第 5 章　策略模式

俗话说，条条大路通罗马。在美剧《越狱》中，主角Michael Scofield就设计了两条越狱的道路。这两条道路都可以到达靠近监狱外墙的医务室。

同样，在现实中，很多时候也有多种途径到达同一个目的地。比如我们要去某个地方旅游，可以根据具体的实际情况来选择出行的线路。

- 如果没有时间但是不在乎钱，可以选择坐飞机。
- 如果没有钱，可以选择坐大巴或者火车。
- 如果再穷一点，可以选择骑自行车。

在程序设计中，我们也常常遇到类似的情况，要实现某一个功能有多种方案可以选择。比如一个压缩文件的程序，既可以选择zip算法，也可以选择gzip算法。

这些算法灵活多样，而且可以随意互相替换。这种解决方案就是本章将要

介绍的策略模式。

策略模式的定义是：定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换。

5.1 使用策略模式计算奖金

策略模式有着广泛的应用。本节我们就以年终奖的计算为例进行介绍。

很多公司的年终奖是根据员工的工资基数和年底绩效情况来发放的。例如，绩效为S的人年终奖有4倍工资，绩效为A的人年终奖有3倍工资，而绩效为B的人年终奖是2倍工资。假设财务部要求我们提供一段代码，来方便他们计算员工的年终奖。

1. 最初的代码实现

我们可以编写一个名为calculateBonus 的函数来计算每个人的奖金数额。很显然，calculateBonus 函数要正确工作，就需要接收两个参数：员工的工资数额和他的绩效考核等级。代码如下：

```
var calculateBonus = function( performanceLevel, salary ){

    if ( performanceLevel === 'S' ){
        return salary * 4;
    }

    if ( performanceLevel === 'A' ){
        return salary * 3;
    }

    if ( performanceLevel === 'B' ){
        return salary * 2;
    }

};

calculateBonus( 'B', 20000  );      // 输出：40000
calculateBonus( 'S', 6000 );      // 输出：24000
```

可以发现，这段代码十分简单，但是存在着显而易见的缺点。

- calculateBonus 函数比较庞大，包含了很多if-else 语句，这些

语句需要覆盖所有的逻辑分支。

- calculateBonus 函数缺乏弹性，如果增加了一种新的绩效等级C，或者想把绩效S的奖金系数改为5，那我们必须深入calculateBonus函数的内部实现，这是违反开放-封闭原则的。
- 算法的复用性差，如果在程序的其他地方需要重用这些计算奖金的算法呢？我们的选择只有复制和粘贴。

因此，我们需要重构这段代码。

2. 使用组合函数重构代码

一般最容易想到的办法就是使用组合函数来重构代码，我们把各种算法封装到一个个的小函数里面，这些小函数有着良好的命名，可以一目了然地知道它对应着哪种算法，它们也可以被复用在程序的其他地方。代码如下：

```
var performanceS = function( salary ){
    return salary * 4;
};

var performanceA = function( salary ){
    return salary * 3;
};

var performanceB = function( salary ){
    return salary * 2;
};

var calculateBonus = function( performanceLevel, salary ){

    if ( performanceLevel === 'S' ){
        return performanceS( salary );
    }

    if ( performanceLevel === 'A' ){
        return performanceA( salary );
    }

    if ( performanceLevel === 'B' ){
        return performanceB( salary );
    }
```

```
};

calculateBonus(  'A' , 10000 );    // 输出：30000
```

目前，我们的程序得到了一定的改善，但这种改善非常有限，我们依然没有解决最重要的问题：calculateBonus 函数有可能越来越庞大，而且在系统变化的时候缺乏弹性。

3. 使用策略模式重构代码

经过思考，我们想到了更好的办法——使用策略模式来重构代码。策略模式指的是定义一系列的算法，把它们一个个封装起来。将不变的部分和变化的部分隔开是每个设计模式的主题，策略模式也不例外，策略模式的目的就是将算法的使用与算法的实现分离开来。

在这个例子里，算法的使用方式是不变的，都是根据某个算法取得计算后的奖金数额。而算法的实现是各异和变化的，每种绩效对应着不同的计算规则。

一个基于策略模式的程序至少由两部分组成。第一个部分是一组策略类，策略类封装了具体的算法，并负责具体的计算过程。 第二个部分是环境类Context，Context接受客户的请求，随后把请求委托给某一个策略类。要做到这点，说明Context中要维持对某个策略对象的引用。

