TURING 图灵原创

AlloyTeam 出品

JavaScript 设计模式与开发实践

曾探◎著

全面涵盖专门针对JavaScript的16个设计模式

深入剖析面向对象设计原则、编程技巧及代码重构

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

曾探

2007年毕业于吉林大学软件学院，目前就职于腾讯AlloyTeam前端团队，高级工程师。

曾参与Web QQ、QQ群、Q+开发者网站、微云、QQ兴趣部落等大型前端项目的开发。有Java、Python和JavaScript的开发经验，业余作品有HTML5版街头霸王等。

平时喜欢电影和音乐，业余时间是一名健身教练。

封面设计 @CHOCDESIGN

TURING 图灵原创

JavaScript 设计模式与开发实践

曾探◎著

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

JavaScript设计模式与开发实践 / 曾探著. -- 北京:
人民邮电出版社, 2015.5
（图灵原创）
ISBN 978-7-115-38888-9

Ⅰ. ①J… Ⅱ. ①曾… Ⅲ. ①JAVA语言－程序设计
Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第072098号

内 容 提 要

本书根据 JavaScript 语言的特性，全面总结了实际工作中常用的设计模式。全书共分为三个部分，第一部分讲解了 JavaScript 语言面向对象和函数式编程的知识及其在设计模式方面的作用；第二部分通过一步步完善示例代码，由浅入深地讲解了 16 个设计模式；第三部分讲述了面向对象的设计原则及其在设计模式中的体现，以及一些常见的面向对象编程技巧和日常开发中的代码重构。

书中所有示例均来自作者长期的开发实践，与实际开发密切相关，适用于初、中、高级 Web 前端开发人员，尤其适合想往架构师晋级的中高级程序员阅读。

◆ 著　　　　曾　探
责任编辑　王军花
执行编辑　张　霞
责任印制　杨林杰
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷
◆ 开本：800×1000　1/16
印张：19.5
字数：461千字　　　　　　　2015年 5 月第 1 版
印数：1－4 000册　　　　　 2015年 5 月北京第 1 次印刷

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

序

如果时间倒退一点，很难想象我这样的“懒人”会花上近一年的业余时间来完成这本书。

这本书的原型是我发表在腾讯内部KM论坛的一篇文章《JavaScript常用设计模式》。这篇文章反响不错，还位列2012年KM十大热门文章第一名。不过说老实话，当时自己也是模式的初学者，和网上大部分讨论设计模式的文章一样，这篇文章里其实存在一些错误，这里要诚恳地说声抱歉。也正是由于这个原因，近两年我重新投身于对设计模式的研究之中。尽管如此，当在电脑上敲下本书第一行字的时候，我心中还是非常忐忑。一是我自己本身并非理论派，大部分工作时间都在做上层应用开发，很多偏理论的知识对于我来说，也是一个学习加总结的过程，二是不确保自己能否牺牲如此多的业余时间，毕竟很难削减玩LOL的时间。

无论如何，它终于和大家见面了。

本书结构

本书共分为三大部分。

第一部分讲解了JavaScript面向对象和函数式编程方面的知识，主要包括静态类型语言和动态类型语言的区别及其在实现设计模式时的异同，以及封装、继承、多态在动态类型语言中的体现，此外还介绍了JavaScript基于原型继承的面向对象系统的来龙去脉，给学习设计模式做好铺垫。

第二部分是核心部分，通过从普通到更好的代码示例，由浅到深地讲解了16个设计模式。

第三部分主要讲解面向对象的设计原则及其在设计模式中的体现，还介绍了一些常见的面向对象编程技巧和日常开发中的代码重构。

目标读者

本书主要面向初中级JavaScript开发人员。本书虽然以设计模式为主题，但也讲述了一些JavaScript开发中需要的基础知识，初级程序员也能从这里找到自己需要的东西。而对于中级程序员而言，学习设计模式的过程，可能正是往高级进阶的过程。

示例代码与勘误

本书提供了丰富的示例，示例代码可以在图灵社区本书主页（http://www.ituring.com.cn/book/1632）的“随书下载”中下载使用。

另外，由于作者的水平和时间所限，本书中难免存在一些遗憾。如果大家发现有什么问题，或者对本书有任何建议，欢迎到图灵社区本书主页提交勘误，也可以发送邮件到svenzeng@tencent.com来讨论，先谢谢！☺

致谢

虽然在写作过程中经历了不少曲折，但最终顺利完成。在这里，我想感谢为我提供帮助的所有人。

感谢图灵的美女编辑Alice，没有她的帮助，这本书不可能完成。

感谢AlloyTeam团队中每一个成员对我的指导和帮助，在这里工作不仅是工作，也是生活很重要的一部分。

感谢贺师俊、王集鹄、易郑超、程劭非几位老师提供的技术指导和宝贵建议。

感谢设计师“出壳设计”设计的插画和封面，它们让内容更加生动有趣。

最后，感谢我的妻子Annie，遇见你，是最美丽的意外。

前　　言

《设计模式》一书自1995年成书一来，一直是程序员谈论的“高端”话题之一。许多程序员从设计模式中学到了设计软件的灵感，或者找到了问题的解决方案。在社区中，既有人对模式无比崇拜，也有人对模式充满误解。有些程序员把设计模式视为圣经，唯模式至上；有些人却认为设计模式只在C++或者Java中有用武之地，JavaScript这种动态语言根本就没有设计模式一说。

那么，在进入设计模式的学习之前，我们最好还是从模式的起源说起，分别听听这些不同的声音。

设计模式并非是软件开发的专业术语。实际上，“模式”最早诞生于建筑学。20世纪70年代，哈佛大学建筑学博士Christopher Alexander和他的研究团队花了约20年的时间，研究了为解决同一个问题而设计出的不同建筑结构，从中发现了那些高质量设计中的相似性，并且用“模式”来指代这种相似性。

受Christopher Alexander工作的启发，Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides四人（人称Gang Of Four，GoF）把这种“模式”观点应用于面向对象的软件设计中，并且总结了23种常见的软件开发设计模式，录入《设计模式：可复用面向对象软件的基础》一书。

> 设计模式的定义是：在面向对象软件设计过程中针对特定问题的简洁而优雅的解决方案。

通俗一点说，设计模式是在某种场合下对某个问题的一种解决方案。如果再通俗一点说，设计模式就是给面向对象软件开发中的一些好的设计取个名字。

GoF成员之一John Vlissides在他的另一本关于设计模式的著作《设计模式沉思录》中写过这样一段话：

> 设想有一个电子爱好者，虽然他没有经过正规的培训，但是却日积月累地设计并制造出许多有用的电子设备：业余无线电、盖革计数器、报警器等。有一天这个爱好者决定重新回到学校去攻读电子学学位，来让自己的才能得到真实的认可。随着课程的展开，这个爱好者突然发现课程内容都似曾相识。似曾相识的并不是术语或者表述的方式，而是背后的概念。这个爱好者不断学到一些名称和原理，虽然这些名称和原理原来他不知道，但事实上他多年来一直都在使用。整个过程只不过是一个接一个的顿悟。

软件开发中的设计也是如此。这些“好的设计”并不是GoF发明的，而是早已存在于软件开发中。一个稍有经验的程序员也许在不知不觉中数次使用过这些设计模式。GoF最大的功绩是把这些“好的设计”从浩瀚的面向对象世界中挑选出来，并且给予它们一个好听又好记的名字。

那么，给模式一个名字有什么意义呢？上述故事中的电子爱好者在未进入学校之前，一点都不知道这些关于电器的概念有一些特定的名称，但这不妨碍他制造出一些电子设备。

实际上给“模式”取名的意义非常重要。人类可以走到生物链顶端的前两个原因分别是会“使用名字”和“使用工具”。在软件设计中，一个好的设计方案有了名字之后，才能被更好地传播，人们才有更多的机会去分享和学习它们。

也许这个小故事可以说明名字对于模式的重要性：假设你是一名足球教练，正在球场边指挥一场足球赛。通过一段时间的观察后，发现对方的后卫技术精湛，身体强壮，但边后卫速度较慢，中后卫身高和头球都非常一般。于是你在场边大声指挥队员：“用速度突破对方边后卫之后，往球门方向踢出高球，中路接应队员抢点头球攻门。”

在机会稍纵即逝的足球场上，教练这样费尽口舌地指挥队员比赛无疑是荒谬的。实际上这种战术有一个名字叫作“下底传中”。正因为战术有了对应的名字，在球场上教练可以很方便地和球员交流。“下底传中”这种战术即是足球场上的一种“模式”。

在软件设计中亦是如此。我们都知道设计经验非常重要。也许我们都有过这种感觉：这个问题发生的场景似曾相识，以前我遇到并解决过这个问题，但是我不知道怎么跟别人去描述它。我们非常希望给这个问题出现的场景和解决方案取一个统一的名字，当别人听到这个名字的时候，便知道我想表达什么。比如一个JavaScript新手今天学会了编写each函数，each函数用来迭代一个数组。他很难想到这个each函数其实就是迭代器模式。于是他向别人描述这个函数结构和意图的时候会遇到困难，而一旦大家对迭代器模式这个名字达成了共识，剩下的交流便是自然而然的事情。

学习模式的作用

小说家很少从头开始设计剧情，足球教练也很少从头开始发明战术，他们总是沿袭一些已经存在的模式。当足球教练看到对方边后卫速度慢，中后卫身高矮时，自然会想到“下底传中”这种模式。

同样，在软件设计中，模式是一些经过了大量实际项目验证的优秀解决方案。熟悉这些模式的程序员，对某些模式的理解也许形成了条件反射。当合适的场景出现时，他们可以很快地找到某种模式作为解决方案。

