9 命令模式

假设有一个快餐店，而我是服务员，那么客人点餐时候我的工作就是把客人需求写在清单上交给厨房，客人不需要关心他的菜是谁炒的，当时它还可以支持预定上菜或者撤销订单，只要有订单在就可以满足客人需求

> 💡 这些记录着订餐信息的清单，便是命令模式中的命令对象

命令模式的用途

命令模式是最简单和优雅的模式之一，其中的命令（command）指的是一个执行某些特定事情的指令

常见的应用场景：

- 需要向某些对象发送请求实现自己的诉求
- 但是不知道明确的接收者与被请求的操作是什么
- 松耦合的设计使得请求者和请求接收者可以解耦

相对于偏向过程化的请求调用，command对象拥有更长的生命周期，所以我们可以更加灵活的再任意时刻执行这个command，例如预定1个小时之后炒菜

例子-菜单程序

这里有一个需求：

- 编写一个用户界面，至少数十个Button
- 复杂，需要分工，一个程序员负责绘制按钮和UI，另一个负责编写按钮点击逻辑

这时候我们必须解开按钮和负责行为对象之间的耦合，不变的是click之后执行逻辑，变化的是执行的具体逻辑，这里我们就可以借助命令对象的帮助来尝试实现：

```
/**
 * 菜单程序例子
 */
var button1 = document.getElementById('button1');
var button2 = document.getElementById('button2');
var button3 = document.getElementById('button3');

// 定义setCommand函数 - 负责安装命令 - 封装不变的部分
```

```
var setCommand = function(button, command){
    button.onclick = function(){
        command.execute();
    }
}

// 负责编写按钮点击逻辑的程序员实现逻辑功能
var MenuBar = {
    refresh: function(){
        console.log('刷新菜单目录');
    }
}

var SubMenu = {
    add: function(){
        console.log('增加子菜单');
    },
    delete: function(){
        console.log('删除子菜单');
    }
}

// 封装行为到命令类中
var RefreshMenuBarCommnad = function(receiver){
    this.receiver = receiver;
}

var AddSubMenuBarCommnad = function(receiver){
    this.receiver = receiver;
}

var DeleteSubMenuBarCommnad = function(receiver){
    this.receiver = receiver;
}

RefreshMenuBarCommnad.prototype.execute = function(){
    this.receiver.refresh();
}

AddSubMenuBarCommnad.prototype.execute = function(){
    this.receiver.add();
}

DeleteSubMenuBarCommnad.prototype.execute = function(){
    this.receiver.delete();
}

```

```
// 把命令接收者传入command对象，并且安装command到button上
var refreshMenuBarCommnad = new RefreshMenuBarCommnad(MenuBar);
var addSubMenuBarCommnad = new AddSubMenuBarCommnad(SubMenu);
var deleteSubMenuBarCommnad = new DeleteSubMenuBarCommnad(SubMenu);

setCommand(button1, refreshMenuBarCommnad);
setCommand(button2, addSubMenuBarCommnad);
setCommand(button3, deleteSubMenuBarCommnad);
```

Javascript中的命令模式

上面的命令模式可能让你感觉奇怪，无非是把对象的某个方法取了execute的名字，还引入了command和receiver这两个把事情复杂化的对象，用下面的代码也可以实现相同的功能：

```
/**
 * 简单的实现
 */

var bindClick = function(button,func){
    button.onclick = func;
}

var MenuBar = {
    refresh: function(){
        console.log('刷新')
    }
}

var SubMenu = {
    add: function(){},
    delete: function(){}
}

bindClick(button1, MenuBar.refresh);
bindClick(button2, SubMenu.add);
```

上述的做法是正确的，因为第二节中是模拟面向对象语言的命令模式的实现，将请求封装在command对象的execute方法中并通过传递command对象来调用命令，客户不关心事情具体如何进行的

⚽ 命令模式的由来，其实是回调函数（callback）的一个面向对象的替代品

所以在把函数作为一等对象的JS中，跟策略模式一样，命令模式本身就已经融入到了JS语言中，运算块不必封装在command的execute方法中，也可以封装在普通函数中来自然的四处传递

这里我们再来看看基于闭包实现的命令模式，将接收者封闭在闭包环境中，执行命令其实就是执行存下来的execute函数：

```
/**
 * 基于闭包的实现
 */
var setCommand = function(button,command){
    button.onclick = function(){
        command.execute();
    }
}

var MenuBar = {
    refresh: function(){
        console.log('刷新')
    }
}

var RefreshMenuBarCommnad = function(receiver){
    return {
        execute: function(){
            receiver.refresh();
        }
    }
}

var refreshMenuBarCommnad = RefreshMenuBarCommnad(MenuBar);
setCommand(button1, refreshMenuBarCommnad);
```

撤销命令

命令模式有一个关键作用是很方便地给命令对象增加撤销操作。类似客人通过电话取消订餐一样。

先看下我们的目标，给Animate类增加动画，来让小球水平移动，一个input输入框输入移动后的位置，一个button按钮触发移动，接着还需要让小球还原到开始之前的位置，我们看一下基础实现：

```
/**
 * 小球移动Demo
 */