现在用策略模式来重构上面的代码。第一个版本是模仿传统面向对象语言中的实现。我们先把每种绩效的计算规则都封装在对应的策略类里面：

```
var performanceS = function(){};

performanceS.prototype.calculate = function( salary ){
    return salary * 4;
};

var performanceA = function(){};

performanceA.prototype.calculate = function( salary ){
    return salary * 3;
};
```

```
var performanceB = function(){};

performanceB.prototype.calculate = function( salary ){
    return salary * 2;
};
```

接下来定义奖金类Bonus：

```
var Bonus = function(){
    this.salary = null;       // 原始工资
    this.strategy = null;     // 绩效等级对应的策略对象
};

Bonus.prototype.setSalary = function( salary ){
    this.salary = salary;     // 设置员工的原始工资
};

Bonus.prototype.setStrategy = function( strategy ){
    this.strategy = strategy;     // 设置员工绩效等级对应的策略对象
};

Bonus.prototype.getBonus = function(){     // 取得奖金数额
    return this.strategy.calculate( this.salary );     // 把计算奖金的操作委
};
```

在完成最终的代码之前，我们再来回顾一下策略模式的思想：

> 定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换[1]。

[1] “并且使它们可以相互替换”，这句话在很大程度上是相对于静态类型语言而言的。因为静态类型语言中有类型检查机制，所以各个策略类需要实现同样的接口。当它们的真正类型被隐藏在接口后面时，它们才能被相互替换。而在JavaScript这种“类型模糊”的语言中没有这种困扰，任何对象都可以被替换使用。因此，JavaScript中的“可以相互替换使用”表现为它们具有相同的目标和意图。

这句话如果说得更详细一点，就是：定义一系列的算法，把它们各自封装

成策略类，算法被封装在策略类内部的方法里。在客户对Context发起请求的时候，Context总是把请求委托给这些策略对象中间的某一个进行计算。

现在我们来完成这个例子中剩下的代码。先创建一个bonus 对象，并且给bonus 对象设置一些原始的数据，比如员工的原始工资数额。接下来把某个计算奖金的策略对象也传入bonus 对象内部保存起来。当调用bonus.getBonus() 来计算奖金的时候，bonus 对象本身并没有能力进行计算，而是把请求委托给了之前保存好的策略对象：

```
var bonus = new Bonus();

bonus.setSalary( 10000 );
bonus.setStrategy( new performanceS() );  // 设置策略对象

console.log( bonus.getBonus() );    // 输出：40000

bonus.setStrategy( new performanceA() );  // 设置策略对象
console.log( bonus.getBonus() );    // 输出：30000
```

刚刚我们用策略模式重构了这段计算年终奖的代码，可以看到通过策略模式重构之后，代码变得更加清晰，各个类的职责更加鲜明。但这段代码是基于传统面向对象语言的模仿，下一节我们将了解用JavaScript实现的策略模式。

5.2 JavaScript版本的策略模式

在5.1节中，我们让strategy 对象从各个策略类中创建而来，这是模拟一些传统面向对象语言的实现。实际上在JavaScript语言中，函数也是对象，所以更简单和直接的做法是把strategy 直接定义为函数：

```
var strategies = {
    "S": function( salary ){
        return salary * 4;
    },
    "A": function( salary ){
        return salary * 3;
    },
    "B": function( salary ){
        return salary * 2;
    }
};
```

同样，Context也没有必要必须用Bonus 类来表示，我们依然用calculateBonus 函数充当Context来接受用户的请求。经过改造，代码的结构变得更加简洁：

```
var strategies = {
    "S": function( salary ){
        return salary * 4;
    },
    "A": function( salary ){
        return salary * 3;
    },
    "B": function( salary ){
        return salary * 2;
    }
};

var calculateBonus = function( level, salary ){
    return strategies[ level ]( salary );
};

console.log( calculateBonus( 'S', 20000 ) );      // 输出：80000
```

```
console.log( calculateBonus( 'A', 10000 ) );      // 输出：30000
```

