第 9 章　命令模式

假设有一个快餐店，而我是该餐厅的点餐服务员，那么我一天的工作应该是这样的：当某位客人点餐或者打来订餐电话后，我会把他的需求都写在清单上，然后交给厨房，客人不用关心是哪些厨师帮他炒菜。我们餐厅还可以满足客人需要的定时服务，比如客人可能当前正在回家的路上，要求1个小时后才开始炒他的菜，只要订单还在，厨师就不会忘记。客人也可以很方便地打电话来撤销订单。另外如果有太多的客人点餐，厨房可以按照订单的顺序排队炒菜。

这些记录着订餐信息的清单，便是命令模式中的命令对象。

9.1 命令模式的用途

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

命令模式最常见的应用场景是：有时候需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是什么。此时希望用一种松耦合的方式来设计程序，使得请求发送者和请求接收者能够消除彼此之间的耦合关系。

拿订餐来说，客人需要向厨师发送请求，但是完全不知道这些厨师的名字和联系方式，也不知道厨师炒菜的方式和步骤。 命令模式把客人订餐的请求封装成command 对象，也就是订餐中的订单对象。这个对象可以在程序中被四处传递，就像订单可以从服务员手中传到厨师的手中。这样一来，客人不需要知道厨师的名字，从而解开了请求调用者和请求接收者之间的耦合关系。

另外，相对于过程化的请求调用，command 对象拥有更长的生命周期。对象的生命周期是跟初始请求无关的，因为这个请求已经被封装在了command 对象的方法中，成为了这个对象的行为。我们可以在程序运行的任意时刻去调用这个方法，就像厨师可以在客人预定1个小时之后才帮他炒菜，相当于程序在1个小时之后才开始执行command 对象的方法。

除了这两点之外，命令模式还支持撤销、排队等操作，本章稍后将会详细讲解。

9.2 命令模式的例子——菜单程序

假设我们正在编写一个用户界面程序，该用户界面上至少有数十个Button按钮。因为项目比较复杂，所以我们决定让某个程序员负责绘制这些按钮，而另外一些程序员则负责编写点击按钮后的具体行为，这些行为都将被封装在对象里。

在大型项目开发中，这是很正常的分工。对于绘制按钮的程序员来说，他完全不知道某个按钮未来将用来做什么，可能用来刷新菜单界面，也可能用来增加一些子菜单，他只知道点击这个按钮会发生某些事情。那么当完成这个按钮的绘制之后，应该如何给它绑定onclick 事件呢?

回想一下命令模式的应用场景:

> 有时候需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是什么，此时希望用一种松耦合的方式来设计软件，使得请求发送者和请求接收者能够消除彼此之间的耦合关系。

我们很快可以找到在这里运用命令模式的理由：点击了按钮之后，必须向某些负责具体行为的对象发送请求，这些对象就是请求的接收者。但是目前并不知道接收者是什么对象，也不知道接收者究竟会做什么。此时我们需要借助命令对象的帮助，以便解开按钮和负责具体行为对象之间的耦合。

设计模式的主题总是把不变的事物和变化的事物分离开来，命令模式也不例外。按下按钮之后会发生一些事情是不变的，而具体会发生什么事情是可变的。通过command 对象的帮助，将来我们可以轻易地改变这种关联，因此也可以在将来再次改变按钮的行为。

下面进入代码编写阶段，首先在页面中完成这些按钮的“绘制”:

```
<body>
    <button id="button1">点击按钮1</button>
    <button id="button2">点击按钮2</button>
    <button id="button3">点击按钮3</button>
</body>

<script>
```

```
    var button1 = document.getElementById( 'button1' ),
    var button2 = document.getElementById( 'button2' ),
    var button3 = document.getElementById( 'button3' );
</script>
```

接下来定义setCommand 函数，setCommand 函数负责往按钮上面安装命令。可以肯定的是，点击按钮会执行某个command 命令，执行命令的动作被约定为调用command 对象的execute() 方法。虽然还不知道这些命令究竟代表什么操作，但负责绘制按钮的程序员不关心这些事情，他只需要预留好安装命令的接口，command 对象自然知道如何和正确的对象沟通：

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

最后，负责编写点击按钮之后的具体行为的程序员总算交上了他们的成果，他们完成了刷新菜单界面、增加子菜单和删除子菜单这几个功能，这几个功能被分布在MenuBar 和SubMenu 这两个对象中：

```
var MenuBar = {
    refresh: function(){
        console.log( '刷新菜单目录' );
    }
};

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

在让button 变得有用起来之前，我们要先把这些行为都封装在命令类中：

```
var RefreshMenuBarCommand = function( receiver ){
    this.receiver = receiver;
};

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

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

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

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

DelSubMenuCommand.prototype.execute = function(){
    console.log( '删除子菜单' );
};
```

最后就是把命令接收者传入到command 对象中，并且把command 对象安装到button 上面：

```
var refreshMenuBarCommand = new RefreshMenuBarCommand( MenuBar );
var addSubMenuCommand = new AddSubMenuCommand( SubMenu );
var delSubMenuCommand = new DelSubMenuCommand( SubMenu );

setCommand( button1, refreshMenuBarCommand );
setCommand( button2, addSubMenuCommand );
setCommand( button3, delSubMenuCommand );
```

以上只是一个很简单的命令模式示例，但从中可以看到我们是如何把请求发送者和请求接收者解耦开的。

9.3 JavaScript中的命令模式

也许我们会感到很奇怪，所谓的命令模式，看起来就是给对象的某个方法取了execute 的名字。引入command 对象和receiver 这两个无中生有的角色无非是把简单的事情复杂化了，即使不用什么模式，用下面寥寥几行代码就可以实现相同的功能：

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

var MenuBar = {
    refresh: function(){
        console.log( '刷新菜单界面' );
    }
};

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

bindClick( button1, MenuBar.refresh );
bindClick( button2, SubMenu.add );
bindClick( button3, SubMenu.del );
```

这种说法是正确的，9.2节中的示例代码是模拟传统面向对象语言的命令模式实现。命令模式将过程式的请求调用封装在command 对象的execute 方法里，通过封装方法调用，我们可以把运算块包装成形。command 对象可以被四处传递，所以在调用命令的时候，客户（Client）不需要关心事情是如何进行的。

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

JavaScript作为将函数作为一等对象的语言，跟策略模式一样，命令模式也早已融入到了JavaScript语言之中。运算块不一定要封装在command.execute 方法中，也可以封装在普通函数中。函数作为一等对象，本身就可以被四处传递。即使我们依然需要请求“接收者”，那也未必使用面向对象的方式，闭包可以完成同样的功能。

在面向对象设计中，命令模式的接收者被当成command 对象的属性保存起来，同时约定执行命令的操作调用command.execute 方法。在使用闭包的命令模式实现中，接收者被封闭在闭包产生的环境中，执行命令的操作可以更加简单，仅仅执行回调函数即可。无论接收者被保存为对象的属性，还是被封闭在闭包产生的环境中，在将来执行命令的时候，接收者都能被顺利访问。用闭包实现的命令模式如下代码所示：

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

var MenuBar = {
    refresh: function(){
        console.log( '刷新菜单界面' );
    }
};

var RefreshMenuBarCommand = function( receiver ){
    return function(){
        receiver.refresh();
    }
};

var refreshMenuBarCommand = RefreshMenuBarCommand( MenuBar );

setCommand( button1, refreshMenuBarCommand );
```

当然，如果想更明确地表达当前正在使用命令模式，或者除了执行命令之外，将来有可能还要提供撤销命令等操作。那我们最好还是把执行函数改为调用execute 方法：

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

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

var refreshMenuBarCommand = RefreshMenuBarCommand( MenuBar );
setCommand( button1, refreshMenuBarCommand );
```

9.4 撤销命令

命令模式的作用不仅是封装运算块，而且可以很方便地给命令对象增加撤销操作。就像订餐时客人可以通过电话来取消订单一样。下面来看撤销命令的例子。

本节的目标是利用5.4节中的Animate 类来编写一个动画，这个动画的表现是让页面上的小球移动到水平方向的某个位置。现在页面中有一个input文本框和一个button按钮，文本框中可以输入一些数字，表示小球移动后的水平位置，小球在用户点击按钮后立刻开始移动，代码如下：

```
<body>
    <div id="ball" style="position:absolute;background:#000;width:50px;h
    输入小球移动后的位置：<input id="pos"/>
    <button id="moveBtn">开始移动</button>
</body>

<script>
    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' );
    };
</script>
```

如果文本框输入200，然后点击moveBtn按钮，可以看到小球顺利地移动到水平方向200px的位置。现在我们需要一个方法让小球还原到开始移动之前的位置。当然也可以在文本框中再次输入-200，并且点击moveBtn按钮，这也是一个办法，不过显得很笨拙。页面上最好有一个撤销按钮，点击撤销按钮之后，小球便能回到上一次的位置。

在给页面中增加撤销按钮之前，先把目前的代码改为用命令模式实现：

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

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

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

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

var moveCommand;

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

接下来增加撤销按钮：

```
<body>
    <div id="ball" style="position:absolute;background:#000;width:50px;h
    输入小球移动后的位置：<input id="pos"/>
    <button id="moveBtn">开始移动</button>
    <button id="cancelBtn">cancel</cancel> <!--增加取消按钮-->
</body>
```

撤销操作的实现一般是给命令对象增加一个名为unexecude 或者undo 的方法，在该方法里执行execute 的反向操作。在command.execute 方法让小球开始真正运动之前，我们需要先记录小球的当前位置，在unexecude 或者undo 操作中，再让小球回到刚刚记录下的位置，代码如下：

```
<script>
    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.re
        // 记录小球开始移动前的位置
    };

    MoveCommand.prototype.undo = function(){
        this.receiver.start( 'left', this.oldPos, 1000, 'strongEaseOut'
        // 回到小球移动前记录的位置
    };

    var moveCommand;

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

    cancelBtn.onclick = function(){
        moveCommand.undo();         // 撤销命令
    };
</script>
```

现在通过命令模式轻松地实现了撤销功能。如果用普通的方法调用来实现，也许需要每次都手工记录小球的运动轨迹，才能让它还原到之前的位置。而命令模式中小球的原始位置在小球开始移动前已经作为command 对象的属性被保存起来，所以只需要再提供一个undo 方法，并且在undo 方法中让小球回到刚刚记录的原始位置就可以了。

撤销是命令模式里一个非常有用的功能，试想一下开发一个围棋程序的时候，我们把每一步棋子的变化都封装成命令，则可以轻而易举地实现悔棋功能。同样，撤销命令还可以用于实现文本编辑器的Ctrl+Z功能。

9.5 撤消和重做

上一节我们讨论了如何撤销一个命令。很多时候，我们需要撤销一系列的命令。比如在一个围棋程序中，现在已经下了10步棋，我们需要一次性悔棋到第5步。在这之前，我们可以把所有执行过的下棋命令都储存在一个历史列表中，然后倒序循环来依次执行这些命令的undo 操作，直到循环执行到第5个命令为止。

然而，在某些情况下无法顺利地利用undo 操作让对象回到execute 之前的状态。比如在一个Canvas画图的程序中，画布上有一些点，我们在这些点之间画了N条曲线把这些点相互连接起来，当然这是用命令模式来实现的。但是我们却很难为这里的命令对象定义一个擦除某条曲线的undo 操作，因为在Canvas画图中，擦除一条线相对不容易实现。

这时候最好的办法是先清除画布，然后把刚才执行过的命令全部重新执行一遍，这一点同样可以利用一个历史列表堆栈办到。记录命令日志，然后重复执行它们，这是逆转不可逆命令的一个好办法。

在我编写的HTML5版《街头霸王》游戏中，命令模式可以用来实现播放录像功能。原理跟Canvas画图的例子一样，我们把用户在键盘的输入都封装成命令，执行过的命令将被存放到堆栈中。播放录像的时候只需要从头开始依次执行这些命令便可，代码如下：

```
<html>
    <body>
        <button id="replay">播放录像</button>
    </body>

    <script>
        var Ryu = {
            attack: function(){
                console.log( '攻击' );
            },
            defense: function(){
                console.log( '防御' );
            },
            jump: function(){
                console.log( '跳跃' );
            },
            crouch: function(){
```

```
            console.log( '蹲下' );
        }
    };

    var makeCommand = function( receiver, state ){        // 创建命令
        return function(){
            receiver[ state ]();
        }
    };

    var commands = {
        "119": "jump",         // W
        "115": "crouch",     // S
        "97": "defense",     // A
        "100": "attack"     // D
    };

    var commandStack = [];      // 保存命令的堆栈

    document.onkeypress = function( ev ){
        var keyCode = ev.keyCode,
            command = makeCommand( Ryu, commands[ keyCode ] );

        if ( command ){
            command();     // 执行命令
            commandStack.push( command );      // 将刚刚执行过的命令保存
        }
    };

    document.getElementById( 'replay' ).onclick = function(){     //
        var command;
        while( command = commandStack.shift() ){      // 从堆栈里依次取
            command();
        }
    };

  </script>
</html>
```

可以看到，当我们在键盘上敲下W、A、S、D这几个键来完成一些动作之后，再按下Replay按钮，此时便会重复播放之前的动作。

9.6 命令队列

在订餐的故事中，如果订单的数量过多而厨师的人手不够，则可以让这些订单进行排队处理。第一个订单完成之后，再开始执行跟第二个订单有关的操作。

队列在动画中的运用场景也非常多，比如之前的小球运动程序有可能遇到另外一个问题：有些用户反馈，这个程序只适合于APM小于20的人群，大部分用户都有快速连续点击按钮的习惯，当用户第二次点击button的时候，此时小球的前一个动画可能尚未结束，于是前一个动画会骤然停止，小球转而开始第二个动画的运动过程。但这并不是用户的期望，用户希望这两个动画会排队进行。

把请求封装成命令对象的优点在这里再次体现了出来，对象的生命周期几乎是永久的，除非我们主动去回收它。也就是说，命令对象的生命周期跟初始请求发生的时间无关，`command` 对象的`execute` 方法可以在程序运行的任何时刻执行，即使点击按钮的请求早已发生，但我们的命令对象仍然是有生命的。

所以我们可以把div的这些运动过程都封装成命令对象，再把它们压进一个队列堆栈，当动画执行完，也就是当前`command` 对象的职责完成之后，会主动通知队列，此时取出正在队列中等待的第一个命令对象，并且执行它。

我们比较关注的问题是，一个动画结束后该如何通知队列。通常可以使用回调函数来通知队列，除了回调函数之外，还可以选择发布-订阅模式。即在一个动画结束后发布一个消息，订阅者接收到这个消息之后，便开始执行队列里的下一个动画。读者可以尝试按照这个思路来自行实现一个队列动画。

9.7 宏命令

宏命令是一组命令的集合，通过执行宏命令的方式，可以一次执行一批命令。想象一下，家里有一个万能遥控器，每天回家的时候，只要按一个特别的按钮，它就会帮我们关上房间门，顺便打开电脑并登录QQ。

下面我们看看如何逐步创建一个宏命令。首先，我们依然要创建好各种Command：

```
var closeDoorCommand = {
    execute: function(){
        console.log( '关门' );
    }
};

var openPcCommand = {
    execute: function(){
        console.log( '开电脑' );
    }
};

var openQQCommand = {
    execute: function(){
        console.log( '登录QQ' );
    }
};
```

接下来定义宏命令MacroCommand，它的结构也很简单。macroCommand.add 方法表示把子命令添加进宏命令对象，当调用宏命令对象的execute 方法时，会迭代这一组子命令对象，并且依次执行它们的execute 方法：

```
var MacroCommand = function(){
    return {
        commandsList: [],
        add: function( command ){
            this.commandsList.push( command );
        },
```

```
        execute: function(){
            for ( var i = 0, command; command = this.commandsList[ i++ ]
               command.execute();
            }
        }
    }
};

var macroCommand = MacroCommand();
macroCommand.add( closeDoorCommand );
macroCommand.add( openPcCommand );
macroCommand.add( openQQCommand );

macroCommand.execute();
```

当然我们还可以为宏命令添加撤销功能，跟macroCommand.execute 类似，当调用macroCommand.undo 方法时，宏命令里包含的所有子命令对象要依次执行各自的undo 操作。

宏命令是命令模式与组合模式的联用产物，关于组合模式的知识，我们将在第10章详细介绍。

9.8 智能命令与傻瓜命令

再看一下我们在9.7节创建的命令：

```
var closeDoorCommand = {
    execute: function(){
        console.log( '关门' );
    }
};
```

很奇怪，closeDoorCommand 中没有包含任何receiver 的信息，它本身就包揽了执行请求的行为，这跟我们之前看到的命令对象都包含了一个receiver 是矛盾的。

一般来说，命令模式都会在command 对象中保存一个接收者来负责真正执行客户的请求，这种情况下命令对象是“傻瓜式”的，它只负责把客户的请求转交给接收者来执行，这种模式的好处是请求发起者和请求接收者之间尽可能地得到了解耦。

但是我们也可以定义一些更“聪明”的命令对象，“聪明”的命令对象可以直接实现请求，这样一来就不再需要接收者的存在，这种“聪明”的命令对象也叫作智能命令。没有接收者的智能命令，退化到和策略模式非常相近，从代码结构上已经无法分辨它们，能分辨的只有它们意图的不同。策略模式指向的问题域更小，所有策略对象的目标总是一致的，它们只是达到这个目标的不同手段，它们的内部实现是针对“算法”而言的。而智能命令模式指向的问题域更广，command 对象解决的目标更具发散性。命令模式还可以完成撤销、排队等功能。