64 | 状态模式：游戏、工作流引擎中常用的状态机是如何实现的？

2020-03-30 王争

设计模式之美 进入课程 >

讲述：冯永吉
时长 10:13 大小 9.36M

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。

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！

什么是有限状态机？

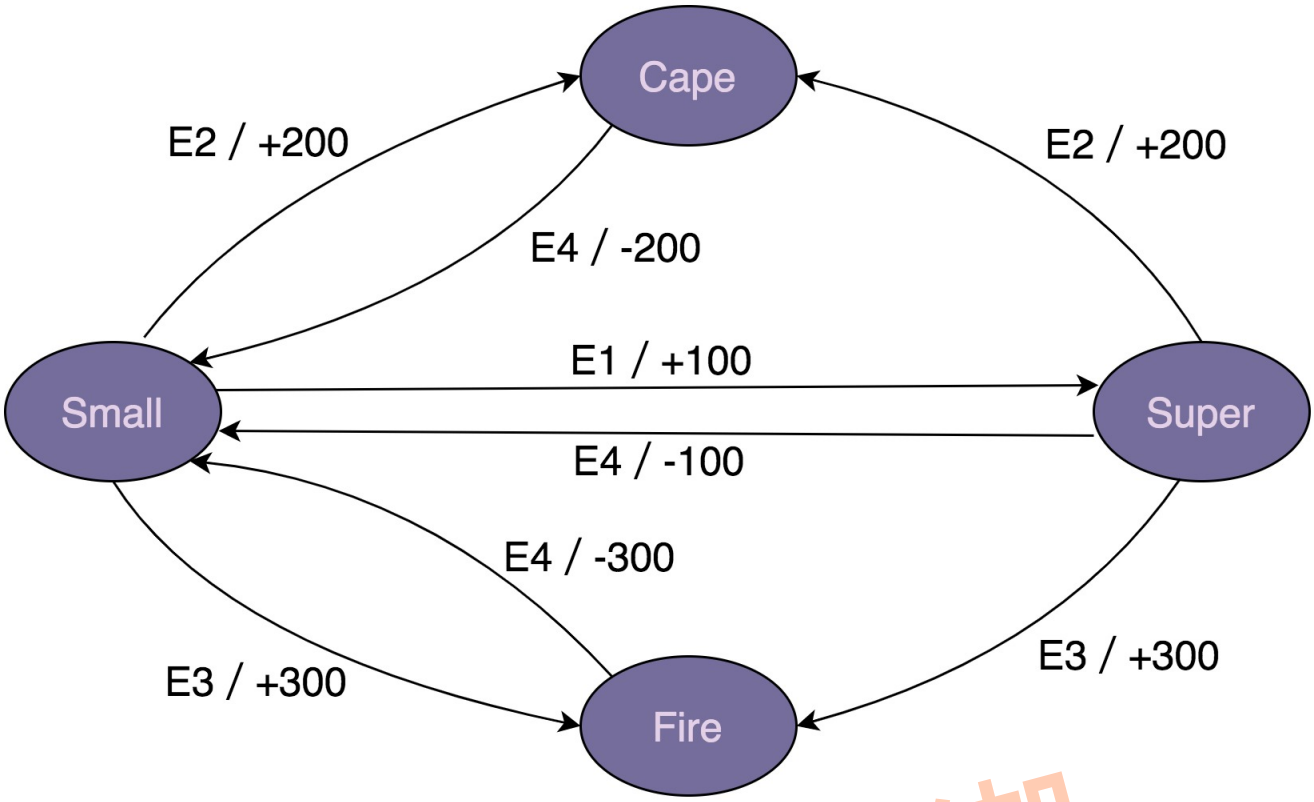
有限状态机，英文翻译是 Finite State Machine，缩写为 FSM，简称为状态机。状态机有 3 个组成部分：状态（State）、事件（Event）、动作（Action）。其中，事件也称为转移条件（Transition Condition）。事件触发状态的转移及动作的执行。不过，动作不是必须的，也可能只转移状态，不执行任何动作。

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，马里奥可以变身为多种形态，比如小马里奥（Small Mario）、超级马里奥（Super Mario）、火焰马里奥（Fire Mario）、斗篷马里奥（Cape Mario）等等。在不同的游戏情节下，各个形态会互相转化，并相应的增减积分。比如，初始形态是小马里奥，吃了蘑菇之后就会变成超级马里奥，并且增加 100 积分。