```

```
var ball = document.getElementById('ball');
var pos = document.getElementById('pos');
var moveBtn = document.getElementById('moveBtn');

moveBtn.onclick = function(){
    var animate = new Animate(ball);
    animate.start('left',pos.value, 1000, 'strongEaseOut');
}
```

如果希望撤回，就输入一个-200再点击button，但是似乎有点蠢，我们看一下如何借助命令模式+撤销功能来实现：

```
/**
 * 命令模式+撤销
 */
var ball = document.getElementById('ball');
var pos = document.getElementById('pos');
var moveBtn = document.getElementById('moveBtn');
var cancelBtn = document.getElementById('cancelBtn');

var MoveCommand = function(receiver, pos){
    this.receiver = receiver;
    this.pos = pos;
    this.oldPos = null;
}

MoveCommand.prototype.execute = function(){
    this.receiver.start('left',this.pos, 1000, 'strongEaseOut');
    this.oldPos = this.receiver.dom.getBoundingClientRect()[this.receiver.proper
}

MoveCommand.prototype.undo = function(){
    this.receiver.start('left',this.oldPos, 1000, 'strongEaseOut');
}

var moveCommand;

moveBtn.onclick = function(){
    var animate = new Animate(ball);
    moveCommand = new MoveCommand(animate, pos.value);
    moveCommand.execute();
}

cancelBtn.onclick = function(){
    moveCommand.undo();
```

```
34  }
```

上述方法通过命令模式实现了撤销，需要撤销时记录的原始位置在执行execute时候就已经记录下来了，只需要提供一个undo方法实现即可

撤销和重做

有的时候我们可能会需要撤销一系列的命令，此外我们有时候也可能无法通过undo操作回到execute之前的状态，需要从0开始执行部分已经执行过的命令，即重做操作。

这时候最好的方法就是记录一个操作历史堆栈，记录命令日志，然后可以方便的撤销一系列命令或者从0执行一遍，代码我就不贴了，核心就是在执行命令的时候记录:

```
1  command();
2  commandStack.push(command);
```

命令队列

例如订餐场景中，有时候出现订单过多而厨师不够情况，这时候就需要让订单排队处理，即第一个订单完成之后再执行第二个订单相关的操作

队列在动画场景中应用也很多，例如有一系列动画是具有前后的关系的，都需要排队进行

所以这里将请求封装成command的优势就体现了出来，因为此时command生命周期几乎是永久的，除非我们主动回收。我们可以将有队列需求的命令压入队列堆栈中，当第一段动画命令执行完成之后 -> 通知队列堆栈 -> 取下一个命令执行，即使中间有冗余用户操作发生，但是不会影响对象以及生命周期

这里需要关注的主要是通知机制，一般可以通过callback或者发布-订阅模式实现即可

宏命令

宏命令是一组命令的集合，一次可以执行一批命令。可以想象一下家里有一个万能遥控器，每天回家时候只需要按一个按钮，就可以执行关上房间门，打开电脑，登录QQ的一系列操作

我们看一个实现的例子：

```
1  /**
2   * 宏命令
3   */
4  let closeDoorCommand = {
```

```
    execute: () => {
        console.log('关门');
    }
}

let openPCCommand = {
    execute: () => {
        console.log('打开电脑');
    }
}

let openQQCommand = {
    execute: () => {
        console.log('打开QQ');
    }
}

const MacroCommand = () => {
    return {
        commandsList: [],
        add: (command) => {
            this.commandsList.push(command);
        },
        execute: () => {
            for(let i = 0, command; command = this.commandsList[i++];){
                command.execute();
            }
        }
    }
}

const macroCommand = MacroCommand();
macroCommand.add(closeDoorCommand);
macroCommand.add(openPCCommand);
macroCommand.add(openQQCommand);
macroCommand.execute();
```

当然我们也可以用类似思路实现宏命令的undo等，宏命令其实是命令模式和组合模式的产物

智能命令与傻瓜命令

```
let closeDoorCommand = {
    execute: () => {
        console.log('关门');
```

```
    }
}
```

上面这段command其实你分析会发现和之前不一样，因为内部没有包含receiver，对比之前：

```
var MoveCommand = function(receiver, pos){
    this.receiver = receiver;
    this.pos = pos;
    this.oldPos = null;
}

MoveCommand.prototype.execute = function(){
    this.receiver.start('left',this.pos, 1000, 'strongEaseOut');
    this.oldPos = this.receiver.dom.getBoundingClientRect()[this.receiver.proper
}
```

包含receiver的命名对象一般称为傻瓜式的，因为它只负责把客户请求转给receiver执行就完事了，优势在于发起者和接收者之间尽可能解耦。

但是我们也可以定义一些更加“聪明”的命令对象，让其直接实现请求，例如上面的closeDoorCommand，这种没有接收者的智能命令其实退化到了策略模式非常相近，代码结构上甚至几乎无法分辨。策略模式一般来说针对的问题域更加小和聚焦，目标是一致的，而智能命令模式指向问题域更广，具有发散性。

小结

JS其实可以用高阶函数方便实现命令模式，命令模式其实在JS中是一种隐形的模式