比如，当他们看到系统中存在一些大量的相似对象，这些对象给系统的内存带来了较大的负担。如果他们熟悉享元模式，那么第一时间就可以想到用享元模式来优化这个系统。再比如，系统中某个接口的结构已经不能符合目前的需求，但他们又不想去改动这个被灰尘遮住的老接口，一个熟悉模式的程序员将很快地找到适配器模式来解决这个问题。

如果我们还没有学习全部的模式，当遇到一个问题时，我们冥冥之中觉得这个问题出现的几率很高，说不定别人也遇到过同样的问题，并且已经把它整理成了模式，提供了一种通用的解决方案。这时候去翻翻《设计模式》这本书也许就会有意外的收获。

模式在不同语言之间的区别

《设计模式》一书的副标题是“可复用面向对象软件的基础”。《设计模式》这本书完全是从面向对象设计的角度出发的，通过对封装、继承、多态、组合等技术的反复使用，提炼出一些可重复使用的面向对象设计技巧。所以有一种说法是设计模式仅仅是就面向对象的语言而言的。

《设计模式》最初讲的确实是静态类型语言中的设计模式，原书大部分代码由C++写成，但设计模式实际上是解决某些问题的一种思想，与具体使用的语言无关。模式社区和语言一直都在发展，如今，除了主流的面向对象语言，函数式语言的发展也非常迅猛。在函数式或者其他编程范型的语言中，设计模式依然存在。

人类飞上天空需要借助飞机等工具，而鸟儿天生就有翅膀。在Dota游戏里，牛头人的人生目标是买一把跳刀（跳刀可以使用跳跃技能），而敌法师天生就有跳跃技能。因为语言的不同，一些设计模式在另外一些语言中的实现也许跟我们在《设计模式》一书中看到的大相径庭，这一点也不令人意外。

Google的研究总监Peter Norvig早在1996年一篇名为“动态语言设计模式”的演讲中，就指出了GoF所提出的23种设计模式，其中有16种在Lisp语言中已经是天然的实现。比如，Command模式在Java中需要一个命令类，一个接收者类，一个调用者类。Command模式把运算块封装在命令对象的方法内，成为该对象的行为，并把命令对象四处传递。但在Lisp或者JavaScript这些把函数当作一等对象的语言中，函数便能封装运算块，并且函数可以被当成对象一样四处传递，这样一来，命令模式在Lisp或者JavaScript中就成为了一种隐形的模式。

在Java这种静态编译型语言中，无法动态地给已存在的对象添加职责，所以一般通过包装类的方式来实现装饰者模式。但在JavaScript这种动态解释型语言中，给对象动态添加职责是再简单不过的事情。这就造成了JavaScript语言的装饰者模式不再关注于给对象动态添加职责，而是关注于给函数动态添加职责。

设计模式的适用性

设计模式被一些人认为只是夸夸其谈的东西，这些人认为设计模式并没有多大用途。毕竟我们用普通的方法就能解决的问题，使用设计模式可能会增加复杂度，或带来一些额外的代码。如果对一些设计模式使用不当，事情还可能变得更糟。

从某些角度来看，设计模式确实有可能带来代码量的增加，或许也会把系统的逻辑搞得更复杂。但软件开发的成本并非全部在开发阶段，设计模式的作用是让人们写出可复用和可维护性高

的程序。假设有一个空房间，我们要日复一日地往里面放一些东西。最简单的办法当然是把这些东西直接扔进去，但是时间久了，就会发现很难从这个房子里找到自己想要的东西，要调整某几样东西的位置也不容易。所以在房间里做一些柜子也许是个更好的选择，虽然柜子会增加我们的成本，但它可以在维护阶段为我们带来好处。使用这些柜子存放东西的规则，或许就是一种模式。

所有设计模式的实现都遵循一条原则，即“找出程序中变化的地方，并将变化封装起来”。一个程序的设计总是可以分为可变的部分和不变的部分。当我们找出可变的部分，并且把这些部分封装起来，那么剩下的就是不变和稳定的部分。这些不变和稳定的部分是非常容易复用的。这也是设计模式为什么描写的是可复用面向对象软件基础的原因。

设计模式被人误解的一个重要原因是人们对它的误用和滥用，比如将一些模式用在了错误的场景中，或者说在不该使用模式的地方刻意使用模式。特别是初学者在刚学会使用一个模式时，恨不得把所有的代码都用这个模式来实现。锤子理论在这里体现得很明显：当我们有了一把锤子，看什么都是钉子。拿足球比赛的例子来说，我们的目标只是进球，“下底传中”这种“模式”仅仅是达到进球目标的一种手段。当我们面临密集防守时，下底传中或许是一种好的选择；但如果我们的球员获得了一个直接面对对方守门员的单刀机会，那么是否还要把球先传向边路队友，再由边路队友来一个边路传中呢？答案是显而易见的，模式应该用在正确的地方。而哪些才算正确的地方，只有在我们深刻理解了模式的意图之后，再结合项目的实际场景才会知道。

分辨模式的关键是意图而不是结构

在设计模式的学习中，有人经常发出这样的疑问：代理模式和装饰者模式，策略模式和状态模式，策略模式和智能命令模式，这些模式的类图看起来几乎一模一样，它们到底有什么区别？

实际上这种情况是普遍存在的，许多模式的类图看起来都差不多，模式只有放在具体的环境下才有意义。比如我们的手机，把它当电话的时候，它就是电话；把它当闹钟的时候，它就是闹钟；用它玩游戏的时候，它就是游戏机。我看到有人手中拿着iPhone18，但那实际上可能只是一个吹风机。有很多模式的类图和结构确实很相似，但这不太重要，辨别模式的关键是这个模式出现的场景，以及为我们解决了什么问题。

对JavaScript设计模式的误解

虽然JavaScript是一门完全面向对象的语言，但在很长一段时间内，JavaScript在人们的印象中只是用来验证表单，或者完成一些简单动画特效的脚本语言。在JavaScript语言上运用设计模式难免显得小题大做。但目前JavaScript已成为最流行的语言之一，在许多大型Web项目中，JavaScript代码的数量已经非常多了。我们绝对有必要把一些优秀的设计模式借鉴到JavaScript这门语言中。许多优秀的JavaScript开源框架也运用了不少设计模式。

JavaScript设计模式的社区目前还几乎是一片荒漠。网络上有一些讨论JavaScript设计模式的

资料和文章，但这些资料和文章大多都存在两个问题。

第一个问题是习惯把静态类型语言的设计模式照搬到JavaScript中，比如有人为了模拟JavaScript版本的工厂方法（Factory Method）模式，而生硬地把创建对象的步骤延迟到子类中。实际上，在Java等静态类型语言中，让子类来“决定”创建何种对象的原因是为了让程序迎合依赖倒置原则（DIP）。在这些语言中创建对象时，先解开对象类型之间的耦合关系非常重要，这样才有机会在将来让对象表现出多态性。

而在JavaScript这种类型模糊的语言中，对象多态性是天生的，一个变量既可以指向一个类，又可以随时指向另外一个类。JavaScript不存在类型耦合的问题，自然也没有必要刻意去把对象“延迟”到子类创建，也就是说，JavaScript实际上是不需要工厂方法模式的。模式的存在首先是能为我们解决什么问题，这种牵强的模拟只会让人觉得设计模式既难懂又没什么用处。

另一个问题是习惯根据模式的名字去臆测该模式的一切。比如命令模式本意是把请求封装到对象中，利用命令模式可以解开请求发送者和请求接受者之间的耦合关系。但命令模式经常被人误解为只是一个名为execute的普通方法调用。这个方法除了叫作execute之外，其实并没有看出其他用处。所以许多人会误会命令模式的意图，以为它其实没什么用处，从而联想到其他设计模式也没有用处。

这些误解都影响了设计模式在JavaScript语言中的发展。

模式的发展

前面说过，模式的社区一直在发展。GoF在1995年提出了23种设计模式。但模式不仅仅局限于这23种。在近20年的时间里，也许有更多的模式已经被人发现并总结了出来。比如一些JavaScript图书中会提到模块模式、沙箱模式等。这些“模式”能否被世人公认并流传下来，还有待时间验证。不过某种解决方案要成为一种模式，还是有几个原则要遵守的。这几个原则即是“再现”“教学”和“能够以一个名字来描述这种模式”。

不管怎样，在一些模式被公认并流行起来之前，需要慎重地冠之以某种模式的名称。否则模式也许很容易泛滥，导致人人都在发明模式，这反而增加了交流的难度。说不准哪天我们就能听到有人说全局变量模式、加模式、减模式等。

在《设计模式》出版后的近20年里，也出现了另外一批讲述设计模式的优秀读物。其中许多都获得过Jolt大奖。数不清的程序员从设计模式中获益，也许是改善了自己编写的某个软件，也许是从设计模式的学习中更好地理解了面向对象编程思想。无论如何，相信对我们这些大多数的普通程序员来说，系统地学习设计模式并没有坏处，相反，你会在模式的学习过程中受益匪浅。

目　　录

第一部分　基础知识

面向对象的 JavaScript

1

1.1　动态类型语言和鸭子类型　2
1.2　多态　4
1.3　封装　12
1.4　原型模式和基于原型继承的 JavaScript 对象系统　14

this、call 和 apply

2

2.1　this　24
2.2　call 和 apply　29

闭包和高阶函数

3

3.1　闭包　35
3.2　高阶函数　44
3.3　小结　58

第二部分　设计模式

单例模式

4

4.1　实现单例模式　60
4.2　透明的单例模式　61
4.3　用代理实现单例模式　62
4.4　JavaScript 中的单例模式　63
4.5　惰性单例　65
4.6　通用的惰性单例　68
4.7　小结　70

