

25 | 行为型：模版、策略和状态模式有什么区别？

2022-11-15 石川 来自北京

天下无鱼
<https://shikey.com/>

《JavaScript进阶实战课》

[课程介绍 >](#)



讲述：石川

时长 08:18 大小 7.57M



你好，我是石川。

今天我们来说说设计模式中剩下的几种行为型模式。我个人觉得剩下这六种模式可以大致分为两类，一类是偏向“策略模型”的设计模式，这里包含了策略、状态和模版这三种模式。另外一大类是偏向“数据传递”的设计模式，这里就包含了中介、命令和职责链这几种模式。这些类别的模式，有什么共同和不同呢？我们先从它们各自的思想 and 实现来看看。

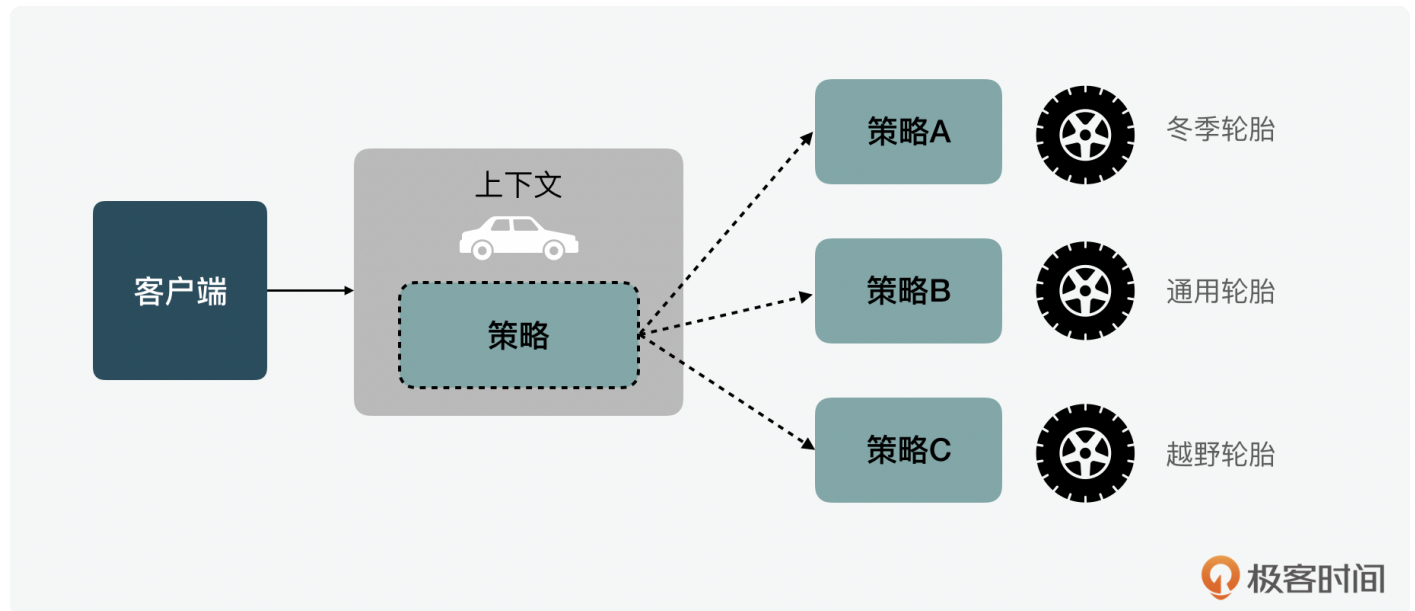
策略模型类的行为模式

首先，我们来看看策略、状态和模版这三种偏向“策略模型”的设计模式吧。

策略模式

先说策略模式（strategy），它的核心思想是**在运行时基于场景选择策略**。

我们可以举一个例子，我们的汽车轮胎适配就算是一种策略模式，比如在冰天雪地的西伯利亚，可以选择冬季轮胎；如果是平时用的买菜车，就选择普通轮胎；如果是去草原狂奔，就换上越野轮胎。



