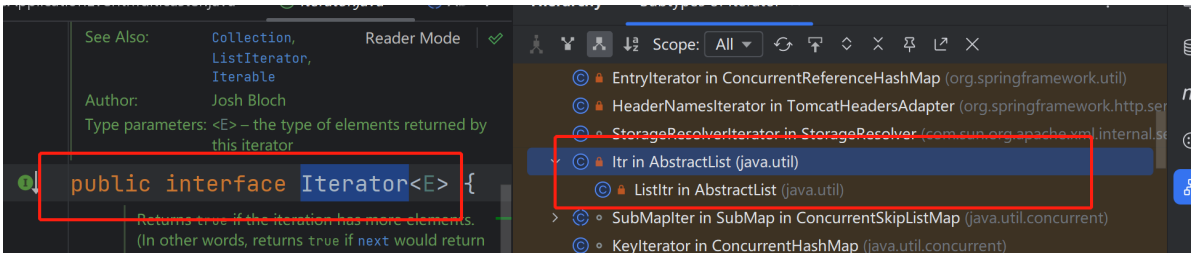
在Java中，许多集合类（如ArrayList）已经实现了内置的迭代器。下面是一个使用迭代器遍历ArrayList的示例，以及相应的中文注释：

```
public class Main {
    public static void main(String[] args) {
        // 创建一个ArrayList对象
        ArrayList<String> names = new ArrayList<>();

        // 向ArrayList中添加元素
        names.add("张三");
        names.add("李四");
        names.add("王五");

        // 获取ArrayList的迭代器对象
        Iterator<String> iterator = names.iterator();

        // 使用迭代器遍历ArrayList
        while (iterator.hasNext()) {
            // 使用next()方法获取下一个元素
            String name = iterator.next();
            System.out.println(name);
        }
    }
}
```

在这个例子中，我们首先创建了一个ArrayList对象并添加了一些字符串。然后，我们通过调用`names.iterator()`方法获得了一个迭代器对象。接下来，我们使用`hasNext()`和`next()`方法遍历ArrayList中的元素。

2、源码

我们可以发现，在jdk提供中，迭代器接口位于`java.util.Iterator`，他有很多具体的实现，我们选取我们日常工作中最常见的ArrayList为例，给大家深入讲解，其继承结构如下：



在AbstractList抽象类中，定义了如下的内部类，我们只看主干方法：

1、hasNext()方法的逻辑是，资源游标未到之后一个就认为其有下一个。

2、next()方法的处理逻辑是：调用子类的get方法（模板方法设计模式），并让游标后移一位。

3、remove()方法的逻辑是：调用子类的remove方法。

在这三个核心方法中我们先不去看中间的一些成员变量，主干内容很好理解：

```
private class Itr implements Iterator<E> {
    /**
     * Index of element to be returned by subsequent call to next.
     */
    int cursor = 0;

    /**
     * Index of element returned by most recent call to next or
     * previous.  Reset to -1 if this element is deleted by a call
     * to remove.
     */
    int lastRet = -1;

    /**
     * The modCount value that the iterator believes that the backing
     * List should have.  If this expectation is violated, the iterator
     * has detected concurrent modification.
     */
    int expectedModCount = modCount;

    public boolean hasNext() {
        return cursor != size();
    }

    public E next() {
        checkForComodification();
        try {
            int i = cursor;
            E next = get(i);
            lastRet = i;
            cursor = i + 1;
            return next;
        } catch (IndexOutOfBoundsException e) {
```

```
            checkForComodification();
            throw new NoSuchElementException(e);
        }
    }

    public void remove() {
        if (lastRet < 0)
            throw new IllegalStateException();
        checkForComodification();

        try {
            AbstractList.this.remove(lastRet);
            if (lastRet < cursor)
                cursor--;
            lastRet = -1;
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException e) {
            throw new ConcurrentModificationException();
        }
    }

    final void checkForComodification() {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
    }
}
```

事实上，抽象的父类提供了公用的方法：

```
public Iterator<E> iterator() {
    return new Itr();
}
```

我们可以使用该共享方法为其子类创建迭代器。

3、迭代时增删元素

如果我们在使用迭代器迭代一个集合时，使用集合原生的方法进行删除，会发生什么呢？

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

```
        // 创建一个ArrayList对象
        ArrayList<String> names = new ArrayList<>();

        // 向ArrayList中添加元素
        names.add("张三");
        names.add("李四");
        names.add("王五");

        // 获取ArrayList的迭代器对象
        Iterator<String> iterator = names.iterator();

        // 使用迭代器遍历ArrayList
        while (iterator.hasNext()) {
            // 使用next()方法获取下一个元素
            String name = iterator.next();
            names.remove(name);
            System.out.println(name);
        }
    }
}
```

如果你尝试了，你会发现会发生如下的异常;

```
"C:\Program Files\jdk-17.0.5\bin\java.exe" ...
张三
Exception in thread "main" java.util.ConcurrentModificationException Create breakpoint
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:967)
    at com.ydlclass.mybatissource.dao.AccountDao.main(AccountDao.java:28)
```

> q: 遍历集合的同时，为什么不能增删集合元素?

道理很简单，在通过迭代器来遍历集合元素的同时，增加或者删除集合中的元素时，有可能会导致**某个元素被重复遍历或遍历不到**。不过，并不是所有情况下都会遍历出错，有的时候也可以正常遍历，所以，这种行为称为**结果不可预期行为**或者**未决行为**。

事实上抛出异常，也是因为实现的迭代器中做了特殊处理。当然，我们编写的简单迭代器并不会抛出异常，只是因为我们并没有做特殊处理。

注：我们要知道不抛出异常不代表可以正常遍历，抛出异常只是更加友好的提示。

我们看一下这个题目：我想把下边的集合中的lucy全部删除?

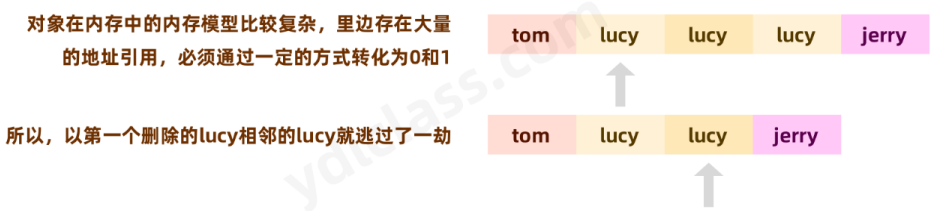
```
public void add() {
    List<String> names = new ArrayList<>();
    names.add("tom");
    names.add("lucy");
    names.add("lucy");
    names.add("lucy");
    names.add("jerry");
}
```

（1）for循环

首先我们想到的就是以下使用for循环，进行判断和删除：

```
public void testDelByFor(){
    for (int i = 0; i < names.size(); i++) {
        if("lucy".equals(names.get(i))){
            names.remove(i);
        }
    }
    System.out.println(names);
}

结果：
[tom, lucy, jerry]
```

我们发现并没有删除干净，中间的lucy好像被遗忘了。



合适的解决方式有两种：

第一种：回调指针

```
for (int i = 0; i < names.size(); i++) {
    if("lucy".equals(names.get(i))){
        names.remove(i);
        // 回调指针：
        i--;
    }
}
System.out.println(names);


结果：
[tom, jerry]
```

第二种：逆序遍历

```
for (int i = names.size()-1; i > 0; i--) {
    if("lucy".equals(names.get(i))){
        names.remove(i);
    }
}
System.out.println(names);

结果：
[tom, jerry]
```

结合指针移动和链表的位置移动过程，我们思考一下，为什么这两种方案，可以解决遍历中删除的问题。

（3）使用迭代器删除元素

事实上，使用迭代器的api进行删除是没有问题的：

```
public static void main(String[] args){
    Iterator<String> iterator = names.iterator();
    while (iterator.hasNext()){
        // 记住next(),只能调用一次，因为每次调用都会选择下一个
        String name = iterator.next();
        if("lucy".equals(name)){
            iterator.remove();
        }
    }
    System.out.println(names);
}
```

根本原因是迭代器的remove使用的指针回拨解决了这个问题。

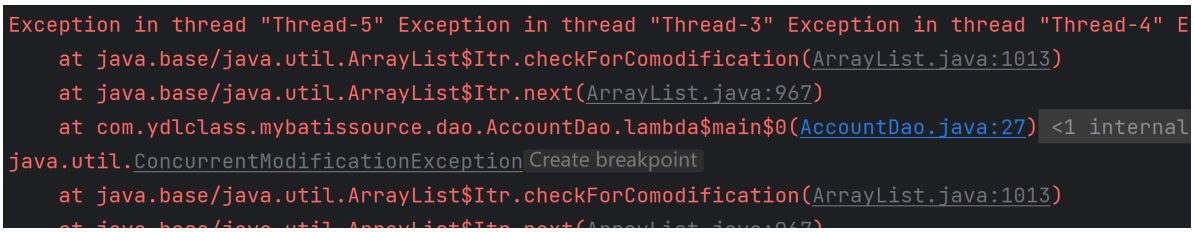
4、并发迭代时删除

事实上，我们上边代码中看到的异常，也叫并发修改异常，也就意味着这个异常的产生本质上应该是在并发修改一个集合时产生的，我们将代码改为如下的内容：

```
public static void main(String[] args) {
    // 创建一个ArrayList对象
    ArrayList<String> names = new ArrayList<>();

    // 向ArrayList中添加元素
    names.add("张三");
    names.add("李四");
    names.add("王五");

    for (int i = 0; i < 20; i++) {
        new Thread(() -> {
            Iterator<String> iterator = names.iterator();
            while (iterator.hasNext()){
                // 记住next(),只能调用一次，因为每次调用都会选择下一个
                String name = iterator.next();
                if("李四".equals(name)){
                    iterator.remove();
                }
                System.out.println(names);
            }
        }).start();
    }
```

```
}
```

事实上，我们确实再次看到了这个异常：



他家可以想一下，我正在迭代一个集合，此时有人却修改了这个集合，你说你气不气，为了防止这样的事情发生引入了这个异常，我们可以并发读，单不允许并发改。

那java中，集合是确定是并发修改呢?

在 ArrayList 中定义一个成员变量 modCount，记录集合被修改的次数，集合每调用一次增加或删除元素的函数，就会给 modCount 加 1。

当通过调用集合上的 iterator() 函数来创建迭代器的时候，我们把 modCount 值传递给迭代器的 expectedModCount 成员变量，之后每次调用迭代器上的 hasNext()、next()、currentItem() 函数，我们都会检查集合上的 modCount 是否等于 expectedModCount，也就是看，在创建完迭代器之后，modCount 是否改变过。

如果两个值不相同，那就说明集合存储的元素已经改变了，要么增加了元素，要么删除了元素，之前创建的迭代器已经不能正确运行了，再继续使用就会产生不可预期的结果，所以我们选择 **fail-fast** 解决方式，抛出运行时异常，结束掉程序，让程序员尽快修复这个因为不正确使用迭代器而产生的 bug。

5、利用快照解决并发问题

在这个示例中，我们为迭代器添加了 `remove()` 方法，允许在迭代过程中删除元素。注意，这里的删除操作会影响原始数据结构，但不会影响快照。

（1）修改迭代器接口，添加 `remove()` 方法：

```
public interface SnapshotIterator {
    boolean hasNext();
    Integer next();
    void remove();
}
```

（2）修改支持快照功能的迭代器类，每次构建迭代器，需要将数据拷贝，实现 `remove()` 方法：

```
public class SnapshotIteratorImpl implements SnapshotIterator {
    private DataStructure dataStructure;
    private List<Integer> snapshot;
    private int index;

    // 构造方法，创建一个当前数据结构的快照
    public SnapshotIteratorImpl(DataStructure dataStructure) {
        this.dataStructure = dataStructure;
        this.snapshot = new ArrayList<>(dataStructure.size());
        for (int i = 0; i < dataStructure.size(); i++) {
            snapshot.add(dataStructure.get(i));
        }
        this.index = 0;
    }

    // 判断是否有下一个元素
    @Override
    public boolean hasNext() {
        return index < snapshot.size();
    }

    // 获取下一个元素
    @Override
    public Integer next() {
        if (hasNext()) {
            return snapshot.get(index++);
        }
        return null;
    }

    // 删除当前元素
    @Override
    public void remove() {
        if (index > 0) {
            Integer itemToRemove = snapshot.get(index--);
            dataStructure.removeItem(itemToRemove);
        }
    }
}
```