策略模式 5

5.1 使用策略模式计算奖金 72
5.2 JavaScript 版本的策略模式 75
5.3 多态在策略模式中的体现 76
5.4 使用策略模式实现缓动动画 76
5.5 更广义的“算法” 80
5.6 表单校验 80
5.7 策略模式的优缺点 86
5.8 一等函数对象与策略模式 86
5.9 小结 87

代理模式 6

6.1 第一个例子——小明追 MM 的故事 88
6.2 保护代理和虚拟代理 91
6.3 虚拟代理实现图片预加载 91
6.4 代理的意义 93
6.5 代理和本体接口的一致性 94
6.6 虚拟代理合并 HTTP 请求 95
6.7 虚拟代理在惰性加载中的应用 97
6.8 缓存代理 99
6.9 用高阶函数动态创建代理 100
6.10 其他代理模式 101
6.11 小结 102

迭代器模式 7

7.1 jQuery 中的迭代器 103
7.2 实现自己的迭代器 104
7.3 内部迭代器和外部迭代器 104
7.4 迭代类数组对象和字面量对象 106
7.5 倒序迭代器 106
7.6 中止迭代器 107
7.7 迭代器模式的应用举例 107
7.8 小结 109

发布–订阅模式 8

8.1 现实中的发布—订阅模式 110
8.2 发布—订阅模式的作用 110
8.3 DOM 事件 111
8.4 自定义事件 112
8.5 发布—订阅模式的通用实现 113
8.6 取消订阅的事件 115
8.7 真实的例子——网站登录 115
8.8 全局的发布—订阅对象 117
8.9 模块间通信 119
8.10 必须先订阅再发布吗 120
8.11 全局事件的命名冲突 121
8.12 JavaScript 实现发布—订阅模式的便利性 124
8.13 小结 124

命令模式 9

9.1 命令模式的用途 125
9.2 命令模式的例子——菜单程序 126
9.3 JavaScript 中的命令模式 128
9.4 撤销命令 130
9.5 撤消和重做 132
9.6 命令队列 134
9.7 宏命令 134
9.8 智能命令与傻瓜命令 135
9.9 小结 136

组合模式 10

10.1 回顾宏命令 138
10.2 组合模式的用途 139
10.3 请求在树中传递的过程 139
10.4 更强大的宏命令 140
10.5 抽象类在组合模式中的作用 143
10.6 透明性带来的安全问题 144
10.7 组合模式的例子——扫描文件夹 145
10.8 一些值得注意的地方 147
10.9 引用父对象 148
10.10 何时使用组合模式 150
10.11 小结 150

模板方法模式 11

11.1 模板方法模式的定义和组成 151
11.2 第一个例子——Coffee or Tea 151
11.3 抽象类 156
11.4 模板方法模式的使用场景 159
11.5 钩子方法 160
11.6 好莱坞原则 162
11.7 真的需要“继承”吗 162
11.8 小结 164

享元模式 12

12.1 初识享元模式 165
12.2 内部状态与外部状态 166
12.3 享元模式的通用结构 167
12.4 文件上传的例子 167
12.5 享元模式的适用性 173
12.6 再谈内部状态和外部状态 173
12.7 对象池 175
12.8 小结 178

职责链模式 13

13.1 现实中的职责链模式 179
13.2 实际开发中的职责链模式 180
13.3 用职责链模式重构代码 181
13.4 灵活可拆分的职责链节点 183
13.5 异步的职责链 184
13.6 职责链模式的优缺点 185
13.7 用 AOP 实现职责链 186
13.8 用职责链模式获取文件上传对象 187
13.9 小结 188

中介者模式 14

14.1 现实中的中介者 190
14.2 中介者模式的例子——泡泡堂游戏 191
14.3 中介者模式的例子——购买商品 199
14.4 小结 207

装饰者模式 15

15.1 模拟传统面向对象语言的装饰者模式 210
15.2 装饰者也是包装器 211
15.3 回到 JavaScript 的装饰者 212
15.4 装饰函数 212
15.5 用 AOP 装饰函数 214
15.6 AOP 的应用实例 216
15.7 装饰者模式和代理模式 222
15.8 小结 223

状态模式 16

16.1 初识状态模式 224
16.2 状态模式的定义 230
16.3 状态模式的通用结构 230
16.4 缺少抽象类的变通方式 231
16.5 另一个状态模式示例——文件上传 232
16.6 状态模式的优缺点 241
16.7 状态模式中的性能优化点 241
16.8 状态模式和策略模式的关系 241
16.9 JavaScript 版本的状态机 242
16.10 表驱动的有限状态机 244
16.11 实际项目中的其他状态机 245
16.12 小结 245

适配器模式 17

17.1 现实中的适配器 246
17.2 适配器模式的应用 247
17.3 小结 250

第三部分　设计原则和编程技巧

单一职责原则 18

18.1 设计模式中的 SRP 原则 252
18.2 何时应该分离职责 256
18.3 违反 SRP 原则 256
18.4 SRP 原则的优缺点 257

最少知识原则 19

19.1 减少对象之间的联系 258
19.2 设计模式中的 LKP 原则 259
19.3 封装在 LKP 原则中的体现 261

开放—封闭原则 20

20.1 扩展 `window.onload` 函数 263
20.2 开放和封闭 264
20.3 用对象的多态性消除条件分支 265
20.4 找出变化的地方 266
20.5 设计模式中的开放—封闭原则 268
20.6 开放—封闭原则的相对性 270
20.7 接受第一次愚弄 270

接口和面向接口编程 21

21.1 回到 Java 的抽象类 271
21.2 `interface` 276
21.3 JavaScript 语言是否需要抽象类和 `interface` 275
21.4 用鸭子类型进行接口检查 277
21.5 用 TypeScript 编写基于 `interface` 的命令模式 278

代码重构 22

22.1 提炼函数 282
22.2 合并重复的条件片段 283
22.3 把条件分支语句提炼成函数 284
22.4 合理使用循环 285
22.5 提前让函数退出代替嵌套条件分支 285
22.6 传递对象参数代替过长的参数列表 286
22.7 尽量减少参数数量 287
22.8 少用三目运算符 288
22.9 合理使用链式调用 288
22.10 分解大型类 289
22.11 用 `return` 退出多重循环 290

参考文献 293

第一部分

基础知识

作为本书的第一部分，我们在进入设计模式的学习之前，需要先了解一些相关的周边知识，例如一些面向对象的基础知识、`this` 等重要概念，还要掌握一些函数式编程的技巧。这些都是学习设计模式的必要铺垫。

第 1 章

面向对象的 JavaScript

JavaScript 没有提供传统面向对象语言中的类式继承，而是通过原型委托的方式来实现对象与对象之间的继承。JavaScript 也没有在语言层面提供对抽象类和接口的支持。正因为存在这些跟传统面向对象语言不一致的地方，我们在用设计模式编写代码的时候，更要跟传统面向对象语言加以区别。所以在正式学习设计模式之前，我们有必要先了解一些 JavaScript 在面向对象方面的知识。

1.1 动态类型语言和鸭子类型

编程语言按照数据类型大体可以分为两类，一类是*静态类型语言*，另一类是*动态类型语言*。

静态类型语言在编译时便已确定变量的类型，而动态类型语言的变量类型要到程序运行的时候，待变量被赋予某个值之后，才会具有某种类型。

静态类型语言的优点首先是在编译时就能发现类型不匹配的错误，编辑器可以帮助我们提前避免程序在运行期间有可能发生的一些错误。其次，如果在程序中明确地规定了数据类型，编译器还可以针对这些信息对程序进行一些优化工作，提高程序执行速度。

静态类型语言的缺点首先是迫使程序员依照强契约来编写程序，为每个变量规定数据类型，归根结底只是辅助我们编写可靠性高程序的一种手段，而不是编写程序的目的，毕竟大部分人编写程序的目的是为了完成需求交付生产。其次，类型的声明也会增加更多的代码，在程序编写过程中，这些细节会让程序员的精力从思考业务逻辑上分散开来。

动态类型语言的优点是编写的代码数量更少，看起来也更加简洁，程序员可以把精力更多地放在业务逻辑上面。虽然不区分类型在某些情况下会让程序变得难以理解，但整体而言，代码量越少，越专注于逻辑表达，对阅读程序是越有帮助的。

动态类型语言的缺点是无法保证变量的类型，从而在程序的运行期有可能发生跟类型相关的错误。这好像在商店买了一包牛肉辣条，但是要真正吃到嘴里才知道是不是牛肉味。

在 JavaScript 中，当我们对一个变量赋值时，显然不需要考虑它的类型，因此，JavaScript 是一门典型的动态类型语言。

动态类型语言对变量类型的宽容给实际编码带来了很大的灵活性。由于无需进行类型检测，我们可以尝试调用任何对象的任意方法，而无需去考虑它原本是否被设计为拥有该方法。

这一切都建立在鸭子类型（duck typing）的概念上，鸭子类型的通俗说法是："如果它走起路来像鸭子，叫起来也是鸭子，那么它就是鸭子。"

我们可以通过一个小故事来更深刻地了解鸭子类型。

> 从前在 JavaScript 王国里，有一个国王，他觉得世界上最美妙的声音就是鸭子的叫声，于是国王召集大臣，要组建一个 1000 只鸭子组成的合唱团。大臣们找遍了全国，终于找到 999 只鸭子，但是始终还差一只，最后大臣发现有一只非常特别的鸡，它的叫声跟鸭子一模一样，于是这只鸡就成为了合唱团的最后一员。