下面，我们可以通过一个红绿灯程序来看一下这一概念的实现。在这个例子中，我们可以看到交通控制（**TrafficControl**）就决定了运行时环境的上下文，它可以通过转换（**turn**）这个方法来切换不同的策略。红绿灯（**TrafficLight**）是一个抽象类的策略，它可以根据环境需要，延伸出具体类的策略。

 复制代码

```
1 // encapsulation
2 class TrafficControl {
3     turn(trafficlight) {
4         return trafficlight.trafficcolor();
5     }
6 }
7
8 class TrafficLight {
9     trafficcolor() {
10         return this.colorDesc;
11     }
12 }
13
14 // strategy 1
15 class RedLight extends TrafficLight {
16     constructor() {
17         super();
18         this.colorDesc = "Stop";
19     }
20 }
21
```

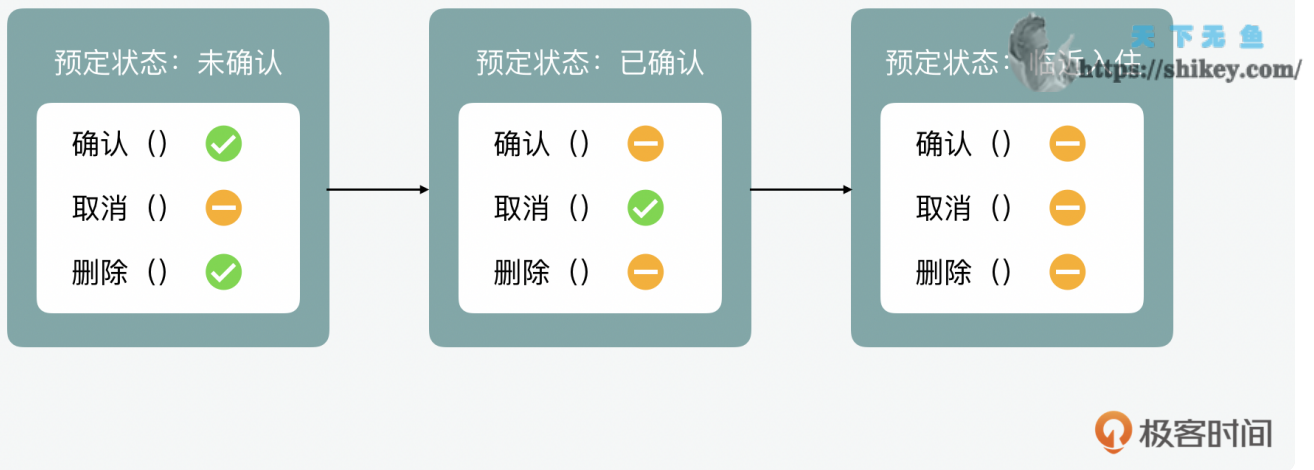
```
22 // strategy 2
23 class YellowLight extends TrafficLight {
24     constructor() {
25         super();
26         this.colorDesc = "Wait";
27     }
28 }
29
30 // strategy 3
31 class GreenLight extends TrafficLight {
32     constructor() {
33         super();
34         this.colorDesc = "Go";
35     }
36 }
37
38 // usage
39 var trafficControl = new TrafficControl();
40
41 console.log(trafficControl.turn(new RedLight())); // Stop
42 console.log(trafficControl.turn(new YellowLight())); // Wait
43 console.log(trafficControl.turn(new GreenLight())); // Go
```



状态模式

下面我们再看看状态模式（state），它的核心概念是**根据运行时状态的不同，切换不同的策略**。所以我们可以说它是策略模式的一个延伸。

这里，我们可以拿酒店预定举个例子，比如我们都有在一些文旅类门户网站上预定酒店的经验。在预定的时候，通常有几种不同的状态，比如当我们下单支付前，订单状态可能是“未确认”，这时我们可以确认或删除，但是因为还没有预定成功，所以没有取消的选项。但是当我们已确认并完成支付，就没有再次确认或删除的动作了，这时，我们只能选择取消。再然后，一般很多酒店都规定只能在入住前 24 小时选择取消，而如果在临近入住的 24 小时之内，那么在这个区间内连取消的按钮可能都失效了。这时，我们只能选择入住或和客服沟通取消。这就是状态模式，也就是说程序依据不同运行时状态，做不同的策略反应。