在接下来的缓动动画和表单验证的例子中，我们用到的都是这种函数形式的策略对象。

5.3　多态在策略模式中的体现

通过使用策略模式重构代码，我们消除了原程序中大片的条件分支语句。所有跟计算奖金有关的逻辑不再放在Context中，而是分布在各个策略对象中。Context并没有计算奖金的能力，而是把这个职责委托给了某个策略对象。每个策略对象负责的算法已被各自封装在对象内部。当我们对这些策略对象发出“计算奖金”的请求时，它们会返回各自不同的计算结果，这正是对象多态性的体现，也是“它们可以相互替换”的目的。替换Context中当前保存的策略对象，便能执行不同的算法来得到我们想要的结果。

5.4　使用策略模式实现缓动动画

如果让一些不太了解前端开发的程序员来投票，选出他们眼中JavaScript语言在Web开发中的两大用途，我想结果很有可能是这样的：

- 编写一些让div飞来飞去的动画
- 验证表单

虽然这只是一句玩笑话，但从中可以看到动画在Web前端开发中的地位。一些别出心裁的动画效果可以让网站增色不少。

有一段时间网页游戏非常流行，HTML5版本的游戏可以达到不逊于Flash游戏的效果。我曾经编写过HTML5版本的街头霸王游戏，让游戏的主角跳跃或是移动，实际上只是让这个div按照一定的缓动算法进行运动而已。

如果我们明白了怎样让一个小球运动起来，那么离编写一个完整的游戏就不遥远了，剩下的只是一些把逻辑组织起来的体力活。本节并不会从头到尾地编写一个完整的游戏，我们首先要做的是让一个小球按照不同的算法进行运动。

5.4.1　实现动画效果的原理

用JavaScript实现动画效果的原理跟动画片的制作一样，动画片是把一些差距不大的原画以较快的帧数播放，来达到视觉上的动画效果。在JavaScript中，可以通过连续改变元素的某个CSS属性，比如`left`、`top`、`background-position`来实现动画效果。图5-1就是通过改变节点的`background-position`，让人物动起来的。

图　5-1

5.4.2　思路和一些准备工作

我们目标是编写一个动画类和一些缓动算法，让小球以各种各样的缓动效果在页面中运动。

现在来分析实现这个程序的思路。在运动开始之前，需要提前记录一些有用的信息，至少包括以下信息： * 动画开始时，小球所在的原始位置；

- 小球移动的目标位置；
- 动画开始时的准确时间点；
- 小球运动持续的时间。

随后，我们会用`setInterval`创建一个定时器，定时器每隔19ms循环一次。在定时器的每一帧里，我们会把动画已消耗的时间、小球原始位置、小球目标位置和动画持续的总时间等信息传入缓动算法。该算法会通过这几个参数，计算出小球当前应该所在的位置。最后再更新该div对应的CSS属性，小球就能够顺利地运动起来了。

5.4.3 让小球运动起来

在实现完整的功能之前，我们先了解一些常见的缓动算法，这些算法最初来自Flash，但可以非常方便地移植到其他语言中。

这些算法都接受4个参数，这4个参数的含义分别是动画已消耗的时间、小球原始位置、小球目标位置、动画持续的总时间，返回的值则是动画元素应该处在的当前位置。代码如下：

```
var tween = {
linear: function( t, b, c, d ){
    return c*t/d + b;
},
easeIn: function( t, b, c, d ){
    return c * ( t /= d ) * t + b;
},
strongEaseIn: function(t, b, c, d){
    return c * ( t /= d ) * t * t * t * t + b;
},
strongEaseOut: function(t, b, c, d){
    return c * ( ( t = t / d - 1) * t * t * t * t + 1 ) + b;
},
sineaseIn: function( t, b, c, d ){
    return c * ( t /= d) * t * t + b;
```

```
},
sineaseOut: function(t,b,c,d){
    return c * ( ( t = t / d - 1) * t * t + 1 ) + b;
}
};
```

现在我们开始编写完整的代码，下面代码的思想来自jQuery库，由于本节的目标是演示策略模式，而非编写一个完整的动画库，因此我们省去了动画的队列控制等更多完整功能。

现在进入代码实现阶段，首先在页面中放置一个div：

```
<body>
    <div style="position:absolute;background:blue" id="div">我是div</div
</body>
```