这个故事告诉我们，国王要听的只是鸭子的叫声，这个声音的主人到底是鸡还是鸭并不重要。鸭子类型指导我们只关注对象的行为，而不关注对象本身，也就是关注 HAS-A, 而不是 IS-A。

下面我们用代码来模拟这个故事。

```
var duck = {
    duckSinging: function(){
        console.log( '嘎嘎嘎' );
    }
};

var chicken = {
    duckSinging: function(){
        console.log( '嘎嘎嘎' );
```

```
    }
};

var choir = [];    // 合唱团

var joinChoir = function( animal ){
    if ( animal && typeof animal.duckSinging === 'function' ){
        choir.push( animal );
        console.log( '恭喜加入合唱团' );
        console.log( '合唱团已有成员数量:' + choir.length );
    }
};

joinChoir( duck );    // 恭喜加入合唱团
joinChoir( chicken );    // 恭喜加入合唱团
```

我们看到，对于加入合唱团的动物，大臣们根本无需检查它们的类型，而是只需要保证它们拥有 duckSinging 方法。如果下次期望加入合唱团的是一只小狗，而这只小狗刚好也会鸭子叫，我相信这只小狗也能顺利加入。

在动态类型语言的面向对象设计中，鸭子类型的概念至关重要。利用鸭子类型的思想，我们不必借助超类型的帮助，就能轻松地在动态类型语言中实现一个原则："面向接口编程，而不是面向实现编程"。例如，一个对象若有 push 和 pop 方法，并且这些方法提供了正确的实现，它就可以被当作栈来使用。一个对象如果有 length 属性，也可以依照下标来存取属性（最好还要拥有 slice 和 splice 等方法），这个对象就可以被当作数组来使用。

在静态类型语言中，要实现"面向接口编程"并不是一件容易的事情，往往要通过抽象类或者接口等将对象进行向上转型。当对象的真正类型被隐藏在它的超类型身后，这些对象才能在类型检查系统的"监视"之下互相被替换使用。只有当对象能够被互相替换使用，才能体现出对象多态性的价值。

"面向接口编程"是设计模式中最重要的思想，但在 JavaScript 语言中，"面向接口编程"的过程跟主流的静态类型语言不一样，因此，在 JavaScript 中实现设计模式的过程与在一些我们熟悉的语言中实现的过程会大相径庭。

1.2　多态

"多态"一词源于希腊文 polymorphism，拆开来看是 poly（复数）+ morph（形态）+ ism，从字面上我们可以理解为复数形态。

多态的实际含义是：同一操作作用于不同的对象上面，可以产生不同的解释和不同的执行结果。换句话说，给不同的对象发送同一个消息的时候，这些对象会根据这个消息分别给出不同的反馈。

从字面上来理解多态不太容易，下面我们来举例说明一下。

主人家里养了两只动物，分别是一只鸭和一只鸡，当主人向它们发出“叫”的命令时，鸭会“嘎嘎嘎”地叫，而鸡会“咯咯咯”地叫。这两只动物都会以自己的方式来发出叫声。它们同样“都是动物，并且可以发出叫声”，但根据主人的指令，它们会各自发出不同的叫声。

其实，其中就蕴含了多态的思想。下面我们通过代码进行具体的介绍。

1.2.1　一段“多态”的JavaScript代码

我们把上面的故事用 JavaScript 代码实现如下：

```
var makeSound = function( animal ){
    if ( animal instanceof Duck ){
        console.log( '嘎嘎嘎' );
    }else if ( animal instanceof Chicken ){
        console.log( '咯咯咯' );
    }
};

var Duck = function(){};
var Chicken = function(){};

makeSound( new Duck() );      // 嘎嘎嘎
makeSound( new Chicken() );   // 咯咯咯
```

这段代码确实体现了“多态性”，当我们分别向鸭和鸡发出“叫唤”的消息时，它们根据此消息作出了各自不同的反应。但这样的“多态性”是无法令人满意的，如果后来又增加了一只动物，比如狗，显然狗的叫声是“汪汪汪”，此时我们必须得改动 makeSound 函数，才能让狗也发出叫声。修改代码总是危险的，修改的地方越多，程序出错的可能性就越大，而且当动物的种类越来越多时，makeSound 有可能变成一个巨大的函数。

多态背后的思想是将“做什么”和“谁去做以及怎样去做”分离开来，也就是将“不变的事物”与 “可能改变的事物”分离开来。在这个故事中，动物都会叫，这是不变的，但是不同类型的动物具体怎么叫是可变的。把不变的部分隔离出来，把可变的部分封装起来，这给予了我们扩展程序的能力，程序看起来是可生长的，也是符合开放–封闭原则的，相对于修改代码来说，仅仅增加代码就能完成同样的功能，这显然优雅和安全得多。

1.2.2　对象的多态性

下面是改写后的代码，首先我们把不变的部分隔离出来，那就是所有的动物都会发出叫声：

```
var makeSound = function( animal ){
    animal.sound();
};
```

然后把可变的部分各自封装起来，我们刚才谈到的多态性实际上指的是对象的多态性：

```
var Duck = function(){}

Duck.prototype.sound = function(){
    console.log( '嘎嘎嘎' );
};

var Chicken = function(){}

Chicken.prototype.sound = function(){
    console.log( '咯咯咯' );
};

makeSound( new Duck() );        // 嘎嘎嘎
makeSound( new Chicken() );     // 咯咯咯
```

现在我们向鸭和鸡都发出"叫唤"的消息，它们接到消息后分别作出了不同的反应。如果有一天动物世界里又增加了一只狗，这时候只要简单地追加一些代码就可以了，而不用改动以前的 makeSound 函数，如下所示：

```
var Dog = function(){}

Dog.prototype.sound = function(){
    console.log( '汪汪汪' );
};

makeSound( new Dog() );     // 汪汪汪
```

1.2.3　类型检查和多态

类型检查是在表现出对象多态性之前的一个绕不开的话题，但 JavaScript 是一门不必进行类型检查的动态类型语言，为了真正了解多态的目的，我们需要转一个弯，从一门静态类型语言说起。

我们在 1.1 节已经说明过静态类型语言在编译时会进行类型匹配检查。以 Java 为例，由于在代码编译时要进行严格的类型检查，所以不能给变量赋予不同类型的值，这种类型检查有时候会让代码显得僵硬，代码如下：

```
String str;

str = "abc";    // 没有问题
str = 2;    // 报错
```

现在我们尝试把上面让鸭子和鸡叫唤的例子换成 Java 代码：

```
public class Duck {        // 鸭子类
    public void makeSound(){
        System.out.println( "嘎嘎嘎" );
    }
}
```

```
public class Chicken {        // 鸡类
    public void makeSound(){
        System.out.println( "咯咯咯" );
    }
}


public class AnimalSound {
    public void makeSound( Duck duck ){    // (1)
        duck.makeSound();
    }

}

public class Test {
    public static void main( String args[] ){
        AnimalSound animalSound = new AnimalSound();
        Duck duck = new Duck();
        animalSound.makeSound( duck );    // 输出：嘎嘎嘎
    }
}
```

我们已经顺利地让鸭子可以发出叫声，但如果现在想让鸡也叫唤起来，我们发现这是一件不可能实现的事情。因为(1)处 AnimalSound 类的 makeSound 方法，被我们规定为只能接受 Duck 类型的参数：

```
public class Test {
    public static void main( String args[] ){
        AnimalSound animalSound = new AnimalSound();
        Chicken chicken = new Chicken();
        animalSound.makeSound( chicken );    // 报错，只能接受 Duck 类型的参数
    }
}
```

某些时候，在享受静态语言类型检查带来的安全性的同时，我们亦会感觉被束缚住了手脚。

为了解决这一问题，静态类型的面向对象语言通常被设计为可以向上转型：当给一个类变量赋值时，这个变量的类型既可以使用这个类本身，也可以使用这个类的超类。这就像我们在描述天上的一只麻雀或者一只喜鹊时，通常说“一只麻雀在飞”或者“一只喜鹊在飞”。但如果想忽略它们的具体类型，那么也可以说“一只鸟在飞”。

同理，当 Duck 对象和 Chicken 对象的类型都被隐藏在超类型 Animal 身后，Duck 对象和 Chicken 对象就能被交换使用，这是让对象表现出多态性的必经之路，而多态性的表现正是实现众多设计模式的目标。

1.2.4 使用继承得到多态效果

使用继承来得到多态效果，是让对象表现出多态性的最常用手段。继承通常包括实现继承和

接口继承。本节我们讨论实现继承，接口继承的例子请参见第 21 章。

我们先创建一个 Animal 抽象类，再分别让 Duck 和 Chicken 都继承自 Animal 抽象类，下述代码中(1)处和(2)处的赋值语句显然是成立的，因为鸭子和鸡也是动物：

```
public abstract class Animal {
    abstract void makeSound();   // 抽象方法
}

public class Chicken extends Animal{
    public void makeSound(){
        System.out.println( "咯咯咯" );
    }
}

public class Duck extends Animal{
    public void makeSound(){
        System.out.println( "嘎嘎嘎" );
    }
}

Animal duck = new Duck();       // (1)
Animal chicken = new Chicken();    // (2)
```

现在剩下的就是让 AnimalSound 类的 makeSound 方法接受 Animal 类型的参数，而不是具体的 Duck 类型或者 Chicken 类型：

```
public class AnimalSound{
    public void makeSound( Animal animal ){    // 接受 Animal 类型的参数
        animal.makeSound();
    }
}

public class Test {
    public static void main( String args[] ){
        AnimalSound animalSound= new AnimalSound ();
        Animal duck = new Duck();
        Animal chicken = new Chicken();
        animalSound.makeSound( duck );    // 输出嘎嘎嘎
        animalSound.makeSound( chicken );        // 输出咯咯咯
    }
}
```