同样，我们可以通过讲策略模式时的红绿灯案例做一些改造，加入状态 `state`，看看会发生什么。这里，我们可以看到每次当我们执行 `turn` 在做切换的时候，随着状态在红、黄、绿三种状态之间循环更新，红绿灯的指示也跟着更新。

复制代码

```
1 class TrafficControl {
2   constructor() {
3     this.states = [new GreenLight(), new RedLight(), new YellowLight()];
4     this.current = this.states[0];
5   }
6   turn() {
7     const totalStates = this.states.length;
8     let currentIndex = this.states.findIndex(light => light === this.current);
9     if (currentIndex + 1 < totalStates) this.current = this.states[currentIndex + 1];
10    else this.current = this.states[0];
11  }
12  desc() {
13    return this.current.desc();
14  }
15 }
16
17 class TrafficLight {
18   constructor(light) {
19     this.light = light;
20   }
21 }
22
23 class RedLight extends TrafficLight {
24   constructor() {
25     super('red');
26   }
27   desc() {
28     return 'Stop';
29   }
30 }
```

```
30 }
31
32 class YellowLight extends TrafficLight {
33     constructor() {
34         super('yellow');
35     }
36     desc() {
37         return 'Wait';
38     }
39 }
40
41 class GreenLight extends TrafficLight {
42     constructor() {
43         super('green');
44     }
45     desc() {
46         return 'Go';
47     }
48 }
49
50 // usage
51 var trafficControl = new TrafficControl();
52 console.log(trafficControl.desc()); // 'Go'
53 trafficControl.turn();
54 console.log(trafficControl.desc()); // 'Stop'
55 trafficControl.turn();
56 console.log(trafficControl.desc()); // 'Wait'
```



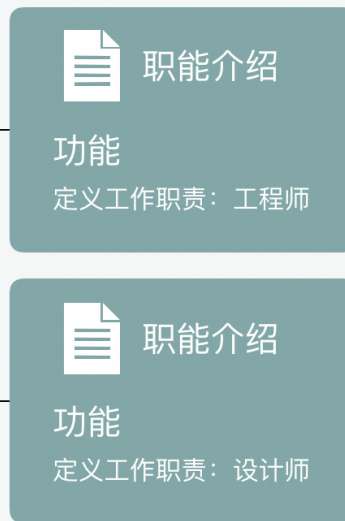
模版模式

最后，我们再来看看模版模式（template）。它的核心思想是在一个方法中定义一个业务逻辑模版，并将某些步骤推迟到子类中实现。所以它和策略模式有些类似。

抽象类



实体类



实例



下面我们可以看一个实现的例子。在这个例子里，我们看到员工 `employee` 里的工作 `work` 就是一个模版，它里面的任务 `tasks` 是延迟到开发 `developer` 和设计 `designer` 两个子类中去实现的。这就是一个简单的模版模式的设计实现。

复制代码

```
1 class Employee {
2   constructor(name, salary) {
3     this.name = name;
4     this.salary = salary;
5   }
6   work() {
7     return `${this.name}负责${this.tasks()}`;
8   }
9   getPaid() {
10    return `${this.name}薪资是${this.salary}`;
11  }
12 }
13
14 class Developer extends Employee {
15   constructor(name, salary) {
16     super(name, salary);
17   }
18   // 细节由子类实现
19   tasks() {
20     return '写代码';
21   }
22 }
23
24 class Designer extends Employee {
25   constructor(name, salary) {
```

```
26     super(name, salary);
27 }
28 // 细节由子类实现
29 tasks() {
30     return '做设计';
31 }
32 }
33
34 // usage
35 var dev = new Developer('张三', 10000);
36 console.log(dev.getPaid()); // '张三薪资是10000'
37 console.log(dev.work()); // '张三负责写代码'
38 var designer = new Designer('李四', 11000);
39 console.log(designer.getPaid()); // '李四薪资是11000'
40 console.log(designer.work()); // '李四负责做设计'
```

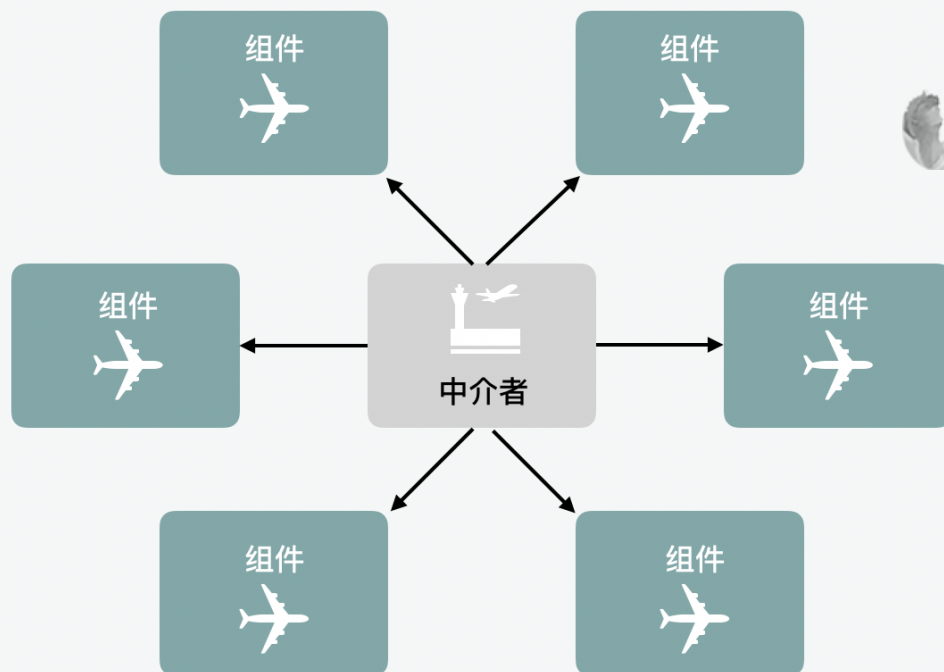


这里我先做个阶段性小结，从上面的例子中，我们可以看出，无论是策略、状态还是模版模式，它们都是基于某种“策略模型”来实现的。比如策略模式中的策略是基于上行文来切换；在状态模式中是根据状态来做切换；而最后在模版模式的例子中，某些策略模版在父类中定义，有些则在子类中实现。

信息传递类的行为模式

中介模式

中介者（mediator）模式的核心是**使组件可以通过一个中心点相互交互**。现实生活中，航空地面塔台就是一个例子，我们不可能让飞机之间交谈，而是通过地面控制台协调。地面塔台人员需要确保所有飞机都接收到安全飞行所需的信息，而不会撞到其他飞机。



我们还是通过一段代码，来看看这种模式的实现。塔台（**TrafficTower**）有着接收每架飞机坐标和获取某架飞机坐标方法。同时，飞机会登记自己的坐标和获取其它飞机的坐标。这些信息都是统一由塔台（**TrafficTower**）来管理的。

复制代码

```
1 class TrafficTower {
2   #airplanes;
3   constructor() {
4     this.#airplanes = [];
5   }
6
7   register(airplane) {
8     this.#airplanes.push(airplane);
9     airplane.register(this);
10  }
11
12  requestCoordinates(airplane) {
13    return this.#airplanes.filter(plane => airplane !== plane).map(plane => pla
14  }
15 }
16
17 class Airplane {
18   constructor(coordinates) {
19     this.coordinates = coordinates;
20     this.trafficTower = null;
21   }
22
23   register(trafficTower) {
24     this.trafficTower = trafficTower;
```



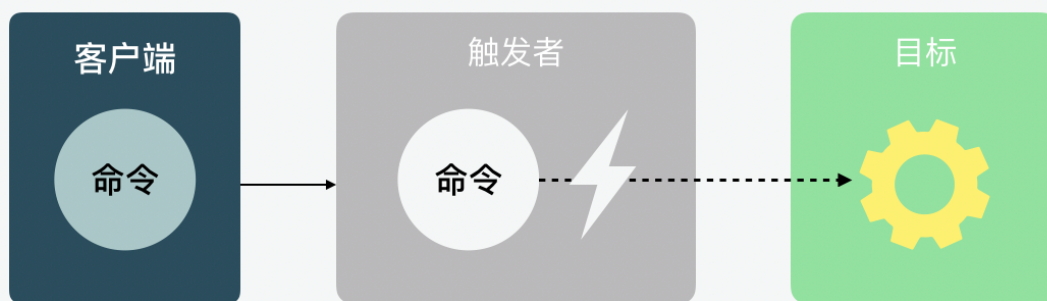
```

25     }
26
27     requestCoordinates() {
28         if (this.trafficTower) return this.trafficTower.requestCoordinates(this);
29         return null;
30     }
31 }
32
33 // usage
34 var tower = new TrafficTower();
35
36 var airplanes = [new Airplane(10), new Airplane(20), new Airplane(30)];
37 airplanes.forEach(airplane => {
38     tower.register(airplane);
39 });
40
41 console.log(airplanes.map(airplane => airplane.requestCoordinates()))
42 // [[20, 30], [10, 30], [10, 20]]

```

命令模式

说完中介模式，我们再来看看命令模式，命令模式（command）允许我们**将命令和发起命令操作的对象分离**，这么做的好处是对于处理具有特定生命周期或者列队执行的命令，它会给我们更多的控制权。并且它还提供了将方法调用作为传参的能力，这样做的好处是可以让方法按需执行。



下面我们可以看看这种模式的样例。事务管理者 `OperationManager` 接到了执行任务，会根据不同的命令，如启动行动（`StartOperationCommand`）、追踪行动状态（`TrackOperationCommand`）及取消行动 `CancelOperationCommand` 等来执行。

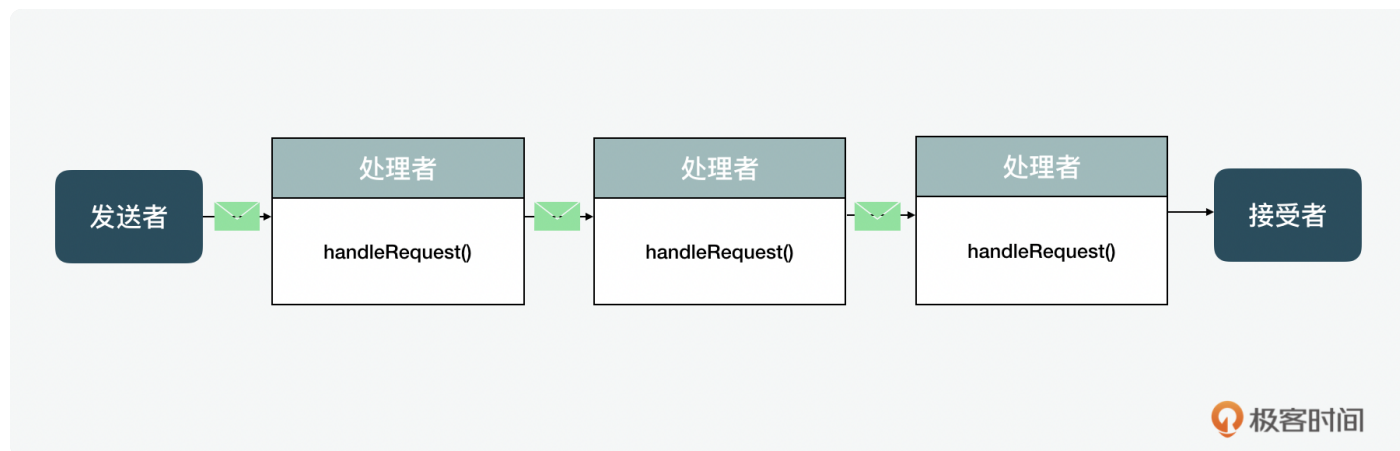


```
1 class OperationManager {
2   constructor() {
3     this.operations = [];
4   }
5
6   execute(command, ...args) {
7     return command.execute(this.operations, ...args);
8   }
9 }
10
11 class Command {
12   constructor(execute) {
13     this.execute = execute;
14   }
15 }
16
17 function StartOperationCommand(operation, id) {
18   return new Command(operations => {
19     operations.push(id);
20     console.log(`你成功的启动了${operation}行动，代号${id}`);
21   });
22 }
23
24 function CancelOperationCommand(id) {
25   return new Command(operations => {
26     operations = operations.filter(operation => operation.id !== id);
27     console.log(`你取消了行动代号${id}`);
28   });
29 }
30
31 function TrackOperationCommand(id) {
32   return new Command(() =>
33     console.log(`你的行动代号${id}，目前正在执行中`)
34   );
35 }
36
37 var manager = new OperationManager();
38
39 manager.execute(new StartOperationCommand("猎豹", "318"));
40 // 返回：你成功的启动了猎豹行动，代号318
41 manager.execute(new TrackOperationCommand("318"));
42 // 返回：你的行动代号318，目前正在执行中
43 manager.execute(new CancelOperationCommand("318"));
44 // 返回：你取消了行动代号318
```

命令模式可以在许多不同的情况下使用，特别是在创建重交互的 UI 上，比如编辑器里撤销的操作，因为它可以让 UI 对象和行为操作做到高度解耦。这种模式也可以用来代替回调函数，这也是因为它更支持模块化地将行为操作在对象之间传递。

职责链模式

最后，再来看下职责链模式，职责链模式（chain of responsibility）核心是**将请求的发送者和接收者解耦**。它的实现是通过一个对象链，链中的每个对象都可以处理请求或将其传递给下一个对象。其实在我们前面讲享元时，就提到过事件捕获和冒泡，JavaScript 内部就是用这个方式来处理事件捕获和冒泡的。同样在享元例子中，我们也提到过，jQuery 是通过职责链每次返回一个对象来做到的链接式调用。



那么这种职责链是如何实现的呢？其实它的实现并不复杂，通过下面的例子我们可以看一下。你也可以很容易实现一个简化版的链式累加。我们通过累加（CumulativeSum）中的加法（add）可以循环上一个对象的结果和参数相加后的结果，作为返回值传给下一个方法。

复制代码

```
1 class CumulativeSum {
2   constructor(intialValue = 0) {
3     this.sum = intialValue;
4   }
5
6   add(value) {
7     this.sum += value;
8     return this;
9   }
10 }
11
12 // usage
13 var sum = new CumulativeSum();
14 console.log(sum.add(10).add(2).add(50).sum); // 62
```

通过上面的三种模式的例子，我们都可以看到数据在不同对象中的传递。中介模式中，我们需要在网状的环境中，信息对多个对象中通过中介进行传输；命令模式中，我们看到了信息在对

象和对象之间的传输；而最后，在职责链的模式中，我们又看到了信息在一个流水线中的传输。因此我说它们是偏向“数据传递”的设计模式。




总结

今天，我带你看了几种不同的行为型设计模式。到现在为止，我们所有的经典模式就都讲完了。


这一讲我们看的这些模式除了在 JavaScript 中会用到以外，在多数其它语言中也都适用，所以算是比较脱离语言本身的几种“普世”模式了。在之后的一讲中，我们会再次看几种在 JavaScript 中特有的一些设计模式。

思考题

如果你用过 Redux 的话，应该用过它的  开发者工具 中的时间旅行式调试，它可以将应用程序的状态向前、向后或移动到任意时间点。你知道这个功能的实现用到了今天学到的哪（些）种行为型设计模式吗？

欢迎在留言区分享你的答案、交流学习心得或者提出问题，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。我们下期再见！

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 0  提建议

© 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。

上一篇 24 | 行为型：通过观察者、迭代器模式看JS异步回调

下一篇 26 | 特殊型：前端有哪些处理加载和渲染的特殊“模式”？



Vue3 企业级项目实战课

进阶高手的 Vue3+Node.js 全栈开发训练

杨文坚

前阿里前端 leader

前腾讯 IMWeb 团队高级前端工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。