接下来定义Animate 类，Animate 的构造函数接受一个参数：即将运动起来的dom 节点。Animate 类的代码如下：

```
var Animate = function( dom ){
    this.dom = dom;                    // 进行运动的dom节点
    this.startTime = 0;                // 动画开始时间
    this.startPos = 0;                 // 动画开始时，dom节点的位置，即dom的初
    this.endPos = 0;                   // 动画结束时，dom节点的位置，即dom的目
    this.propertyName = null;          // dom节点需要被改变的css属性名
    this.easing = null;                // 缓动算法
    this.duration = null;              // 动画持续时间
};
```

接下来Animate.prototype.start 方法负责启动这个动画，在动画被启动的瞬间，要记录一些信息，供缓动算法在以后计算小球当前位置的时候使用。在记录完这些信息之后，此方法还要负责启动定时器。代码如

下：

```
Animate.prototype.start = function( propertyName, endPos, duration, easi
    this.startTime = +new Date;         // 动画启动时间
    this.startPos = this.dom.getBoundingClientRect()[ propertyName ];  /
    this.propertyName = propertyName;  // dom节点需要被改变的CSS属性名
    this.endPos = endPos;  // dom节点目标位置
    this.duration = duration;   // 动画持续事件
    this.easing = tween[ easing ];  // 缓动算法

    var self = this;
    var timeId = setInterval(function(){       // 启动定时器，开始执行动画
        if ( self.step() === false ){          // 如果动画已结束，则清除定时器
            clearInterval( timeId );
        }
    }, 19 );
};
```

Animate.prototype.start 方法接受以下4个参数。

- propertyName： 要改变的CSS属性名，比如'left'、'top'，分别表示左右移动和上下移动。
- endPos： 小球运动的目标位置。
- duration： 动画持续时间。
- easing： 缓动算法。

再接下来是Animate.prototype.step 方法，该方法代表小球运动的每一帧要做的事情。在此处，这个方法负责计算小球的当前位置和调用更新CSS属性值的方法Animate.prototype.update 。代码如下：

```
Animate.prototype.step = function(){
    var t = +new Date;         // 取得当前时间
    if ( t >= this.startTime + this.duration ){        // (1)
        this.update( this.endPos );   // 更新小球的CSS属性值
        return false;
    }
    var pos = this.easing( t - this.startTime, this.startPos,
```

```
        this.endPos - this.startPos, this.duration );
    // pos为小球当前位置
    this.update( pos );     // 更新小球的CSS属性值
};
```

在这段代码中，(1)处的意思是，如果当前时间大于动画开始时间加上动画持续时间之和，说明动画已经结束，此时要修正小球的位置。因为在这一帧开始之后，小球的位置已经接近了目标位置，但很可能不完全等于目标位置。此时我们要主动修正小球的当前位置为最终的目标位置。此外让`Animate.prototype.step`方法返回`false`，可以通知`Animate.prototype.start`方法清除定时器。

最后是负责更新小球CSS属性值的`Animate.prototype.update`方法：

```
Animate.prototype.update = function( pos ){
    this.dom.style[ this.propertyName ] = pos + 'px';
};
```

如果不嫌麻烦，我们可以进行一些小小的测试：

```
var div = document.getElementById( 'div' );
var animate = new Animate( div );

animate.start( 'left', 500, 1000, 'strongEaseOut' );
// animate.start( 'top', 1500, 500, 'strongEaseIn' );
```

通过这段代码，可以看到小球按照我们的期望以各种各样的缓动算法在页面中运动。

本节我们学会了怎样编写一个动画类，利用这个动画类和一些缓动算法就可以让小球运动起来。我们使用策略模式把算法传入动画类中，来达到各

种不同的缓动效果，这些算法都可以轻易地被替换为另外一个算法，这是策略模式的经典运用之一。策略模式的实现并不复杂，关键是如何从策略模式的实现背后，找到封装变化、委托和多态性这些思想的价值。

5.5　更广义的“算法”

策略模式指的是定义一系列的算法，并且把它们封装起来。本章我们介绍的计算奖金和缓动动画的例子都封装了一些算法。

从定义上看，策略模式就是用来封装算法的。但如果把策略模式仅仅用来封装算法，未免有一点大材小用。在实际开发中，我们通常会把算法的含义扩散开来，使策略模式也可以用来封装一系列的“业务规则”。只要这些业务规则指向的目标一致，并且可以被替换使用，我们就可以用策略模式来封装它们。

GoF在《设计模式》一书中提到了一个利用策略模式来校验用户是否输入了合法数据的例子，但GoF未给出具体的实现。刚好在Web开发中，表单校验是一个非常常见的话题。下面我们就看一个使用策略模式来完成表单校验的例子。

5.6 表单校验

在一个Web项目中，注册、登录、修改用户信息等功能的实现都离不开提交表单。

在将用户输入的数据交给后台之前，常常要做一些客户端力所能及的校验工作，比如注册的时候需要校验是否填写了用户名，密码的长度是否符合规定，等等。这样可以避免因为提交不合法数据而带来的不必要网络开销。

假设我们正在编写一个注册的页面，在点击注册按钮之前，有如下几条校验逻辑。

- 用户名不能为空。
- 密码长度不能少于6位。
- 手机号码必须符合格式。

5.6.1 表单校验的第一个版本

现在编写表单校验的第一个版本，可以提前透露的是，目前我们还没有引入策略模式。代码如下：

```
<html>
    <body>
        <form action="http:// xxx.com/register" id="registerForm" method
            请输入用户名：<input type="text" name="userName"/ >
            请输入密码：<input type="text" name="password"/ >
            请输入手机号码：<input type="text" name="phoneNumber"/ >
            <button>提交</button>
        </form>
    <script>
        var registerForm = document.getElementById( 'registerForm' );

        registerForm.onsubmit = function(){
            if ( registerForm.userName.value === '' ){
                alert ( '用户名不能为空' );
                return false;
            }
```

```
            if ( registerForm.password.value.length < 6 ){
                alert ( '密码长度不能少于6位' );
                return false;
            }
            if ( !/(^1[3|5|8][0-9]{9}$)/.test( registerForm.phoneNumber.
                alert ( '手机号码格式不正确' );
                return false;
            }
        }
    </script>
</body>
</html>
```

这是一种很常见的代码编写方式，它的缺点跟计算奖金的最初版本一模一样。

- registerForm.onsubmit 函数比较庞大，包含了很多if-else 语句，这些语句需要覆盖所有的校验规则。
- registerForm.onsubmit 函数缺乏弹性，如果增加了一种新的校验规则，或者想把密码的长度校验从6改成8，我们都必须深入registerForm.onsubmit 函数的内部实现，这是违反开放—封闭原则的。
- 算法的复用性差，如果在程序中增加了另外一个表单，这个表单也需要进行一些类似的校验，那我们很可能将这些校验逻辑复制得漫天遍野。

5.6.2 用策略模式重构表单校验

下面我们将用策略模式来重构表单校验的代码，很显然第一步我们要把这些校验逻辑都封装成策略对象：

```
var strategies = {
    isNonEmpty: function( value, errorMsg ){    // 不为空
        if ( value === '' ){
            return errorMsg ;
        }
    },
```

```
    minLength: function( value, length, errorMsg ){     // 限制最小长度
        if ( value.length < length ){
            return errorMsg;
        }
    },
    isMobile: function( value, errorMsg ){     // 手机号码格式
        if ( !/(^1[3|5|8][0-9]{9}$)/.test( value ) ){
            return errorMsg;
        }
    }
};
```

接下来我们准备实现Validator 类。Validator 类在这里作为Context，负责接收用户的请求并委托给strategy 对象。在给出Validator 类的代码之前，有必要提前了解用户是如何向Validator 类发送请求的，这有助于我们知道如何去编写Validator 类的代码。代码如下：

```
var validataFunc = function(){
    var validator = new Validator();     // 创建一个validator对象

       /***************添加一些校验规则****************/
    validator.add( registerForm.userName, 'isNonEmpty', '用户名不能为空' )
    validator.add( registerForm.password, 'minLength:6', '密码长度不能少于
    validator.add( registerForm.phoneNumber, 'isMobile', '手机号码格式不正

    var errorMsg = validator.start();     // 获得校验结果
    return errorMsg;  // 返回校验结果
}

 var registerForm = document.getElementById( 'registerForm' );
 registerForm.onsubmit = function(){
     var errorMsg = validataFunc();   // 如果errorMsg有确切的返回值，说明未通
     if ( errorMsg ){
         alert ( errorMsg );
         return false;    // 阻止表单提交
     }
};
```

从这段代码中可以看到，我们先创建了一个validator 对象，然后通过validator.add 方法，往validator 对象中添加一些校验规则。validator.add 方法接受3个参数，以下面这句代码说明：

```
validator.add( registerForm.password, 'minLength:6', '密码长度不能少于6位'
```

- registerForm.password 为参与校验的input 输入框。
- 'minLength:6' 是一个以冒号隔开的字符串。冒号前面的minLength 代表客户挑选的strategy 对象，冒号后面的数字6表示在校验过程中所必需的一些参数。'minLength:6' 的意思就是校验registerForm.password 这个文本输入框的value 最小长度为6。如果这个字符串中不包含冒号，说明校验过程中不需要额外的参数信息，比如'isNonEmpty' 。
- 第3个参数是当校验未通过时返回的错误信息。

当我们往validator 对象里添加完一系列的校验规则之后，会调用validator.start() 方法来启动校验。如果validator.start() 返回了一个确切的errorMsg 字符串当作返回值，说明该次校验没有通过，此时需让registerForm.onsubmit 方法返回false 来阻止表单的提交。

最后是Validator 类的实现：

```
var Validator = function(){
    this.cache = [];        // 保存校验规则
};

Validator.prototype.add = function( dom, rule, errorMsg ){
    var ary = rule.split( ':' );    // 把strategy和参数分开
    this.cache.push(function(){  // 把校验的步骤用空函数包装起来，并且放入cach
        var strategy = ary.shift();     // 用户挑选的strategy
        ary.unshift( dom.value );    // 把input的value添加进参数列表
        ary.push( errorMsg );    // 把errorMsg添加进参数列表
        return strategies[ strategy ].apply( dom, ary );
    });
```

```
};

Validator.prototype.start = function(){
    for ( var i = 0, validatorFunc; validatorFunc = this.cache[ i++ ]; ){
        var msg = validatorFunc();    // 开始校验，并取得校验后的返回信息
        if ( msg ){     // 如果有确切的返回值，说明校验没有通过
            return msg;
        }
    }
};
```

使用策略模式重构代码之后，我们仅仅通过“配置”的方式就可以完成一个表单的校验，这些校验规则也可以复用在程序的任何地方，还能作为插件的形式，方便地被移植到其他项目中。

在修改某个校验规则的时候，只需要编写或者改写少量的代码。比如我们想将用户名输入框的校验规则改成用户名不能少于4个字符。可以看到，这时候的修改是毫不费力的。代码如下：

```
validator.add( registerForm.userName, 'isNonEmpty', '用户名不能为空' );

// 改成：
validator.add( registerForm.userName, 'minLength:10', '用户名长度不能小于10
```

5.6.3 给某个文本输入框添加多种校验规则

为了让读者把注意力放在策略模式的使用上，目前我们的表单校验实现留有一点小遗憾：一个文本输入框只能对应一种校验规则，比如，用户名输入框只能校验输入是否为空：

```
validator.add( registerForm.userName, 'isNonEmpty', '用户名不能为空' );
```

如果我们既想校验它是否为空，又想校验它输入文本的长度不小于10呢？我们期望以这样的形式进行校验：

```
validator.add( registerForm.userName, [{
        strategy: 'isNonEmpty',
        errorMsg: '用户名不能为空'
    }, {
        strategy: 'minLength:6',
        errorMsg: '用户名长度不能小于10位'
    }] );
```