1.2.5　JavaScript的多态

从前面的讲解我们得知，多态的思想实际上是把“做什么”和“谁去做”分离开来，要实现这一点，归根结底先要消除类型之间的耦合关系。如果类型之间的耦合关系没有被消除，那么我们在 makeSound 方法中指定了发出叫声的对象是某个类型，它就不可能再被替换为另外一个类型。在 Java 中，可以通过向上转型来实现多态。

而 JavaScript 的变量类型在运行期是可变的。一个 JavaScript 对象，既可以表示 Duck 类型的对象，又可以表示 Chicken 类型的对象，这意味着 JavaScript 对象的多态性是与生俱来的。

这种与生俱来的多态性并不难解释。JavaScript 作为一门动态类型语言，它在编译时没有类型检查的过程，既没有检查创建的对象类型，又没有检查传递的参数类型。在 1.2.2 节的代码示例中，我们既可以往 makeSound 函数里传递 duck 对象当作参数，也可以传递 chicken 对象当作参数。

由此可见，某一种动物能否发出叫声，只取决于它有没有 makeSound 方法，而不取决于它是否是某种类型的对象，这里不存在任何程度上的“类型耦合”。这正是我们从上一节的鸭子类型中领悟的道理。在 JavaScript 中，并不需要诸如向上转型之类的技术来取得多态的效果。

1.2.6　多态在面向对象程序设计中的作用

有许多人认为，多态是面向对象编程语言中最重要的技术。但我们目前还很难看出这一点，毕竟大部分人都不关心鸡是怎么叫的，也不想知道鸭是怎么叫的。让鸡和鸭在同一个消息之下发出不同的叫声，这跟程序员有什么关系呢？

Martin Fowler 在《重构：改善既有代码的设计》里写到：

> 多态的最根本好处在于，你不必再向对象询问“你是什么类型”而后根据得到的答案调用对象的某个行为——你只管调用该行为就是了，其他的一切多态机制都会为你安排妥当。

换句话说，多态最根本的作用就是通过把过程化的条件分支语句转化为对象的多态性，从而消除这些条件分支语句。

Martin Fowler 的话可以用下面这个例子很好地诠释：

> 在电影的拍摄现场，当导演喊出“action”时，主角开始背台词，照明师负责打灯光，后面的群众演员假装中枪倒地，道具师往镜头里撒上雪花。在得到同一个消息时，每个对象都知道自己应该做什么。如果不利用对象的多态性，而是用面向过程的方式来编写这一段代码，那么相当于在电影开始拍摄之后，导演每次都要走到每个人的面前，确认它们的职业分工（类型），然后告诉他们要做什么。如果映射到程序中，那么程序中将充斥着条件分支语句。

利用对象的多态性，导演在发布消息时，就不必考虑各个对象接到消息后应该做什么。对象应该做什么并不是临时决定的，而是已经事先约定和排练完毕的。每个对象应该做什么，已经成为了该对象的一个方法，被安装在对象的内部，每个对象负责它们自己的行为。所以这些对象可以根据同一个消息，有条不紊地分别进行各自的工作。

将行为分布在各个对象中，并让这些对象各自负责自己的行为，这正是面向对象设计的优点。

再看一个现实开发中遇到的例子，这个例子的思想和动物叫声的故事非常相似。

假设我们要编写一个地图应用，现在有两家可选的地图 API 提供商供我们接入自己的应用。目前我们选择的是谷歌地图，谷歌地图的 API 中提供了 show 方法，负责在页面上展示整个地图。示例代码如下：

```
var googleMap = {
    show: function(){
        console.log( '开始渲染谷歌地图' );
    }
};

var renderMap = function(){
    googleMap.show();
};

renderMap();    // 输出：开始渲染谷歌地图
```

后来因为某些原因，要把谷歌地图换成百度地图，为了让 renderMap 函数保持一定的弹性，我们用一些条件分支来让 renderMap 函数同时支持谷歌地图和百度地图：

```
var googleMap = {
    show: function(){
        console.log( '开始渲染谷歌地图' );
    }
};

var baiduMap = {
    show: function(){
        console.log( '开始渲染百度地图' );
    }
};

var renderMap = function( type ){
    if ( type === 'google' ){
        googleMap.show();
    }else if ( type === 'baidu' ){
        baiduMap.show();
    }
};

renderMap( 'google' );    // 输出：开始渲染谷歌地图
renderMap( 'baidu' );     // 输出：开始渲染百度地图
```

可以看到，虽然 renderMap 函数目前保持了一定的弹性，但这种弹性是很脆弱的，一旦需要替换成搜搜地图，那无疑必须得改动 renderMap 函数，继续往里面堆砌条件分支语句。

我们还是先把程序中相同的部分抽象出来，那就是显示某个地图：

```
var renderMap = function( map ){
    if ( map.show instanceof Function ){
        map.show();
    }
};
```

```
renderMap( googleMap );    // 输出：开始渲染谷歌地图
renderMap( baiduMap );     // 输出：开始渲染百度地图
```

现在来找找这段代码中的多态性。当我们向谷歌地图对象和百度地图对象分别发出“展示地图”的消息时，会分别调用它们的 show 方法，就会产生各自不同的执行结果。对象的多态性提示我们，“做什么”和“怎么去做”是可以分开的，即使以后增加了搜搜地图，renderMap 函数仍然不需要做任何改变，如下所示：

```
var sosoMap = {
    show: function(){
        console.log( '开始渲染搜搜地图' );
    }
};

renderMap( sosoMap );     // 输出：开始渲染搜搜地图
```

在这个例子中，我们假设每个地图 API 提供展示地图的方法名都是 show，在实际开发中也许不会如此顺利，这时候可以借助适配器模式来解决问题。

1.2.7　设计模式与多态

GoF 所著的《设计模式》一书的副书名是“可复用面向对象软件的基础”。该书完全是从面向对象设计的角度出发的，通过对封装、继承、多态、组合等技术的反复使用，提炼出一些可重复使用的面向对象设计技巧。而多态在其中又是重中之重，绝大部分设计模式的实现都离不开多态性的思想。

拿命令模式[①]来说，请求被封装在一些命令对象中，这使得命令的调用者和命令的接收者可以完全解耦开来，当调用命令的 execute 方法时，不同的命令会做不同的事情，从而会产生不同的执行结果。而做这些事情的过程是早已被封装在命令对象内部的，作为调用命令的客户，根本不必去关心命令执行的具体过程。

在组合模式[②]中，多态性使得客户可以完全忽略组合对象和叶节点对象之前的区别，这正是组合模式最大的作用所在。对组合对象和叶节点对象发出同一个消息的时候，它们会各自做自己应该做的事情，组合对象把消息继续转发给下面的叶节点对象，叶节点对象则会对这些消息作出真实的反馈。

在策略模式[③]中，Context 并没有执行算法的能力，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”的消息时，它们会返回各自不同的计算结果。

---

① 参见第 9 章。
② 参见第 10 章。
③ 参见第 5 章。

在 JavaScript 这种将函数作为一等对象的语言中，函数本身也是对象，函数用来封装行为并且能够被四处传递。当我们对一些函数发出“调用”的消息时，这些函数会返回不同的执行结果，这是“多态性”的一种体现，也是很多设计模式在 JavaScript 中可以用高阶函数来代替实现的原因。

1.3　封装

封装的目的是将信息隐藏。一般而言，我们讨论的封装是封装数据和封装实现。这一节将讨论更广义的封装，不仅包括封装数据和封装实现，还包括封装类型和封装变化。

1.3.1　封装数据

在许多语言的对象系统中，封装数据是由语法解析来实现的，这些语言也许提供了 `private`、`public`、`protected` 等关键字来提供不同的访问权限。

但 JavaScript 并没有提供对这些关键字的支持，我们只能依赖变量的作用域来实现封装特性，而且只能模拟出 `public` 和 `private` 这两种封装性。

除了 ECMAScript 6 中提供的 `let` 之外，一般我们通过函数来创建作用域：

```
var myObject = (function(){
    var __name = 'sven';    // 私有（private）变量
    return {
        getName: function(){    // 公开（public）方法
            return __name;
        }
    }
})();

console.log( myObject.getName() );    // 输出：sven
console.log( myObject.__name )    // 输出：undefined
```

另外值得一提的是，在 ECAMScript 6 中，还可以通过 Symbol 创建私有属性。详情可参阅 https://github.com/lukehoban/es6features，二维码见右边。

1.3.2　封装实现

上一节描述的封装，指的是数据层面的封装。有时候我们喜欢把封装等同于封装数据，但这是一种比较狭义的定义。

封装的目的是将信息隐藏，封装应该被视为“任何形式的封装”，也就是说，封装不仅仅是隐藏数据，还包括隐藏实现细节、设计细节以及隐藏对象的类型等。

从封装实现细节来讲，封装使得对象内部的变化对其他对象而言是透明的，也就是不可见的。对象对它自己的行为负责。其他对象或者用户都不关心它的内部实现。封装使得对象之间的耦合

变松散，对象之间只通过暴露的 API 接口来通信。当我们修改一个对象时，可以随意地修改它的内部实现，只要对外的接口没有变化，就不会影响到程序的其他功能。

封装实现细节的例子非常之多。拿迭代器来说明，迭代器的作用是在不暴露一个聚合对象的内部表示的前提下，提供一种方式来顺序访问这个聚合对象。我们编写了一个 `each` 函数，它的作用就是遍历一个聚合对象，使用这个 `each` 函数的人不用关心它的内部是怎样实现的，只要它提供的功能正确便可以。即使 `each` 函数修改了内部源代码，只要对外的接口或者调用方式没有变化，用户就不用关心它内部实现的改变。