实际上，马里奥形态的转变就是一个状态机。其中，马里奥的不同形态就是状态机中的“状态”，游戏情节（比如吃了蘑菇）就是状态机中的“事件”，加减积分就是状态机中的“动作”。比如，吃蘑菇这个事件，会触发状态的转移：从小马里奥转移到超级马里奥，以及触发动作的执行（增加 100 积分）。

为了方便接下来的讲解，我对游戏背景做了简化，只保留了部分状态和事件。简化之后的状态转移如下图所示：



E1：吃了蘑菇　　E2：获得斗篷

E3：获得火焰　　E4：遇到怪物

极客时间

我们如何编程来实现上面的状态机呢？换句话说，如何将上面的状态转移图翻译成代码呢？

我写了一个骨架代码，如下所示。其中，obtainMushRoom()、obtainCape()、obtainFireFlower()、meetMonster() 这几个函数，能够根据当前的状态和事件，更新状态和增减积分。不过，具体的代码实现我暂时并没有给出。你可以把它当做面试题，试着补全一下，然后再来看我下面的讲解，这样你的收获会更大。

复制代码

```
public enum State {
  SMALL(0),
  SUPER(1),
  FIRE(2),
  CAPE(3);

  private int value;
```

```
    private State(int value) {
        this.value = value;
    }

    public int getValue() {
        return this.value;
    }
}

public class MarioStateMachine {
    private int score;
    private State currentState;

    public MarioStateMachine() {
        this.score = 0;
        this.currentState = State.SMALL;
    }

    public void obtainMushRoom() {
        //TODO
    }

    public void obtainCape() {
        //TODO
    }

    public void obtainFireFlower() {
        //TODO
    }

    public void meetMonster() {
        //TODO
    }

    public int getScore() {
        return this.score;
    }

    public State getCurrentState() {
        return this.currentState;
    }
}

public class ApplicationDemo {
    public static void main(String[] args) {
        MarioStateMachine mario = new MarioStateMachine();
        mario.obtainMushRoom();
        int score = mario.getScore();
        State state = mario.getCurrentState();
        System.out.println("mario score: " + score + "; state: " + state);
    }

```

```
60 }
```

状态机实现方式一：分支逻辑法

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，我把这种方法暂且命名为分支逻辑法。

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：

复制代码

```
1  public class MarioStateMachine {
2    private int score;
3    private State currentState;
4
5    public MarioStateMachine() {
6      this.score = 0;
7      this.currentState = State.SMALL;
8    }
9
10   public void obtainMushRoom() {
11     if (currentState.equals(State.SMALL)) {
12       this.currentState = State.SUPER;
13       this.score += 100;
14     }
15   }
16
17   public void obtainCape() {
18     if (currentState.equals(State.SMALL) || currentState.equals(State.SUPER) )
19       this.currentState = State.CAPE;
20       this.score += 200;
21     }
22   }
23
24   public void obtainFireFlower() {
25     if (currentState.equals(State.SMALL) || currentState.equals(State.SUPER) )
26       this.currentState = State.FIRE;
27       this.score += 300;
28     }
29   }
30
31   public void meetMonster() {
32     if (currentState.equals(State.SUPER)) {
33       this.currentState = State.SMALL;
34       this.score -= 100;
```

```
35        return;
36      }
37
38      if (currentState.equals(State.CAPE)) {
39        this.currentState = State.SMALL;
40        this.score -= 200;
41        return;
42      }
43
44      if (currentState.equals(State.FIRE)) {
45        this.currentState = State.SMALL;
46        this.score -= 300;
47        return;
48      }
49    }
50
51    public int getScore() {
52      return this.score;
53    }
54
55    public State getCurrentState() {
56      return this.currentState;
57    }
58  }
```

对于简单的状态机来说，分支逻辑这种实现方式是可以接受的。但是，对于复杂的状态机来说，这种实现方式极易漏写或者错写某个状态转移。除此之外，代码中充斥着大量的 if-else 或者 switch-case 分支判断逻辑，可读性和可维护性都很差。如果哪天修改了状态机中的某个状态转移，我们要在冗长的分支逻辑中找到对应的代码进行修改，很容易改错，引入 bug。

状态机实现方式二：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。

实际上，除了用状态转移图来表示之外，状态机还可以用二维表来表示，如下所示。在这个二维表中，第一维表示当前状态，第二维表示事件，值表示当前状态经过事件之后，转移到的新状态及其执行的动作。

	E1(Got MushRoom)	E2(Got Cape)	E3(Got Fire Flower)	E4(Met Monster)
Small	Super/+100	Cape/+200	Fire/+300	/
Super	/	Cape/+200	Fire/+300	Small/-100
Cape	/	/	/	Small/-200
Fire	/	/	/	Small/-300
备注：表中的斜杠表示不存在这种状态转移。				

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。具体的代码如下所示：


```
public enum Event {
  GOT_MUSHROOM(0),
  GOT_CAPE(1),
  GOT_FIRE(2),
  MET_MONSTER(3);

  private int value;

  private Event(int value) {
    this.value = value;
  }

  public int getValue() {
    return this.value;
  }
}

public class MarioStateMachine {
  private int score;
  private State currentState;

  private static final State[][] transitionTable = {
          {SUPER, CAPE, FIRE, SMALL},
          {SUPER, CAPE, FIRE, SMALL},
          {CAPE, CAPE, CAPE, SMALL},
          {FIRE, FIRE, FIRE, SMALL}
  };

```

```
29    private static final int[][] actionTable = {
30            {+100, +200, +300, +0},
31            {+0, +200, +300, -100},
32            {+0, +0, +0, -200},
33            {+0, +0, +0, -300}
34    };
35
36    public MarioStateMachine() {
37      this.score = 0;
38      this.currentState = State.SMALL;
39    }
40
41    public void obtainMushRoom() {
42      executeEvent(Event.GOT_MUSHROOM);
43    }
44
45    public void obtainCape() {
46      executeEvent(Event.GOT_CAPE);
47    }
48
49    public void obtainFireFlower() {
50      executeEvent(Event.GOT_FIRE);
51    }
52
53    public void meetMonster() {
54      executeEvent(Event.MET_MONSTER);
55    }
56
57    private void executeEvent(Event event) {
58      int stateValue = currentState.getValue();
59      int eventValue = event.getValue();
60      this.currentState = transitionTable[stateValue][eventValue];
61      this.score = actionTable[stateValue][eventValue];
62    }
63
64    public int getScore() {
65      return this.score;
66    }
67
68    public State getCurrentState() {
69      return this.currentState;
70    }
71
72  }
```

状态机实现方式三：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积分加减，所以，我们用一个 int 类型的二维数组 actionTable 就能表示，二维数组中的值表示积分的加减值。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（比如加减积分、写数据库，还有可能发送消息通知等等），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。

其中，IMario 是状态的接口，定义了所有的事件。SmallMario、SuperMario、CapeMario、FireMario 是 IMario 接口的实现类，分别对应状态机中的 4 个状态。原来所有的状态转移和动作执行的代码逻辑，都集中在 MarioStateMachine 类中，现在，这些代码逻辑被分散到了这 4 个状态类中。

复制代码

```
public interface IMario { //所有状态类的接口
  State getName();
  //以下是定义的事件
  void obtainMushRoom();
  void obtainCape();
  void obtainFireFlower();
  void meetMonster();
}

public class SmallMario implements IMario {
  private MarioStateMachine stateMachine;

  public SmallMario(MarioStateMachine stateMachine) {
    this.stateMachine = stateMachine;
  }

  @Override
  public State getName() {
    return State.SMALL;
  }

  @Override
  public void obtainMushRoom() {
```

```
    stateMachine.setCurrentState(new SuperMario(stateMachine));
    stateMachine.setScore(stateMachine.getScore() + 100);
  }

  @Override
  public void obtainCape() {
    stateMachine.setCurrentState(new CapeMario(stateMachine));
    stateMachine.setScore(stateMachine.getScore() + 200);
  }

  @Override
  public void obtainFireFlower() {
    stateMachine.setCurrentState(new FireMario(stateMachine));
    stateMachine.setScore(stateMachine.getScore() + 300);
  }

  @Override
  public void meetMonster() {
    // do nothing...
  }
}

public class SuperMario implements IMario {
  private MarioStateMachine stateMachine;

  public SuperMario(MarioStateMachine stateMachine) {
    this.stateMachine = stateMachine;
  }

  @Override
  public State getName() {
    return State.SUPER;
  }

  @Override
  public void obtainMushRoom() {
    // do nothing...
  }

  @Override
  public void obtainCape() {
    stateMachine.setCurrentState(new CapeMario(stateMachine));
    stateMachine.setScore(stateMachine.getScore() + 200);
  }

  @Override
  public void obtainFireFlower() {
    stateMachine.setCurrentState(new FireMario(stateMachine));
    stateMachine.setScore(stateMachine.getScore() + 300);
  }

  @Override
```

```
76    public void meetMonster() {
77      stateMachine.setCurrentState(new SmallMario(stateMachine));
78      stateMachine.setScore(stateMachine.getScore() - 100);
79    }
80  }
81
82  // 省略CapeMario、FireMario类...
83
84  public class MarioStateMachine {
85    private int score;
86    private IMario currentState; // 不再使用枚举来表示状态
87
88    public MarioStateMachine() {
89      this.score = 0;
90      this.currentState = new SmallMario(this);
91    }
92
93    public void obtainMushRoom() {
94      this.currentState.obtainMushRoom();
95    }
96
97    public void obtainCape() {
98      this.currentState.obtainCape();
99    }
100
101   public void obtainFireFlower() {
102     this.currentState.obtainFireFlower();
103   }
104
105   public void meetMonster() {
106     this.currentState.meetMonster();
107   }
108
109   public int getScore() {
110     return this.score;
111   }
112
113   public State getCurrentState() {
114     return this.currentState.getName();
115   }
116
117   public void setScore(int score) {
118     this.score = score;
119   }
120
121   public void setCurrentState(IMario currentState) {
122     this.currentState = currentState;
123   }
124 }
```

上面的代码实现不难看懂，我只强调其中的一点，即 MarioStateMachine 和各个状态类之间是双向依赖关系。MarioStateMachine 依赖各个状态类是理所当然的，但是，反过来，各个状态类为什么要依赖 MarioStateMachine 呢？这是因为，各个状态类需要更新 MarioStateMachine 中的两个变量，score 和 currentState。

实际上，上面的代码还可以继续优化，我们可以将状态类设计成单例，毕竟状态类中不包含任何成员变量。但是，当将状态类设计成单例之后，我们就无法通过构造函数来传递 MarioStateMachine 了，而状态类又要依赖 MarioStateMachine，那该如何解决这个问题呢？

实际上，在第 42 讲单例模式的讲解中，我们提到过几种解决方法，你可以回过头去再查看一下。在这里，我们可以通过函数参数将 MarioStateMachine 传递进状态类。根据这个设计思路，我们对上面的代码进行重构。重构之后的代码如下所示：

复制代码

```
public interface IMario {
  State getName();
  void obtainMushRoom(MarioStateMachine stateMachine);
  void obtainCape(MarioStateMachine stateMachine);
  void obtainFireFlower(MarioStateMachine stateMachine);
  void meetMonster(MarioStateMachine stateMachine);
}

public class SmallMario implements IMario {
  private static final SmallMario instance = new SmallMario();
  private SmallMario() {}
  public static SmallMario getInstance() {
    return instance;
  }

  @Override
  public State getName() {
    return State.SMALL;
  }

  @Override
  public void obtainMushRoom(MarioStateMachine stateMachine) {
    stateMachine.setCurrentState(SuperMario.getInstance());
    stateMachine.setScore(stateMachine.getScore() + 100);
  }

  @Override
  public void obtainCape(MarioStateMachine stateMachine) {
    stateMachine.setCurrentState(CapeMario.getInstance());
```

```
    stateMachine.setScore(stateMachine.getScore() + 200);
  }

  @Override
  public void obtainFireFlower(MarioStateMachine stateMachine) {
    stateMachine.setCurrentState(FireMario.getInstance());
    stateMachine.setScore(stateMachine.getScore() + 300);
  }

  @Override
  public void meetMonster(MarioStateMachine stateMachine) {
    // do nothing...
  }
}

// 省略SuperMario、CapeMario、FireMario类...

public class MarioStateMachine {
  private int score;
  private IMario currentState;

  public MarioStateMachine() {
    this.score = 0;
    this.currentState = SmallMario.getInstance();
  }

  public void obtainMushRoom() {
    this.currentState.obtainMushRoom(this);
  }

  public void obtainCape() {
    this.currentState.obtainCape(this);
  }

  public void obtainFireFlower() {
    this.currentState.obtainFireFlower(this);
  }

  public void meetMonster() {
    this.currentState.meetMonster(this);
  }

  public int getScore() {
    return this.score;
  }

  public State getCurrentState() {
    return this.currentState.getName();
  }

  public void setScore(int score) {
    this.score = score;
```

```
82    }
83
84    public void setCurrentState(IMario currentState) {
85      this.currentState = currentState;
86    }
87  }
```

实际上，像游戏这种比较复杂的状态机，包含的状态比较多，我优先推荐使用查表法，而状态模式会引入非常多的状态类，会导致代码比较难维护。相反，像电商下单、外卖下单这种类型的状态机，它们的状态并不多，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会比较复杂，所以，更加推荐使用状态模式来实现。

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。

课堂讨论

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？

欢迎留言和我分享你的想法。如果有收获，欢迎你把这篇文章分享给你的朋友。

上一篇　63 | 职责链模式（下）：框架中常用的过滤器、拦截器是如何实现的？

下一篇　65 | 迭代器模式（上）：相比直接遍历集合数据，使用迭代器有哪些优势？

精选留言 (30)

写留言

张先生、

2020-03-30

关于课堂讨论，可以在接口和实现类中间加一层抽象类解决此问题，抽象类实现状态接口，状态类继承抽象类，只需要重写需要的方法即可

展开

11

J.D.
2020-03-30

Flutter里引入了Bloc框架后，就是非常典型的状态模式（或是有限状态机）。https://bloclibrary.dev/#/coreconcepts

展开

4

李小四
2020-03-30

设计模式_63:
作业
组合优于继承
- 即使不需要，也必须实现所有的函数
>>> 最小接口原则，每个函数拆分到单独的接口中...

展开

4

下雨天
2020-03-30

课后题
最小接口原则

具体做法:状态类只关心与自己相关的接口，将状态接口中定义的事件函数按事件分类，拆分到不同接口中，通过这些新接口的组合重新实现状态类即可！

展开

3

小晏子
2020-03-30

课后思考：要解决这个问题可以有两种方式1. 直接使用抽象类替代接口，抽象类中对每个时间有个默认的实现，比如抛出unimplemented exception，这样子类要使用的话必须自己实现。2. 就是还是使用接口定义事件，但是额外创建一个抽象类实现这个接口，然后具体的状态实现类继承这个抽象类，这种方式好处在于可扩展性强，可以处理将来有不相关的事件策略加入进来的情况。

展开

2

test
2020-03-30

课堂讨论：给新增的方法一个默认实现。

展开

2

Geek_Zjy
2020-03-31

课后作业，与过滤器上的解决方法一样:

……

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。...

展开

1

jaryoung
2020-03-31

代码中：

```
public enum State {
    SMALL(0),
    SUPER(1),
    FIRE(2),...
```

展开

1

Geek_54edc1
2020-03-30

思考题，可以用回调来替换接口，状态机类的方法增加一个回调对象的入参

1

Jxin
2020-03-30

1.解决方法的话，java可以用接口的def函数解决，也可以在实现类和接口间加一个def实现来过度。但这都是不好的设计。事实上接口def函数的实现是一种无奈之举，我们在使用接口时应依旧遵循其语意限制？而非滥用语言特性。

2.所以上诉解决方案，个人认为最好的方式就是细分接口包含的函数，对现有的函数重新...

展开

1

Frank
2020-03-30

打卡 今日学习状态模式，收获如下：
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。与策略模式一样，状态模式可以解决if-else或着switch-case分支逻辑过多的问题。同时也了解到了有限状态机的概念，以前在看一些资料时遇到这个概念，之前不太理解这个状态机时干嘛用的，通过今天的学习，理解了状态机就是一种数学模型，该模型...
展开

1

业余爱好者
2020-03-30

（一直觉得状态机是个非常高大上的东西，心中一直有疑问，今天才算是基本弄懂了。）

对于一个全局对象的依赖，当做方法参数传递是个不错的设计。像之前提到的servlet中的过滤器的过滤方法中，参数就有FilterChain这一对象。一个方法需要依赖（广义）一个对象，无非来自于对象属性和方法自身。前者叫做组合，后者叫做依赖。在接口设计中，...
展开

1 1

Tommy
2020-03-30

老师，状态机模式怎么防止状态回退呢？
展开

2 1

eason2017
2020-04-01

也可以基于jdk8的接口提供默认实现来做。
展开

Jesse
2020-04-01

方法一：可以使用将事件拆分成不同接口，不同的状态实现不同事件。
方法二：用java1.8 接口提供的default实现。
展开

jaryoung

2020-04-01

查表法中代码 this.score = actionTable[stateValue][eventValue]; // 61 line
是否修改成 this.score += actionTable[stateValue][eventValue]; 才是正确呢?


Bern
2020-04-01

添加一个默认抽象实现，空实现，所有状态类继承默认抽象实现。


Geek_3b1096
2020-04-01

查表法好用

展开


攻城拔寨
2020-03-31

多写个抽象类默认实现接口，实现类继承抽象类就行了

展开


南山
2020-03-31

1.接口增加默认设置，方法体中抛出 UnsupportOperateException
2.增加一个抽象类

涉及到多个状态转换的场景，状态机真的是可以提高代码的可读程度，也能保证状态的正常流转。

展开