下面提供的代码可用于一个文本输入框对应多种校验规则：

```
<html>
    <body>
        <form action="http:// xxx.com/register" id="registerForm" method
            请输入用户名：<input type="text" name="userName"/ >
            请输入密码：<input type="text" name="password"/ >
            请输入手机号码：<input type="text" name="phoneNumber"/ >
            <button>提交</button>
        </form>
    <script>

      /***********************策略对象*************************/

        var strategies = {
            isNonEmpty: function( value, errorMsg ){
                if ( value === '' ){
                    return errorMsg;
                }
            },
            minLength: function( value, length, errorMsg ){
                if ( value.length < length ){
                    return errorMsg;
                }
            },
            isMobile: function( value, errorMsg ){
                if ( !/(^1[3|5|8][0-9]{9}$)/.test( value ) ){
                    return errorMsg;
                }
            }
        };
```

```
/***********************Validator 类*************************/

var Validator = function(){
    this.cache = [];
};

Validator.prototype.add = function( dom, rules ){

    var self = this;

    for ( var i = 0, rule; rule = rules[ i++ ]; ){
        (function( rule ){
            var strategyAry = rule.strategy.split( ':' );
            var errorMsg = rule.errorMsg;

            self.cache.push(function(){
                var strategy = strategyAry.shift();
                strategyAry.unshift( dom.value );
                strategyAry.push( errorMsg );
                return strategies[ strategy ].apply( dom, strate
            });
        })( rule )
    }

};

Validator.prototype.start = function(){
    for ( var i = 0, validatorFunc; validatorFunc = this.cache[
        var errorMsg = validatorFunc();
        if ( errorMsg ){
            return errorMsg;
        }
    }
};

/***********************客户调用代码*************************/

var registerForm = document.getElementById( 'registerForm' );

var validataFunc = function(){
    var validator = new Validator();

    validator.add( registerForm.userName, [{
        strategy: 'isNonEmpty',
        errorMsg: '用户名不能为空'
    }, {
        strategy: 'minLength:6',
        errorMsg: '用户名长度不能小于10位'
    }]);
```

```
            validator.add( registerForm.password, [{
                strategy: 'minLength:6',
                errorMsg: '密码长度不能小于6位'
            }]);

            validator.add( registerForm.phoneNumber, [{
                strategy: 'isMobile',
                errorMsg: '手机号码格式不正确'
            }]);

            var errorMsg = validator.start();
            return errorMsg;
        }

        registerForm.onsubmit = function(){
             var errorMsg = validataFunc();

             if ( errorMsg ){
                  alert ( errorMsg );
                  return false;
             }
         };


     </script>
    </body>
</html>
```

5.7 策略模式的优缺点

策略模式是一种常用且有效的设计模式，本章提供了计算奖金、缓动动画、表单校验这三个例子来加深大家对策略模式的理解。从这三个例子中，我们可以总结出策略模式的一些优点。

- 策略模式利用组合、委托和多态等技术和思想，可以有效地避免多重条件选择语句。
- 策略模式提供了对开放—封闭原则的完美支持，将算法封装在独立的strategy中，使得它们易于切换，易于理解，易于扩展。
- 策略模式中的算法也可以复用在系统的其他地方，从而避免许多重复的复制粘贴工作。
- 在策略模式中利用组合和委托来让Context拥有执行算法的能力，这也是继承的一种更轻便的替代方案。

当然，策略模式也有一些缺点，但这些缺点并不严重。

首先，使用策略模式会在程序中增加许多策略类或者策略对象，但实际上这比把它们负责的逻辑堆砌在Context中要好。

其次，要使用策略模式，必须了解所有的strategy，必须了解各个strategy之间的不同点，这样才能选择一个合适的strategy。比如，我们要选择一种合适的旅游出行路线，必须先了解选择飞机、火车、自行车等方案的细节。此时strategy要向客户暴露它的所有实现，这是违反最少知识原则的。

5.8 一等函数对象与策略模式

本章提供的几个策略模式示例，既有模拟传统面向对象语言的版本，也有针对JavaScript语言的特有实现。在以类为中心的传统面向对象语言中，不同的算法或者行为被封装在各个策略类中，Context将请求委托给这些策略对象，这些策略对象会根据请求返回不同的执行结果，这样便能表现出对象的多态性。

Peter Norvig在他的演讲中曾说过：“在函数作为一等对象的语言中，策略模式是隐形的。`strategy` 就是值为函数的变量。”在JavaScript中，除了使用类来封装算法和行为之外，使用函数当然也是一种选择。这些“算法”可以被封装到函数中并且四处传递，也就是我们常说的“高阶函数”。实际上在JavaScript这种将函数作为一等对象的语言里，策略模式已经融入到了语言本身当中，我们经常用高阶函数来封装不同的行为，并且把它传递到另一个函数中。当我们对这些函数发出“调用”的消息时，不同的函数会返回不同的执行结果。在JavaScript中，“函数对象的多态性”来得更加简单。

在前面的学习中，为了清楚地表示这是一个策略模式，我们特意使用了 `strategies` 这个名字。如果去掉`strategies`，我们还能认出这是一个策略模式的实现吗？代码如下：

```
var S = function( salary ){
     return salary * 4;
};

var A = function( salary ){
    return salary * 3;
};

var B = function( salary ){
    return salary * 2;
};

var calculateBonus = function( func, salary ){
    return func( salary );
};

calculateBonus( S, 10000  );    // 输出：40000
```