1.3.3 封装类型

封装类型是静态类型语言中一种重要的封装方式。一般而言，封装类型是通过抽象类和接口来进行的[①]。把对象的真正类型隐藏在抽象类或者接口之后，相比对象的类型，客户更关心对象的行为。在许多静态语言的设计模式中，想方设法地去隐藏对象的类型，也是促使这些模式诞生的原因之一。比如工厂方法模式、组合模式等。

当然在 JavaScript 中，并没有对抽象类和接口的支持。JavaScript 本身也是一门类型模糊的语言。在封装类型方面，JavaScript 没有能力，也没有必要做得更多。对于 JavaScript 的设计模式实现来说，不区分类型是一种失色，也可以说是一种解脱。在后面章节的学习中，我们可以慢慢了解这一点。

1.3.4 封装变化

从设计模式的角度出发，封装在更重要的层面体现为*封装变化*。

《设计模式》一书曾提到如下文字：

> *“考虑你的设计中哪些地方可能变化，这种方式与关注会导致重新设计的原因相反。它不是考虑什么时候会迫使你的设计改变，而是考虑你怎样才能够在不重新设计的情况下进行改变。这里的关键在于封装发生变化的概念，这是许多设计模式的主题。”*

这段文字即是《设计模式》提到的“找到变化并封装之”。《设计模式》一书中共归纳总结了 23 种设计模式。从意图上区分，这 23 种设计模式分别被划分为创建型模式、结构型模式和行为型模式。

拿创建型模式来说，要创建一个对象，是一种抽象行为，而具体创建什么对象则是可以变化的，创建型模式的目的就是封装创建对象的变化。而结构型模式封装的是对象之间的组合关系。行为型模式封装的是对象的行为变化。

通过封装变化的方式，把系统中稳定不变的部分和容易变化的部分隔离开来，在系统的演变过程中，我们只需要替换那些容易变化的部分，如果这些部分是已经封装好的，替换起来也相对

---

① 详情可参阅 1.2 节中的 `Animal` 示例。

容易。这可以最大程度地保证程序的稳定性和可扩展性。

从《设计模式》副标题“可复用面向对象软件的基础”可以知道，这本书理应教我们如何编写可复用的面向对象程序。这本书把大多数笔墨都放在如何封装变化上面，这跟编写可复用的面向对象程序是不矛盾的。当我们想办法把程序中变化的部分封装好之后，剩下的即是稳定而可复用的部分了。

1.4　原型模式和基于原型继承的 JavaScript 对象系统

在 Brendan Eich 为 JavaScript 设计面向对象系统时，借鉴了 Self 和 Smalltalk 这两门基于原型的语言。之所以选择基于原型的面向对象系统，并不是因为时间匆忙，它设计起来相对简单，而是因为从一开始 Brendan Eich 就没有打算在 JavaScript 中加入类的概念。

在以类为中心的面向对象编程语言中，类和对象的关系可以想象成铸模和铸件的关系，对象总是从类中创建而来。而在原型编程的思想中，类并不是必需的，对象未必需要从类中创建而来，一个对象是通过克隆另外一个对象所得到的。就像电影《第六日》一样，通过克隆可以创造另外一个一模一样的人，而且本体和克隆体看不出任何区别。

原型模式不单是一种设计模式，也被称为一种编程泛型。

本节我们将首先学习第一个设计模式——原型模式。随后会了解基于原型的 Io 语言，借助对 Io 语言的了解，我们对 JavaScript 的面向对象系统也将有更深的认识。在本节的最后，我们将详细了解 JavaScript 语言如何通过原型来构建一个面向对象系统。

1.4.1　使用克隆的原型模式

从设计模式的角度讲，原型模式是用于创建对象的一种模式，如果我们想要创建一个对象，一种方法是先指定它的类型，然后通过类来创建这个对象。原型模式选择了另外一种方式，我们不再关心对象的具体类型，而是找到一个对象，然后通过克隆来创建一个一模一样的对象。

既然原型模式是通过克隆来创建对象的，那么很自然地会想到，如果需要一个跟某个对象一模一样的对象，就可以使用原型模式。

假设我们在编写一个飞机大战的网页游戏。某种飞机拥有分身技能，当它使用分身技能的时候，要在页面中创建一些跟它一模一样的飞机。如果不使用原型模式，那么在创建分身之前，无疑必须先保存该飞机的当前血量、炮弹等级、防御等级等信息，随后将这些信息设置到新创建的飞机上面，这样才能得到一架一模一样的新飞机。

如果使用原型模式，我们只需要调用负责克隆的方法，便能完成同样的功能。

原型模式的实现关键，是语言本身是否提供了 `clone` 方法。ECMAScript 5 提供了 `Object.create` 方法，可以用来克隆对象。代码如下：

```
var Plane = function(){
    this.blood = 100;
    this.attackLevel = 1;
    this.defenseLevel = 1;
};

var plane = new Plane();
plane.blood = 500;
plane.attackLevel = 10;
plane.defenseLevel = 7;

var clonePlane = Object.create( plane );
console.log( clonePlane );   // 输出：Object {blood: 500, attackLevel: 10, defenseLevel: 7}
```

在不支持 `Object.create` 方法的浏览器中，则可以使用以下代码：

```
Object.create = Object.create || function( obj ){
    var F = function(){};
    F.prototype = obj;

    return new F();
}
```

1.4.2　克隆是创建对象的手段

通过上一节的代码，我们看到了如何通过原型模式来克隆出一个一模一样的对象。但原型模式的真正目的并非在于需要得到一个一模一样的对象，而是提供了一种便捷的方式去创建某个类型的对象，克隆只是创建这个对象的过程和手段。

在用 Java 等静态类型语言编写程序的时候，类型之间的解耦非常重要。依赖倒置原则提醒我们创建对象的时候要避免依赖具体类型，而用 `new XXX` 创建对象的方式显得很僵硬。工厂方法模式和抽象工厂模式可以帮助我们解决这个问题，但这两个模式会带来许多跟产品类平行的工厂类层次，也会增加很多额外的代码。

原型模式提供了另外一种创建对象的方式，通过克隆对象，我们就不用再关心对象的具体类型名字。这就像一个仙女要送给三岁小女孩生日礼物，虽然小女孩可能还不知道飞机或者船怎么说，但她可以指着商店橱柜里的飞机模型说"我要这个"。

当然在 JavaScript 这种类型模糊的语言中，创建对象非常容易，也不存在类型耦合的问题。从设计模式的角度来讲，原型模式的意义并不算大 。但 JavaScript 本身是一门基于原型的面向对象语言，它的对象系统就是使用原型模式来搭建的，在这里称之为原型编程范型也许更合适。

1.4.3　体验Io语言

前面说过，原型模式不仅仅是一种设计模式，也是一种编程范型。JavaScript 就是使用原型模式来搭建整个面向对象系统的。在 JavaScript 语言中不存在类的概念，对象也并非从类中创建

出来的，所有的 JavaScript 对象都是从某个对象上克隆而来的。

对于习惯了以类为中心语言的人来说，也许一时不容易理解这种基于原型的语言。即使是对于 JavaScript 语言的熟练使用者而言，也可能会有一种“不识庐山真面目，只缘身在此山中”的感觉。事实上，使用原型模式来构造面向对象系统的语言远非仅有 JavaScript 一家。

JavaScript 基于原型的面向对象系统参考了 Self 语言和 Smalltalk 语言，为了搞清 JavaScript 中的原型，我们本该寻根溯源去瞧瞧这两门语言。但由于这两门语言距离现在实在太遥远，我们不妨转而了解一下另外一种轻巧又基于原型的语言——Io 语言。

Io 语言在 2002 年由 Steve Dekorte 发明。可以从http://iolanguage.com下载到 Io 语言的解释器，安装好之后打开 Io 解释器，输入经典的“Hello World”程序。解释器打印出了 Hello World 的字符串，这说明我们已经可以使用 Io 语言来编写一些小程序了，如图 1-1 所示。

```
Io 20110905
Io> "Hello World" print
Hello World==> Hello World
Io>
```

图　1-1

作为一门基于原型的语言，Io 中同样没有类的概念，每一个对象都是基于另外一个对象的克隆。

就像吸血鬼的故事里必然有一个吸血鬼祖先一样，既然每个对象都是由其他对象克隆而来的，那么我们猜测 Io 语言本身至少要提供一个根对象，其他对象都发源于这个根对象。这个猜测是正确的，在 Io 中，根对象名为 Object。

这一节我们依然拿动物世界的例子来讲解 Io 语言。在下面的代码中，通过克隆根对象 `Object`，就可以得到另外一个对象 `Animal`。虽然 `Animal` 是以大写开头的，但是记住 Io 中没有类，`Animal` 跟所有的数据一样都是对象。

```
Animal := Object clone    // 克隆动物对象
```

现在通过克隆根对象 `Object` 得到了一个新的 `Animal` 对象，所以 `Object` 就被称为 `Animal` 的原型。目前 `Animal` 对象和它的原型 `Object` 对象一模一样，还没有任何属于它自己方法和能力。我们假设在 Io 的世界里，所有的动物都会发出叫声，那么现在就给 `Animal` 对象添加 `makeSound` 方法吧。代码如下：

```
Animal makeSound := method( "animal makeSound " print );
```

好了，现在所有的动物都能够发出叫声了，那么再来继续创建一个 `Dog` 对象。显而易见，`Animal` 对象可以作为 `Dog` 对象的原型，`Dog` 对象从 `Animal` 对象克隆而来：

```
Dog := Animal clone
```

