

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？

王争 · 设计模式之美

状态模式

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？

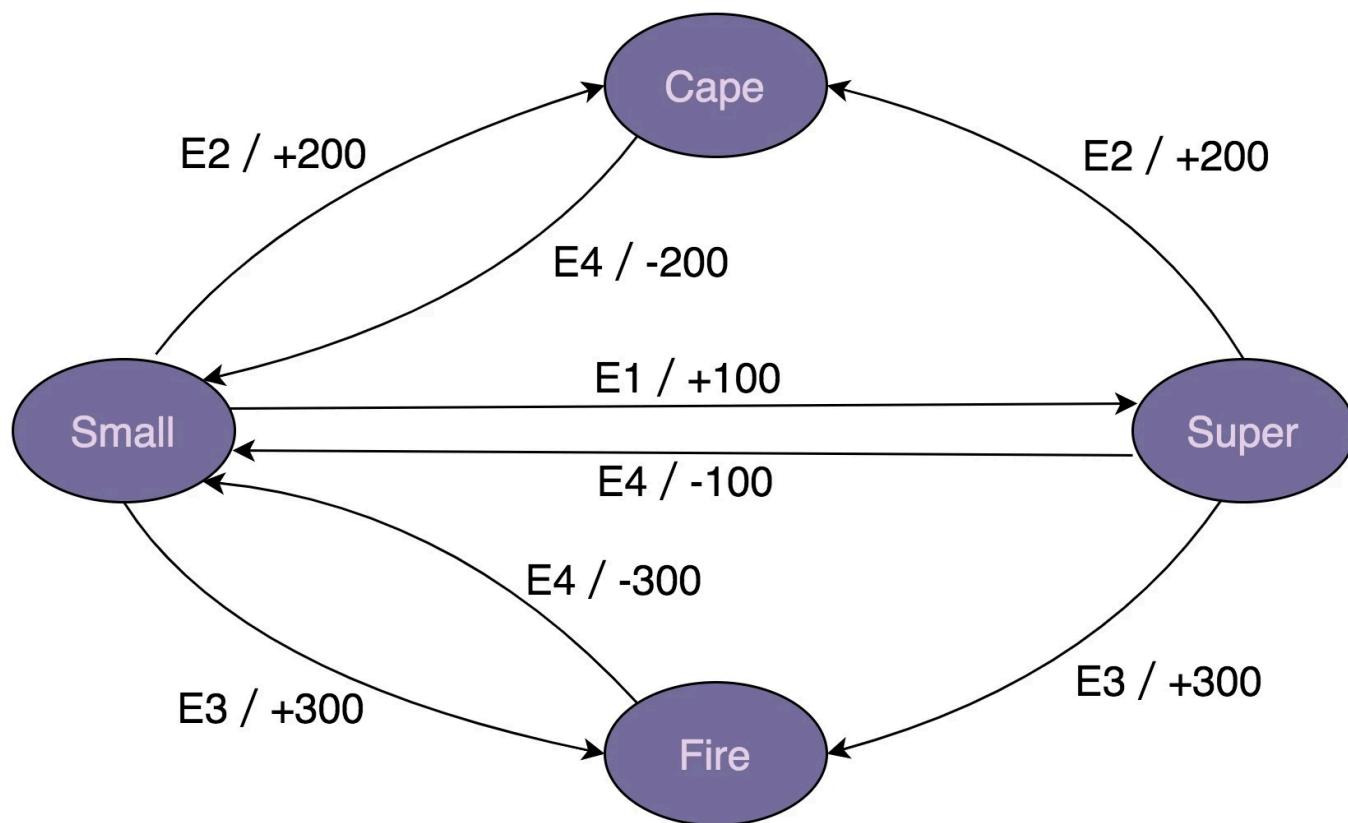
有限状态机，英文翻译是 Finite State Machine，缩写为 FSM，简称为状态机。状态机有 3 个组成部分：状态（State）、事件（Event）、动作（Action）。其中，事件也称为转移条件（Transition Condition）。事件触发状态的转移及动作的执行。不过，动作不是必须的，也可能只转移状态，不执行任何动作。

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



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

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


```
4     FIRE(2),
5     CAPE(3);
6
7     private int value;
8
9     private State(int value) {
10         this.value = value;
11     }
12
13     public int getValue() {
14         return this.value;
15     }
16 }
17
18 public class MarioStateMachine {
19     private int score;
20     private State currentState;
21
22     public MarioStateMachine() {
23         this.score = 0;
24         this.currentState = State.SMALL;
25     }
26
27     public void obtainMushRoom() {
28         //TODO
29     }
30
31     public void obtainCape() {
32         //TODO
33     }
34
35     public void obtainFireFlower() {
36         //TODO
37     }
38
39     public void meetMonster() {
40         //TODO
41     }
42
43     public int getScore() {
44         return this.score;
45     }
46
47     public State getCurrentState() {
48         return this.currentState;
49     }
50 }
51
52 public class ApplicationDemo {
```

```
53     public static void main(String[] args) {
54         MarioStateMachine mario = new MarioStateMachine();
55         mario.obtainMushRoom();
56         int score = mario.getScore();
57         State state = mario.getCurrentState();
58         System.out.println("mario score: " + score + "; state: " + state);
59     }
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：

复制代码

```
1 public enum Event {
2     GOT_MUSHROOM(0),
3     GOT_CAPE(1),
4     GOT_FIRE(2),
5     MET_MONSTER(3);
6
7     private int value;
8
9     private Event(int value) {
10         this.value = value;
11     }
12 }
```

```

13     public int getValue() {
14         return this.value;
15     }
16 }
17
18 public class MarioStateMachine {
19     private int score;
20     private State currentState;
21
22     private static final State[][] transitionTable = {
23         {SUPER, CAPE, FIRE, SMALL},
24         {SUPER, CAPE, FIRE, SMALL},
25         {CAPE, CAPE, CAPE, SMALL},
26         {FIRE, FIRE, FIRE, SMALL}
27     };
28
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

```
1 public interface IMario { //所有状态类的接口
```

```
2     State getName();
3     //以下是定义的事件
4     void obtainMushRoom();
5     void obtainCape();
6     void obtainFireFlower();
7     void meetMonster();
8 }
9
10 public class SmallMario implements IMario {
11     private MarioStateMachine stateMachine;
12
13     public SmallMario(MarioStateMachine stateMachine) {
14         this.stateMachine = stateMachine;
15     }
16
17     @Override
18     public State getName() {
19         return State.SMALL;
20     }
21
22     @Override
23     public void obtainMushRoom() {
24         stateMachine.setCurrentState(new SuperMario(stateMachine));
25         stateMachine.setScore(stateMachine.getScore() + 100);
26     }
27
28     @Override
29     public void obtainCape() {
30         stateMachine.setCurrentState(new CapeMario(stateMachine));
31         stateMachine.setScore(stateMachine.getScore() + 200);
32     }
33
34     @Override
35     public void obtainFireFlower() {
36         stateMachine.setCurrentState(new FireMario(stateMachine));
37         stateMachine.setScore(stateMachine.getScore() + 300);
38     }
39
40     @Override
41     public void meetMonster() {
42         // do nothing...
43     }
44 }
45
46 public class SuperMario implements IMario {
47     private MarioStateMachine stateMachine;
48
49     public SuperMario(MarioStateMachine stateMachine) {
50         this.stateMachine = stateMachine;
```

```
51     }
52
53     @Override
54     public State getName() {
55         return State.SUPER;
56     }
57
58     @Override
59     public void obtainMushRoom() {
60         // do nothing...
61     }
62
63     @Override
64     public void obtainCape() {
65         stateMachine.setCurrentState(new CapeMario(stateMachine));
66         stateMachine.setScore(stateMachine.getScore() + 200);
67     }
68
69     @Override
70     public void obtainFireFlower() {
71         stateMachine.setCurrentState(new FireMario(stateMachine));
72         stateMachine.setScore(stateMachine.getScore() + 300);
73     }
74
75     @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：

```

1  public interface IMario {
2      State getName();
3      void obtainMushRoom(MarioStateMachine stateMachine);
4      void obtainCape(MarioStateMachine stateMachine);
5      void obtainFireFlower(MarioStateMachine stateMachine);
6      void meetMonster(MarioStateMachine stateMachine);
7  }
8
9  public class SmallMario implements IMario {
10     private static final SmallMario instance = new SmallMario();
11     private SmallMario() {}
12     public static SmallMario getInstance() {
13         return instance;
14     }
15
16     @Override
17     public State getName() {
18         return State.SMALL;
19     }
20
21     @Override
22     public void obtainMushRoom(MarioStateMachine stateMachine) {
23         stateMachine.setCurrentState(SuperMario.getInstance());
24         stateMachine.setScore(stateMachine.getScore() + 100);
25     }
26
27     @Override
28     public void obtainCape(MarioStateMachine stateMachine) {
29         stateMachine.setCurrentState(CapeMario.getInstance());
30         stateMachine.setScore(stateMachine.getScore() + 200);
31     }
32
33     @Override
34     public void obtainFireFlower(MarioStateMachine stateMachine) {
35         stateMachine.setCurrentState(FireMario.getInstance());
36         stateMachine.setScore(stateMachine.getScore() + 300);
37     }
38
39     @Override
40     public void meetMonster(MarioStateMachine stateMachine) {
41         // do nothing...
42     }
43 }
44
45 // 省略SuperMario、CapeMario、FireMario类...
46
47 public class MarioStateMachine {

```

```
48     private int score;
49     private IMario currentState;
50
51     public MarioStateMachine() {
52         this.score = 0;
53         this.currentState = SmallMario.getInstance();
54     }
55
56     public void obtainMushRoom() {
57         this.currentState.obtainMushRoom(this);
58     }
59
60     public void obtainCape() {
61         this.currentState.obtainCape(this);
62     }
63
64     public void obtainFireFlower() {
65         this.currentState.obtainFireFlower(this);
66     }
67
68     public void meetMonster() {
69         this.currentState.meetMonster(this);
70     }
71
72     public int getScore() {
73         return this.score;
74     }
75
76     public State getCurrentState() {
77         return this.currentState.getName();
78     }
79
80     public void setScore(int score) {
81         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。

AI智能总结

本文介绍了游戏中常用的状态机的实现方式，以及状态机的基本概念。作者首先介绍了有限状态机的组成部分：状态、事件、动作，并通过“超级马里奥”游戏的例子解释了状态机的概念。接着，作者提出了三种状态机的实现方式：分支逻辑法、查表法和状态模式。在分支逻辑法中，作者通过代码示例展示了如何使用if-else或switch-case分支判断逻辑来实现状态机，但指出了其可读性和可维护性较差的问题。而在查表法中，作者介

绍了利用二维表来表示状态机的实现方式，并展示了相应的代码示例，强调了其代码实现更加清晰、可读性和可维护性更好的优点。最后，作者提到了状态模式的实现方式，并对比了这三种实现方式的优劣和应用场景。通过本文，读者可以快速了解状态机的基本概念和不同的实现方式，以及它们的优缺点和适用场景。文章内容深入浅出，适合技术人员了解状态机的实现方式及其应用。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (95)

最新 精选



DFighting

2020-11-21

不太赞同老师的这种抽象方式，状态机中的每个状态应该具有事件、触发条件、转移（方法）列表，每一个应该都可以抽象为一个接口或者泛型类，状态机作为一个单例这个没问题，但是状态机应该只是作为一个状态的注册工厂，里面具有的应该是多种状态，状态间的流转才是状态机最重要的功能抽象。score放在状态和状态机中都不合适，这应该是状态机操纵的一个对象/资源，应该单独抽象出来，在状态间流转

作者回复: 好吧

共 4 条评论 >

👍 29



进击的巨人

2020-11-27

感觉状态模式就是一个策略模式

作者回复: 嗯嗯

共 2 条评论 >

👍 2



吃饭睡觉打酱油

2020-06-20

老师，在使用状态机的时候，初始状态应该是可以支持初始化的吧。

作者回复: 嗯嗯



1



Geek_78eadb

2020-11-28

课后题：

1. 使用抽象工厂模式实现 Action 类：即根据不同的状态实现 ObtainMushRoom 等动作类，比如 ObtainMushRoom 的初始化需要传入目前状态和目前分数，ObtainMushRoom 可以利用状态机的查表法进行状态转移和其它动作

作者回复： 嗯嗯 🤔🤔🤔🤔🤔🤔

共 2 条评论 >



打工人233号

2020-11-13

如果一种状态转移对应多种状态如何处理呢？

作者回复： 这个好像没有这种情况吧



慕枫技术笔记

2020-07-29

查表法在状态新增的情况下怎么做到不修改代码的？

作者回复： 没法做到一点都不修改代码的哦

共 3 条评论 >



pippin

2020-04-13

查表法的这个数组有问题吧，transitionTable[0][0]应该是SMALL吧。这里应该是写错了吧。

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

作者回复: 没有啊, 你看看上面那个表格

共 2 条评论 >



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

共 6 条评论 >

153



Darren

2020-04-14

Java中Spring也有有限状态机的框架: <https://github.com/spring-projects/spring-state-machine>



57



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!

共 2 条评论 >

34