（2）修改 `DataStructure` 类，添加 `removeItem()` 方法：

```
// 自定义一个数据结构类
public class DataStructure {
    private List<Integer> items;

    public DataStructure(List<Integer> items) {
        this.items = new ArrayList<>(items);
    }

    public Integer get(int index) {
        return items.get(index);
    }

    public int size() {
        return items.size();
    }

    public void removeItem(Integer item) {
        items.remove(item);
    }
}
```

（3）使用示例：

```
public class Main {
    public static void main(String[] args) {
        List<Integer> items = Arrays.asList(1, 2, 3, 4, 5);
        DataStructure dataStructure = new DataStructure(items);

        SnapshotIterator iterator = new SnapshotIteratorImpl(dataStructure);
        while (iterator.hasNext()) {
            Integer item = iterator.next();
            System.out.println(item);

            if (item % 2 == 0) {
                iterator.remove(); // 删除偶数
            }
        }

        System.out.println("After removal:");
```

```
        SnapshotIterator iterator2 = new SnapshotIteratorImpl(dataStructure);
        while (iterator2.hasNext()) {
            System.out.println(iterator2.next());
        }
    }
}
```

在这个示例中，我们在 `SnapshotIterator` 接口中添加了 `remove()` 方法，同时在 `SnapshotIteratorImpl` 类中实现了该方法。`remove()` 方法会从原始数据结构中删除元素，但不会影响快照。我们还为 `DataStructure` 类添加了一个 `removeItem()` 方法以支持删除操作。在主函数中，我们演示了在演示示例中，我们迭代数据结构并删除所有偶数元素。然后，我们创建一个新的迭代器，展示删除操作后的数据结构。

```
public class Main {
    public static void main(String[] args) {
        List<Integer> items = Arrays.asList(1, 2, 3, 4, 5);
        DataStructure dataStructure = new DataStructure(items);

        SnapshotIterator iterator = new SnapshotIteratorImpl(dataStructure);
        System.out.println("Original list:");
        while (iterator.hasNext()) {
            Integer item = iterator.next();
            System.out.println(item);

            if (item % 2 == 0) {
                iterator.remove(); // 删除偶数
            }
        }

        System.out.println("After removal:");
        SnapshotIterator iterator2 = new SnapshotIteratorImpl(dataStructure);
        while (iterator2.hasNext()) {
            System.out.println(iterator2.next());
        }
    }
}
```

运行这个程序，你将看到以下输出：

```
Original list:
After removal:
```

如上所示，程序首先迭代原始列表并打印元素。在迭代过程中，我们删除了所有偶数元素。然后我们创建了一个新的迭代器并打印出删除操作后的数据结构。可以看到，删除操作已成功删除了偶数元素。

这个实现使得迭代器可以在迭代过程中修改原始数据结构，同时保持快照功能。

事实上，以上的实现在多线程环境下可能会遇到**线程安全问题**。如果有多个线程同时访问和修改原始数据结构（`DataStructure` 类的实例），可能会导致**不可预期的行为和数据不一致**。为了确保线程安全，我们可以采用以下方法之一：

（1）使用同步关键字（`synchronized`）：

在 `DataStructure` 类的方法中，添加 `synchronized` 关键字以确保在同一时间只有一个线程可以访问这些方法。这样可以保证在多线程环境下的线程安全性。

```
public class DataStructure {
    // ...

    public synchronized Integer get(int index) {
        return items.get(index);
    }

    public synchronized int size() {
        return items.size();
    }

    public synchronized void removeItem(Integer item) {
        items.remove(item);
    }
}
```

（2）使用 `java.util.concurrent.locks` 中的锁（如 `ReentrantLock`）：

另一种确保线程安全的方法是使用显式锁。在这个例子中，我们可以使用 `ReentrantLock`。这种方法的优点是可以实现更细粒度的锁定控制，但需要手动管理锁的获取和释放。

```
public class DataStructure {
    private List<Integer> items;
```

```
    private final Lock lock = new ReentrantLock();

    // ...

    public Integer get(int index) {
        lock.lock();
        try {
            return items.get(index);
        } finally {
            lock.unlock();
        }
    }

    public int size() {
        return items.size();
    }

    public void removeItem(Integer item) {
        lock.lock();
        try {
            items.remove(item);
        } finally {
            lock.unlock();
        }
    }
}
```

请注意，这两种方法在不同程度上牺牲了性能。同步方法或锁机制可能导致线程阻塞，从而降低程序的执行速度。在选择合适的线程安全策略时，请根据具体需求和性能要求进行权衡。

6、优化

我们可不可以这样做，为每个元素保存两个时间戳，一个是添加时间戳 addTimestamp，一个是删除时间戳 delTimestamp。当元素被加入到集合中的时候，我们将 addTimestamp 设置为当前时间，将 delTimestamp 设置成最大长整型值（Long.MAX_VALUE）。当元素被删除时，我们将 delTimestamp 更新为当前时间，表示已经被删除。同时，每个迭代器也保存一个迭代器创建时间戳 snapshotTimestamp，也就是迭代器对应的快照的创建时间戳。当使用迭代器来遍历容器的时候，只有满足 addTimestamp<snapshotTimestamp<delTimestamp 的元素，才是属于这个迭代器的快照。

如果元素的 addTimestamp>snapshotTimestamp，说明元素在创建了迭代器之后才加入的，不属于这个迭代器的快照；如果元素的 delTimestamp<snapshotTimestamp，说明元素在创建迭代器之前就被删除掉了，也不属于这个迭代器的快照。

通过为每个元素和迭代器添加时间戳，可以在遍历过程中准确地识别哪些元素属于迭代器的快照。以下是基于这个思路的 Java 实现：

（1）定义一个包含元素和时间戳的类 `TimestampedItem`：

```
public class TimestampedItem<T> {
    private T item;
    private long addTimestamp;
    private long delTimestamp;

    public TimestampedItem(T item) {
        this.item = item;
        this.addTimestamp = System.currentTimeMillis();
        this.delTimestamp = Long.MAX_VALUE;
    }

    public T getItem() {
        return item;
    }

    public long getAddTimestamp() {
        return addTimestamp;
    }

    public long getDelTimestamp() {
        return delTimestamp;
    }

    public void markAsDeleted() {
        this.delTimestamp = System.currentTimeMillis();
    }
}
```

（2）修改 `DataStructure` 类，使其包含 `TimestampedItem`：

```
public class DataStructure<T> {
    private List<TimestampedItem<T>> items;

    public DataStructure(List<T> items) {
        this.items = new ArrayList<>(items.size());
        for (T item : items) {
            this.items.add(new TimestampedItem<>(item));
        }
    }

    public TimestampedItem<T> get(int index) {
        return items.get(index);
    }

    public int size() {
        return items.size();
    }

    public void removeItem(T item) {
        for (TimestampedItem<T> timestampedItem : items) {
            if (timestampedItem.getItem().equals(item)) {
                timestampedItem.markAsDeleted();
                break;
            }
        }
    }
}
```

（3）修改迭代器接口和实现类，使其适应 `TimestampedItem` 类型的元素，并添加快照时间戳：

```
public interface SnapshotIterator<T> {
    boolean hasNext();
    T next();
    void remove();
}

public class SnapshotIteratorImpl<T> implements SnapshotIterator<T> {
    private DataStructure<T> dataStructure;
    private int index;
    private long snapshotTimestamp;
```

```
    public SnapshotIteratorImpl(DataStructure<T> dataStructure) {
        this.dataStructure = dataStructure;
        this.index = 0;
        this.snapshotTimestamp = System.currentTimeMillis();
    }

    @Override
    public boolean hasNext() {
        // 这个过程会查找时间戳
        while (index < dataStructure.size()) {
            TimestampedItem<T> currentItem = dataStructure.get(index);
            if (currentItem.getAddTimestamp() < snapshotTimestamp &&
                currentItem.getDelTimestamp() > snapshotTimestamp) {
                return true;
            }
            index++;
        }
        return false;
    }

    @Override
    public T next() {
        if (hasNext()) {
            return dataStructure.get(index++).getItem();
        }
        return null;
    }

    @Override
    public void remove() {
        if (index > 0) {
            int currentIndex = index - 1;
            T itemToRemove = dataStructure.get(currentIndex).getItem();
            dataStructure.removeItem(itemToRemove);
        }
    }
}
```

（4）使用示例：

```
public class Main {
    public static void main(String[] args) {
        List<Integer> items = Arrays.asList(1, 2, 3, 4, 5);
        DataStructure<Integer> dataStructure = new DataStructure<>(items);
        SnapshotIterator<Integer> iterator = new SnapshotIteratorImpl<>
(dataStructure);
        System.out.println("Original list:");
        while (iterator.hasNext()) {
            Integer item = iterator.next();
            System.out.println(item);

            if (item % 2 == 0) {
                iterator.remove(); // 删除偶数
            }
        }

        System.out.println("After removal:");
        SnapshotIterator<Integer> iterator2 = new SnapshotIteratorImpl<>
(dataStructure);
        while (iterator2.hasNext()) {
            System.out.println(iterator2.next());
        }
    }
}
```

对于时间戳来说，Java中的`System.currentTimeMillis()`方法在多线程环境下是线程安全的。然而，仅仅依赖时间戳来处理并发问题还是不够的。考虑以下场景：

1. 当两个线程同时尝试删除同一个元素时，可能会出现竞态条件。虽然它们都会尝试将`delTimestamp`更新为当前时间，但这个过程不是原子性的，可能导致不一致的结果。
2. 当一个线程在遍历数据结构时，另一个线程可能正在修改数据结构（例如，添加或删除元素）。这种情况下，依赖于时间戳的迭代器可能无法确保数据的一致性。在某些情况下，这可能导致不稳定的迭代器行为。

因此，尽管时间戳本身是线程安全的，但在实际操作中，还需要额外的同步机制来确保整个过程的线程安全。根据您的需求和性能要求，您可以使用前面提到的方法，例如同步关键字（`synchronized`）或显式锁（如`ReentrantLock`）。

时间戳在这个实现中的主要目的是**跟踪元素的添加和删除状态**，从而在迭代器中准确地识别哪些元素属于迭代器的快照。时间戳可以帮助我们在不复制整个数据结构的情况下，实现一种“轻量级”的快照功能。

以下是时间戳在这个实现中的主要用途：

1. 记录元素的添加时间：`addTimestamp` 用于保存元素添加到数据结构的时间。这样，在创建迭代器时，我们可以根据元素的添加时间判断它是否属于快照。
2. 记录元素的删除时间：`delTimestamp` 用于保存元素从数据结构中删除的时间。这样，当我们使用迭代器遍历数据结构时，可以根据元素的删除时间判断它是否仍属于快照。
3. 记录迭代器快照创建时间：`snapshotTimestamp` 用于保存迭代器创建时的时间戳。通过比较元素的 `addTimestamp` 和 `delTimestamp` 与迭代器的 `snapshotTimestamp`，我们可以判断元素是否属于迭代器的快照。

虽然时间戳在这个场景中提供了一种有效的方法来实现迭代器的快照功能，但它并不能解决多线程环境下的线程安全问题。为了确保线程安全，您需要使用额外的同步机制（如前面提到的同步关键字或显式锁）。总的来说，时间戳在这里主要用于实现快照功能，而非解决线程安全问题。

第七章 访问者模式

一、概述和原理

访问者设计模式（Visitor Pattern）是一种行为型设计模式，它允许你在**不修改现有类结构**的情况下，为类添加**新的操作**。这种模式可以实现良好的**解耦和扩展性**，尤其适用于在**现有类层次结构中添加新功能**的情况。

访问者模式主要包含以下角色：

1. 访问者（Visitor）：定义一个访问具体元素的接口，为每种具体元素类型声明一个访问操作。
2. 具体访问者（ConcreteVisitor）：实现访问者接口，为每种具体元素提供具体的访问操作实现。
3. 元素（Element）：定义一个接口，声明接受访问者的方法。
4. 具体元素（ConcreteElement）：实现元素接口，提供接受访问者的具体实现。
5. 对象结构（ObjectStructure）：包含一个元素集合，提供一个方法以遍历这些元素并让访问者访问它们。

以下是一个简单的访问者模式示例：

假设我们有一个表示计算机组件的类层次结构（如 CPU、内存和硬盘等），我们需要为这些组件实现一个功能，比如展示它们的详细信息。使用访问者模式，我们可以将**【展示详细信息】的功能与【组件类】分离**，从而实现解耦和扩展性。

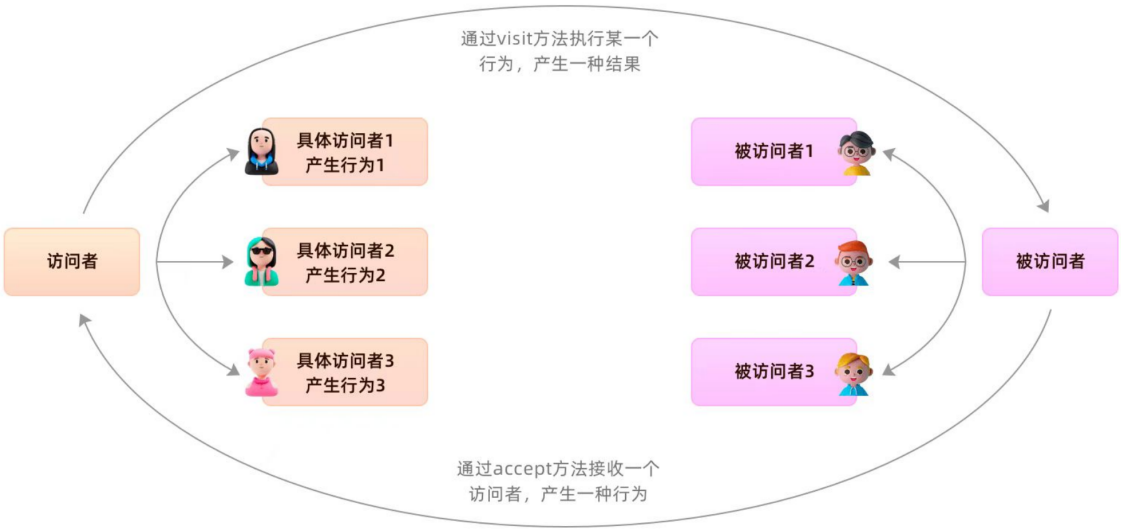
1、不同的元素（被访问的对象）可以接收不同的访问者。

2、不同的访问者会对不同的被访问者产生不同的行为。

3、如果想要扩展，则独立重新实现访问者接口，产生一个新的具体访问者就可以了。

4、他实际解耦的是【被访问者】和【对被访问者的操作】。

简单理解就是**不同的访问者**，到了**同一个被访问对象**的家里会干不同的事。这个【事】就是行为，通过访问者模式，我们可以将**行为和对象分离解耦**，如下图。



下边是一个使用了访问者模式的案例：

```
// 访问者接口
interface ComputerPartVisitor {
    // 访问 Computer 对象
    void visit(Computer computer);
    // 访问 Mouse 对象
    void visit(Mouse mouse);
    // 访问 Keyboard 对象
    void visit(Keyboard keyboard);
}

// 具体访问者
```

```
class ComputerPartDisplayVisitor implements ComputerPartVisitor {
    // 访问 Computer 对象
    @Override
    public void visit(Computer computer) {
        System.out.println("Displaying Computer.");
    }

    // 访问 Mouse 对象
    @Override
    public void visit(Mouse mouse) {
        System.out.println("Displaying Mouse.");
    }

    // 访问 Keyboard 对象
    @Override
    public void visit(Keyboard keyboard) {
        System.out.println("Displaying Keyboard.");
    }
}

// 元素接口
interface ComputerPart {
    // 接受访问者的访问
    void accept(ComputerPartVisitor computerPartVisitor);
}

// 具体元素
class Computer implements ComputerPart {
    // 子元素数组
    ComputerPart[] parts;

    public Computer() {
        // 初始化子元素数组
        parts = new ComputerPart[]{new Mouse(), new Keyboard()};
    }

    // 接受访问者的访问
    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        // 遍历所有子元素并接受访问者的访问
        for (int i = 0; i < parts.length; i++) {
            parts[i].accept(computerPartVisitor);
        }
```

```
        // 访问 Computer 对象本身
        computerPartVisitor.visit(this);
    }
}

// 具体元素：鼠标
class Mouse implements ComputerPart {
    // 接受访问者的访问
    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        // 访问 Mouse 对象
        computerPartVisitor.visit(this);
    }
}

// 具体元素：键盘
class Keyboard implements ComputerPart {
    // 接受访问者的访问
    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        // 访问 Keyboard 对象
        computerPartVisitor.visit(this);
    }
}

// 客户端代码
public class VisitorPatternDemo {
    public static void main(String[] args) {
        // 创建一个 Computer 对象
        ComputerPart computer = new Computer();
        // 创建一个具体访问者
        ComputerPartVisitor visitor = new ComputerPartDisplayVisitor();
        // 让 Computer 对象接受访问者的访问
        computer.accept(visitor);
    }
}
```

在这个示例中，我们定义了一个**表示计算机组件的类层次结构**，包括 `Computer`、`Mouse` 和 `Keyboard`。这些类实现了 `ComputerPart` 接口，该接口声明了一个接受访问者的方法。我们还定义了一个 `ComputerPartVisitor` 接口，用于访问这些计算机组件，并为每种组件类型声明了一个访问操作。

`ComputerPartDisplayVisitor` 类实现了 `ComputerPartVisitor` 接口，为每种计算机组件**提供了展示详细信息的功能。**在客户端代码中，我们创建了一个 `Computer` 对象和一个 `ComputerPartDisplayVisitor` 对象。当我们调用 `computer.accept()` 方法时，计算机的所有组件都会被访问者访问，并显示相应的详细信息。

这个示例展示了如何使用访问者模式将**功能与类结构分离，实现解耦和扩展性。**如果我们需要为计算机组件添加新功能，只需**创建一个新的访问者类，而无需修改现有的组件类。**这使得在不影响现有代码的情况下，为系统添加新功能变得容易。

```
// 添加一个更新计算机部件的访问者实现
class ComputerPartUpdateVisitorImpl implements ComputerPartVisitor {
    // 访问 Computer 对象并执行更新操作
    @Override
    public void visit(Computer computer) {
        System.out.println("Updating Computer.");
    }

    // 访问 Mouse 对象并执行更新操作
    @Override
    public void visit(Mouse mouse) {
        System.out.println("Updating Mouse.");
    }

    // 访问 Keyboard 对象并执行更新操作
    @Override
    public void visit(Keyboard keyboard) {
        System.out.println("Updating Keyboard.");
    }
}

// 客户端代码，几乎不用任何修改
public class VisitorPatternDemo {
    public static void main(String[] args) {
        // 创建一个 Computer 对象
        ComputerPart computer = new Computer();
        // 创建一个具体访问者
        ComputerPartVisitor visitor = new ComputerPartUpdateVisitorImpl();
        // 让 Computer 对象接受访问者的访问
        computer.accept(visitor);
    }
```

```
}
```

访问者模式可以算是 23 种经典设计模式中最难理解的几个之一。因为它难理解、难实现，应用它会导致代码的**可读性、可维护性变差**，所以，访问者模式在实际的软件开发中**很少被用到**，在没有特别必要的情况下，**建议你不要使用访问者模式。**

二、使用场景

1、抽象语法树

访问者模式在实际项目中的一个常见使用场景是**处理抽象语法树（AST）**。例如，在**编译器或解释器**中，我们需要处理**不同类型的语法结构**，如声明、表达式、循环等。使用访问者模式，我们可以将处理这些**结构的功能与结构类分离**，实现解耦和扩展性。

以下是一个简单的示例，展示了如何使用访问者模式处理抽象语法树：

```
// AST 节点基类
abstract class AstNode {
    // 接受访问者的方法
    abstract void accept(AstVisitor visitor);
}

// 访问者接口
interface AstVisitor {
    // 访问表达式节点的方法
    void visit(ExpressionNode node);
    // 访问数字节点的方法
    void visit(NumberNode node);
    // 访问加法节点的方法
    void visit(AdditionNode node);
    // 省略其他节点
}

// 数字节点，表示一个整数值
class NumberNode extends AstNode {
    int value;

    // 构造方法，接收一个整数作为值
```

```
    NumberNode(int value) {
        this.value = value;
    }

    // 实现基类的 accept 方法，接受访问者
    void accept(AstVisitor visitor) {
        visitor.visit(this);
    }
}

// 加法节点，表示两个子节点的相加
class AdditionNode extends AstNode {
    AstNode left;
    AstNode right;

    // 构造方法，接收两个子节点
    AdditionNode(AstNode left, AstNode right) {
        this.left = left;
        this.right = right;
    }

    // 实现基类的 accept 方法，接受访问者
    void accept(AstVisitor visitor) {
        visitor.visit(this);
    }
}

// 表达式节点，包含一个子节点
class ExpressionNode extends AstNode {
    AstNode node;

    // 构造方法，接收一个子节点
    ExpressionNode(AstNode node) {
        this.node = node;
    }

    // 实现基类的 accept 方法，接受访问者
    void accept(AstVisitor visitor) {
        visitor.visit(this);
    }
}
```

现在，我们可以创建一个实现了 `AstVisitor` 接口的类，用于遍历 AST 并计算其结果：

```
class AstEvaluator implements AstVisitor {
    int result;

    AstEvaluator() {
        result = 0;
    }

    void visit(ExpressionNode node) {
        node.node.accept(this);
    }

    void visit(NumberNode node) {
        result = node.value;
    }

    void visit(AdditionNode node) {
        node.left.accept(this);
        int leftValue = result;
        node.right.accept(this);
        int rightValue = result;
        result = leftValue + rightValue;
    }
}
```

最后，我们可以使用这个访问者类计算一个简单的 AST：

```
public class Main {
    public static void main(String[] args) {
        // 创建一个简单的 AST：(2 + 3)
        AstNode ast = new ExpressionNode(
            new AdditionNode(
                new NumberNode(2),
                new NumberNode(3)
            )
        );

        // 创建一个访问者实例
        AstEvaluator evaluator = new AstEvaluator();
```

```
        // 使用访问者计算 AST 的结果
        ast.accept(evaluator);

        // 输出计算结果
        System.out.println("AST 的结果是: " + evaluator.result);
    }
}
```

这个示例将输出 `AST 的结果是: 5`。

2、报表生成

假设我们有一个表示销售订单的类层次结构，包括不同类型的订单。我们需要根据不同的需求生成各种报表，如销售报表、库存报表等。

首先，定义订单类层次结构：

```
// 订单元素接口
interface OrderElement {
    void accept(OrderVisitor visitor);
}

// 零售订单
class RetailOrder implements OrderElement {
    private String id;
    private double amount;

    public RetailOrder(String id, double amount) {
        this.id = id;
        this.amount = amount;
    }

    public String getId() {
        return id;
    }

    public double getAmount() {
        return amount;
    }

    @Override
    public void accept(OrderVisitor visitor) {
```

```
        visitor.visit(this);
    }
}

// 批发订单
class WholesaleOrder implements OrderElement {
    private String id;
    private double amount;

    public WholesaleOrder(String id, double amount) {
        this.id = id;
        this.amount = amount;
    }

    public String getId() {
        return id;
    }

    public double getAmount() {
        return amount;
    }

    @Override
    public void accept(OrderVisitor visitor) {
        visitor.visit(this);
    }
}
```

接着，定义访问者接口和具体的访问者类：

```
// 订单访问者接口
interface OrderVisitor {
    void visit(RetailOrder order);
    void visit(WholesaleOrder order);
}

// 销售报表访问者
class SalesReportVisitor implements OrderVisitor {
    private double totalSales = 0;

    @Override
    public void visit(RetailOrder order) {
```

```
        System.out.println("销售报表 - 零售订单: " + order.getId() + "，金额: " +
order.getAmount());
        totalSales += order.getAmount();
    }

    @Override
    public void visit(WholesaleOrder order) {
        System.out.println("销售报表 - 批发订单: " + order.getId() + "，金额: " +
order.getAmount());
        totalSales += order.getAmount();
    }

    public double getTotalSales() {
        return totalSales;
    }
}
```

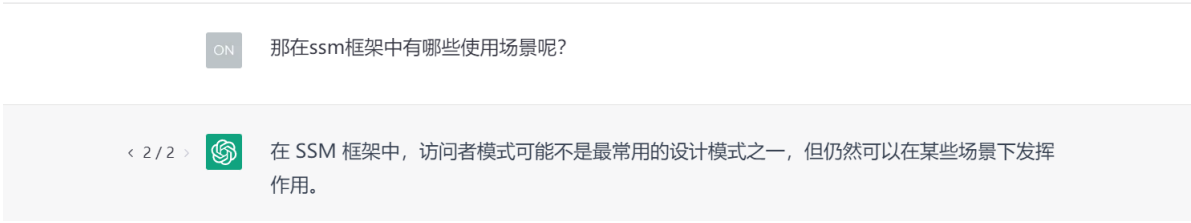
现在，我们可以使用访问者模式生成报表：

```
public class ReportGenerationDemo {
    public static void main(String[] args) {
        List<OrderElement> orders = new ArrayList<>();
        orders.add(new RetailOrder("R1", 100.0));
        orders.add(new RetailOrder("R2", 200.0));
        orders.add(new WholesaleOrder("W1", 1500.0));

        SalesReportVisitor salesReportVisitor = new SalesReportVisitor();
        for (OrderElement order : orders) {
            order.accept(salesReportVisitor);
        }

        System.out.println("总销售额: " + salesReportVisitor.getTotalSales());
    }
}
```

在这个示例中，我们定义了表示订单的类层次结构，包括 `RetailOrder`（零售订单）和 `WholesaleOrder`（批发订单）。这些类实现了 `OrderElement` 接口，该接口声明了一个接受访问者的方法。

三、源码应用

事实上，在源码中我们也很少见到访问者设计模式的具体使用。



1、ASM字节码编辑

但是我们可以看看，ASM库是如何使用访问者设计模式修改字节码文件的：

在这个例子中，我们将展示如何使用 ASM 库和访问者设计模式来修改一个 Java 类的字节码。我们将对一个简单的 Java 类进行修改，为其添加一个方法。以下是我们的示例类：

```
// 示例类
public class ExampleClass {
    public void hello() {
        System.out.println("Hello, world!");
    }
}
```

要使用 ASM 修改这个类，我们需要首先定义一个 ClassVisitor，用于在遍历类结构时接收通知。我们将在 visitEnd 方法中添加一个新的方法。

```
// 自定义 ClassVisitor 类
class AddMethodClassVisitor extends ClassVisitor {
    AddMethodClassVisitor(ClassVisitor classVisitor) {
        super(Opcodes.ASM9, classVisitor);
    }

    // 类结构访问完成时调用
    @Override
    public void visitEnd() {
        // 添加一个新的方法：public void newMethod()
        MethodVisitor mv = cv.visitMethod(Opcodes.ACC_PUBLIC, "newMethod", "
()V", null, null);
        if (mv != null) {
            // 方法开始
            mv.visitCode();
```

```
            // 加载 this 到操作数栈
            mv.visitVarInsn(Opcodes.ALOAD, 0);
            // 调用 System.out.println 方法
            mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "out",
"Ljava/io/PrintStream;");
            mv.visitLdcInsn("This is a new method!");
            mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/PrintStream",
"println", "(Ljava/lang/String;)V", false);
            // 方法返回
            mv.visitInsn(Opcodes.RETURN);
            // 设置方法的最大栈大小和最大局部变量数
            mv.visitMaxs(2, 1);
            // 方法结束
            mv.visitEnd();
        }

        // 结束类访问
        super.visitEnd();
    }
}
```

现在，我们需要使用 ASM 的 ClassReader 和 ClassWriter 类读取和写入字节码，同时将自定义的 AddMethodClassVisitor 应用于类：

```
public class Main {
    public static void main(String[] args) {
        try {
            // 从当前类加载器中获取 ExampleClass 的字节码
            InputStream inputStream =
Main.class.getClassLoader().getResourceAsStream("ExampleClass.class");
            if (inputStream == null) {
                throw new IOException("ExampleClass not found");
            }

            // 使用 ASM 的 ClassReader 读取字节码
            ClassReader classReader = new ClassReader(inputStream);
            // 使用 ASM 的 ClassWriter 生成新的字节码
            ClassWriter classWriter = new
ClassWriter(ClassWriter.COMPUTE_FRAMES);
            // 创建自定义的 ClassVisitor 实例
            AddMethodClassVisitor addMethodClassVisitor = new
AddMethodClassVisitor(classWriter);
```

```
        // 使用自定义的 ClassVisitor 遍历字节码
        classReader.accept(addMethodClassVisitor, 0);

        // 将修改后的字节码写入到新的文件中
        byte[] newBytecode = classWriter.toByteArray();
        try (FileOutputStream fos = new
FileOutputStream("ExampleClassModified.class")) {
            fos.write(newBytecode); }

        System.out.println("ExampleClassModified.class 已生成");

        // 使用自定义的类加载器来加载修改后的类
        CustomClassLoader customClassLoader = new CustomClassLoader();
        Class<?> modifiedClass = customClassLoader.defineClass("ExampleClass",
newBytecode);

        // 创建修改后类的实例并调用原有的 hello 方法
        Object instance = modifiedClass.newInstance();
        modifiedClass.getMethod("hello").invoke(instance);

        // 调用新增的 newMethod 方法
        modifiedClass.getMethod("newMethod").invoke(instance);

    } catch (IOException | ReflectiveOperationException e) {
        e.printStackTrace();
    }
  }

  // 自定义类加载器，用于加载修改后的类
  static class CustomClassLoader extends ClassLoader {
    public Class<?> defineClass(String name, byte[] b) {
        return defineClass(name, b, 0, b.length);
    }
  }
}
```

在这个例子中，我们从当前类加载器中读取了 `ExampleClass` 的字节码，然后使用 ASM 的 `ClassReader` 和自定义的 `AddMethodClassVisitor` 对其进行修改。修改后的字节码被写入到一个名为 `ExampleClassModified.class` 的新文件中。接下来，我们使用自定义的 `CustomClassLoader` 加载修改后的类，并创建一个新的实例。我们分别调用了原有的 `hello` 方法和新增的 `newMethod` 方法。输出应该如下：

```
Hello, world!
This is a new method!
```

这个例子展示了如何使用 ASM 和访问者设计模式在 Java 字节码级别修改类的结构。在这种情况下，我们向现有类中添加了一个新的方法。

第八章 备忘录模式

今天，我们学习另外一种行为型模式，备忘录模式。这个模式理解、掌握起来不难，代码实现比较灵活，应用场景也比较明确和有限，主要是用来防丢失、撤销、恢复等。所以，相对于上两节课，今天的内容学起来相对会比较轻松些。

话不多说，让我们正式开始今天的学习吧！

一、原理与实现

备忘录模式，也叫快照（Snapshot）模式，英文翻译是 Memento Design Pattern。在 GoF 的《设计模式》一书中，备忘录模式是这么定义的：

> Captures and externalizes an object's internal state so that it can be restored later, all without violating encapsulation.

翻译成中文就是：在不违背封装原则的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便之后恢复对象为先前的状态。

在我看来，这个模式的定义主要表达了两部分内容。一部分是，存储副本以便后期恢复。这一部分很好理解。另一部分是，要在不违背封装原则的前提下，进行对象的备份和恢复实现撤销（Undo）和恢复（Redo）操作。这部分不太好理解。接下来，我就结合一个例子来解释一下，特别带你搞清楚这两个问题：

- 为什么存储和恢复副本会违背封装原则？
- 备忘录模式是如何做到不违背封装原则的？

备忘录设计模式涉及三个主要组件：

1. 发起人（Originator）：这是我们希望保存状态的对象。它可以创建一个备忘录（Memento）对象来保存其当前状态，并可以使用备忘录来恢复先前的状态。
2. 备忘录（Memento）：这个对象存储了发起人的内部状态。它应该有一个足够的状态，以便发起人可以恢复到之前的状态。备忘录类通常具有私有的访问权

限，仅发起人可以访问其内部状态。
3. 负责人（Caretaker）：这个对象负责保存和恢复备忘录。它不应该对备忘录的内容进行任何操作，只是将备忘录传递给发起人。

下面是一个使用 Java 实现的简单备忘录设计模式的示例：

```
// 发起人类
class Originator {
    private String state;

    public void setState(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }

    // 创建备忘录
    public Memento saveStateToMemento() {
        return new Memento(state);
    }

    // 恢复状态
    public void getStateFromMemento(Memento memento) {
        state = memento.getState();
    }
}

// 备忘录类
class Memento {
    private final String state;

    public Memento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }
}

// 负责人类
```

```
class Caretaker {
    private final List<Memento> mementoList = new ArrayList<>();

    public void add(Memento state) {
        mementoList.add(state);
    }

    public Memento get(int index) {
        return mementoList.get(index);
    }
}

public class Main {
    public static void main(String[] args) {
        Originator originator = new Originator();
        Caretaker caretaker = new Caretaker();

        // 设置状态并保存到备忘录
        originator.setState("State1");
        caretaker.add(originator.saveStateToMemento());

        originator.setState("State2");
        caretaker.add(originator.saveStateToMemento());

        // 从备忘录中恢复状态
        originator.getStateFromMemento(caretaker.get(0));
        System.out.println("恢复的状态: " + originator.getState()); // 输出：恢复的状态: State1

        originator.getStateFromMemento(caretaker.get(1));
        System.out.println("恢复的状态: " + originator.getState()); // 输出：恢复的状态: State2
    }
}
```

这个例子演示了如何使用备忘录设计模式保存和恢复对象的状态。在这个例子中，我们创建了一个 `Originator` 类来存储状态并创建备忘录对象。`Memento` 类用于保存 `Originator` 的状态。`Caretaker` 类负责保存和恢复备忘录对象。

这个例子展示了如何在需要时从备忘录中恢复对象的状态。这种设计模式在撤销和重做操作时非常有用，因为它允许我们恢复到先前的状态，而不需要修改或者重新实现对象的行为。

下边我们举一个小例子：

假设有这样一道面试题，希望你编写一个小程序，可以接收命令行的输入。用户输入文本时，程序将其追加存储在内存文本中；用户输入“:list”，程序在命令行中输出内存文本的内容；用户输入“:undo”，程序会撤销上一次输入的文本，也就是从内存文本中将上次输入的文本删除掉。

我举了个小例子来解释一下这个需求，如下所示：

```
>hello
>:list
hello
>world
>:list
helloworld
>:undo
>:list
hello
```

怎么来编程实现呢？你可以打开 IDE 自己先试着编写一下，然后再看我下面的讲解。整体上来讲，这个小程序实现起来并不复杂。我写了一种实现思路，如下所示：

```
// 文本缓冲区类，用于存储文本并管理撤销操作
class TextBuffer {
    private StringBuilder text;

    public TextBuffer() {
        text = new StringBuilder();
    }

    // 向文本缓冲区追加新文本
    public void append(String newText) {
        text.append(newText);
    }

    // 获取文本缓冲区的内容
    public String getText() {
        return text.toString();
    }

    // 从历史记录恢复文本
    public void restoreText(String previousText) {
```

```
            text = new StringBuilder(previousText);
        }
    }

    public class Main {
        public static void main(String[] args) {
            TextBuffer textBuffer = new TextBuffer();
            // 我们搞一个list，当做栈来使用记录历史记录
            List<String> history = new ArrayList<>();
            Scanner scanner = new Scanner(System.in);

            while (true) {
                System.out.print("请输入命令： ");
                String input = scanner.nextLine();

                // 列出当前文本缓冲区的内容
                if (":list".equals(input)) {
                    System.out.println(textBuffer.getText());
                }
                // 撤销上一次输入的文本
                else if (":undo".equals(input)) {
                    if (!history.isEmpty()) {
                        history.remove(history.size() - 1);
                        textBuffer.restoreText(history.isEmpty() ? "" : history.get(history.size()
- 1));
                    } else {
                        System.out.println("无法撤销，没有更早的历史记录。");
                    }
                }
                // 将用户输入的文本追加到文本缓冲区
                else {
                    history.add(textBuffer.getText());
                    textBuffer.append(input);
                }
            }
        }
    }
```

实际上，**备忘录模式的实现很灵活**，也没有很固定的实现方式，在不同的业务需求、不同编程语言下，代码实现可能都不大一样。上面的代码基本上已经实现了最基本的备忘录的功能。List集合中存储的历史记录本质就是一个个的备忘录。

但是，如果我们深究一下的话，还有一些问题要解决：

- 第一，使用 List<String> 记录历史，扩展性很差，一旦需要记录更多内容，则必须修改原始代码。
- 第二，集合中的备忘信息不具备封装性，有被篡改的风险。

针对以上问题，我们对代码做两点修改。

其一，定义一个独立的类（Memento类）来表示备忘录，而不是使用 String 类或者其他。这个类只暴露 get() 方法，没有 set() 等任何修改内部状态的方法。

其二，在 TextBuffer类中，我们把 setText() 方法重命名为 restoreFromMemento() 方法，用意更加明确，只用来恢复对象。

按照这个思路，我们对代码进行重构。重构之后的代码如下所示：

```
class TextBuffer {
  private StringBuilder text;

  public TextBuffer() {
    text = new StringBuilder();
  }

  public void append(String newText) {
    text.append(newText);
  }

  public String getText() {
    return text.toString();
  }

  public Memento saveToMemento() {
    return new Memento(text.toString());
  }

  public void restoreFromMemento(Memento memento) {
    text = new StringBuilder(memento.getSavedText());
  }

  // 备忘录
  static class Memento {
    private final String savedText;

    public Memento(String savedText) {
```

```
            this.savedText = savedText;
        }

        public String getSavedText() {
            return savedText;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        TextBuffer textBuffer = new TextBuffer();
        List<TextBuffer.Memento> history = new ArrayList<>();
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.print("请输入命令： ");
            String input = scanner.nextLine();

            if (":list".equals(input)) {
                System.out.println(textBuffer.getText());
            } else if (":undo".equals(input)) {
                if (!history.isEmpty()) {
                    history.remove(history.size() - 1);
                    textBuffer.restoreFromMemento(history.isEmpty() ? new
TextBuffer.Memento("") : history.get(history.size() - 1));
                } else {
                    System.out.println("无法撤销，没有更早的历史记录。");
                }
            } else {
                history.add(textBuffer.saveToMemento());
                textBuffer.append(input);
            }
        }
    }
}
```

实际上，上面的代码实现就是典型的备忘录模式的代码实现，也是很多书籍（包括 GoF 的《设计模式》）中给出的实现方法。

事实上，很多场景下我们使用第一种方式就能满足绝大部分需求，我们要知道，设计模式不是万金油有时候简单的需求使用简单的编码就可以实现，很多时候不必要为了扩展而扩展，为了设计而设计，编码简单也是很重要的。

二、备份优化

前面我们只是简单介绍了备忘录模式的原理和经典实现，现在我们再继续深挖一下。如果要备份的对象数据比较大，备份频率又比较高，那快照占用的内存会比较大，备份和恢复的耗时会比较长。这个问题该如何解决呢？

不同的应用场景下有不同的解决方法。比如，我们前面举的那个例子，应用场景是利用备忘录来实现撤销操作，而且仅仅支持顺序撤销，也就是说，每次操作只能撤销上一次的输入，不能跳过上次输入撤销之前的输入。在具有这样特点的应用场景下，为了节省内存，我们不需要在快照中存储完整的文本，只需要记录少许信息，比如在获取快照当下的文本长度，用这个值结合 InputText 类对象存储的文本来做撤销操作。

我们再举一个例子。假设每当有数据改动，我们都需要生成一个备份，以备之后恢复。如果需要备份的数据很大，这样高频率的备份，不管是对存储（内存或者硬盘）的消耗，还是对时间的消耗，都可能是无法接受的。想要解决这个问题，我们一般会采用**“低频率全量备份”和“高频率增量备份”**相结合的方法。

全量备份就不用讲了，它跟我们上面的例子类似，就是把所有的数据“拍个快照”保存下来。所谓“增量备份”，指的是记录每次操作或数据变动。

当我们需要恢复到某一时间点的备份的时候，如果这一时间点有做全量备份，我们直接拿来恢复就可以了。如果这一时间点没有对应的全量备份，我们就先找到最近的一次全量备份，然后用它来恢复，之后执行此次全量备份跟这一时间点之间的所有增量备份，也就是对应的操作或者数据变动。这样就能减少全量备份的数量和频率，减少对时间、内存的消耗。

三、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。

备忘录模式也叫快照模式，具体来说，就是在不违背封装原则的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便之后恢复对象为先前的状态。这个模式的定义表达了两部分内容：一部分是，存储副本以便后期恢复；另一部分是，要在不违背封装原则的前提下，进行对象的备份和恢复。

备忘录模式的应用场景也比较明确和有限，主要是用来防丢失、撤销、恢复等。它跟平时我们常说的“备份”很相似。两者的主要区别在于，备忘录模式更侧重于代码的设计和实现，备份更侧重架构设计或产品设计。

对于大对象的备份来说，备份占用的存储空间会比较大，备份和恢复的耗时会比较长。针对这个问题，不同的业务场景有不同的处理方式。比如，只备份必要的恢复信息，结合最新的数据来恢复；再比如，全量备份和增量备份相结合，低频全量备份，高频增量备份，两者结合来做恢复。

四、课堂讨论

今天我们讲到，备份在架构或产品设计中比较常见，比如，重启 Chrome 可以选择恢复之前打开的页面，你还能想到其他类似的应用场景吗？

第九章 命令模式

一、概述和原理

1、概述

设计模式模块已经接近尾声了，现在我们只剩下 3 个模式还没有学习，它们分别是：命令模式、解释器模式、中介模式。这 3 个模式**使用频率低、理解难度大**，只在**非常特定的应用场景下才会用到**，所以，不是我们学习的重点，你只需要稍微了解，见了能认识就可以了。

今天呢，我们来学习其中的命令模式。在学习这个模式的过程中，你可能会遇到的最大的疑惑是，感觉命令模式没啥用，是一种过度设计，有更加简单的设计思路可以替代。所以，我今天讲解的重点是这个模式的设计意图，带你搞清楚到底什么情况下才真正需要使用它。

命令模式的英文翻译是 Command Design Pattern。在 GoF 的《设计模式》一书中，它是这么定义的：

> The command pattern encapsulates a request as an object, thereby letting us parameterize other objects with different requests, queue or log requests, and support undoable operations.

命令模式（Command Pattern）是一种行为设计模式，它将**请求封装为一个对象**，从而使得**调用请求的客户端**与**处理请求的服务端**解耦。这使得我们可以将具体的请求、调用者和接收者进行更灵活的组合。命令模式常用于实现任务队列、历史记录（如撤销、重做操作）等场景。

命令模式的主要组成部分有以下几个：

1. 命令接口（Command）：定义了一个执行操作的接口，通常包含一个名为 execute() 的方法。
2. 具体命令（ConcreteCommand）：实现命令接口的具体类，包含一个接收者（Receiver）对象的引用，并在 execute() 方法中调用接收者的相应操作。
3. 调用者（Invoker）：负责调用命令对象的 execute() 方法。调用者并不需要了解命令是如何执行的，只需知道命令接口即可。
4. 接收者（Receiver）：负责执行与命令相关的具体操作。接收者对象通常是一个特定的业务对象。

命令模式的优点有：

- 降低系统的耦合度，使得调用者和接收者之间的依赖关系更加清晰。
- 可以对请求进行排队、记录日志、撤销和重做等操作。
- 可以将复杂的业务逻辑分解为较小的、更易于管理的部分，提高代码的可维护性。

下面是一个简单的 Java 代码示例，展示了命令模式的使用：

```
// 命令接口
interface Command {
    void execute();
}

// 具体命令类
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}

// 接收者
```

```
class Light {
    public void turnOn() {
        System.out.println("Light is on");
    }

    public void turnOff() {
        System.out.println("Light is off");
    }
}

// 请求者
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}

public class Main {
    public static void main(String[] args) {
        // 创建一个灯
        Light light = new Light();
        // 创建一个开灯命令
        Command lightOnCommand = new LightOnCommand(light);

        RemoteControl remoteControl = new RemoteControl();
        remoteControl.setCommand(lightOnCommand);
        remoteControl.pressButton();
    }
}
```

在这个示例中，我们创建了一个简单的家庭自动化系统，可以通过遥控器控制灯光的开关。`Light` 类是接收者，`LightOnCommand` 类是具体命令，`RemoteControl` 类是请求者。我们将命令对象设置到遥控器中，然后按下遥控器的按钮来执行命令。这样，遥控器可以控制灯光，而不需要知道如何实现灯光的开关操作。

使用命令模式有以下好处：

1. 解耦：命令模式将请求发送者（Invoker）与请求接收者（Receiver）解耦，请求发送者不需要知道接收者的具体实现，只需知道如何发送请求。这样可以降低系统各部分之间的耦合度，使系统更易于维护和扩展。
2. 可扩展性：新增命令时，只需实现一个新的具体命令类，而不需要修改请求发送者或请求接收者的代码。这使得系统更容易扩展，符合开闭原则。
3. 灵活性：命令模式允许在运行时动态地更改命令。例如，在上面的示例中，我们可以在运行时为遥控器设置不同的命令，从而控制不同的设备。
4. 可以实现宏命令：命令模式可以方便地实现宏命令（Macro Command），即一组命令的组合。宏命令可以按顺序执行多个命令，甚至可以实现撤销和重做功能。
5. 命令历史和撤销功能：命令模式可以用于记录已执行的命令，从而实现命令历史、撤销和重做等功能。例如，文本编辑器可以使用命令模式记录用户的编辑操作，以便在需要时撤销或重做这些操作。

总之，命令模式通过将操作封装为一个对象，实现了对操作的解耦、记录操作历史、撤销操作等功能。这使得系统更加灵活、可扩展和易于维护。

2、宏命令

下面是一个 Java 代码示例，展示了如何使用命令模式实现宏命令。在这个示例中，我们扩展了之前的家庭自动化系统，使其可以同时控制灯光和空调。

```
// 命令接口
interface Command {
    void execute();
}

// 具体命令类：开灯
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}

// 具体命令类：关灯
```

```
class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOff();
    }
}

// 接收者：灯
class Light {
    public void turnOn() {
        System.out.println("Light is on");
    }

    public void turnOff() {
        System.out.println("Light is off");
    }
}

// 具体命令类：开空调
class AirConditionerOnCommand implements Command {
    private AirConditioner airConditioner;

    public AirConditionerOnCommand(AirConditioner airConditioner) {
        this.airConditioner = airConditioner;
    }

    public void execute() {
        airConditioner.turnOn();
    }
}

// 具体命令类：关空调
class AirConditionerOffCommand implements Command {
    private AirConditioner airConditioner;

    public AirConditionerOffCommand(AirConditioner airConditioner) {
        this.airConditioner = airConditioner;
    }
```

```
    public void execute() {
        airConditioner.turnOff();
    }
}

// 接收者：空调
class AirConditioner {
    public void turnOn() {
        System.out.println("Air conditioner is on");
    }

    public void turnOff() {
        System.out.println("Air conditioner is off");
    }
}

// 宏命令类
class MacroCommand implements Command {
    private List<Command> commands;

    public MacroCommand() {
        commands = new ArrayList<>();
    }

    public void addCommand(Command command) {
        commands.add(command);
    }

    public void execute() {
        for (Command command : commands) {
            command.execute();
        }
    }
}

// 请求者：遥控器
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }
```

```
    public void pressButton() {
        command.execute();
    }
}

// 在 main 方法中添加关闭所有设备的宏命令
public class Main {
    public static void main(String[] args) {
        // ... 前面的代码省略

        // 创建宏命令：开启所有设备
        MacroCommand turnAllOnCommand = new MacroCommand();
        turnAllOnCommand.addCommand(lightOnCommand);
        turnAllOnCommand.addCommand(airConditionerOnCommand);

        // 创建宏命令：关闭所有设备
        MacroCommand turnAllOffCommand = new MacroCommand();
        turnAllOffCommand.addCommand(lightOffCommand);
        turnAllOffCommand.addCommand(airConditionerOffCommand);

        RemoteControl remoteControl = new RemoteControl();

        // 执行宏命令：开启所有设备
        System.out.println("Turn all devices on:");
        remoteControl.setCommand(turnAllOnCommand);
        remoteControl.pressButton();

        // 执行宏命令：关闭所有设备
        System.out.println("\nTurn all devices off:");
        remoteControl.setCommand(turnAllOffCommand);
        remoteControl.pressButton();
    }
}
```

在这个示例中，我们创建了两个宏命令：`turnAllOnCommand` 和 `turnAllOffCommand`。`turnAllOnCommand` 用于开启所有设备，而 `turnAllOffCommand` 用于关闭所有设备。通过为遥控器设置不同的宏命令，我们可以一键控制家中的多个设备。

这个示例展示了如何使用命令模式实现宏命令功能。通过将多个命令组合成一个宏命令，我们可以轻松地实现批量操作。此外，宏命令还可以与其他命令模式功能（例如撤销和重做）结合使用，实现更强大的功能。

3、历史命令和撤销

以下是一个使用命令模式实现命令历史和撤销功能的 Java 代码示例。在这个示例中，我们创建了一个简单的文本编辑器，支持添加文本、删除文本、撤销和重做操作。

```
import java.util.Stack;

// 命令接口
interface Command {
    void execute();
    void undo();
}

// 具体命令类：添加文本
class AddTextCommand implements Command {
    private StringBuilder textEditor;
    private String text;

    public AddTextCommand(StringBuilder textEditor, String text) {
        this.textEditor = textEditor;
        this.text = text;
    }

    public void execute() {
        textEditor.append(text);
    }

    public void undo() {
        textEditor.delete(textEditor.length() - text.length(), textEditor.length());
    }
}

// 请求者：文本编辑器
class TextEditor {
    private StringBuilder text;
    private Stack<Command> commandHistory;

    public TextEditor() {
        text = new StringBuilder();
        commandHistory = new Stack<>();
```

```
    }

    public void addText(String newText) {
        Command command = new AddTextCommand(text, newText);
        command.execute();
        commandHistory.push(command);
    }

    public void undo() {
        if (!commandHistory.isEmpty()) {
            Command lastCommand = commandHistory.pop();
            lastCommand.undo();
        }
    }

    public void printContent() {
        System.out.println(text.toString());
    }
}

public class Main {
    public static void main(String[] args) {
        TextEditor textEditor = new TextEditor();

        // 添加文本
        textEditor.addText("Hello, ");
        textEditor.addText("world!");
        textEditor.printContent(); // 输出：Hello, world!

        // 撤销操作
        textEditor.undo();
        textEditor.printContent(); // 输出：Hello,

        // 再次添加文本
        textEditor.addText("Command Pattern!");
        textEditor.printContent(); // 输出：Hello, Command Pattern!
    }
}
```

在这个示例中，我们创建了一个简单的文本编辑器 `TextEditor`，它使用一个 `StringBuilder` 对象存储文本内容，并使用一个栈（`Stack`）存储命令历史。添加文本操作对应的具体命令类是 `AddTextCommand`，它实现了 `execute()` 和 `undo()` 方法。当执行 `addText()` 方法时，编辑器会创建一个 `AddTextCommand` 对象并执行它，然后将这个命令对象添加到命令历史栈中。当执行 `undo()` 方法时，编辑器会从命令历史栈中弹出上一个命令对象并执行它的 `undo()` 方法，从而撤销该命令。

这个示例展示了如何使用命令模式实现命令历史和撤销功能。通过维护一个命令历史栈，我们可以轻松地记录已执行的命令，并在需要时撤销这些命令。此外，这种方法还可以扩展为支持重做操作。

话不多说，让我们正式开始今天的学习吧！

二、命令模式 VS 策略模式

看了刚才的讲解，你可能会觉得，命令模式跟策略模式、工厂模式非常相似啊，那它们的区别在哪里呢？不仅如此，在留言区中我还看到有不止一个同学反映，感

命令模式和策略模式确实在某些方面相似，但它们的核心目的和应用场景是不同的。

1. 目的：
    - 命令模式（Command Pattern）的主要目的是**将操作封装成对象**，使请求者与执行者之间的耦合度降低。这样可以实现命令的存储、传递、撤销等功能。
    - 策略模式（Strategy Pattern）的主要目的是**定义一系列算法或策略**，并将它们封装起来，使它们**可以互相替换。**策略模式**让算法独立于使用它的客户端，使客户端可以在运行时选择不同的算法或策略。**
2. 应用场景：
    - 命令模式适用于需要对操作进行记录、撤销或重做的场景，或者需要将操作作为参数传递给其他对象的场景。例如，文本编辑器、数据库事务管理等场景。
    - 策略模式适用于需要在多个算法或策略之间进行选择的场景，或者需要将算法或策略独立于客户端的场景。例如，排序算法、压缩算法等场景。
3. 实现方式：
    - 命令模式通常包括命令接口（Command）和具体命令类（ConcreteCommand），以及请求者（Invoker）和接收者（Receiver）。请求者负责调用命令，接收者负责执行命令。
    - 策略模式通常包括策略接口（Strategy）和具体策略类（ConcreteStrategy），以及使用策略的上下文类（Context）。上下文类

通过策略接口调用具体策略的方法，从而实现算法的切换。

尽管命令模式和策略模式在结构上有一定的相似性，但它们解决的问题和应用场景是不同的。在实际项目中，根据需求和目标来选择合适的设计模式。

四、使用场景

命令模式在实际开发中有很多应用场景，以下是一些常见的案例：

1. 撤销和重做操作：在文本编辑器、图形编辑器等软件中，用户可能希望撤销或重做之前的操作。通过使用命令模式，我们可以将每个操作封装为一个命令对象，然后将它们存储在一个历史记录列表中。这样，撤销操作就可以通过反向执行命令列表中的命令来实现，而重做操作则可以通过正向执行命令列表来实现。
2. 菜单和工具栏按钮：在图形用户界面中，菜单项和工具栏按钮通常与特定的操作相关联。命令模式允许我们将操作封装为命令对象，这样菜单项和按钮就可以通过调用命令对象的 execute() 方法来执行相应的操作。这种方式使得菜单和工具栏与底层的操作逻辑解耦，提高了系统的可扩展性和可维护性。
3. 任务队列和后台任务：命令模式可以用于实现任务队列。例如，在一个多线程应用程序中，我们可以将不同的任务封装为命令对象，然后将它们添加到一个任务队列中。后台线程可以从队列中获取命令对象，并执行它们。这样可以实现对任务的并行处理和优先级控制。
4. 宏（Macro）：在一些软件中，用户可以创建宏来执行一系列操作。命令模式可以用于实现这种功能。每个操作都可以封装为一个命令对象，然后将这些命令对象组合成一个宏命令。用户可以通过执行宏命令来一次性执行所有操作。
5. 电商系统中的订单处理：在电商系统中，我们可以将不同的订单操作（例如创建订单、取消订单、支付订单等）封装为命令对象。这样，处理这些操作的代码可以与具体的订单对象解耦，使得系统更加灵活和可维护。
6. 智能家居系统：在智能家居系统中，各种家电设备（如灯光、空调、电视等）的操作可以被封装为命令对象。通过使用命令模式，用户可以通过一个统一的接口（例如手机App或语音助手）来控制不同的设备，而无需关心设备的具体实现。
7. 游戏开发：在游戏开发中，玩家的操作（如移动角色、使用技能等）可以被封装为命令对象。这样，我们可以将玩家操作与游戏逻辑解耦，实现更加灵活的游戏控制和操作记录。
8. 状态模式与命令模式结合：在某些场景下，可以将状态模式与命令模式结合使用。例如，一个文档编辑器可能有多种编辑模式（如插入模式、选择模式等），每种模式对应不同的操作集合。我们可以将这些操作封装为命令对象，并根据当前的编辑模式来选择执行哪个命令。

这些案例展示了命令模式在实际应用中的广泛应用和灵活性。通过使用命令模式，我们可以实现更加松耦合、可扩展和可维护的系统。

第十章 解释器模式

一、解释器模式的原理和实现

解释器模式的英文翻译是 Interpreter Design Pattern。在 GoF 的《设计模式》一书中，它是这样定义的：

> Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

翻译成中文就是：解释器模式为某个语言定义它的语法（或者叫文法）表示，并定义一个解释器用来处理这个语法。

为了让你更好地理解定义，我举一个比较贴近生活的例子来解释一下。

实际上，理解这个概念，我们可以类比中英文翻译。我们知道，把英文翻译成中文是有一定规则的。这个规则就是定义中的“语法”。我们开发一个类似 Google Translate 这样的翻译器，这个翻译器能够根据语法规则，将输入的中文翻译成英文。这里的翻译器就是解释器模式定义中的“解释器”。

解释器模式的主要组成部分如下：

1. 抽象表达式（Abstract Expression）：定义解释器的接口，规定了解释操作的方法。通常包含一个 `interpret()` 方法。
2. 终结符表达式（Terminal Expression）：实现抽象表达式接口的具体类，用于表示语法中的终结符。终结符是不可分解的最小单位，例如数字、变量等。
3. 非终结符表达式（Nonterminal Expression）：实现抽象表达式接口的具体类，用于表示语法中的非终结符。非终结符通常是由终结符组成的复杂结构，例如加法、乘法等。
4. 上下文（Context）：包含解释器需要的全局信息，例如变量与值的映射关系等。
5. 客户端（Client）：构建抽象语法树，然后调用解释器的 `interpret()` 方法来解释语法。

解释器模式的优点：

- 易于实现简单的文法：解释器模式可以方便地实现简单的文法表示和解释。
- 易于扩展新的文法：通过添加新的终结符表达式和非终结符表达式，可以方便地扩展解释器的功能。

解释器模式的缺点：

- 难以应对复杂的文法：对于复杂的文法，解释器模式可能导致大量的类创建，使得系统变得难以维护。
- 效率较低：解释器模式通过递归调用的方式解释语法，这可能导致较低的执行效率。

因此，在实际项目中，如果需要处理的问题可以表示为一种简单的语法结构，那么可以考虑使用解释器模式。然而，对于复杂的语法结构，应该寻找其他更合适的方法，如使用编译器生成器（如 ANTLR）等工具。

以下是一个简单的计算器示例，使用解释器模式解释由数字和加法、减法操作符组成的算术表达式：

```
// 抽象表达式
interface Expression {
    int interpret(Map<String, Integer> variables);
}

// 终结符表达式：变量
class Variable implements Expression {
    private String name;

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

    @Override
    public int interpret(Map<String, Integer> variables) {
        return variables.getOrDefault(name, 0);
    }
}

// 非终结符表达式：加法
class Add implements Expression {
    private Expression left;
    private Expression right;
```

```
    public Add(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int interpret(Map<String, Integer> variables) {
        return left.interpret(variables) + right.interpret(variables);
    }
}

// 非终结符表达式：减法
class Subtract implements Expression {
    private Expression left;
    private Expression right;

    public Subtract(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int interpret(Map<String, Integer> variables) {
        return left.interpret(variables) - right.interpret(variables);
    }
}

public class Main {
    public static void main(String[] args) {
        // 构建语法树
        Expression left = new Variable("a");
        Expression right = new Variable("b");
        Expression add = new Add(left, right);
        Expression subtract = new Subtract(add, right);

        // 设置变量值
        Map<String, Integer> variables = new HashMap<>();
        variables.put("a", 10);
        variables.put("b", 5);

        // 解释表达式
        int result = subtract.interpret(variables);
        System.out.println("Result: " + result); // 输出：Result: 10
```

```
    }
}
```

在这个示例中，我们定义了一个简单的算术表达式语言，包括变量、加法和减法操作。`Expression`是抽象表达式接口，`Variable`是终结符表达式，表示变量；`Add`和`Subtract`是非终结符表达式，表示加法和减法操作。

客户端通过构建语法树来表示算术表达式。在本例中，我们构建了一个表示`(a + b) - b`的语法树。然后，客户端设置变量值，并使用`interpret()`方法解释表达式。最后，输出解释结果：10。

这个示例展示了如何使用解释器模式解释一个简单的算术表达式。解释器模式使得表示和解释这类语言变得容易，同时支持语法规则的修改和扩展。然而，对于复杂的、频繁变化的语法规则，请谨慎使用解释器模式。

解释器设计模式的优势和好处如下：

1. **易于扩展和修改**：解释器模式的语法规则通常表示为一棵语法树，因此可以轻松地扩展和修改这些规则，而不会影响其他部分的代码。例如，可以添加新的终结符或非终结符，或者修改某些规则的解释方式。
2. **灵活性和可重用性**：解释器模式使得语法规则和解释器之间的耦合度降低，从而使得语法规则和解释器可以独立地变化和重用。这种灵活性和可重用性使得解释器模式非常适用于需要频繁修改和扩展语法规则的场景。
3. **可读性和可维护性**：解释器模式的语法规则通常表示为类似于语法树的结构，这使得代码更加易于理解和维护。在实际开发中，通过使用解释器模式，可以将复杂的算法或规则分解成简单的表达式和操作，从而提高代码的可读性和可维护性。
4. **提高安全性和可靠性**：解释器模式通常会检查输入是否符合语法规则，从而提高系统的安全性和可靠性。在某些场景下，解释器模式还可以用于验证和过滤用户输入。

总之，解释器模式适用于需要解释和处理复杂语法规则的场景。通过使用解释器模式，可以轻松地扩展和修改语法规则，提高代码的可读性和可维护性，同时提高系统的安全性和可靠性。

二、解释器模式实战举例

在我们平时的项目开发中，监控系统非常重要，它可以时刻监控业务系统的运行情况，及时将异常报告给开发者。比如，如果每分钟接口出错数超过 100，监控系统就通过短信、微信、邮件等方式发送告警给开发者。

一般来讲，监控系统支持开发者自定义告警规则，比如我们可以用下面这样一个表达式，来表示一个告警规则，它表达的意思是：每分钟 API 总出错数超过 100 或者每分钟 API 总调用数超过 10000 就触发告警。

```
"api_error_per_minute > 100 || api_count_per_minute > 10000"
```

在监控系统中，告警模块只负责根据统计数据和告警规则，判断是否触发告警。至于每分钟 API 接口出错数、每分钟接口调用数等统计数据的计算，是由其他模块来负责的。其他模块将统计数据放到一个 Map 中（数据的格式如下所示），发送给告警模块。接下来，我们只关注告警模块。

```
Map<String, Long> apiStat = new HashMap<>();
apiStat.put("api_error_per_minute", 103);
apiStat.put("api_count_per_minute", 987);
```

为了简化讲解和代码实现，我们假设自定义的告警规则只包含“||、&&、>、<、==”这五个运算符，其中，“>、<、==”运算符的优先级高于“||、&&”运算符，“&&”运算符优先级高于“||”。在表达式中，任意元素之间需要通过空格来分隔。除此之外，用户可以自定义要监控的 key，比如前面的 api_error_per_minute、api_count_per_minute。

我们可以把自定义的告警规则，看作一种特殊“语言”的语法规则。我们实现一个解释器，能够根据规则，针对用户输入的数据，判断是否触发告警。利用解释器模式，我们把解析表达式的逻辑拆分到各个小类中，避免大而复杂的大类的出现。按照这个实现思路，我把刚刚的代码补全，如下所示，你可以拿你写的代码跟我写的对比一下。

```
public interface Expression {
  boolean interpret(Map<String, Long> stats);
}
public class GreaterExpression implements Expression {
  private String key;
  private long value;
  public GreaterExpression(String strExpression) {
    String[] elements = strExpression.trim().split("\\s+");
    if (elements.length != 3 || !elements[1].trim().equals(">")) {
      throw new RuntimeException("Expression is invalid: " + strExpression);
    }
```

```
        this.key = elements[0].trim();
        this.value = Long.parseLong(elements[2].trim());
    }
    public GreaterExpression(String key, long value) {
        this.key = key;
        this.value = value;
    }
    @Override
    public boolean interpret(Map<String, Long> stats) {
        if (!stats.containsKey(key)) {
            return false;
        }
        long statValue = stats.get(key);
        return statValue > value;
    }
}
// LessExpression/EqualExpression跟GreaterExpression代码类似，这里就省略了
public class AndExpression implements Expression {
    private List<Expression> expressions = new ArrayList<>();
    public AndExpression(String strAndExpression) {
        String[] strExpressions = strAndExpression.split("&&");
        for (String strExpr : strExpressions) {
            if (strExpr.contains(">")) {
                expressions.add(new GreaterExpression(strExpr));
            } else if (strExpr.contains("<")) {
                expressions.add(new LessExpression(strExpr));
            } else if (strExpr.contains("==")) {
                expressions.add(new EqualExpression(strExpr));
            } else {
                throw new RuntimeException("Expression is invalid: " +
strAndExpression);
            }
        }
    }
    public AndExpression(List<Expression> expressions) {
        this.expressions.addAll(expressions);
    }
    @Override
    public boolean interpret(Map<String, Long> stats) {
        for (Expression expr : expressions) {
            if (!expr.interpret(stats)) {
                return false;
```

```
            }
        }
        return true;
    }
}
public class OrExpression implements Expression {
    private List<Expression> expressions = new ArrayList<>();
    public OrExpression(String strOrExpression) {
        String[] andExpressions = strOrExpression.split("\\|\\|");
        for (String andExpr : andExpressions) {
            expressions.add(new AndExpression(andExpr));
        }
    }
    public OrExpression(List<Expression> expressions) {
        this.expressions.addAll(expressions);
    }
    @Override
    public boolean interpret(Map<String, Long> stats) {
        for (Expression expr : expressions) {
            if (expr.interpret(stats)) {
                return true;
            }
        }
        return false;
    }
}
public class AlertRuleInterpreter {
    private Expression expression;
    public AlertRuleInterpreter(String ruleExpression) {
        this.expression = new OrExpression(ruleExpression);
    }
    public boolean interpret(Map<String, Long> stats) {
        return expression.interpret(stats);
    }
}
```

三、使用场景

解释器模式通常用于处理具有一定结构和语法的问题，以下是一些常见的使用场景：

1. SQL 解释器：在数据库领域，SQL（结构化查询语言）是一种非常常见的查询语言。SQL 解释器负责将 SQL 语句解释为特定数据库引擎可以理解的命令。这种场景下，可以使用解释器模式来实现 SQL 的解释和执行。
2. 业务规则引擎：在企业级应用中，业务规则引擎用于处理和执行业务规则。这些规则可以表示为一种特定的语法结构，通过使用解释器模式，可以将这些规则解释为具体的操作，从而实现对业务规则的动态管理。
3. 自定义脚本语言：在某些场景下，开发者可能需要为应用程序提供一种简单的脚本语言，以方便用户进行自定义操作。这种情况下，可以使用解释器模式来实现脚本语言的解释和执行。
4. 数学表达式求值器：在编程中，我们有时需要对数学表达式进行求值。可以使用解释器模式将数学表达式解释为一系列操作，并计算出结果。例如，我们可以为加法、减法、乘法和除法等操作创建终结符表达式和非终结符表达式，然后构建抽象语法树来表示数学表达式。

需要注意的是，解释器模式适合处理简单的语法结构，对于复杂的语法结构，可能会导致大量的类创建和较低的执行效率。在实际项目中，应根据具体需求来判断是否使用解释器模式。

第十一章 中介模式

今天，我们来学习 23 种经典设计模式中的最后一个，中介模式。跟前面刚刚讲过的命令模式、解释器模式类似，中介模式也属于**不怎么常用**的模式，应用场景比较特殊、有限，但是，跟它俩不同的是，中介模式理解起来并不难，代码实现也非常简单，学习难度要小很多。

话不多说，让我们正式开始今天的学习吧！

一、原理和实现

中介模式的英文翻译是 Mediator Design Pattern。在 GoF 中的《设计模式》一书中，它是这样定义的：

> Mediator pattern defines a separate (mediator) object that encapsulates the interaction between a set of objects and the objects delegate their interaction to a mediator object instead of interacting with each other directly.

翻译成中文就是：中介模式定义了一个单独的（中介）对象，来封装一组对象之间的交互。将这组对象之间的交互委派给与中介对象交互，来避免对象之间的直接交互。

还记得我们在第 30 节课中讲的“如何给代码解耦”吗？其中一个方法就是引入中间层。

中介者模式是一种行为设计模式，它用于**降低多个对象之间的通信复杂性**。这种模式通过引入一个中介者对象来处理对象之间的通信，使得对象之间不需要直接相互引用，从而降低它们之间的耦合度。

以下是中介者模式的一些关键组件：

1. 中介者（Mediator）：定义了一个接口，用于与各个同事（Colleague）对象通信。
2. 具体中介者（Concrete Mediator）：实现中介者接口并协调各个同事对象之间的交互。
3. 同事（Colleague）：定义了各个对象之间的接口。每个同事对象知道它的中介者对象，但不知道其他同事对象。
4. 具体同事（Concrete Colleague）：实现同事接口并通过中介者与其他同事对象通信。

实际上，中介模式的设计思想跟中间层很像，通过引入中介这个中间层，将一组对象之间的交互关系（或者说依赖关系）从多对多（网状关系）转换为一对多（星状关系）。原来一个对象要跟 n 个对象交互，现在只需要跟一个中介对象交互，从而最小化对象之间的交互关系，降低了代码的复杂度，提高了代码的可读性和可维护性。

下面是一个简单的Java代码实例：

```
interface Mediator {
    void send(String message, Colleague colleague);
}

// 具体中介者
class ConcreteMediator implements Mediator {
    private Colleague1 colleague1;
    private Colleague2 colleague2;

    public void setColleague1(Colleague1 colleague1) {
        this.colleague1 = colleague1;
    }
```

```
    public void setColleague2(Colleague2 colleague2) {
        this.colleague2 = colleague2;
    }

    @Override
    public void send(String message, Colleague colleague) {
        if (colleague == colleague1) {
            colleague2.notify(message);
        } else {
            colleague1.notify(message);
        }
    }
}

// 同事接口
abstract class Colleague {
    protected Mediator mediator;

    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }

    public abstract void send(String message);
    public abstract void notify(String message);
}

// 具体同事1
class Colleague1 extends Colleague {
    public Colleague1(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String message) {
        mediator.send(message, this);
    }

    @Override
    public void notify(String message) {
        System.out.println("Colleague1 receives message: " + message);
    }
}
```

```
// 具体同事2
class Colleague2 extends Colleague {
    public Colleague2(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String message) {
        mediator.send(message, this);
    }

    @Override
    public void notify(String message) {
        System.out.println("Colleague2 receives message: " + message);
    }
}

// 客户端代码
public class MediatorPatternDemo {
    public static void main(String[] args) {
        ConcreteMediator mediator = new ConcreteMediator();

        Colleague1 colleague1 = new Colleague1(mediator);
        Colleague2 colleague2 = new Colleague2(mediator);

        mediator.setColleague1(colleague1);
        mediator.setColleague2(colleague2);

        colleague1.send("Hello, Colleague2!");
        colleague2.send("Hi, Colleague1! How are you?");
        colleague1.send("I'm fine, thank you!");
        colleague2.send("Great, have a nice day!");
    }
}
```

在这个例子中，我们让`Colleague1`和`Colleague2`对象之间互相发送消息。在主函数中，我们创建了一个`ConcreteMediator`对象，并将其分别传递给`Colleague1`和`Colleague2`对象。然后，我们使用中介者对象将两个同事对象关联起来。最后，我们让同事对象互相发送消息。

输出结果如下：

```
Colleague2 receives message: Hello, Colleague2!
Colleague1 receives message: Hi, Colleague1! How are you?
Colleague2 receives message: I'm fine, thank you!
Colleague1 receives message: Great, have a nice day!
```

通过使用中介者模式，我们实现了`Colleague1`和`Colleague2`对象之间的解耦。这样，如果我们需要在系统中引入更多的同事对象，只需创建一个新的具体同事类并在中介者中实现适当的逻辑即可。这大大降低了对象之间的耦合度，提高了代码的可维护性和可扩展性。

二、使用场景

我们可以像chatgpt咨询，这个模式有哪些使用场景，他告诉我们一下的答案：

1. 聊天室：在一个聊天室应用中，用户之间需要相互发送消息。使用中介者模式，我们可以创建一个中介者对象（聊天室服务器）来管理用户之间的消息传递，这样每个用户不需要知道其他用户的存在，只需将消息发送给中介者，然后由中介者负责分发消息。
2. 机场调度系统：机场调度系统通常需要管理多个飞机、跑道和门等资源。在这种情况下，可以使用一个中介者（如调度塔）来协调和管理这些资源之间的通信和安排，以确保系统的正常运行。
3. 图形用户界面（GUI）组件：在复杂的GUI应用程序中，各种组件（如按钮、列表框、文本框等）可能需要相互通信。使用中介者模式，我们可以创建一个中介者对象来处理这些组件之间的通信，避免组件之间直接相互引用，降低它们之间的耦合度。
4. 智能家居系统：在智能家居系统中，各种设备（如灯、窗帘、空调等）需要根据用户的需求进行协调和控制。通过引入一个中介者（如智能家居中心），我们可以实现各种设备之间的协调和通信，从而实现对整个系统的统一管理。
5. MVC（Model-View-Controller）架构：在MVC架构中，控制器（Controller）充当中介者的角色，协调模型（Model）和视图（View）之间的交互。这样，模型和视图之间的耦合度降低，使得它们可以独立地进行修改和扩展。
6. 网络游戏：在多人在线游戏中，游戏服务器可以充当中介者，管理玩家之间的通信和交互。这样，每个玩家只需要与游戏服务器通信，而不必知道其他玩家的详细信息。
7. 电子商务平台：在电子商务平台中，买家和卖家需要进行交易。平台可以作为一个中介者，处理买家和卖家之间的通信和交易，包括订单处理、支付、退款等。这样，买家和卖家之间不需要直接进行通信，降低了交易的复杂性。
8. 事件总线：在事件驱动的系统中，事件总线可以作为一个中介者，负责处理和分发事件。系统中的组件可以将事件发布到事件总线，而不必直接与其他组件通信。这样，组件之间的耦合度降低，系统变得更加模块化。

9. 工作流引擎：在工作流引擎中，各个流程节点之间需要进行通信和协调。工作流引擎可以作为一个中介者，负责管理流程节点之间的通信，确保整个流程的正确执行。

这些案例展示了中介者模式在不同领域和场景中的应用。通过引入中介者，我们可以降低对象之间的耦合度，简化系统的设计和维护。

1、多人游戏案例

在这个案例中，我们将展示如何在多人在线游戏中使用中介者模式。我们将创建一个简单的游戏，如五子棋，其中有两个玩家，分别为PlayerA和PlayerB。游戏服务器(GameServer)将充当中介者，负责处理玩家之间的消息传递。

首先，我们定义中介者接口：

```
interface GameServerMediator {
    void sendMessage(String message, Player player);
}
```

接下来，实现具体的中介者类：

```
class GameServer implements GameServerMediator {
    private PlayerA playerA;
    private PlayerB playerB;

    public void setPlayerA(PlayerA playerA) {
        this.playerA = playerA;
    }

    public void setPlayerB(PlayerB playerB) {
        this.playerB = playerB;
    }

    @Override
    public void sendMessage(String message, Player player) {
        if (player == playerA) {
            playerB.receiveMessage(message);
        } else {
            playerA.receiveMessage(message);
        }
    }
}
```

定义玩家抽象类：

```
abstract class Player {
    protected GameServerMediator mediator;

    public Player(GameServerMediator mediator) {
        this.mediator = mediator;
    }

    public abstract void send(String message);
    public abstract void receiveMessage(String message);
}
```

实现具体的玩家类：

```
class PlayerA extends Player {
    public PlayerA(GameServerMediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String message) {
        mediator.sendMessage(message, this);
    }

    @Override
    public void receiveMessage(String message) {
        System.out.println("PlayerA收到消息: " + message);
    }
}

class PlayerB extends Player {
    public PlayerB(GameServerMediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String message) {
        mediator.sendMessage(message, this);
    }

    @Override
    public void receiveMessage(String message) {
```

```
        System.out.println("PlayerB收到消息: " + message);
    }
}
```

最后，在客户端代码中创建玩家对象并进行通信：

```
public class MultiplayerGameDemo {
    public static void main(String[] args) {
        GameServer gameServer = new GameServer();

        PlayerA playerA = new PlayerA(gameServer);
        PlayerB playerB = new PlayerB(gameServer);

        gameServer.setPlayerA(playerA);
        gameServer.setPlayerB(playerB);

        playerA.send("你好，PlayerB!");
        playerB.send("你好，PlayerA! 一起玩游戏吗？");
        playerA.send("好的，我们开始吧！");
    }
}
```

输出结果如下：

```
PlayerB收到消息: 你好，PlayerB!
PlayerA收到消息: 你好，PlayerA! 一起玩游戏吗?
PlayerB收到消息: 好的，我们开始吧!
```

在这个例子中，我们创建了一个简单的多人在线游戏，其中GameServer充当中介者，负责管理PlayerA和PlayerB之间的通信。玩家对象不直接相互通信，而是通过GameServer进行消息传递。这样，如果我们需要添加更多玩家，只需在GameServer中进行相应的修改即可，而无需修改现有的玩家类。这降低了系统的耦合度，提高了可扩展性和可维护性。

例如，假设我们要在游戏中添加第三个玩家PlayerC。首先，我们需要创建一个新的玩家类：

```
class PlayerC extends Player {
    public PlayerC(GameServerMediator mediator) {
        super(mediator);
    }

    @Override
```

```
    public void send(String message) {
        mediator.sendMessage(message, this);
    }

    @Override
    public void receiveMessage(String message) {
        System.out.println("PlayerC收到消息: " + message);
    }
}
```

然后，我们需要更新GameServer类以支持第三个玩家：

```
class GameServer implements GameServerMediator {
    private PlayerA playerA;
    private PlayerB playerB;
    private PlayerC playerC;

    // ...省略setPlayerA()和setPlayerB()方法...

    public void setPlayerC(PlayerC playerC) {
        this.playerC = playerC;
    }

    @Override
    public void sendMessage(String message, Player player) {
        if (player == playerA) {
            playerB.receiveMessage(message);
            playerC.receiveMessage(message);
        } else if (player == playerB) {
            playerA.receiveMessage(message);
            playerC.receiveMessage(message);
        } else {
            playerA.receiveMessage(message);
            playerB.receiveMessage(message);
        }
    }
}
```

最后，在客户端代码中添加第三个玩家，并进行通信：

```
public class MultiplayerGameDemo {
    public static void main(String[] args) {
        GameServer gameServer = new GameServer();
```

```
        PlayerA playerA = new PlayerA(gameServer);
        PlayerB playerB = new PlayerB(gameServer);
        PlayerC playerC = new PlayerC(gameServer);

        gameServer.setPlayerA(playerA);
        gameServer.setPlayerB(playerB);
        gameServer.setPlayerC(playerC);

        playerA.send("你好，PlayerB和PlayerC!");
        playerB.send("你好，PlayerA和PlayerC! 一起玩游戏吗？");
        playerC.send("我也加入！让我们开始吧！");
    }
}
```

输出结果如下：

```
makefileCopy code
PlayerB收到消息: 你好，PlayerB和PlayerC!
PlayerC收到消息: 你好，PlayerB和PlayerC!
PlayerA收到消息: 你好，PlayerA和PlayerC! 一起玩游戏吗？
PlayerC收到消息: 你好，PlayerA和PlayerC! 一起玩游戏吗？
PlayerA收到消息: 我也加入！让我们开始吧！
PlayerB收到消息: 我也加入！让我们开始吧！
```

通过使用中介者模式，我们可以轻松地在游戏中添加更多玩家，而不需要修改现有玩家类。这使得游戏的设计更加灵活和可扩展。

2、机场调度案例

在机场调度系统的例子中，我们可以使用中介者模式来协调飞机、跑道和其他设施之间的交互。在这个例子中，我们将创建一个简化的调度系统，其中有两个飞机和两个跑道。机场调度塔(AirportControlTower)将作为中介者，负责协调飞机和跑道之间的通信。

首先，定义中介者接口：

```
interface ControlTowerMediator {
    void requestRunway(Flight flight);
    void releaseRunway(Flight flight);
}
```

接下来，实现具体的中介者类：

```
class AirportControlTower implements ControlTowerMediator {
    private Runway runway1;
    private Runway runway2;

    public void setRunway1(Runway runway1) {
        this.runway1 = runway1;
    }

    public void setRunway2(Runway runway2) {
        this.runway2 = runway2;
    }

    @Override
    public void requestRunway(Flight flight) {
        if (!runway1.isOccupied()) {
            runway1.assignFlight(flight);
        } else if (!runway2.isOccupied()) {
            runway2.assignFlight(flight);
        } else {
            System.out.println("所有跑道均被占用，请稍候再试。");
        }
    }

    @Override
    public void releaseRunway(Flight flight) {
        if (runway1.getCurrentFlight() == flight) {
            runway1.release();
        } else if (runway2.getCurrentFlight() == flight) {
            runway2.release();
        }
    }
}
```

定义跑道类：

```
class Runway {
    private Flight currentFlight;
    private String name;

    public Runway(String name) {
        this.name = name;
```

```
    }

    public void assignFlight(Flight flight) {
        currentFlight = flight;
        System.out.println("跑道" + name + "分配给航班" + flight.getFlightNumber());
    }

    public void release() {
        System.out.println("跑道" + name + "释放，航班" +
currentFlight.getFlightNumber() + "已离开。");
        currentFlight = null;
    }

    public Flight getCurrentFlight() {
        return currentFlight;
    }

    public boolean isOccupied() {
        return currentFlight != null;
    }
}
```

定义飞机类：

```
class Flight {
    private String flightNumber;
    private ControlTowerMediator mediator;

    public Flight(String flightNumber, ControlTowerMediator mediator) {
        this.flightNumber = flightNumber;
        this.mediator = mediator;
    }

    public void requestRunway() {
        mediator.requestRunway(this);
    }

    public void releaseRunway() {
        mediator.releaseRunway(this);
    }

    public String getFlightNumber() {
        return flightNumber;
```

```
    }
}
```

最后，在客户端代码中创建飞机和跑道对象，并模拟调度过程：

```
public class AirportControlSystemDemo {
    public static void main(String[] args) {
        AirportControlTower controlTower = new AirportControlTower();

        Runway runway1 = new Runway("1");
        Runway runway2 = new Runway("2");

        controlTower.setRunway1(runway1);
        controlTower.setRunway2(runway2);
        Flight flight1 = new Flight("CA001", controlTower);
        Flight flight2 = new Flight("MU002", controlTower);
        Flight flight3 = new Flight("CZ003", controlTower);

        // 航班1请求跑道
        flight1.requestRunway();
        // 航班2请求跑道
        flight2.requestRunway();
        // 航班3请求跑道，但所有跑道都被占用
        flight3.requestRunway();

        // 航班1释放跑道
        flight1.releaseRunway();

        // 航班3再次请求跑道，此时可以分配
        flight3.requestRunway();
    }
}
```

输出结果如下：

跑道1分配给航班CA001 跑道2分配给航班MU002 所有跑道均被占用，请稍候再试。 跑道1释放，航班CA001已离开。 跑道1分配给航班CZ003

在这个例子中，我们创建了一个简化的机场调度系统，其中AirportControlTower充当中介者，负责管理飞机和跑道之间的交互。飞机和跑道对象之间没有直接通信，而是通过AirportControlTower进行调度。这使得系统的设计更加模块化，可以方便地添加更多飞机、跑道或其他设施，而无需修改现有的类。

上述示例展示了如何在机场调度系统中使用中介者模式。AirportControlTower作为中介者负责协调飞机和跑道之间的交互。当飞机需要请求或释放跑道时，它们会通过AirportControlTower进行操作。这样的设计降低了系统的耦合度，提高了可扩展性和可维护性。

三、中介模式 VS 观察者模式

中介者模式和观察者模式都是用于解决对象之间的通信问题，但它们的**关注点和应用场景**有所不同。下面我们对比一下它们之间的主要区别：

1. 关注点：

- 中介者模式：关注的是如何减少对象之间的直接耦合，从而提高系统的可扩展性和可维护性。在中介者模式中，对象之间不直接通信，而是通过中介者对象进行交互。中介者对象负责处理其他对象之间的交互逻辑，使各个对象之间的依赖关系得到解耦。
- 观察者模式：关注的是如何在对象间实现一种一对多的依赖关系，使得当一个对象的状态发生改变时，其他依赖于它的对象都能得到通知并自动更新。观察者模式中的对象之间具有明确的被观察者（Subject）和观察者（Observer）角色。

1. 应用场景：

- 中介者模式：适用于那些具有复杂对象交互关系的系统，例如界面组件之间的交互、协同工作的多个模块之间的通信等。通过引入中介者，可以降低系统的耦合度，使得各个对象之间更加独立，便于修改和扩展。
- 观察者模式：适用于那些需要在对象间实现一种一对多的依赖关系的场景，例如数据驱动的UI更新、事件驱动的系统等。观察者模式允许被观察者对象在其状态发生改变时通知所有关注它的观察者对象，实现了状态同步。

1. 实现方式：

- 中介者模式：中介者模式中，对象之间通常通过注册到中介者，然后中介者负责协调它们之间的交互。这样，对象之间不需要直接引用其他对象，从而降低了耦合度。
- 观察者模式：观察者模式中，被观察者对象维护了一个观察者列表，当其状态发生改变时，会通知列表中的所有观察者。观察者对象通常通过实现一个特定的接口来实现对被观察者的关注。

总结一下，中介者模式和观察者模式虽然都可以用于解决对象间的通信问题，但它们侧重的点和适用场景有所不同。中介者模式主要用于降低对象间的耦合度，而观察者模式主要用于实现一

前面讲观察者模式的时候，我们讲到，观察者模式有多种实现方式。虽然经典的实现方式没法彻底解耦观察者和被观察者，观察者需要注册到被观察者中，被观察者状态更新需要调用观察者的 update() 方法。但是，在跨进程的实现方式中，我们可以利用消息队列实现彻底解耦，观察者和被观察者都只需要跟消息队列交互，观察者完全不知道被观察者的存在，被观察者也完全不知道观察者的存在。

我们前面提到，中介模式也是为了解耦对象之间的交互，所有的参与者都只与中介进行交互。而观察者模式中的消息队列，就有点类似中介模式中的“中介”，观察者模式的中观察者和被观察者，就有点类似中介模式中的“参与者”。那问题来了：中介模式和观察者模式的区别在哪里呢？什么时候选择使用中介模式？什么时候选择使用观察者模式呢？

在观察者模式中，尽管一个参与者既可以是观察者，同时也可以是被观察者，但是，大部分情况下，交互关系往往都是单向的，一个参与者要么是观察者，要么是被观察者，不会兼具两种身份。也就是说，在观察者模式的应用场景中，参与者之间的交互关系比较有条理。

而中介模式正好相反。只有当参与者之间的交互关系错综复杂，维护成本很高的时候，我们才考虑使用中介模式。毕竟，中介模式的应用会带来一定的副作用，前面也讲到，它有可能会产生大而复杂的上帝类。除此之外，如果一个参与者状态的改变，其他参与者执行的操作有一定先后顺序的要求，这个时候，中介模式就可以利用中介类，通过先后调用不同参与者的方法，来实现顺序的控制，而观察者模式是无法实现这样的顺序要求的。