可以确定，Dog 一定懂得怎么吃食物，所以接下来给 `Dog` 对象添加 `eat` 方法：

```
Dog eat = method( "dog eat " print );
```

现在已经完成了整个动物世界的构建，通过一次次克隆，Io 的对象世界里不再只有形单影只的根对象 Object，而是多了两个新的对象：Animal 对象和 Dog 对象。其中 Dog 的原型是 Animal，Animal 对象的原型是 Object。最后我们来测试 Animal 对象和 Dog 对象的功能。

先尝试调用 Animal 的 makeSound 方法，可以看到，动物顺利发出了叫声：

```
Animal makeSound      // 输出：animal makeSound
```

然后再调用 Dog 的 eat 方法，同样我们也看到了预期的结果：

```
Dog eat    // 输出：dog eat
```

1.4.4 原型编程范型的一些规则

从上一节的讲解中，我们看到了如何在 Io 语言中从无到有地创建一些对象。跟使用“类”的语言不一样的地方是，Io 语言中最初只有一个根对象 Object，其他所有的对象都克隆自另外一个对象。如果 A 对象是从 B 对象克隆而来的，那么 B 对象就是 A 对象的原型。

在上一小节的例子中，Object 是 Animal 的原型，而 Animal 是 Dog 的原型，它们之间形成了一条原型链。这个原型链是很有用处的，当我们尝试调用 Dog 对象的某个方法时，而它本身却没有这个方法，那么 Dog 对象会把这个请求委托给它的原型 Animal 对象，如果 Animal 对象也没有这个属性，那么请求会顺着原型链继续被委托给 Animal 对象的原型 Object 对象，这样一来便能得到继承的效果，看起来就像 Animal 是 Dog 的“父类”，Object 是 Animal 的“父类”。

这个机制并不复杂，却非常强大，Io 和 JavaScript 一样，基于原型链的委托机制就是原型继承的本质。

我们来进行一些测试。在 Io 的解释器中执行 Dog makeSound 时，Dog 对象并没有 makeSound 方法，于是把请求委托给了它的原型 Animal 对象，而 Animal 对象是有 makeSound 方法的，所以该条语句可以顺利得到输出，如图 1-2 所示。

```
Io> Dog makeSound
animal makeSound==> animal makeSound
Io>
```

图 1-2

现在我们明白了原型编程中的一个重要特性，即当对象无法响应某个请求时，会把该请求委托给它自己的原型。

最后整理一下本节的描述，我们可以发现原型编程范型至少包括以下基本规则。

- 所有的数据都是对象。
- 要得到一个对象，不是通过实例化类，而是找到一个对象作为原型并克隆它。

- 对象会记住它的原型。
- 如果对象无法响应某个请求，它会把这个请求委托给它自己的原型。

1.4.5　JavaScript中的原型继承

刚刚我们已经体验过同样是基于原型编程的 Io 语言，也已经了解了在 Io 语言中如何通过原型链来实现对象之间的继承关系。在原型继承方面，JavaScript 的实现原理和 Io 语言非常相似，JavaScript 也同样遵守这些原型编程的基本规则。

- 所有的数据都是对象。
- 要得到一个对象，不是通过实例化类，而是找到一个对象作为原型并克隆它。
- 对象会记住它的原型。
- 如果对象无法响应某个请求，它会把这个请求委托给它自己的原型。

下面我们来分别讨论 JavaScript 是如何在这些规则的基础上来构建它的对象系统的。

1. 所有的数据都是对象

JavaScript 在设计的时候，模仿 Java 引入了两套类型机制：基本类型和对象类型。基本类型包括 `undefined`、`number`、`boolean`、`string`、`function`、`object`。从现在看来，这并不是一个好的想法。

按照 JavaScript 设计者的本意，除了 `undefined` 之外，一切都应是对象。为了实现这一目标，`number`、`boolean`、`string` 这几种基本类型数据也可以通过“包装类”的方式变成对象类型数据来处理。

我们不能说在 JavaScript 中所有的数据都是对象，但可以说绝大部分数据都是对象。那么相信在 JavaScript 中也一定会有一个根对象存在，这些对象追根溯源都来源于这个根对象。

事实上，JavaScript 中的根对象是 `Object.prototype` 对象。`Object.prototype` 对象是一个空的对象。我们在 JavaScript 遇到的每个对象，实际上都是从 `Object.prototype` 对象克隆而来的，`Object.prototype` 对象就是它们的原型。比如下面的 `obj1` 对象和 `obj2` 对象：

```
var obj1 = new Object();
var obj2 = {};
```

可以利用 ECMAScript 5 提供的 `Object.getPrototypeOf` 来查看这两个对象的原型：

```
console.log( Object.getPrototypeOf( obj1 ) === Object.prototype );    // 输出：true
console.log( Object.getPrototypeOf( obj2 ) === Object.prototype );    // 输出：true
```

2. 要得到一个对象，不是通过实例化类，而是找到一个对象作为原型并克隆它

在 Io 语言中，克隆一个对象的动作非常明显，我们可以在代码中清晰地看到 `clone` 的过程。比如以下代码：

```
Dog :=  Animal clone
```

但在 JavaScript 语言里，我们并不需要关心克隆的细节，因为这是引擎内部负责实现的。我们所需要做的只是显式地调用 `var obj1 = new Object()`或者 `var obj2 = {}`。此时，引擎内部会从 `Object.prototype` 上面克隆一个对象出来，我们最终得到的就是这个对象。

再来看看如何用 `new` 运算符从构造器中得到一个对象，下面的代码我们再熟悉不过了：

```
function Person( name ){
    this.name = name;
};

Person.prototype.getName = function(){
    return this.name;
};

var a = new Person( 'sven' )

console.log( a.name );    // 输出：sven
console.log( a.getName() );     // 输出：sven
console.log( Object.getPrototypeOf( a ) === Person.prototype );     // 输出：true
```

在 JavaScript 中没有类的概念，这句话我们已经重复过很多次了。但刚才不是明明调用了 `new Person()`吗？

在这里 `Person` 并不是类，而是函数构造器，JavaScript 的函数既可以作为普通函数被调用，也可以作为构造器被调用。当使用 `new` 运算符来调用函数时，此时的函数就是一个构造器。 用 `new` 运算符来创建对象的过程，实际上也只是先克隆 `Object.prototype` 对象，再进行一些其他额外操作的过程。[①]

在 Chrome 和 Firefox 等向外暴露了对象`__proto__`属性的浏览器下，我们可以通过下面这段代码来理解 `new` 运算的过程：

```
function Person( name ){
    this.name = name;
};

Person.prototype.getName = function(){
    return this.name;
};

var objectFactory = function(){
    var obj = new Object(),    // 从 Object.prototype 上克隆一个空的对象
        Constructor = [].shift.call( arguments );    // 取得外部传入的构造器，此例是 Person
```

① JavaScript 是通过克隆 `Object.prototype` 来得到新的对象，但实际上并不是每次都真正地克隆了一个新的对象。从内存方面的考虑出发，JavaScript 还做了一些额外的处理，具体细节可以参阅周爱民老师编著的《JavaScript 语言精髓与编程实践》。这里不做深入讨论，我们暂且把创建对象的过程看成完完全全的克隆。

```
    obj.__proto__ = Constructor.prototype;    // 指向正确的原型
    var ret = Constructor.apply( obj, arguments );    // 借用外部传入的构造器给 obj 设置属性

    return typeof ret === 'object' ? ret : obj;     // 确保构造器总是会返回一个对象
};

var a = objectFactory( Person, 'sven' );

console.log( a.name );    // 输出：sven
console.log( a.getName() );     // 输出：sven
console.log( Object.getPrototypeOf( a ) === Person.prototype );      // 输出：true
```

我们看到，分别调用下面两句代码产生了一样的结果：

```
var a = objectFactory( A, 'sven' );
var a = new A( 'sven' );
```

3. 对象会记住它的原型

如果请求可以在一个链条中依次往后传递，那么每个节点都必须知道它的下一个节点。同理，要完成 Io 语言或者 JavaScript 语言中的原型链查找机制，每个对象至少应该先记住它自己的原型。

目前我们一直在讨论“对象的原型”，就 JavaScript 的真正实现来说，其实并不能说对象有原型，而只能说对象的构造器有原型。对于“对象把请求委托给它自己的原型”这句话，更好的说法是对象把请求委托给它的构造器的原型。那么对象如何把请求顺利地转交给它的构造器的原型呢？

JavaScript 给对象提供了一个名为`__proto__`的隐藏属性，某个对象的`__proto__`属性默认会指向它的构造器的原型对象，即`{Constructor}.prototype`。在一些浏览器中，`__proto__`被公开出来，我们可以在 Chrome 或者 Firefox 上用这段代码来验证：

```
var a = new Object();
console.log ( a.__proto__=== Object.prototype );    // 输出：true
```

实际上，`__proto__`就是对象跟“对象构造器的原型”联系起来的纽带。正因为对象要通过`__proto__`属性来记住它的构造器的原型，所以我们用上一节的 `objectFactory` 函数来模拟用 `new` 创建对象时，需要手动给 `obj` 对象设置正确的`__proto__`指向。

```
obj.__proto__ = Constructor.prototype;
```

通过这句代码，我们让 `obj.__proto__` 指向 `Person.prototype`，而不是原来的 `Object.prototype`。

4. 如果对象无法响应某个请求，它会把这个请求委托给它的构造器的原型

这条规则即是原型继承的精髓所在。从对 Io 语言的学习中，我们已经了解到，当一个对象无法响应某个请求的时候，它会顺着原型链把请求传递下去，直到遇到一个可以处理该请求的对象为止。

JavaScript 的克隆跟 Io 语言还有点不一样，Io 中每个对象都可以作为原型被克隆，当 Animal 对象克隆自 Object 对象，Dog 对象又克隆自 Animal 对象时，便形成了一条天然的原型链，如图 1-3 所示。

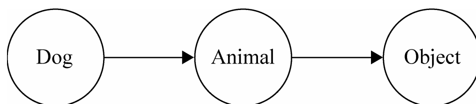


图　1-3

而在 JavaScript 中，每个对象都是从 Object.prototype 对象克隆而来的，如果是这样的话，我们只能得到单一的继承关系，即每个对象都继承自 Object.prototype 对象，这样的对象系统显然是非常受限的。

实际上，虽然 JavaScript 的对象最初都是由 Object.prototype 对象克隆而来的，但对象构造器的原型并不仅限于 Object.prototype 上，而是可以动态指向其他对象。这样一来，当对象 a 需要借用对象 b 的能力时，可以有选择性地把对象 a 的构造器的原型指向对象 b，从而达到继承的效果。下面的代码是我们最常用的原型继承方式：

```
var obj = { name: 'sven' };

var A = function(){};
A.prototype = obj;

var a = new A();
console.log( a.name );    // 输出：sven
```

我们来看看执行这段代码的时候，引擎做了哪些事情。

- 首先，尝试遍历对象 a 中的所有属性，但没有找到 name 这个属性。
- 查找 name 属性的这个请求被委托给对象 a 的构造器的原型，它被 a.__proto__ 记录着并且指向 A.prototype，而 A.prototype 被设置为对象 obj。
- 在对象 obj 中找到了 name 属性，并返回它的值。

当我们期望得到一个“类”继承自另外一个“类”的效果时，往往会用下面的代码来模拟实现：

```
var A = function(){};
A.prototype = { name: 'sven' };

var B = function(){};
B.prototype = new A();

var b = new B();
console.log( b.name );    // 输出：sven
```

再看这段代码执行的时候，引擎做了什么事情。

- 首先，尝试遍历对象 b 中的所有属性，但没有找到 name 这个属性。

- 查找 name 属性的请求被委托给对象 b 的构造器的原型，它被 b.__proto__ 记录着并且指向 B.prototype，而 B.prototype 被设置为一个通过 new A()创建出来的对象。
- 在该对象中依然没有找到 name 属性，于是请求被继续委托给这个对象构造器的原型 A.prototype。
- 在 A.prototype 中找到了 name 属性，并返回它的值。

和把 B.prototype 直接指向一个字面量对象相比，通过 B.prototype = new A()形成的原型链比之前多了一层。但二者之间没有本质上的区别，都是将对象构造器的原型指向另外一个对象，继承总是发生在对象和对象之间。

最后还要留意一点，原型链并不是无限长的。现在我们尝试访问对象 a 的 address 属性。而对象 b 和它构造器的原型上都没有 address 属性，那么这个请求会被最终传递到哪里呢？

实际上，当请求达到 A.prototype，并且在 A.prototype 中也没有找到 address 属性的时候，请求会被传递给 A.prototype 的构造器原型 Object.prototype，显然 Object.prototype 中也没有 address 属性，但 Object.prototype 的原型是 null，说明这时候原型链的后面已经没有别的节点了。所以该次请求就到此打住，a.address 返回 undefined。

```
a.address        // 输出：undefined
```

1.4.6　原型继承的未来

设计模式在很多时候其实都体现了语言的不足之处。Peter Norvig 曾说，设计模式是对语言不足的补充，如果要使用设计模式，不如去找一门更好的语言。这句话非常正确。不过，作为 Web 前端开发者，相信 JavaScript 在未来很长一段时间内都是唯一的选择。虽然我们没有办法换一门语言，但语言本身也在发展，说不定哪天某个模式在 JavaScript 中就已经是天然的存在，不再需要拐弯抹角来实现。比如 Object.create 就是原型模式的天然实现。使用 Object.create 来完成原型继承看起来更能体现原型模式的精髓。目前大多数主流浏览器都提供了 Object.create 方法。

但美中不足是在当前的 JavaScript 引擎下，通过 Object.create 来创建对象的效率并不高，通常比通过构造函数创建对象要慢。此外还有一些值得注意的地方，比如通过设置构造器的 prototype 来实现原型继承的时候，除了根对象 Object.prototype 本身之外，任何对象都会有一个原型。而通过 Object.create(null)可以创建出没有原型的对象。

另外，ECMAScript 6 带来了新的 Class 语法。这让 JavaScript 看起来像是一门基于类的语言，但其背后仍是通过原型机制来创建对象。通过 Class 创建对象的一段简单示例代码[①]如下所示：

```
class Animal {
  constructor(name) {
    this.name = name;
  }
```

① 这段代码来自 http://jurberg.github.io/blog/2014/07/12/javascript-prototype/。

```
  getName() {
    return this.name;
  }
}

class Dog extends Animal {
  constructor(name) {
    super(name);
  }
  speak() {
    return "woof";
  }
}

var dog = new Dog("Scamp");
console.log(dog.getName() + ' says ' + dog.speak());
```

1.4.6　小结

本节讲述了本书的第一个设计模式——原型模式。原型模式是一种设计模式，也是一种编程泛型，它构成了 JavaScript 这门语言的根本。本节首先通过更加简单的 Io 语言来引入原型模式的概念，随后学习了 JavaScript 中的原型模式。原型模式十分重要，和 JavaScript 开发者的关系十分密切。通过原型来实现的面向对象系统虽然简单，但能力同样强大。

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托给对应的策略对象
};
```

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]。

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

---

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。

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 的初始位置
    this.endPos = 0;                // 动画结束时，dom 节点的位置，即 dom 的目标位置
    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, easing ){
    this.startTime = +new Date;        // 动画启动时间
    this.startPos = this.dom.getBoundingClientRect()[ propertyName ];  // dom 节点初始位置
    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="post">
            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.value ) ){
                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 6 位' );
    validator.add( registerForm.phoneNumber, 'isMobile', '手机号码格式不正确' );

    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，说明未通过校验
     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 cache
        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 位' );
```

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="post">
            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, strategyAry );
            });
        })( rule )
    }

};

Validator.prototype.start = function(){
    for ( var i = 0, validatorFunc; validatorFunc = this.cache[ i++ ]; ){
        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
```

5.9　小结

本章我们既提供了接近传统面向对象语言的策略模式实现，也提供了更适合 JavaScript 语言的策略模式版本。在 JavaScript 语言的策略模式中，策略类往往被函数所代替，这时策略模式就成为一种“隐形”的模式。尽管这样，从头到尾地了解策略模式，不仅可以让我们对该模式有更加透彻的了解，也可以使我们明白使用函数的好处。

关注图灵教育 关注图灵社区

iTuring.cn

在线出版 电子书《码农》杂志 图灵访谈 ……

QQ联系我们

读者QQ群：218139230 164939616

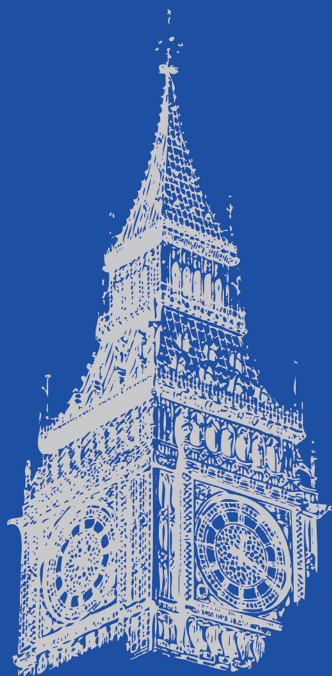
微博联系我们

官方账号：@图灵教育 @图灵社区 @图灵新知
市场合作：@图灵袁野
写作本版书：@图灵小花 @图灵张霞
翻译英文书：@李松峰 @朱巍ituring @楼伟珊
翻译日文书或文章：@图灵乐馨
翻译韩文书：@图灵陈曦
电子书合作：@hi_jeanne
图灵访谈 /《码农》杂志：@李盼ituring
加入我们：@王子是好人

微信联系我们

图灵教育
turingbooks

图灵访谈
ituring_interview



图灵社区读者评价

- “深入浅出，讲解得很好!”

——starj3221

- “看了样章，很不错！有点迫不及待地想看全书了！”

——天才少年

- “看了几章真心感觉不错的。突然之间感觉，我领会了一点JS OOP的精髓了。”

——339025450

业内推荐

- “这本书由浅入深，讲解得很细致，对学习JavaScript很有帮助。”

——于涛，腾讯AlloyTeam负责人

- “内容浅显易懂，覆盖范围全面，对部分常用的模式有深入的剖析。”

——林挺，微众银行前端工程师

JavaScript 设计模式与开发实践

设计模式是软件设计中经过了大量实际项目验证的可复用的优秀解决方案，它有助于程序员写出可复用和可维护性高的程序。许多优秀的JavaScript开源框架都运用了不少设计模式，越来越多的程序员从设计模式中获益，也许是改善了自己编写的某个软件，也许是更好地理解了面向对象的编程思想。无论如何，系统地学习设计模式都会令你受益匪浅。

本书针对JavaScript语言特性全面总结了16个常用的设计模式，讲解了JavaScript面向对象和函数式编程方面的基础知识，介绍了面向对象的设计原则及其在设计模式中的体现，还分享了面向对象编程技巧和日常开发中的代码重构。本书将教会你如何把经典的设计模式应用到JavaScript语言中，编写出优美高效、结构化和可维护的代码。

ISBN 978-7-115-38888-9

定价：59.00元

图灵社区：iTuring.cn
热线：(010) 51095186 转 600

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn