

JavaScript 设计模式

张容铭 著

- + 全面涵盖专门针对 JavaScript 的 36 个设计模式，帮助读者尽快提高开发效率
- + 深入剖析面向对象的设计原则及代码重构，帮助读者快速融入团队项目开发中
- + 本书通过职场主人公“小白”实战历练，介绍了他从菜鸟到高级程序员的蜕变过程，值得每一个程序员借鉴和学习
- + 各种设计模式的原则和准确定义、应用方法和最佳实践

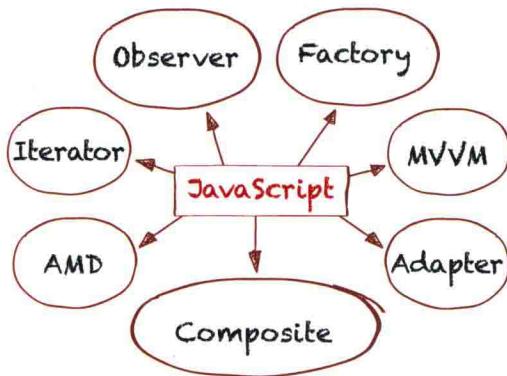


中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

JavaScript 设计模式



- + 如何尽快提高工作效率
- + 如何尽快融入团队项目开发
- + 如何设计出高质量的页面

本书通过主人公“小白”的实战历练，
给读者一个实实在在的学习路线图，
他经历的一切就是你即将或正在遇到的困惑，
本书将给出——解答。
他的成功你可以复制和借鉴！

封面设计：董志桢

分类建议：计算机 / 程序设计 / JavaScript
计算机 / Web前端技术
人民邮电出版社网址：www.ptpress.com.cn

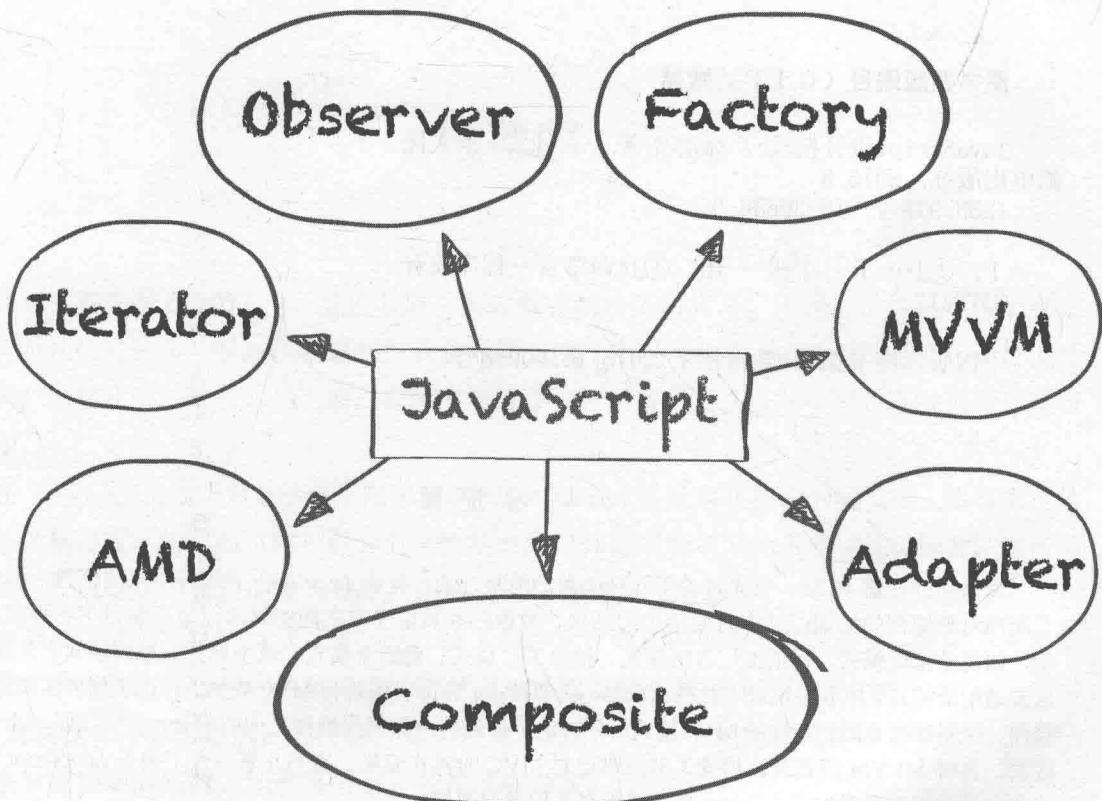


ISBN 978-7-115-39686-0



ISBN 978-7-115-39686-0

定价：59.00 元



JavaScript 设计模式

张容铭 著

人民邮电出版社

北京

图书在版编目 (C I P) 数据

JavaScript设计模式 / 张容铭著. -- 北京 : 人民邮电出版社, 2015.8
ISBN 978-7-115-39686-0

I. ①J... II. ①张... III. ①JAVA语言—程序设计
IV. ①TP312

中国版本图书馆CIP数据核字(2015)第160968号

内 容 提 要

本书共分 6 篇 40 章。首先讨论了几种函数的编写方式，体会 JavaScript 在编程中的灵活性；然后讲解了面向对象编程的知识，其中讨论了类的创建、数据的封装以及类之间的继承；最后探讨了各种模式的技术，如简单工厂模式，包括工厂方法模式、抽象工厂模式、建造者模式、原型模式、单例模式、外观模式，以及适配器模式。本书还讲解了几种适配器、代理模式、装饰者模式和 MVC 模式，讨论了如何实现对数据、视图、控制器的分离。在讲解 MVP 模式时，讨论了如何解决数据与视图之间的耦合，并实现了一个模板生成器；讲解 MVVM 模式时，讨论了双向绑定对 MVC 的模式演化。本书几乎包含了关于 JavaScript 设计模式的全部知识，是进行 JavaScript 高效编程必备的学习手册。

本书适合 JavaScript 初学者、前端设计者、JavaScript 程序员学习，也可以作为大专院校相关专业师生的学习用书，以及培训学校的教材。

-
- ◆ 著 张容铭
 - 责任编辑 张涛
 - 责任印制 张佳莹 焦志炜
 - ◆ 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>
 - 三河市海波印务有限公司印刷
 - ◆ 开本：800×1000 1/16
 - 印张：21
 - 字数：449 千字 2015 年 8 月第 1 版
 - 印数：1-3 000 册 2015 年 8 月河北第 1 次印刷
-

定价：59.00 元

读者服务热线：(010) 81055410 印装质量热线：(010) 81055316
反盗版热线：(010) 81055315

推荐序

认识张容铭是在 2012 年年底的时候，那时张容铭来公司实习，大家都觉得这小伙子实力不俗，而且很爱钻研。得知张容铭利用业余时间完成了本书的创作，作为他的朋友，真替他感到高兴！短短几年，进步如此迅速，在前端实战开发方面有着这么多的积累，有时也会让我自愧不如。

在 Web 应用日益丰富的今天，越来越多的 JavaScript 被运用在我们的网页中。随着用户体验日益受到重视，前端的可维护性、前端性能对用户体验的影响开始备受关注，因此，如何编写高效的、可维护的代码，成为众多互联网公司争相研究的对象。

本书通过情境对话的方式，详细地介绍了各种设计模式的原则、准确定义、应用方法和最佳实践，全方位比较各种同类模式之间的异同，详细讲解不同模式的使用方法。

“极具趣味，容易理解，但讲解又极为严谨和透彻”是本书的写作风格和最大特点。希望大家在学到知识的同时，能够感受到作者的风趣幽默。

最后，希望本书能够帮助业界同仁打造出更为卓越的 Web 产品。

阿里巴巴集团 高级前端研发工程师 王鹏飞

在百度工作的时间里，和张容铭共事过一段时光，在相处的过程中就发现张容铭对设计模式的研究和应用有很大的热情，投入精力很多，且在应用上有很好的理解和收获。本书可以说是张容铭多年来积累的技术和经验的总结，本书涵盖了绝大多数设计模式；本书写作上很有特色，采用新人与导师对话的方式，风趣幽默、通俗易懂，让读者易学、易用、易理解，非常适合 JavaScript 初学者和前段开发工程师学习。

百度 高级前端研发工程师 杨坤

前　　言

一年前如果有人问我是否会写一本书的话，我会直接而坦诚地说我不会。不过随着团队开发中，对同事编写的代码的阅读中我发现，有的人写的代码难懂且臃肿，很难继续编写下去；有的人写的代码简明而灵活，即使再多的需求也很容易实现。我一直在思考，为何为同一需求编写的代码会有这么大的差别？

随着团队项目的开发，我发现，当对类似的需求以类似的模板去解决时，开发成本会减少许多，而且他人也会轻松介入项目的开发。这样，按照同样的流程去解决问题，开发效率得以提高。而将这些解决问题的模板提炼出来，会发现复杂的问题也会简单许多，书写的代码思路清晰且结构简明，这些模板是一种解决问题的方式，或者说是一种模式。

复杂的问题可以分解成一个个小的模块，然后像拼图一样将这些通过模式解决的模块拼凑成一个完整的需求。同时可将余下的精力用去研究其他烦琐问题的解决模式。积攒的模式越多，在工作中以这些模式来解决问题，工作效率就越高。

于是将这些模式总结出来，编写成一本 JavaScript 设计模式书。考虑到对技术的探讨有时是很枯燥的，为了降低读者的学习难度，不至于在阅读中出现倦意，我把工作实战中的角色引入书中，通过他们工作中的情境故事来表达每种模式的内涵与应用，也借此希望更多的人读懂，并领悟更多的设计模式，以便应用到自己的项目中。

目标读者

本书不是一本入门级别的书，本书适用于希望将自身 JavaScript 经验技巧提升一个层次的读者，所以，本书对 JavaScript 基础知识点，如数据类型、运算符、语句等未进行讨论。本书将面向如下 4 类读者。

第一类读者有点 JavaScript 基础，想要更深入地学习 JavaScript，并成为一名标准 Web 开发人员或者前端工程师。想深入了解面向对象编程思想，并且提高自己模块化开发能力，写出可维护、高效率、可拓展的代码的程序员。

第二类读者主要是以前从事 Java 或者 C++ 等编程语言的程序员，现如今想转行从事前端开发，他们可能对于 JavaScript 这种语言比较陌生，但是对于面向对象思想以及设计模式了解较多，只是对于将这些思想运用于前端不是十分清楚，因此，通过阅读本书，对于他们实现前端编程开发很有意义。

第三类读者主要是对设计模式感兴趣，并且想更多了解设计模式在 JavaScript 高效应用的研发人员。通过阅读本书，他们可以体会 JavaScript 中设计模式的实现，突破以往面向对象语言中的实现，用更具灵活的方式解决问题。

第四类读者主要是那些从事前端开发的专业人员。他们能熟练应用 JavaScript 开发，但是还希望提升自己，使自己在团队开发中更具有价值。阅读本书后，他们可以更深入地了解面向对象编程，掌握各种设计模式，使自己的编程技术更灵活，他们会懂得在何种情况下使用那种设计模式解决问题效果会更佳。因此，他们可以自由而熟练地运用设计模式重构现有的代码，使其更灵活、高效、可拓展，即使出现复杂的问题也会编写出一目了然、结构清晰的代码。

诚然会有一些不了解 JavaScript 以及设计模式的读者。他们可能很难看懂书中示例的代码。因此，本书用一种更通俗易懂的方法编写，力求深入浅出，尽可能让更多不同层面的读者理解。

本书特色

本书突破以往填鸭式著书风格，以生动有趣的故事情节推出一个个精彩的设计模式实践。文中以大学刚毕业的小白同学的编程工作经历为主线，在阅读时可以跟着他的经历来学习这些设计模式的具体应用。对于每种模式我们首先提出该模式的定义，这也就声明了该模式的用途。随后交代应用背景，这往往就是该种设计模式的某种应用场境。随着故事的演进，小白所经历的往往是读者在项目中所经历的，因此，很有可能会遇到小白所遇到的问题，这也正是我们需要学习的地方。最后，通过项目经理、小铭等人的帮助使小白顺利地解决一道道难题，从而使小白从初学者一步步进入了工程师的角色。

本书内容

本书分为 6 篇，共 40 章。第一篇主要讲述 JavaScript 面向对象编程基础知识，章节之间知识点连贯，因此，建议读者顺序阅读，并且该篇也是后续 5 篇的基础，因此，一定要按顺序阅读。后 5 篇则是讲述各个设计模式，因此，读者可以根据自己的兴趣选择性阅读。但 5 篇各自侧重点不同，第二篇主要讲述创建型模式，第三篇主要讲述结构型模式，第四篇主要讲述行为型模式，第五篇主要讲述技巧型模式，第六篇主要讲述架构型模式。

第一篇包括第 1 章和第 2 章。

第 1 章介绍 JavaScript 基础知识，讨论了几种函数编写方式，让读者体会 JavaScript 在编程中的灵活性。

第 2 章介绍面向对象编程，讨论了类的创建、数据的封装以及类之间的继承。

第二篇包括第 3 章到第 8 章。

第 3 章介绍简单工厂模式，讨论了对象创建的几种方式。

第 4 章介绍工厂方法模式，讨论了创建多类对象以及一种安全的创建方式。

第 5 章介绍抽象工厂模式，讨论了抽象类以及如何定义一种类簇。

第 6 章介绍建造者模式，讨论了如何更灵活地创建一种复杂的对象。

第 7 章介绍原型模式，讨论了 JavaScript 的核心继承方式——原型式继承。

第 8 章介绍单例模式，讨论了单例对象及其实现与用途。

第三篇包括第 9 章到第 15 章。

第 9 章介绍外观模式，讨论了如何通过外观模式简化接口的使用。

第 10 章介绍适配器模式，讨论了几种用途的适配器。

第 11 章介绍代理模式，讨论了代理思想对于跨域的解决方案。

第 12 章介绍装饰者模式，讨论了装饰者模式更友好地对于已有功能的拓展。

第 13 章介绍桥接模式，讨论了桥接模式解决对象之间的依赖。

第 14 章介绍组合模式，讨论了组合模式如何优化系统的可拓展性。

第 15 章介绍享元模式，讨论了享元模式如何优化系统、提高性能。

第四篇包括第 16 章到第 26 章。

第 16 章介绍模板方法模式，讨论了基于模板类的拓展与创建。

第 17 章介绍观察者模式，讨论了观察者模式解决团队开发中的模块间通信的实践。

第 18 章介绍状态模式，讨论了状态模式中状态在交互中的保存与执行。

第 19 章介绍策略模式，讨论了策略模式如何丰富交互算法。

第 20 章介绍职责链模式，讨论了如何实现一个需求链。

第 21 章介绍命令模式，讨论了如何定义命令集合及运用。

第 22 章介绍访问者模式，讨论了借助已有对象解决已有问题。

第 23 章介绍中介者模式，讨论了中介者如何管理对象之间的通信交互。

第 24 章介绍备忘录模式，讨论了如何更好地处理数据缓存问题。

第 25 章介绍迭代器模式，讨论了迭代器的易用性及其对性能的优化。

第 26 章介绍解释器模式，讨论了通过解释器解决规定的需求。

第五篇包括第 27 章到第 34 章。

第 27 章介绍链模式，讨论了一种高效的方法调用模式。

第 28 章介绍委托模式，讨论了事件委托对性能的优化。

第 29 章介绍数据访问对象模式，讨论了数据访问对象模式对数据库操作对象的封装。

第 30 章介绍节流模式，讨论了如何优化页面中的高频事件以及交互动画。

第 31 章介绍简单模板模式，讨论了一种新的生成页面视图的方法。

第 32 章介绍惰性模式，讨论了对方法的加载以及执行的优化。

第 33 章介绍参与者模式，讨论了一种宽松地为对象绑定方法的方式。

第 34 章介绍等待者模式，讨论了对于异步执行方法回调函数的处理。

第六篇包括第 35 章到第 40 章。

第 35 章介绍同步模块模式，讨论了如何模块化封装代码。

第 36 章介绍异步模块模式，讨论了一种更适合的前端模块化开发实践方式。

第 37 章介绍 Widget 模式，讨论了当今流行的组件式开发，并实现了一个简单的模板引擎。

第 38 章介绍 MVC 模式，讨论了如何实现对数据、视图、控制器的分离。

第 39 章介绍 MVP 模式，讨论了如何解决数据与视图之间的耦合，并实现了一个模板生成器。

第 40 章介绍 MVVM 模式，讨论了双向绑定对 MVC 的模式演化。

本书约定

标 题	意 义
模式定义	指明该模式的定义与用途
故事背景	指明该模式的某种使用场合
故事情节	通过该模式解决的一类问题
下章剧透	点明下一章故事内容
忆之获	回顾本章知识点
我问你答	给出问题，请读者思考与解答，并深入体会该模式

感谢

设计模式是工作经验的结晶，如此多的模式是我一个人无法做到的，因此，本书取得的成果是在前人工作经验总结的基础上提出的。能够完成本书需要感谢太多太多的人。

从百度空间，到百度首页，从百度翻译，再到百度图片搜索，期间经历了太多的团队，得到了太多同事的帮助，因此要感谢每一个人。感谢慧总、冬叔、璇姐、辉哥。感谢慧总让我加入百度工作；感谢冬叔对我工作的支持；感谢璇姐对我的关怀与帮助，让我们的团队气氛无比融洽；感谢辉哥为我提供百度的图搜工作机会，你的决策使我更加坚定前端的工作。

百度新首页的重构是我工作中经历的最难忘的一段日子，感谢大家给予的帮助，感谢坤哥、周全、鹏飞、锡月、王晨、亚斌、研婷。

除了自己所在团队同事给予的帮助，还要感谢很多帮助过我的人，他们是王潇、尚实、先烈、王凯、冯振兴等。

感谢我的新团队，王群、尊程、晓晨、乔岳、腾飞、茗名、李毅、佳佳、阳阳、潇潇、琳琳、胜敏、王敏。有你们在，工作和生活变得如此融洽。

感谢我的老师，刘嘉敏老师，是你让我认识了计算机世界。

本书能够出版最该感谢的就是人民邮电出版社，感谢我们这次融洽的合作，尤其要感谢张涛编辑，没有你对我的支持与帮助，本书可能不会这么顺利地出版，你是一名专业的编辑。当然还要感谢本书默默无闻的编辑，你们辛苦的审校才使本书顺利地出版。

最后，在此感谢我的家人，你们培养了我，从一个对计算机一无所知的孩子，到如今的一名工程师，感谢你们，感谢你们对我的付出。你们虽然对我的工作不是很了解，但每天依旧是那么关心、支持着我。希望你们每天健康而开心地生活。

本书读者答疑 QQ 群为：471118627。

编辑联系邮箱：zhangtao@ptpress.com.cn。

目 录

第一篇 面向对象编程

第1章 灵活的语言——JavaScript 2

1.1 入职第一天 2
1.2 函数的另一种形式 2
1.3 用对象收编变量 3
1.4 对象的另一种形式 4
1.5 真假对象 4
1.6 类也可以 5
1.7 一个检测类 5
1.8 方法还可以这样用 6
1.9 函数的祖先 7
1.10 可以链式添加吗 8
1.11 换一种方式使用方法 9
下章剧透 10
忆之获 10
我问你答 10

第2章 写的都是看到的——面向 对象编程 11

2.1 两种编程风格——面向过程与 面向对象 11
2.2 包装明星——封装 12
2.3 传宗接代——继承 19
2.4 老师不止一位——多继承 27
2.5 多种调用方式——多态 29
下章剧透 30
忆之获 31
我问你答 31

第二篇 创建型设计模式

第3章 神奇的魔术师——简单工厂模式 34

3.1 工作中的第一次需求 34
3.2 如果类太多，那么提供一个 35

3.3 一个对象有时也可代替许多类 37

3.4 你的理解决定你选择的方式 38
下章剧透 39
忆之获 39
我问你答 39

第4章 给我一张名片——工厂方法模式 40

4.1 广告展现 40
4.2 方案的抉择 41
4.3 安全模式类 42
4.4 安全的工厂方法 43
下章剧透 44
忆之获 44
我问你答 44

第5章 出现的都是幻觉——抽象 工厂模式 45

5.1 带头模范——抽象类 45
5.2 幽灵工厂——抽象工厂模式 46
5.3 抽象与实现 47
下章剧透 49
忆之获 49
我问你答 49

第6章 分即是合——建造者模式 50

6.1 发布简历 50
6.2 创建对象的另一种形式 50
6.3 创建一位应聘者 52
下章剧透 53
忆之获 53
我问你答 54

第7章 语言之魂——原型模式 55

7.1 语言中的原型 55
7.2 创建一个焦点图 55

7.3 最优的解决方案	56	第 11 章 牛郎织女——代理模式	79
7.4 原型的拓展	58	11.1 无法获取图片上传模块数据	79
7.5 原型继承	58	11.2 一切只因跨域	79
下章剧透	59	11.3 站长统计	80
忆之获	60	11.4 JSONP	81
我问你答	60	11.5 代理模板	81
第 8 章 一个人的寂寞——单例模式	61	下章剧透	83
8.1 滑动特效	61	忆之获	83
8.2 命名空间的管理员	62	我问你答	84
8.3 模块分明	63		
8.4 创建一个小型代码库	63		
8.5 无法修改的静态变量	64		
8.6 惰性单例	65		
下章剧透	65		
忆之获	66		
我问你答	66		
第三篇 结构型设计模式			
第 9 章 套餐服务——外观模式	68	第 12 章 房子装修——装饰者模式	85
9.1 添加一个点击事件	68	12.1 为输入框的新需求	85
9.2 兼容方式	69	12.2 装饰已有的功能对象	86
9.3 除此之外	70	12.3 为输入框添砖加瓦	86
9.4 小型代码库	70	下章剧透	87
下章剧透	71	忆之获	87
忆之获	71	我问你答	88
我问你答	72		
第 10 章 水管弯弯——适配器模式	73	第 13 章 城市间的公路——桥接模式	89
10.1 引入 jQuery	73	13.1 添加事件交互	89
10.2 生活中的适配器	73	13.2 提取共同点	90
10.3 jQuery 适配器	74	13.3 事件与业务逻辑之间的桥梁	90
10.4 适配异类框架	74	13.4 多元化对象	91
10.5 参数适配器	75	下章剧透	93
10.6 数据适配	76	忆之获	93
10.7 服务器端数据适配	77	我问你答	93
下章剧透	77		
忆之获	77		
我问你答	78		
第 14 章 超值午餐——组合模式	94		
14.1 新闻模块十万火急	94		
14.2 餐厅里的套餐业务	95		
14.3 每个成员要有祖先	95		
14.4 组合要有容器类	96		
14.5 创建一个新闻类	97		
14.6 把新闻模块创建出来	99		
14.7 表单中的应用	100		
下章剧透	101		
忆之获	101		
我问你答	102		

第 15 章 城市公交车——享元模式	103	第 18 章 超级玛丽——状态模式	126
15.1 翻页需求	103	18.1 最美图片	126
15.2 冗余的结构	104	18.2 分支判断的思考	126
15.3 享元对象	104	18.3 状态对象的实现	127
15.4 实现需求	105	18.4 状态对象演练	127
15.5 享元动作	106	18.5 超级玛丽	128
下章剧透	108	18.6 状态的优化	129
忆之获	108	18.7 两种使用方式	130
我问你答	108	下章剧透	131
		忆之获	131
		我问你答	131
第四篇 行为型设计模式			
第 16 章 照猫画虎——模板方法模式	110	第 19 章 活诸葛——策略模式	132
16.1 提示框归一化	110	19.1 商品促销	132
16.2 美味的蛋糕	111	19.2 活诸葛	132
16.3 创建基本提示框	111	19.3 策略对象	133
16.4 模板的原型方法	112	19.4 诸葛奇谋	133
16.5 根据模板创建类	113	19.5 缓冲函数	134
16.6 继承类也可作为模板类	113	19.6 表单验证	134
16.7 创建一个提示框	114	19.7 算法拓展	135
16.8 创建多类导航	114	19.8 算法调用	135
16.9 创建导航更容易	116	下章剧透	136
下章剧透	116	忆之获	136
忆之获	116	我问你答	137
我问你答	117		
第 17 章 通信卫星——观察者模式	118	第 20 章 有序车站——职责链模式	138
17.1 团队开发的坎坷	118	20.1 “半成品”需求	138
17.2 卫星的故事	118	20.2 分解需求	139
17.3 创建一个观察者	119	20.3 第一站——请求模块	139
17.4 拉出来溜溜	121	20.4 下一站——响应数据适配模块	140
17.5 使用前的思考	121	20.5 终点站——创建组件模块	141
17.6 大显身手	121	20.6 站点检测——单元测试	141
17.7 对象间解耦	123	20.7 方案确定	142
17.8 课堂演练	124	下章剧透	142
下章剧透	125	忆之获	143
忆之获	125	我问你答	143
我问你答	125		
第 21 章 命令模式	144		
21.1 自由化创建视图	144		

21.2 命令对象	145	忆之获	166
21.3 视图创建	145	我问你答	166
21.4 视图展示	146	第 25 章 点钞机——迭代器模式	167
21.5 命令接口	147	25.1 简化循环遍历	167
21.6 大功告成	147	25.2 迭代器	167
21.7 绘图命令	148	25.3 实现迭代器	168
21.8 写一条命令	150	25.4 小试牛刀	170
下章剧透	150	25.5 数组迭代器	171
忆之获	151	25.6 对象迭代器	171
我问你答	151	25.7 试用迭代器	172
第 22 章 驻华大使——访问者模式	152	25.8 同步变量迭代器	172
22.1 设置样式	152	25.9 分支循环嵌套问题	174
22.2 自娱自乐的 IE	152	25.10 解决方案	176
22.3 访问操作元素	153	下章剧透	177
22.4 事件自定义数据	153	忆之获	177
22.5 原生对象构造器	154	我问你答	177
22.6 对象访问器	154	第 26 章 语言翻译——解释器模式	178
22.7 操作类数组	155	26.1 统计元素路径	178
下章剧透	156	26.2 描述文法	179
忆之获	156	26.3 解释器	179
我问你答	156	26.4 同级兄弟元素遍历	180
第 23 章 媒婆——中介者模式	157	26.5 遍历文档树	180
23.1 导航设置层	157	26.6 小试牛刀	181
23.2 创建中介者对象	158	下章剧透	182
23.3 试试看，可否一用	159	忆之获	182
23.4 攻克需求	159	我问你答	182
23.5 订阅消息	160		
23.6 发布消息	161		
下章剧透	162		
忆之获	162		
我问你答	162		
第 24 章 做好笔录——备忘录模式	163		
24.1 新闻展示	163		
24.2 缓存数据	164		
24.3 新闻缓存器	164		
24.4 工作中的备忘录	166		
下章剧透	166		
		第五篇 技巧型设计模式	
		第 27 章 永无尽头——链模式	184
		27.1 深究 jQuery	184
		27.2 原型式继承	184
		27.3 找位助手	185
		27.4 获取元素	185
		27.5 一个大问题	186
		27.6 覆盖获取	187
		27.7 方法丢失	187
		27.8 对比 jQuery	188

27.9	丰富元素获取	189	第 30 章 执行控制——节流模式	214	
27.10	数组与对象	190	30.1	返回顶部	214
27.11	方法拓展	191	30.2	节流器	214
27.12	添加方法	192	30.3	优化浮层	216
27.13	大功告成	195	30.4	创建浮层类	216
	下章剧透	196	30.5	添加节流器	217
	忆之获	196	30.6	图片的延迟加载	218
	我问你答	196	30.7	延迟加载图片类	218
			30.8	获取容器内的图片	219
			30.9	加载图片	220
			30.10	筛选需加载的图片	220
			30.11	获取纵坐标	221
			30.12	节流器优化加载	221
			30.13	大功告成	222
			30.14	统计打包	222
			30.15	组装统计	222
				下章剧透	224
				忆之获	224
				我问你答	224
第 28 章 未来预言家——委托模式		197	第 31 章 卡片拼图——简单模板模式	225	
28.1	点击日历交互	197	31.1	展示模板	225
28.2	委托父元素	197	31.2	实现方案	225
28.3	预言未来	198	31.3	创建文字列表视图	226
28.4	内存外泄	199	31.4	新方案	227
28.5	数据分发	200	31.5	再次优化	228
	下章剧透	201	31.6	模板生成器	228
	忆之获	201	31.7	最佳方案	229
	我问你答	201		下章剧透	229
				忆之获	230
				我问你答	230
第 29 章 数据管理器——数据访问					
对象模式		202			
29.1	用户引导	202			
29.2	数据访问对象类	203			
29.3	数据操作状态	203			
29.4	增添数据	204			
29.5	查找数据	205			
29.6	删除数据	206			
29.7	检验 DAO	207			
29.8	MongoDB	208			
29.9	在 nodejs 中写入配置项	208			
29.10	连接 MongoDB	208			
29.11	操作集合	209			
29.12	插入操作	209			
29.13	删除操作	210			
29.14	更新操作	211			
29.15	查找操作	211			
29.16	操作其他集合	212			
	下章剧透	212			
	忆之获	212			
	我问你答	213			
			第 32 章 机器学习——惰性模式	231	
			32.1	对事件的思考	231
			32.2	机器学习	232
			32.3	加载即执行	232
			32.4	惰性执行	233
			32.5	创建 XHR 对象	233
			32.6	第一种方案	234

32.7 第二种方案	235
下章剧透	235
忆之获	235
我问你答	235
第 33 章 异国战场——参与者模式	236
33.1 传递数据	236
33.2 函数绑定	237
33.3 应用于事件	238
33.4 原生 bind 方法	239
33.5 函数柯里化	239
33.6 重构 bind	240
33.7 兼容版本	241
下章剧透	242
忆之获	242
我问你答	243
第 34 章 入场仪式——等待者模式	244
34.1 接口拆分	244
34.2 入场仪式	244
34.3 等待者对象	245
34.4 监控对象	246
34.5 完善接口方法	247
34.6 学以致用	248
34.7 异步方法	248
34.8 结果如何	249
34.9 框架中的等待者	250
34.10 封装异步请求	250
34.11 轮询	251
下章剧透	251
忆之获	251
我问你答	252
第六篇 架构型设计模式	
第 35 章 死心眼——同步模块模式	254
35.1 排队开发	254
35.2 模块化开发	255
35.3 模块管理器与创建方法	255
35.4 创建模块	256
35.5 模块调用方法	258
35.6 调用模块	259
下章剧透	259
忆之获	260
我问你答	260
第 36 章 大心脏——异步模块模式	261
36.1 异步加载文件中的模块	261
36.2 异步模块	262
36.3 闭包环境	262
36.4 创建与调度模块	263
36.5 加载模块	264
36.6 设置模块	265
36.7 学以致用	266
36.8 实现交互	267
下章剧透	267
忆之获	267
我问你答	267
第 37 章 分而治之——Widget 模式	268
37.1 视图模块化	268
37.2 模板引擎	269
37.3 实现原理	269
37.4 模板引擎模块	269
37.5 处理数据	270
37.6 获取模板	270
37.7 处理模板	271
37.8 编译执行	272
37.9 几种模板	273
37.10 实现组件	274
下章剧透	274
忆之获	275
我问你答	275
第 38 章 三人行——MVC 模式	276
38.1 小白的顾虑	276
38.2 一个传说——MVC	276
38.3 数据层	277
38.4 视图层	278
38.5 控制器	279

38.6	侧边导航栏	279	39.12	MVP 构造函数	295
38.7	侧边导航栏数据模型层	280	39.13	增添管理器	295
38.8	侧边导航栏视图层	281	39.14	增加一个模块	295
38.9	侧边导航栏控制器层	282		下章剧透	297
38.10	执行控制器	284		忆之获	297
38.11	增加一个模块	284		我问你答	297
	下章剧透	285			
	忆之获	285			
	我问你答	286			
第 39 章	三军统帅——MVP 模式	287	第 40 章	视图的逆袭——MVVM 模式	298
39.1	数据模型层与视图层联姻的代价	287	40.1	视图层的思考	298
39.2	MVP 模式	287	40.2	滚动条与进度条	299
39.3	数据层的填补	288	40.3	组件的探讨	299
39.4	视图层的大刀阔斧	289	40.4	视图模型层	299
39.5	模板创建的分层处理	289	40.5	创建进度条	300
39.6	处理一个元素	291	40.6	创建滑动条	301
39.7	改头换面的管理器	292	40.7	让滑动条动起来	302
39.8	一个案例	293	40.8	为组件点睛	303
39.9	用数据装扮导航	293	40.9	寻找我的组件	303
39.10	千呼万唤始出来的导航	294	40.10	展现组件	304
39.11	模块开发中的应用	294		下章剧透	304
				忆之获	304
				我问你答	305
			附录 A		307

第一篇

面向对象编程

面向对象编程（Object-oriented programming, OOP）是一种程序设计范型。它将对象作为程序的基本单元，将程序和数据封装其中，以提高程序的重用性、灵活性和扩展性。

第1章 灵活的语言——JavaScript

第2章 写的都是看到的——面向
对象编程

第1章 灵活的语言——JavaScript

结束了4年的大学学习生活，小白信心满满地来到应聘的M公司。今天是入职的第一天，项目经理分下来一个验证表单功能的任务，内容不多，仅需要验证用户名、邮箱、密码等。

1.1 入职第一天

小白接到需求看了看，感觉很简单，于是便写下几个函数。

```
function checkName() {  
    // 验证姓名  
}  
function checkEmail() {  
    // 验证邮箱  
}  
function checkPassword() {  
    // 验证密码  
}  
.....
```

于是要把自己的代码提交到团队项目里。

正在此时，一位工作多年的程序员小铭看到小白要提交的代码摇了摇头说：“小白，等一下，先不要提交。”

“怎么了？”

“你创建了很多全局变量呀。”

“变量？我只是写了几个函数而已。”

“函数不是变量么？”小铭反问道。

此时小白不知所措，心想：“难道函数是变量？”脸瞬间沉了下来。

1.2 函数的另一种形式

小铭见此情形忙笑着说：“别着急，你看，如果我这么声明几个变量来实现你的功能你可以么？”

```

var checkName = function() {
    // 验证姓名
}
var checkEmail = function() {
    // 验证邮箱
}
var checkPassword = function() {
    // 验证密码
}

```

“一样的，只不过……”

“对，只不过这个在用的时候要提前声明，但是这么看你就发现你创建了3个函数保存在变量里来实现你的功能，而你写的是将你的变量名放在function后面而已，它也代表了你的变量。所以说你也声明了3个全局变量。”

“这有什么问题呢？”

“从功能上讲当然没问题，但是今天你加入了我们的团队，在团队开发中你所写的代码就不能只考虑自己了，也要考虑不影晌到他人，如果别人也定义了同样的方法就会覆盖掉原有的功能了。如果你定义了很多方法，这种相互覆盖的问题是很不容易察觉到的。”

“那我应该如何避免呢？”小白问道。

“你可以将它们放在一个变量里保存，这样就可减少覆盖或被覆盖的风险，当然一旦被覆盖，所有的功能都会失效，这种现象也是很明显的，你自然也会很轻易觉察到。”

“可是我该如何做呢？”小白迫不及待地追问道。

1.3 用对象收编变量

“一猜你就会问。”

“好吧，请你先简单地说一下。”

“对象你知道吧，它有属性和方法，而如果我们要访问它的属性或者方法时，可通过点语法向下遍历查询得到。我们可以创建一个检测对象，然后把我们的方法放在里面。”

```

var CheckObject = {
    checkName : function() {
        // 验证姓名
    },
    checkEmail : function() {
        // 验证邮箱
    },
    checkPassword : function() {
        // 验证密码
    }
}

```

“此时我们将所有的函数作为CheckObject对象的方法，这样我们就只有一个对象，而我们要想使用它们也很简单，比如检测姓名CheckObject.checkName()，只是在我们原来使用的

函数式前面多了一个对象名称。”

“哦，这样呀，但是我们既然可以通过点语法来使用方法，我们是不是也可以这么创建呢？”

1.4 对象的另一种形式

“当然，不过首先你要声明一个对象，然后给它添加方法，当然在 JavaScript 中函数也是对象，所以你可以这么做：”

```
var CheckObject = function(){};  
CheckObject.checkName = function(){  
    // 验证姓名  
}  
CheckObject.checkEmail = function(){  
    // 验证邮箱  
}  
CheckObject.checkPassword = function(){  
    // 验证密码  
}
```

“使用和前面的方式是一样的，比如 CheckObject.checkName()，”小铭接着说，“现在虽然能满足你的需求，但当别人想用你写的对象方法时就有些麻烦了，因为这个对象不能复制一份，或者说这个对象类在用 new 关键字创建新的对象时，新创建的对象是不能继承这些方法的。”

“但是复制又有什么用呢？”小白不解地问道。

“给你举个例子吧，假如你喜欢设计模式，你买了这本书，然后回去你的小伙伴看见了，感觉很有用，他们也想要怎么办？书就这一本。但如果你买的是台打印机，那么好吧，即使你的小伙伴再多，你也有能力给他们每个人打印一本。”

“哦，有些明白了，但是我该如何做到呢？”

1.5 真假对象

小铭解释说：“如果你想简单地复制一下，你可以将这些方法放在一个函数对象中。”于是小铭将代码写下。

```
var CheckObject = function(){  
    return {  
        checkName : function(){  
            // 验证姓名  
        },  
        checkEmail : function(){  
            // 验证邮箱  
        },  
        checkPassword : function(){  
            // 验证密码  
        }  
    }  
}
```

}

小白看了看代码，思考一下说：“哦，你写的看上去是，当每次调用这个函数的时候，把我们之前写的那个对象返回出来，当别人每次调用这个函数时都返回了一个新对象，这样执行过程中明面上是 CheckObject 对象，可实际上是返回的新对象。这样每个人在使用时就互不影响了。比如想检测邮箱可以像这样吧。”

```
var a = CheckObject ();
a.checkEmail ();
```

1.6 类也可以

“嗯，对”小铭接着说，“虽然通过创建了新对象完成了我们的需求，但是他不是一个真正意义上类的创建方式，并且创建的对象 a 和对象 CheckObject 没有任何关系（返回出来的对象本身就与 CheckObject 对象无关），所以我们还要对其稍加改造一下。”

```
var CheckObject = function() {
  this.checkName = function(){
    // 验证姓名
  }
  this.checkEmail = function(){
    // 验证邮箱
  }
  this.checkPassword = function(){
    // 验证密码
  }
}
```

“像上面这样的对象就可以看成类了。”小铭继续说。

“那么我们使用它还像之前那样创建对象的方法创建么？”小白追问道。

“不，既然是一个类，你就要用关键字 new 来创建了。”

```
var a = new CheckObject ();
a.checkEmail ();
```

“这样你就可以用 CheckObject 类创建出来的对象了。”

“如果我和我的小伙伴们对类实例化了（用类创建对象），那么我们每个人都会有一套属于自己的方法吧。”小白不解地问道。

1.7 一个检测类

“当然，你看，我们是把所有的方法放在函数内部了，通过 this 定义的，所以每一次通过 new 关键字创建新对象的时候，新创建的对象都会对类的 this 上的属性进行复制。所以这些新创建的对象都会有自己的一套方法，然而有时候这么做造成的消耗是很奢侈的，我们需要处理

一下。”

```
var CheckObject = function(){};  
CheckObject.prototype.checkName = function(){  
    // 验证姓名  
}  
CheckObject.prototype.checkEmail = function(){  
    // 验证邮箱  
}  
CheckObject.prototype.checkPassword = function(){  
    // 验证密码  
}
```

“这样创建对象实例的时候，创建出来的对象所拥有的方法就都是一个了，因为它们都要依赖 prototype 原型依次寻找，而找到的方法都是同一个，它们都绑定在 CheckObject 对象类的原型上，”小铭继续说，“这种方式我们要将 prototype 写很多遍，所以你也可以这样做。”

```
var CheckObject = function(){};  
CheckObject.prototype = {  
    checkName : function(){  
        // 验证姓名  
    },  
    checkEmail : function(){  
        // 验证邮箱  
    },  
    checkPassword : function(){  
        // 验证密码  
    }  
}
```

“但有一点你要记住，这两种方式不能混着用，否则一旦混用，如在后面为对象的原型对象赋值新对象时，那么它将会覆盖掉之前对 prototype 对象赋值的方法。”小铭补充说。

“知道了，不过我们要使用这种方式定义的类是不是要像下面这样呢？”小白问道。

```
var a = new CheckObject();  
a.checkName();  
a.checkEmail();  
a.checkPassword();
```

1.8 方法还可以这样用

“没错，但是你发现没，你调用了 3 个方法，但是你对对象 a 书写了 3 遍。这是可以避免的，那就要在你声明的每一个方法末尾处将当前对象返回，在 JavaScript 中 this 指向的就是当前对象，所以你可以将它返回。例如我们开始写的第一个对象还记得么？改动它很简单，像下面这样就可以。”

```
var CheckObject = {  
    checkName : function(){  
        // 验证姓名  
    }
```

```

        return this;
    },
    checkEmail : function(){
        // 验证邮箱
        return this;
    },
    checkPassword : function(){
        // 验证密码
        return this;
    }
}

```

“此时我们要想使用他就可以这样：”

```
CheckObject.checkName().checkEmail().checkPassword();
```

“当然同样的方式还可以放到类的原型对象中。”

```

var CheckObject = function(){}
CheckObject.prototype = {
    checkName : function(){
        // 验证姓名
        return this;
    },
    checkEmail : function(){
        // 验证邮箱
        return this;
    },
    checkPassword : function(){
        // 验证密码
        return this;
    }
}

```

“但使用时候也要先创建一下：”

```
var a = new CheckObject();
a.checkName().checkEmail().checkPassword();
```

1.9 函数的祖先

小白回顾着这些从未见过的代码方式内心很激动，小铭见小白对 JavaScript 如此着迷，于是补充了两句。

“如果你看过 prototype.js 的代码，我想你会想到下面的书写方式。”

“prototype.js 是什么？”小白问道。

“一款 JavaScript 框架，里面为我们方便地封装了很多方法，它最大的特点就是对源生对象（JavaScript 语言为我们提供的对象类，如 Function、Array、Object 等等）的拓展，比如你想给每一个函数都添加一个检测邮箱的方法就可以这么做。”

```
Function.prototype.checkEmail = function(){
    // 验证邮箱
}
```

“这样你在使用这个方法的时候就比较方便了，如果你习惯函数形式，那么你可以这么做。”

```
var f = function(){};
f.checkEmail();
```

“如果你习惯类的形式你也可以这么做。”

```
var f = new Function();
f.checkEmail();
```

“但是你这么做在我们这里是不允许的，因为你污染了原生对象 Function，所以别人创建的函数也会被你创建的函数所污染，造成不必要的开销，但你可以抽象出一个统一添加方法的功能方法。”

```
Function.prototype.addMethod = function(name, fn){
    this[name] = fn;
}
```

“这样如果你想添加邮箱验证和姓名验证方法你可以这样做。”

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

或者

```
var methods = new Function();
methods.addMethod('checkName', function(){
    // 验证姓名
});
methods.addMethod('checkEmail', function(){
    // 验证邮箱
});
methods.checkName();
methods.checkEmail();
```

1.10 可以链式添加吗

“呀，这种方式很奇特呀。不过我想链式添加方法，是不是在 addMethod 中将 this 返回就可以呀，这么做可以么？”

```
Function.prototype.addMethod = function(name, fn){
    this[name] = fn;
    return this;
}
```

“当然，所以你再想添加方法就可以这样了：”

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

```
methods.addMethod('checkName', function(){
  // 验证姓名
}).addMethod('checkEmail', function(){
  // 验证邮箱
});
```

“那么，小白，我问你，我如果想链式使用你知道该如何做么？”

小白想了想说：“既然添加方法的时候可以将 this 返回实现，那么添加的每个方法将 this 返回是不是可以实现呢？”

于是小白这么写下：

```
var methods = function() {};
methods.addMethod('checkName', function(){
  // 验证姓名
  return this;
}).addMethod('checkEmail', function(){
  // 验证邮箱
  return this;
});
```

然后测试一下：

```
methods.checkName().checkEmail();
```

“真的可以呀！”小白兴奋地说。

1.11 换一种方式使用方法

“可是在你测试的时候，你用的是函数式调用方式？对于习惯于类式调用方式的同学来说，他们可以这样简单更改一下。”

```
Function.prototype.addMethod = function(name, fn) {
  this.prototype[name] = fn;
}
```

“此时我们还按照上一种方式添加方法。”

```
var Methods = function() {};
methods.addMethod('checkName', function(){
  // 验证姓名
}).addMethod('checkEmail', function(){
  // 验证邮箱
});
```

“但是我们在使用的时候要注意了，不能直接使用，要通过 new 关键字来创建新对象了。”

```
var m = new Methods();
m.checkEmail()
```

小白兴奋地看着这一行行的代码情不自禁地叫了一声“这正是一种艺术”。

小铭笑着说：“JavaScript 是一种灵活的语言，当然函数在其中扮演着一等公民。所以使用 JavaScript，你可以编写出更多优雅的艺术代码。”

下章剧透

在欢乐的学习中小白的第一天工作结束了，兴奋、痴迷、感慨。明天小白将去看看同事们丰富的编程世界。在那时小白将领略封装、继承的魅力。

忆之获

小白工作第一天的故事结束，通过对小白与小铭对函数的多样化创建与使用，我们对 JavaScript 这门语言有了新的认识，“灵活性”是这门语言特有的气质，不同的人可以写出不同风格的代码，这是 JavaScript 给予我们的财富，不过我们要在团队开发中慎重挥霍，尽量保证团队开发代码风格的一致性，这也是团队代码易开发、可维护以及代码规范的必然要求。

我问你答

真假对象一节中如何实现方法的链式调用呢？

试着定义一个可以为函数添加多个方法的 addMethod 方法。

试着定义一个既可为函数原型添加方法又可为其自身添加方法的 addMethod 方法。

第2章 写的都是看到的——面向对象编程

第一天的经历使小白深深认识到校园学到的知识与实际工作中的偏差，所以想见识见识公司团队里大家都是如何书写代码并完成需求的。早晨走进公司的时候恰巧遇见了项目经理。

2.1 两种编程风格——面向过程与面向对象

“早！小白，今天是你来的第二天，这一周你熟悉一下我们团队的项目吧。”项目经理对小白说。

“好呀，项目经理，我也正想跟大家学习学习呢。”于是项目经理带着小白将项目中的代码下载下来。可小白打开一看傻眼了：“函数，昨天探讨的函数呢？”小白想了半天还是没找到自己以前熟悉的代码。于是走去问小铭：“为何大家在解决需求时都不按照需求规定的功能写函数呢？怎么都是一个一个对象呢？”

“函数？对象？看来你还是习惯于按照传统流程编写一个一个函数来解决需求的方式。昨天跟你说过，那样做不利于团队开发，比如你昨天写的3个对输入框中输入的数据校验功能方法，用了3个函数，这是一种面向过程的实现方式，然而在这种方式中，你会发现无端地在页面中添加了很多全局变量，而且不利于别人重复使用。一旦别人使用以前提供的方法，你就不能轻易地去修改这些方法，这不利于团队代码维护。因此你现在要接受咱们团队这边的编程风格——面向对象编程。”

“面向对象编程？我不太理解，你可以跟我说一说么？”小白问道。

“面向对象编程就是将你的需求抽象成一个对象，然后针对这个对象分析其特征（属性）与动作（方法）。这个对象我们称之为类。面向对象编程思想其中有一个特点就是封装，就是说把你需要的功能放在一个对象里。比如你大学毕业你来公司携带的行李物品没有一件一件拿过来，而是要将他们放在一个旅行箱里，这样不论携带还是管理都会更方便一些。遗憾的是对于JavaScript这种解释性的弱类型语言没有经典强类型语言中那种通过class等关键字实现的类的封装方式，JavaScript中都是通过一些特性模仿实现的，但这也带来了极高的灵活性，让我们编写的代码更自由。”

2.2 包装明星——封装

2.2.1 创建一个类

“在 JavaScript 中创建一个类很容易，首先声明一个函数保存在一个变量里。按编程习惯一般将这个代表类的变量名首字母大写。然后在这个函数（类）的内部通过对 this（函数内部自带的一个变量，用于指向当前这个对象）变量添加属性或者方法来实现对类添加属性或者方法，例如：”

```
var Book = function(id, bookname, price){
    this.id = id;
    this.bookname = bookname;
    this.price = price;
}
```

“也可以通过在类的原型（类也是一个对象，所以也有原型 prototype）上添加属性和方法，有两种方式，一种是一一为原型对象属性赋值，另一种则是将一个对象赋值给类的原型对象。但这两种不要混用。例如：”

```
Book.prototype.display = function(){
    // 展示这本书
};
```

或者

```
Book.prototype = {
    display : function(){}
};
```

“这样我们将所需要的方法和属性都封装在我们抽象的 Book 类里面了，当使用功能方法时，我们不能直接使用这个 Book 类，需要用 new 关键字来实例化（创建）新的对象。使用实例化对象的属性或者方法时，可以通过点语法访问，例如：”

```
var book = new Book(10, 'JavaScript 设计模式', 50);
console.log(book.bookname) // JavaScript 设计模式
```

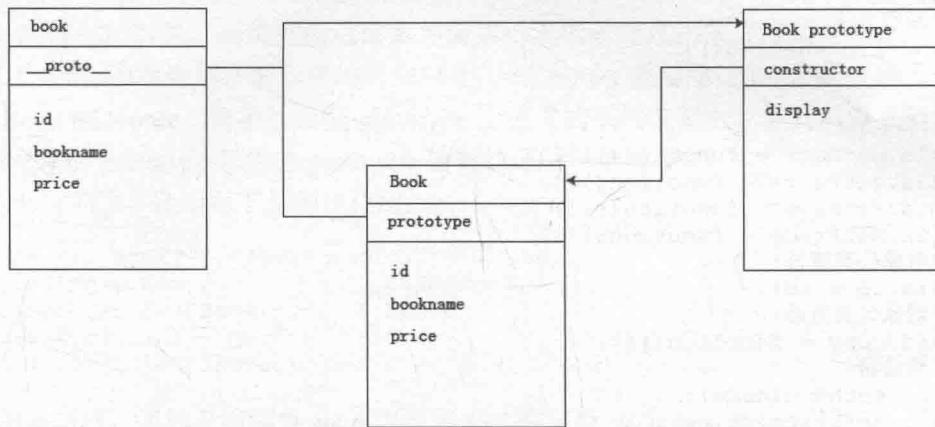
小白看了看对类添加的属性和方法部分，感觉不是很理解，于是问：“通过 this 添加的属性和方法同在 prototype 中添加的属性和方法有什么区别呀？”

“通过 this 添加的属性、方法是在当前对象上添加的，然而 JavaScript 是一种基于原型 prototype 的语言，所以每创建一个对象时（当然在 JavaScript 中函数也是一种对象），它都有一个原型 prototype 用于指向其继承的属性、方法。这样通过 prototype 继承的方法并不是对象自身的，所以在使用这些方法时，需要通过 prototype 一级一级查找来得到。这样你会发现通过 this 定义的属性或者方法是该对象自身拥有的，所以我们每次通过类创建一个新对象时，this

指向的属性和方法都会得到相应的创建，而通过 `prototype` 继承的属性或者方法是每个对象通过 `prototype` 访问到，所以我们每次通过类创建一个新对象时这些属性和方法不会再次创建。（如图 2-1 所示）。”

“哦，对了，解析图中的 `constructor` 又是指的什么呀。”

“`constructor` 是一个属性，当创建一个函数或者对象时都会为其创建一个原型对象 `prototype`，在 `prototype` 对象中又会像函数中创建 `this` 一样创建一个 `constructor` 属性，那么 `constructor` 属性指向的就是拥有整个原型对象的函数或对象，例如在本例中 `Book prototype` 中的 `constructor` 属性指向的就是 `Book` 类对象。”



▲图 2-1 原型对象 `prototype`

2.2.2 这些都是我的——属性与方法封装

“原来是这样，”小白似乎明白些，“面向对象思想在学校里也学过，说的就是对一些属性方法的隐藏与暴露，比如私有属性、私有方法、共有属性、共有方法、保护方法等等，那么 JavaScript 中也有这些么？”

“你能想到这些很好。说明你有一定面向对象的基础了。不过你说的这些在 JavaScript 中没有显性的存在，但是我们可以通过一些灵活的技巧来实现它。”小铭继续解释说，“面向对象思想你可以想象成一个人，比如一位明星为了在社会中保持一个良好形象，她就会将一些隐私隐藏在心里，然而对于这位明星，她的家人认识她，所以会了解一些关于她的事情。外界的人不认识她，即使外界人通过某种途径认识她也仅仅了解一些她暴露出来的事情，不会了解她的隐私。如果想了解更多关于她的事情怎么办？对，还可以通过她的家人来了解，但是这位明星自己内心深处的隐私是永远不会被别人知道的。”

“那么在 JavaScript 中又是如何实现的呢？”小白问。

“由于 JavaScript 的函数级作用域，声明在函数内部的变量以及方法在外界是访问不到的，通过此特性即可创建类的私有变量以及私有方法。然而在函数内部通过 `this` 创建的属性和方

法，在类创建对象时，每个对象自身都拥有一份并且可以在外部访问到。因此通过 this 创建的属性可看作是对象共有属性和对象共有方法，而通过 this 创建的方法，不但可以访问这些对象的共有属性与共有方法，而且还能访问到类（创建时）或对象自身的私有属性和私有方法，由于这些方法权利比较大，所以我们又将它看作特权方法。在对象创建时通过使用这些特权方法我们可以初始化实例对象的一些属性，因此这些在创建对象时调用的特权方法还可以看作是类的构造器。如下面的例子。”

```
// 私有属性与私有方法，特权方法，对象公有属性和对象共有方法，构造器
var Book = function(id, name, price){
    //私有属性
    var num = 1;
    //私有方法
    function checkId(){

    };
    //特权方法
    this.getName = function(){};
    this.getPrice = function(){};
    this.setName = function(){};
    this.setPrice = function(){};
    //对象公有属性
    this.id = id;
    //对象公有方法
    this.copy = function(){};
    //构造器
    this.setName(name);
    this.setPrice(price);
};
```

小白心中暗喜：“原来是这样呀，通过 JavaScript 函数级作用域的特征来实现在函数内部创建外界就访问不到的私有化变量和私有化方法。通过 new 关键字实例化对象时，由于对类执行一次，所以类的内部 this 上定义的属性和方法自然就可以复制到新创建的对象上，成为对象公有化的属性与方法，而其中的一些方法能访问到类的私有属性和方法，就像例子中家人对明星了解得比外界多，因此比外界权利大，因而得名特权方法。而我们在通过 new 关键字实例化对象时，执行了一遍类的函数，所以里面通过调用特权方法自然就可以初始化对象的一些属性了。可是在类的外部通过点语法定义的属性和方法以及在外部通过 prototype 定义的属性和方法又有什么作用呢？”

“通过 new 关键字创建新对象时，由于类外面通过点语法添加的属性和方法没有执行到，所以新创建的对象中无法获取他们，但是可以通过类来使用。因此在类外面通过点语法定义的属性以及方法被称为类的静态共有属性和类的静态共有方法。而类通过 prototype 创建的属性或者方法在类实例的对象中是可以通过 this 访问到的（如图 2.1 新创建的对象的__proto__指向了类的原型所指向的对象），所以我们将 prototype 对象中的属性和方法称为共有属性和共有方法，如：”

```
//类静态公有属性（对象不能访问）
Book.isChinese = true;
//类静态公有方法（对象不能访问）
Book.resetTime = function(){
    console.log('new Tiem')
};

Book.prototype = {
    //公有属性
    isJSBook : false,
    //公有方法
    display : function(){}
}
```

“通过 new 关键字创建的对象实质是对新对象 this 的不断赋值，并将 prototype 指向类的 prototype 所指向的对象，而类的构造函数外面通过点语法定义的属性方法是不会添加到新创建的对象上去的。因此要想在新创建的对象中使用 isChinese 就得通过 Book 类使用而不能通过 this，如 Book.isChinese，而类的原型 prototype 上定义的属性在新对象里就可以直接使用，这是因为新对象的 prototype 和类的 prototype 指向的是同一个对象。”

于是小白半信半疑地写下了测试代码：

```
var b = new Book(11,'JavaScript 设计模式',50);
console.log(b.num);           // undefined
console.log(b.isJSBook);      // false
console.log(b.id);            // 11
console.log(b.isChinese);     // undefined
```

“真的是这样，类的私有属性 num 以及静态共有属性 isChinese 在新创建的 b 对象里是访问不到的。而类的共有属性 isJSBook 在 b 对象中却可以通过点语法访问到。”

“但是类的静态公有属性 isChinese 可以通过类的自身访问。”

```
console.log(Book.isChinese);   // true
Book.resetTime();             // new Tiem
```

2.2.3 你们看不到我——闭包实现

“有时我们经常将类的静态变量通过闭包来实现。”

```
// 利用闭包实现
var Book = (function() {
    //静态私有变量
    var bookNum = 0;
    //静态私有方法
    function checkBook(name) {
    }
    //返回构造函数
    return function(newId, newName, newPrice) {
        //私有变量
        var name, price;
        //私有方法
        function checkID(id){}
```

```

//特权方法
this.getName = function(){};
this.getPrice = function(){};
this.setName = function(){};
this.setPrice = function(){};
//公有属性
this.id = newId;
//公有方法
this.copy = function(){};
bookNum++;
if(bookNum > 100)
    throw new Error('我们仅出版 100 本书。');
//构造器
this.setName(name);
this.setPrice(price);
}
})();
}

Book.prototype = {
    //静态公有属性
    isJSBook : false,
    //静态公有方法
    display : function(){}
};

```

“小白，你知道闭包么？”

“不太了解。你能说说么？”

“闭包是有权访问另外一个函数作用域中变量的函数，即在一个函数内部创建另外一个函数。我们将这个闭包作为创建对象的构造函数，这样它既是闭包又是可实例对象的函数，即可访问到类函数作用域中的变量，如 bookNum 这个变量，此时这个变量叫静态私有变量，并且 checkBook() 可称之为静态私有方法。当然闭包内部也有其自身的私有变量以及私有方法如 price, checkID()。但是，在闭包外部添加原型属性和方法看上去像似脱离了闭包这个类，所以有时候在闭包内部实现一个完整的类然后将其返回，看下面的例子。”

```

// 利用闭包实现
var Book = (function() {
    //静态私有变量
    var bookNum = 0;
    //静态私有方法
    function checkBook(name) {}
    //创建类
    function book(newId, newName, newPrice) {
        //私有变量
        var name, price;
        //私有方法
        function checkID(id){}
        //特权方法
        this.getName = function(){};
        this.getPrice = function(){};
        this.setName = function(){};
        this.setPrice = function(){};
    }
});

```

```

//公有属性
this.id = newId;
//公有方法
this.copy = function() {};
bookNum++;
if(bookNum > 100)
    throw new Error('我们仅出版 100 本书。');
//构造器
this.setName(name);
this.setPrice(price);
}
//构建原型
_book.prototype = {
    //静态公有属性
    isJSBook : false,
    //静态公有方法
    display : function(){}
};
//返回类
return _book;
})();

```

“哦，这样看上去更像一个整体。”

2.2.4 找位检察长——创建对象的安全模式

“对于你们初学者来说，在创建对象上由于不适应这种写法，所以经常容易忘记使用 new 而犯错误。”

“可是对于我们来说，这种错误发生也是不可避免的，毕竟不像你们工作了这么多年。但是你有什么好办法么？”

“哈哈，那是当然，如果你们犯错误有人实时监测不就解决了么，所以赶快找一位检察长吧。比如 JavaScript 在创建对象时有一种安全模式就完全可以解决你们这类问题。”

```

// 图书类
var Book = function(title, time, type){
    this.title = title;
    this.time = time;
    this.type = type;
}
// 实例化一本书
var book = Book('JavaScript', '2014', 'js');

```

“小白，你猜 book 这个变量是个什么？”

“Book 类的一个实例吧。”为了验证自己的想法，小白写下测试代码。

```
console.log(book); // undefined
```

“怎么会是这样？为什么是一个 undefined（未定义）？”小白不解。

“别着急，你来看看我的测试代码。”

```
console.log(window.title); // JavaScript
console.log(window.time); // 2014
console.log(window.type); // js
```

“怎么样发现问题了么”，小铭问道。

“明明创建了一个 Book 对象，并且添加了 title、time、type3 个属性，怎么会添加到 window 上面去了，而且 book 这个变量还是 undefined。”小白又看了看实例中的代码恍然大悟，“哦，原来是忘记了用 new 关键字来实例化了，可是为什么会出现这个结果呢？”

“别着急，首先你要明白一点，new 关键字的作用可以看作是对当前对象的 this 不停地赋值，然而例子中没有用 new，所以就会直接执行这个函数，而这个函数在全局作用域中执行了，所以在全局作用域中 this 指向的当前对象自然就是全局变量，在你的页面里全局变量就是 window 了，所以添加的属性自然就会被添加到 window 上面了，而我们这个 book 变量最终的作用是要得到 Book 这个类（函数）的执行结果，由于函数中没有 return 语句，这个 Book 类自然不会告诉 book 变量的执行结果了，所以就是 undefined（未定义）。”

“原来是这样，看来创建时真是不小心呀，可是该如何避免呢？”小白感叹道。

“去找位检察长’呀，哈哈，使用安全模式吧。”

```
// 图书安全类
var Book = function(title, time, type) {
    // 判断执行过程中 this 是否是当前这个对象（如果是说明是用 new 创建的）
    if(this instanceof Book) {
        this.title = title;
        this.time = time;
        this.type = type;
    } else {
        return new Book(title, time, type);
    }
}
var book = Book('JavaScript', '2014', 'js');
```



“好了小白，测试一下吧。”

```
console.log(book); // Book
console.log(book.title); // JavaScript
console.log(book.time); // 2014
console.log(book.type); // js
console.log(window.title); // undefined
console.log(window.time); // undefined
console.log(window.type); // undefined
```

“真的是这样呀，太好了，再也不用担心创建对象忘记使用 new 关键字的问题了。”

“好了说了很多，你也休息一下，好好回顾一下，后面还有个更重要的面向对象等着你一继承，这可是许多设计模式设计的灵魂。”

2.3 传宗接代——继承

“小白，看继承呢？”小铭忙完自己的事情走过来。

“是呀，刚才学习类，发现每个类都有3个部分，第一部分是构造函数内的，这是供实例化对象复制用的，第二部分是构造函数外的，直接通过点语法添加的，这是供类使用的，实例化对象是访问不到的，第三部分是类的原型中的，实例化对象可以通过其原型链间接地访问到，也是为供所有实例化对象所共用的。然而在继承中所涉及的不仅仅是一个对象。”

“对呀，不过继承这种思想却很简单，如千年文明能够流传至今靠的就是传承，将这些有用的文化一年一年地流传下来，又如我们祖先一代一代地繁衍，才有了今天的我们。所以继承涉及的不仅仅是一个对象。如人类的传宗接代，父母会把自己的一些特点传给孩子，孩子具有了父母的一些特点，但又不完全一样，总会有自己的特点，所以父母与孩子又是不同的个体。”

“可是JavaScript并没有继承这一现有的机制，它又是如何实现的呢？”

2.3.1 子类的原型对象——类式继承

“对呀，也正因为JavaScript少了这些显性的限制才使得其具有了一定的灵活性，所以我们可以根据不同的需求实现多样式的继承。比如常见的类式继承。”

```
// 类式继承
// 声明父类
function SuperClass() {
    this.superValue = true;
}
// 为父类添加共有方法
SuperClass.prototype.getSuperValue = function() {
    return this.superValue;
};
// 声明子类
function SubClass() {
    this.subValue = false;
}

// 继承父类
SubClass.prototype = new SuperClass();
// 为子类添加共有方法
SubClass.prototype.getSubValue = function () {
    return this.subValue;
};
```

“很像，真的很像！”小白很惊讶。

“像什么？”小铭不解地问。

“刚才看过的封装呀，不同的是这里声明了2个类，而且第二个类的原型prototype被赋予了第一个类的实例。”小白解释道。

“很对，继承很简单，就是声明 2 个类而已，不过类式继承需要将第一个类的实例赋值给第二个类的原型。但你知道为何要这么做么？”

“类的原型对象的作用就是为类的原型添加共有方法，但类不能直接访问这些属性和方法，必须通过原型 `prototype` 来访问。而我们实例化一个父类的时候，新创建的对象复制了父类的构造函数内的属性与方法并且将原型 `_proto_` 指向了父类的原型对象，这样就拥有了父类的原型对象上的属性与方法，并且这个新创建的对象可直接访问到父类原型对象上的属性与方法。如果我们将这个新创建的对象赋值给子类的原型，那么子类的原型就可以访问到父类的原型属性和方法。”小白还有些不自信。

“对，你分析得很准确。补充一点，你说的新创建的对象不仅可以访问父类原型上的属性和方法，同样也可访问从父类构造函数中复制的属性和方法。你将这个对象赋值给子类的原型，那么这个子类的原型同样可以访问父类原型上的属性和方法与从父类构造函数中复制的属性和方法。这正是类式继承原理。”

“原来是这样，但是我们要如何使用子类呢？”小白问道。

“使用很简单，像下面这样即可。”小铭说。

```
var instance = new SubClass();
console.log(instance.getSuperValue());      //true
console.log(instance.getSubValue());        //false
```

“另外，我们还可以通过 `instanceof` 来检测某个对象是否是某个类的实例，或者说某个对象是否继承了某个类。这样就可以判断对象与类之间的继承关系了。”小铭补充说。

“`instanceof`？它如何就知道对象与类之间的继承关系呢？”小白不解。

“`instanceof` 是通过判断对象的 `prototype` 链来确定这个对象是否是某个类的实例，而不关心对象与类的自身结构。”

“原来是这样。”于是小白写下测试代码。

```
console.log(instance instanceof SuperClass);    //true
console.log(instance instanceof SubClass);      //true
console.log(SubClass instanceof SuperClass);    //false
```

“我们说 `subClass` 继承 `superClass`，可是为什么 `SubClass instanceof SuperClass` 得到的结果是 `false` 呢？”小白不解。

“前面说了，`instanceof` 是判断前面的对象是否是后面类（对象）的实例，它并不表示两者的继承，这一点你不要弄混，其次我们看看前面的代码，你看我们在实现 `subClass` 继承 `superClass` 时是通过将 `superClass` 的实例赋值给 `subClass` 的原型 `prototype`，所以说 `SubClass.prototype` 继承了 `superClass`。”小铭解释说。

于是小白半信半疑地写下测试代码。

```
console.log(SubClass.prototype instanceof SuperClass); //true
```

“真的是这样。”小白惊呼。

“是呀。这也是类式继承的一个特点。问你一个问题，你所创建的所有对象都是谁的实例？”

“Object 吗？”小白回答说。

“对，正是 JavaScript 为我们提供的原生对象 Object，所以你看下面的检测代码返回的就是 true。”

```
console.log(instance instanceof Object); //true
```

“哦，这么说 Object 就是所有对象的祖先了。”小白笑着说。

“哈哈，可是你知道吗，这种类式继承还有 2 个缺点。其一，由于子类通过其原型 prototype 对父类实例化，继承了父类。所以说父类中的共有属性要是引用类型，就会在子类中被所有实例共用，因此一个子类的实例更改子类原型从父类构造函数中继承来的共有属性就会直接影响到其他子类，比如你看下面的代码。”

```
function SuperClass() {
  this.books = ['JavaScript', 'html', 'css'];
}
function SubClass(){}
SubClass.prototype = new SuperClass();
var instance1 = new SubClass();
var instance2 = new SubClass();
console.log(instance2.books); // ["JavaScript", "html", "css"]
instance1.books.push('设计模式');
console.log(instance2.books); // ["JavaScript", "html", "css", "设计模式"]
```

“instance1 的一个无意的修改就会无情地伤害了 instance2 的 book 属性，这在编程中很容易埋藏陷阱。其二，由于子类实现的继承是靠其原型 prototype 对父类的实例化实现的，因此在创建父类的时候，是无法向父类传递参数的，因而在实例化父类的时候也无法对父类构造函数内的属性进行初始化。”

“那我们要如何解决这些问题呢？”小白好奇地追问。

2.3.2 创建即继承——构造函数继承

“别着急，JavaScript 是灵活的，自然也会有其他继承方法来解决，比如常见的构造函数继承。”

```
//构造函数式继承
// 声明父类
function SuperClass(id) {
  // 引用类型共有属性
  this.books = ['JavaScript', 'html', 'css'];
  // 值类型共有属性
  this.id = id;
}
// 父类声明原型方法
SuperClass.prototype.showBooks = function() {
```

```

        console.log(this.books);
    }
    // 声明子类
    function SubClass(id) {
        // 继承父类
        SuperClass.call(this, id);
    }
    // 创建第一个子类的实例
    var instance1 = new SubClass(10);
    // 创建第二个子类的实例
    var instance2 = new SubClass(11);

    instance1.books.push("设计模式");
    console.log(instance1.books);      // ["JavaScript", "html", "css", "设计模式"]
    console.log(instance1.id);         // 10
    console.log(instance2.books);      // ["JavaScript", "html", "css"]
    console.log(instance2.id);         // 11

    instance1.showBooks();            // TypeError

```

“小白，注意这里。SuperClass.call(this, id);这条语句是构造函数式继承的精华，由于 call这个方法可以更改函数的作用环境，因此在子类中，对 superClass 调用这个方法就是将子类中的变量在父类中执行一遍，由于父类中是给 this 绑定属性的，因此子类自然也就继承了父类的共有属性。由于这种类型的继承没有涉及原型 prototype，所以父类的原型方法自然不会被子类继承，而如果要想被子类继承就必须要放在构造函数中，这样创建出来的每个实例都会单独拥有一份而不能共用，这样就违背了代码复用的原则。为了综合这两种模式的优点，后来有了组合式继承。”

2.3.3 将优点为我所用——组合继承

“组合继承是不是说将这两种继承模式综合到一起呀？那么它又是如何做到的呢？”

“别着急，我们先总结一下之前两种模式的特点，类式继承是通过子类的原型 prototype 对父类实例化来实现的，构造函数式继承是通过在子类的构造函数作用环境中执行一次父类的构造函数来实现的，所以只要在继承中同时做到这两点即可，看下面的代码。”

```

// 组合式继承
// 声明父类
function SuperClass(name) {
    // 值类型共有属性
    this.name = name;
    // 引用类型共有属性
    this.books = ["html", "css", "JavaScript"];
}
// 父类原型共有方法
SuperClass.prototype.getName = function() {
    console.log(this.name);
};
// 声明子类
function SubClass(name, time) {

```

```

// 构造函数式继承父类 name 属性
SuperClass.call(this, name);
// 子类中新增共有属性
this.time = time;
}
// 类式继承 子类原型继承父类
SubClass.prototype = new SuperClass();
// 子类原型方法
SubClass.prototype.getTime = function(){
    console.log(this.time);
};

```

“小白看到没，在子类构造函数中执行父类构造函数，在子类原型上实例化父类就是组合模式，这样就融合了类式继承和构造函数继承的优点，并且过滤掉其缺点，你测试看看。”

于是小白写下测试代码。

```

var instance1 = new SubClass("js book", 2014);
instance1.books.push("设计模式");
console.log(instance1.books);      // ["html", "css", "JavaScript", "设计模式"]
instance1.getName();              // js book
instance1.getTime();              // 2014

var instance2 = new SubClass("css book", 2013);
console.log(instance2.books);      // ["html", "css", "JavaScript"]
instance2.getName();              // css book
instance2.getTime();              // 2013

```

“真的是这样呀，”小白兴奋地说，“子类的实例中更改父类继承下来的引用类型属性如 books，根本不会影响到其他实例，并且子类实例化过程中又能将参数传递到父类的构造函数中，如 name。这种模式真的很强大，所以这应该是继承中最完美的版本吧？”

“还不是，因为我们在使用构造函数继承时执行了一遍父类的构造函数，而在实现子类原型的类式继承时又调用了一遍父类构造函数。因此父类构造函数调用了两遍，所以这还不是最完美的方式。”

“难道还有更好的方式么？”

“那当然，JavaScript 很灵活嘛。不过在学习这种方式之前我们先学习一个简单而很常用的方式。”

2.3.4 洁净的继承者——原型式继承

“2006 年道格拉斯·克罗克福德发表一篇《JavaScript 中原型式继承》的文章，他的观点是，借助原型 prototype 可以根据已有的对象创建一个新的对象，同时不必创建新的自定义对象类型。大师的话理解起来可能很困难，不过我们还是先看一下他实现的代码吧。”

```

// 原型是继承
function inheritObject(o){
    // 声明一个过渡函数对象
    function F() {}

```

```

    // 过渡对象的原型继承父对象
    F.prototype = o;
    // 返回过渡对象的一个实例，该实例的原型继承了父对象
    return new F();
}

```

“这种方式怎么和类式继承有些像呢？”

“对，它是对类式继承的一个封装，其实其中的过渡对象就相当于类式继承中的子类，只不过在原型式中作为一个过渡对象出现的，目的是为了创建要返回的新的实例化对象。”

“如果是这样，是不是类式继承中的问题在这里也会出现呢？”小白追问。

“是这样的，”小铭接着说，“不过这种方式由于F过渡类的构造函数中无内容，所以开销比较小，使用起来也比较方便。当然如果你感觉有必要可以将F过渡类缓存起来，不必每次创建一个新过渡类F。当然这种顾虑也是不必要的。随着对这种思想的深入，后来就出现了Object.create()的方法。”

“创建的新对象会不会影响到父类对象呢？”于是小白写下测试用例（测试代码）。

```

var book = {
  name: "js book",
  alikeBook: ["css book", "html book"]
};
var newBook = inheritObject(book);
newBook.name = "ajax book";
newBook.alikeBook.push("xml book");

var otherBook = inheritObject(book);
otherBook.name = "flash book";
otherBook.alikeBook.push("as book");

console.log(newBook.name);           // ajax book
console.log(newBook.alikeBook);      // ["css book", "html book", "xml book",
"as book"]
console.log(otherBook.name);         // flash book
console.log(otherBook.alikeBook);    // ["css book", "html book", "xml book",
"as book"]
console.log(book.name);             // js book
console.log(book.alikeBook);        // ["css book", "html book", "xml book",
"as book"]

```

“跟类式继承一样，父类对象book中的值类型的属性被复制，引用类型的属性被共用。”小白感叹道。

“然而道格拉斯·克罗克福德推广的继承并不只这一种，他在此基础上做了一些增强而推出一种寄生式继承。”

2.3.5 如虎添翼——寄生式继承

“寄生式继承？这还头一次听说，它是怎么实现的？”

“不着急，大师对该模式论述的话我们就不深究了，我们还是看看这种继承的实现吧。”

```
// 寄生式继承
// 声明基对象
var book = {
  name: "js book",
  alikeBook: ["css book", "html book"]
};

function createBook(obj) {
  // 通过原型继承方式创建新对象
  var o = new inheritObject(obj);
  // 拓展新对象
  o.getName = function() {
    console.log(name);
  };
  // 返回拓展后的新对象
  return o;
}
```

“看懂了吗？”小铭问小白，“其实寄生式继承就是对原型继承的第二次封装，并且在这第二次封装过程中对继承的对象进行了拓展，这样新创建的对象不仅仅有父类中的属性和方法而且还添加新的属性和方法。”

“哦，这种类型的继承果如其名，寄生大概指的就是像寄生虫一样寄托于某个对象内部生长。当然寄生式继承这种增强新创建对象的继承思想也是寄托于原型继承模式吧。”

“嗯，是这个道理，而这种思想的作用也是为了寄生组合式继承模式的实现。”

2.3.6 终极继承者——寄生组合式继承

“寄生组合式继承？”小白好奇地问道。

“嗯，之前我们学习了组合式继承，那时候我们将类式继承同构造函数继承组合使用，但是这种方式有一个问题，就是子类不是父类的实例，而子类的原型是父类的实例，所以才有了寄生组合式继承。但是你知道是哪两种模式的组合么？”

“寄生当然是寄生式继承，寄生式继承依托于原型继承，原型继承又与类式继承相像，另外一种就不应该是这些模式了，所以另外一种继承模式应该是构造函数继承了吧。当然，子类不是父类实例的问题是由于类式继承引起的。”小白回答道。

“对，正是这两种继承，但是这里寄生式继承有些特殊，这里它处理的不是对象，而是类的原型。我们再次来看看道格拉斯·克罗克福德对寄生式继承的一个改造。”

```
/**
 * 寄生式继承 继承原型
 * 传递参数 subClass 子类
 * 传递参数 superClass 父类
 */
function inheritPrototype(subClass, superClass) {
  // 复制一份父类的原型副本保存在变量中
  var p = inheritObject(superClass.prototype);
  // 修正因为重写子类原型导致子类的 constructor 属性被修改
  p.constructor = subClass;
  // 设置子类的原型
```

```

    subClass.prototype = p;
}

```

“组合式继承中，通过构造函数继承的属性和方法是没有问题的，所以这里我们主要探究通过寄生式继承重新继承父类的原型。我们需要继承的仅仅是父类的原型，不再需要调用父类的构造函数，换句话说，在构造函数继承中我们已经调用了父类的构造函数。因此我们需要的就是父类的原型对象的一个副本，而这个副本我们通过原型继承便可得到，但是这么直接赋值给子类会有问题的，因为对父类原型对象复制得到的复制对象 p 中的 constructor 指向的不是 subClass 子类对象，因此在寄生式继承中要对复制对象 p 做一次增强，修复其 constructor 属性指向不正确的问题，最后将得到的复制对象 p 赋值给子类的原型，这样子类的原型就继承了父类的原型并且没有执行父类的构造函数。”

“看上去好复杂呀。”小白惊叹道。

“所以你要去测一测呀。测试很简单，与组合模式相比只有一个地方做了修改，就是子类原型继承父类原型这一处，测试看看吧。”

“好吧。”于是小白写下了测试用例。

```

// 定义父类
function SuperClass(name) {
    this.name = name;
    this.colors = ["red", "blue", "green"];
}
// 定义父类原型方法
SuperClass.prototype.getName = function() {
    console.log(this.name);
};
// 定义子类
function SubClass(name, time) {
    // 构造函数式继承
    SuperClass.call(this, name);
    // 子类新增属性
    this.time = time;
}
// 寄生式继承父类原型
inheritPrototype(SubClass, SuperClass);
// 子类新增原型方法
SubClass.prototype.getTime = function() {
    console.log(this.time);
};
// 创建两个测试方法
var instance1 = new SubClass("js book", 2014);
var instance2 = new SubClass("css book", 2013);

```

小白首先创建了父类，以及父类的原型方法，然后创建子类，并在构造函数中实现构造函数式继承，然后又通过寄生式继承了父类原型，最后又对子类添加了一些原型方法。

最后小白测试了一下，结果如下：

```

instance1.colors.push("black");

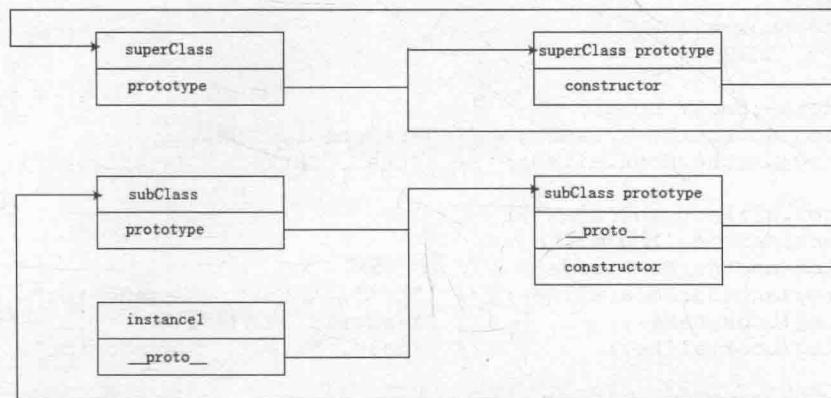
```

```

console.log(instance1.colors); //["red", "blue", "green", "black"]
console.log(instance2.colors); //["red", "blue", "green"]
instance2.getName(); //css book
instance2.getTime(); //2013

```

“现在你明白了吧，其实这种方式继承如图 2-2 所示，其中最大的改变就是对子类原型的处理，被赋予父类原型的一个引用，这是一个对象，因此这里有一点你要注意，就是子类再想添加原型方法必须通过 `prototype` 对象，通过点语法的形式一个一个添加方法了，否则直接赋予对象就会覆盖掉从父类原型继承的对象了。”



▲图 2-2 继承原理

2.4 老师不止一位——多继承

“是这样呀，对了，我记得有一些面向对象语言中支持多继承，在 JavaScript 中能实现么？”

“嗯，不过是有一些局限性的。你知道，在 JavaScript 中继承是依赖于原型 `prototype` 链实现的，只有一条原型链，所以理论上是不能继承多个父类的。然而 JavaScript 是灵活的，通过一些技巧方法你却可以继承多个对象的属性来实现类似的多继承。”小铭接着说，“讲解多继承之前先跟你说一下当前很流行的一个用来继承单对象属性的 `extend` 方法。”

```

// 单继承 属性复制
var extend = function(target, source) {
  // 遍历源对象中的属性
  for (var property in source) {
    // 将源对象中的属性复制到目标对象中
    target[property] = source[property];
  }
  // 返回目标对象
  return target;
};

```

“原来 `extend` 方法的实现就是对对象中的属性的一个复制过程呀。”小白惊讶地说。

“嗯，是这样，我们的这个 `extend` 方法是一个浅复制过程，他只能复制值类型的属性，对于引用类型的属性它无能为力。而在 `jquery` 等一些框架中实现了深复制，就是将源对象中的引用类型的属性再执行一遍 `extend` 方法而实现的。我们这里实现得比较简单，所以你测试也比较容易。”

于是小白写下如下测试代码。

```
var book = {
    name : 'JavaScript 设计模式',
    alike : ['css', 'html', 'JavaScript']
}
var anotherBook = {
    color : 'blue'
}
extend(anotherBook, book);
console.log(anotherBook.name); // JavaScript 设计模式
console.log(anotherBook.alike); // ["css", "html", "JavaScript"]

anotherBook.alike.push('ajax');
anotherBook.name = '设计模式';
console.log(anotherBook.name); // 设计模式
console.log(anotherBook.alike); // ["css", "html", "JavaScript", "ajax"]
console.log(book.name); // JavaScript 设计模式
console.log(book.alike); // ["css", "html", "JavaScript", "ajax"]
```

“真的是这样。但是多继承呢？”

“很容易，既然上面的方法可以实现对一个对象属性的复制继承，那么如果我们传递多个对象呢？”

```
// 多继承 属性复制
var mix = function() {
    var i = 1,
        len = arguments.length,           // 从第二个参数起为被继承的对象
        target = arguments[0],            // 获取参数长度
        arg;                            // 第一个对象为目标对象
        arg;                            // 缓存参数对象

    // 遍历被继承的对象
    for(; i < len; i++) {
        // 缓存当前对象
        arg = arguments[i];
        // 遍历被继承对象中的属性
        for (var property in arg) {
            // 将被继承对象中的属性复制到目标对象中
            target[property] = arg[property];
        }
    }

    // 返回目标对象
    return target;
};
```

“`mix` 方法的作用就是将传入的多个对象的属性复制到源对象中，这样即可实现对多个对

象的属性的继承。”

“这是实现方式真不错，可是使用的时候需要传入目标对象（第一个参数——需要继承的对象）。”

“当然你也可以将它绑定到原生对象 Object 上，这样所有的对象就可以拥有这个方法了。”

```
Object.prototype.mix = function(){
    var i = 0,                                // 从第一个参数起为被继承的对象
        len = arguments.length,                // 获取参数长度
        arg;                                  // 缓存参数对象
    // 遍历被继承的对象
    for(; i < len; i++) {
        // 缓存当前对象
        arg = arguments[i];
        // 遍历被继承对象中的属性
        for (var property in arg) {
            // 将被继承对象中的属性复制到目标对象中
            this[property] = arg[property];
        }
    }
}
```

“这样我们就可以在对象上直接调用了。如……”

```
otherBook.mix(book1, book2);
console.log(otherBook); // Object {color: "blue", name: "JavaScript 设计模式",
" , mix: function, about: "一本 JavaScript 书"}
```

“在 JavaScript 中实现的多继承是如此的美妙。”

2.5 多种调用方式——多态

“小铭，在面向对象编程中不是还有一种特性叫作多态么？在 JavaScript 中可以实现么？”

“多态，就是同一个方法多种调用方式吧。在 JavaScript 中也是可以实现的，只不过要对传入的参数做判断以实现多种调用方式，如我们定义一个 add 方法，如果不传参数则返回 10，如果传一个参数则返回 10+参数，如果传两个参数则返回两个参数相加的结果。”

```
//多态
function add(){
    // 获取参数
    var arg = arguments,
        // 获取参数长度
        len = arg.length;
    switch(len) {
        // 如果没有参数
        case 0:
            return 10;
        // 如果只有一个参数
        case 1:
            return 10 + arg[0];
    }
}
```

```

    // 如果有两个参数
    case 2:
        return arg[0] + arg[1];
    }
}
// 测试用例
console.log(add());           // 10
console.log(add(5));          // 15
console.log(add(6, 7));        // 13

```

“当然我们还可以让其转化成更易懂的类形式：”

```

function Add(){
    // 无参数算法
    function zero(){
        return 10;
    }
    // 一个参数算法
    function one(num){
        return 10 + num;
    }
    // 两个参数算法
    function two(num1, num2){
        return num1 + num2;
    }
    // 相加共有方法
    this.add = function(){
        var arg = arguments,
            // 获取参数长度
            len = arg.length;
        switch(len){
            // 如果没有参数
            case 0:
                return zero();
            // 如果只有一个参数
            case 1:
                return one(arg[0]);
            // 如果有两个参数
            case 2:
                return two(arg[0], arg[1]);
        }
    }
}
// 实例化类
var A = new Add();
// 测试
console.log(A.add());           // 10
console.log(A.add(5));          // 15
console.log(A.add(6, 7));        // 13

```

“对于多态类，当我们调用 add 运算方法时，他会根据传参不同做相应运算，当然我们将不同运算方法封装在类内，这样代码更易懂。”

下章剧透

两天的学习让小白深深陶醉于一种新的编程方式——面向对象编程，可是这种编程方式又

有什么用呢？下章我们将伴随小白看看面向对象编程在实战中的应用，并开始学习设计模式。

忆之获

封装与继承是面向对象中的两个主要特性，继承即是对原有对象的封装，从中创建私有属性、私有方法、特权方法、共有属性、共有方法等，对于每种属性与每种方法特点是不一样的，有的不论对类如何实例化，它只创建一次，那么这类属性或者方法我们称之为静态的。有的只被类所拥有，那么这类属性和方法又是静态类方法与静态类属性。当然可被继承的方法与属性无外乎两类，一类在构造函数中，这类属性与方法在对象实例化时被复制一遍。另一类在类的原型对象中，这类属性与方法在对象实例化时被所有实例化对象所共用。

提到类的实例化我们就引出了继承，当然如果实例化的是对象那么则为对象继承，如果实例化的是类（当然类也是一种对象，只不过是用来创建对象的），那么就是一种类的继承。对于类的继承我们根据继承的方式又分为很多种，通过原型链继承的方式我们称之为类式继承，通过构造函数继承的方式我们称之为构造函数式继承，那么将这两种方式组合起来的继承方式我们称之为组合继承，由于类式继承过程中会实例化父类，这样如果父类构造函数极其复杂，那么这种方式对构造函数的开销是不值得的，此时有了一种新的继承方式，通过在一个函数内的过渡对象实现继承并返回新对象的方式我们称之为寄生式继承，此时我们在结合构造函数时继承，这样再融合构造函数继承中的优点并去除其缺点，得到的继承方式我们称之为寄生组合式继承。当然有时候子类对父类实现继承可以通过拷贝方法与属性的方式来实现，这就有了多继承，即将多个父类（对象）的属性与方法拷贝给子类实现继承。

对于面向对象中的多态，在 JavaScript 中实现起来就容易得多了，通过对传递的参数判断来决定执行逻辑，即可实现一种多态处理机制。

我问你答

多继承中我们通过对对象属性与方法浅复制实现继承，想一想如何才能实现深复制呢？（浅复制中复制对象的方法对象实质是一种指向引用，所以在深复制中要把该对象中的引用类型属性细化成值类型拷贝到目标对象中。）

第二篇

创建型设计模式

创建型设计模式是一类处理对象创建的设计模式，通过某种方式控制对象的创建来避免基本对象创建时可能导致设计上的问题或增加设计上的复杂度。

第3章 神奇的魔术师——简单工厂模式

第4章 给我一张名片——工厂方法模式

第5章 出现的都是幻觉——抽象工厂模式

第6章 分即是合——建造者模式

第7章 语言之魂——原型模式

第8章 一个人的寂寞——单例模式

第3章 神奇的魔术师——简单工厂模式

简单工厂模式（Simple Factory）：又叫静态工厂方法，由一个工厂对象决定创建某一种产品对象类的实例。主要用来创建同一类对象。

小白经过几天对团队代码的学习，终于对面向对象思想有所认知，自己跳动的心脏跃跃欲试，信心满满准备大显身手……

3.1 工作中的第一次需求

“小白，这几天学习得怎么样？登录模块的需求你能来处理一下么？”项目经理问。

“没问题。”小白答道。

“不过用户名输入框这里如果用户输入的内容不符合规范就自定义个警示框警示一句‘用户名不能多于 16 个字母或数字’。”

“好的。”于是小白简简单单写下了一个警示框类。

```
var LoginAlert = function(text) {
    this.content = text;
}
LoginAlert.prototype.show = function() {
    // 显示警示框
}
var userNameAlert = new LoginAlert('用户名不能多于 16 个字母或数字');
userNameAlert.show();
```

“关于用户输入的密码也有一个需求，就是当用户输入的密码错误时也提示一句‘输入的密码不正确’的提示文案吧。”

“没问题，”小白心中暗喜，“面向对象编程就是好用，这不刚写完的类，这么快就用上了，小 case。”

```
var passwordAlert = new LoginAlert('输入的密码不正确');
passwordAlert.show();
```

“小白，用户登录时如果用户名不存在你提示一句‘您的用户名不存在，请重新输入’”。

“没问题”，小白心中暗喜。

“不过这里有些变化，就是要在警示框中添加一个注册按钮”。

“添加一个按钮……”，小白傻眼了，心想，“以前的功能这可怎么复用呀，唉，又得创建一个类了。”

```
var loginConfirm = function(text) {
    this.content = text;
}
loginConfirm.prototype.show = function(){
    // 显示确认框
}
var loginFailConfirm = new LoginConfirm('您的用户名不存在，请重新输入');
loginFailConfirm.show();
```

“小白，登录成功后给出一个自定义提示框，除了有确定取消按钮，也提示一句‘欢迎回来，请输入您今天的心情吧’。”

“这又是一个新类……”小白感叹道。于是只好添加一个类。

```
var LoginPrompt = function(text) {
    this.content = text;
}
LoginPrompt.prototype.show = function(){
    // 显示提示框
}
```

利用了大半天时间，小白终于把这些需求写完了。

“小铭，注册模块你来做吧，当用户输入的新用户名规范不正确的时候提示‘用户名不能多于 16 个字母或数字’，当用户输入的两次密码不正确时提示‘您两次输入的密码不统一，请重新输入’，当用户的邮箱不规范时提示……”项目经理吩咐着小铭余下的工作。

“项目经理，登录模块谁做的，里面是不是也有提示框之类的需求呀？”小铭问项目经理。

“小白负责的，不过他那里的情况要比这还多几种。”

“好的，”于是小铭来找小白，“小白，你写了登录模块吧，把你写的方法借我用一下”。

“好呀，我是通过对类实例化对象实现的。”小白回应说。

“那太好了，我这边就可以无缝使用了，快给我。”

3.2 如果类太多，那么提供一个

于是小白将 LoginAlert、LoginConfirm、LoginPrompt 3 个类告诉了小铭。

“怎么这么多？还是以 Login 为前缀，这样吧，你写个简单工厂给我吧。”

“简单工厂？这是什么？”小白好奇地问。

“这是一种模式呀，我的意思是说你给我 3 个类，在每次创建时还要找到对应的类，太麻烦了，而且在注册（regist）模块用你的 login 前缀也不太好，所以你最好封装在一个函数里，这样我只需要记住这个函数，然后通过这个函数就可以创建我需要的对象为我所用岂不是更好

么？这样不但对我，而且其他人都不用再关注创建这些对象到底依赖于哪个基类了，而我们知道这个函数就可以了。这个函数通常也被称为工厂函数，这种模式叫简单工厂模式，它是工厂模式中最简单的一种形式。你看它很像一个会变戏法的魔术师，你想要魔术师为你变的礼物，不过你不需要知道魔术师是用什么变的，你只需要知道有这位魔术师就行，然后找他就能帮你得到你想要的东西。”

“原来是这样，可是我该如何改呢？”

“举个例子，比如说体育商品店卖体育器材，里面有很多体育用品，及其相关介绍等。当你来到体育用品店买一个篮球及其相关介绍时，你只需要问售货员，他会帮你找到你所要的东西。”

```
// 篮球基类
var Basketball = function(){
    this.intro = '篮球盛行于美国';
}
Basketball.prototype = {
    getMember : function(){
        console.log('每个队伍需要 5 名队员');
    },
    getBallSize : function(){
        console.log('篮球很大');
    }
}
// 足球基类
var Football = function(){
    this.intro = '足球在世界范围内很流行';
}
Football.prototype = {
    getMember : function(){
        console.log('每个队伍需要 11 名队员');
    },
    getBallSize : function(){
        console.log('足球很大');
    }
}
// 网球基类
var Tennis = function(){
    this.intro = '每年有很多网球系列赛';
}
Tennis.prototype = {
    getMember : function(){
        console.log('每个队伍需要 1 名队员');
    },
    getBallSize : function(){
        console.log('网球很小');
    }
}
// 运动工厂
var SportsFactory = function(name){
    switch(name){
        case 'NBA':
            return new Basketball();
        case 'wordCup':
            return new Football();
        case 'Tennis':
            return new Tennis();
    }
}
```

```

        return new Football();
    case 'FrenchOpen':
        return new Tennis();
    }
}

```

“当你想和小伙伴踢足球，只需要告诉店员我要买个足球即可。你使用这个商店工厂时仅仅需要记住 SportsFactory 这个工厂对象就好了。这个工厂魔术师会帮你找到你需要的一切。”

```

// 为世界杯创建一个足球，只需要记住运动工厂 SportsFactory，调用并创建
var footnall = SportsFactory('wordCup');
console.log(footnall);
console.log(footnall.intro);
footnall.getMember();

```

“很简单！好了，去把你的代码改一下吧，别忘了告诉我你创建的那位‘魔术师’。”很快小白就把自定义弹框这位“魔术师”请来了。

```

var PopFactory = function(name) {
    switch(name) {
        case 'alert':
            return new LoginAlert();
        case 'confirm':
            return new LoginConfirm();
        case 'prompt':
            return new LoginPrompt();
    }
}

```

“很不错！”小铭夸道。然后又看了看小白之前写的 LoginAlert、LoginConfirm、LoginPrompt 3 个类，皱了一下眉头，说：“这 3 个类有很多地方是相同的，是可以抽象提取出来共用的，你也可以用简单工厂的方式实现他们。”

3.3 一个对象有时也可代替许多类

“这怎么实现？还是跟上面一样的结构？”小白很惊讶。

“不太一样，详细跟你说一下吧，简单工厂模式的理念就是创建对象，像上面那种方式是对不同的类实例化，不过除此之外简单工厂模式还可以用来创建相似对象。而你创建的这 3 个类（对象）很多地方都比较相似，比如都有关闭按钮，都有提示文案等。所以你可以通过将这些相似东西提取，不相似的针对性处理即可。这有点类似我们之前学习继承时的寄生模式，很像，但是又不太一样，因为这里没有父类，所以无需做任何继承，你只需简单创建一个对象即可，然后通过对这个对象大量拓展方法和属性，并在最终将对象返回出来。”

“举个例子，比如你想创建一些书，那么这些书都有一些相似的地方，比如目录、页码等。也有一些不相似的地方，如书名、出版时间、书的类型等，对于创建的对象相似的属性好处理，对于不同的属性就要有针对性地进行处理了，比如我们将不同的属性作为参数传递进来处理。”

```
//工厂模式
function createBook(name, time, type) {
    // 创建一个对象，并对对象拓展属性和方法
    var o = new Object();
    o.name = name;
    o.time = time;
    o.type = type;
    o.getName = function() {
        console.log(this.name);
    };
    // 将对象返回
    return o;
}

var book1 = createBook("js book", 2014, "js");
var book2 = createBook("css book", 2013, "css");

book1.getName();
book2.getName();
```

“真的很想寄生式继承，只不过这里的 o 没有继承任何类或对象。”

“所以你这 3 个类要改成一个工厂模式也就很简单了，首先抽象他们的相同点，比如共有属性 this.content，原型共有方法 show，当然也有不同点，比如确认框与提示框的确定按钮，比如提示框的用户输入框等等，所以你就可以像下面这样创建了。”

```
function createPop(type, text) {
    // 创建一个对象，并对对象拓展属性和方法
    var o = new Object();
    o.content = text;
    o.show = function() {
        // 显示方法
    };
    if(type == 'alert') {
        // 警示框差异部分
    }
    if(type == 'prompt') {
        // 提示框差异部分
    }
    if(type == 'confirm') {
        // 确认框差异部分
    }
    // 将对象返回
    return o;
}
// 创建警示框
var userNameAlert = createPop('alert', '用户名只能是 26 个字母和数字');
```

3.4 你的理解决定你选择的方式

“但是小白这两个‘魔术师’工厂还是有一定区别的，好好想一想，你知道是什么吗？”小铭问。

“第一种是通过类实例化对象创建的，第二种是通过创建一个新对象然后包装增强其属性和功能来实现的。他们之间的差异性也造成前面通过类创建的对象，如果这些类继承同一父类，那么他们的父类原型上的方法是可以共用的。而后面寄生方式创建的对象都是一个新的个体，所以他们的方法就不能共用了。当然选择哪种工厂方式来实现你的需求还要看你是如何分析你的需求的。”

下章剧透

陪伴着小白第一次公司团队小项目实战让我们认识了简单工厂模式，不过这是工厂中最简单的一种。明天我们将同小白一起学习一种更复杂的工厂模式，不见不散哦。

忆之获

团队项目开发不同于个人开发，其对全局变量的限制很大，所以我们要尽量少地创建全局变量。对于同一类对象在不同需求中的重复性使用，很多时候不需要重复创建，代码复用是面向对象编程的一条准则。通过对简单工厂来创建一些对象，可以让这些对象共用一些资源而又私有一些资源，这是一种很不错的实践。不过对于简单工厂模式，它的使用场合通常也就限制在创建单一对象。

我问你答

谈谈简单工厂模式与类的异同点。

第4章 给我一张名片——工厂方法模式

工厂方法模式（Factory Method）：通过对产品类的抽象使其创建业务主要负责用于创建多类产品的实例。

广告是公司主要的一个经济来源，这不，很多企业等着要来公司首页打广告呢。

4.1 广告展现

“小白，咱们新来了一批广告资源需要投放，关于计算机培训的。一批是 Java 的，用绿色字体。还有一批是 PHP 的，要用黄色字体，红色背景。”

“没问题，于是小白准备创建两个类，然后通过实例对象方式来完成这个需求”。

```
// 创建 Java 学科类
var Java = function(content) {
    // 将内容保存在 content 里面以备日后使用
    this.content = content;
    // 创建对象时，通过闭包，直接执行，将内容按需求的样式插入到页面内
    (function(content) {
        var div = document.createElement('div');
        div.innerHTML = content;
        div.style.color = 'green';
        document.getElementById('container').appendChild(div);
    })(content);
}
// 创建 PHP 学科类
var Php : function(content) {
    this.content = content;
    (function(content) {
        var div = document.createElement('div');
        div.innerHTML = content;
        div.style.color = 'yellow';
        div.style.background = 'red';
        document.getElementById('container').appendChild(div);
    })(content);
}
```

刚写完就听到身后的喊声：“小白，又来了一批广告，关于 JavaScript 的，要求背景色是

粉色……”

“好吧，”突然间小白想起昨天学习的简单工厂模式，心想，“正好派上用场了，就用简单工厂模式去实现吧，这样日后再创建对象直接找工厂就好了。”

```
// 创建 Java 学科类
var Java = function(content) {
    // .....
}

// 创建 PHP 学科类
var Php = function(content) {
    // .....
}

// 创建 JavaScript 学科
var JavaScript = function(content) {
    this.content = content;
    (function(content) {
        var div = document.createElement('div');
        div.innerHTML = content;
        div.style.background = 'pink';
        document.getElementById('container').appendChild(div);
    })(content);
}

// 学科类工厂
function JobFactory(type, content) {
    switch(type) {
        case 'java' :
            return new Java(content);
        case 'php' :
            return new Php(content);
        case 'JavaScript' :
            return new JavaScript(content);
    }
}
```

然后写了一个测试案例：

```
JobFactory('JavaScript', 'JavaScript 哪家强');
```

“小白，又来了一批 UI 学科，红色边框……”

小白沉默了……

4.2 方案的抉择

小铭见状走了过来。“怎么了，小白？”

“需求总在变，我不知道哪种解决方式更好，开始需求简单，我就直接创建对象，后来需求多了，我就用简单工厂方法重构，而现在又变了，我不仅仅要添加类，还要修改工厂函数。而我更担心的是未来的需求还会变……”

“这样呀，”小铭微笑一下，“不用担心，需求变是正常的，而我们还有更好的模式可以应

用，刚才你用简单工厂模式遇到的问题就是每添加一个类就要修改两处是吧，所以说你可以试用一下工厂方法模式。这样以后每需要一个类，你只需要添加这个类就行，其他的不用操心了。”

“工厂方法？这是一个什么样的模式？”

“工厂方法模式本意是说将实际创建对象工作推迟到子类当中。这样核心类就成了抽象类，不过对于 JavaScript 不必这么深究，JavaScript 没有像传统创建抽象类那样的方式轻易创建抽象类，所以在 JavaScript 中实现工厂方法模式我们只需要参考它的核心思想即可。所以我们可以将工厂方法看作是一个实例化对象的工厂类。安全起见，我们采用安全模式类，而我们将创建对象的基类放在工厂方法类的原型中即可。”

4.3 安全模式类

小白有些不懂，打断小铭的话：“小铭，什么叫安全模式类？你说的我不是很懂，你能详细说明一下么？”

“安全模式类是说可以屏蔽使用这对类的错误使用造成的错误。比如对于一个类（我们暂且称之为 Demo 类）的创建，我们知道类的前面是需要有 new 关键字的（如 var d = new Demo()），不过如果其他人不知道这个对象（Demo）是一个类，那么在使用时很可能忽略 new 关键字直接执行类（如 var d = Demo();），此时我们得到的并不是我们预期的对象，如下所示。”

```
var Demo = function() {}
Demo.prototype = {
  show : function(){
    console.log('成功获取！');
  }
}
var d = new Demo();
d.show(); // 成功获取!
var d = Demo();
d.show(); // Uncaught TypeError: Cannot read property 'show' of undefined
```

“那么你所说的安全模式就是为了解决这种问题吧。”

“当然，这也是避免像你一样的那些新来同学可能犯的错误。当然解决方案很简单，就是在构造函数开始时先判断当前对象 this 指代是不是类（Demo），如果是则通过 new 关键字创建对象，如果不是说明类在全局作用域中执行（通常情况下），那么既然在全局作用域中执行当然 this 就会指向 window 了（通常情况下，如非浏览器环境可为其他全局对象），这样我们就要重新返回新创建的对象了。”

```
var Demo = function(){
  if(!(this instanceof Demo)){
    return new Demo();
  }
}
var d = Demo();
d.show(); // 成功获取!
```

“有了安全模式我们就可以将这种技术应用在我们的工厂方法中了。”

4.4 安全的工厂方法

```
// 安全模式创建的工厂类
var Factory = function(type, content) {
    if(this instanceof Factory){
        var s = new this[type](content);
        return s;
    }else{
        return new Factory(type, content);
    }
}
// 工厂原型中设置创建所有类型数据对象的基类
Factory.prototype = {
    Java : function(content){
        // .....
    },
    JavaScript : function(content){
        // .....
    },
    UI : function(content){
        this.content = content;
        (function(content){
            var div = document.createElement('div');
            div.innerHTML = content;
            div.style.border = '1px solid red';
            document.getElementById('container').appendChild(div);
        })(content);
    },
    php : function(content){
        // .....
    }
};
```

“这样我们以后如果想添加其他类时，是不是只需写在 Factory 这个工厂类的原型里面就可以了？”

“嗯，是这样，你再也不必担心创建时做任何修改。就好比你在 Factory 类的原型里面注册了一张名片，以后需要哪类直接拿着这张名片，查找上面的信息就能找到这个类了，所以你就不用担心使用时找不到基类的问题了。”

“小白。这里是今天要添加广告的数据，给你，现在就给添加上吧。”经理走过来对小白说。

```
var data = [
    {type:'JavaScript', content:'JavaScript 哪家强'},
    {type:'Java', content:'Java 哪家强'},
    {type:'php', content:'php 哪家强'},
    {type:'UI', content:'UI 哪家强'},
    {type:'UI', content:'UI 哪家强'},
    {type:'JavaScript', content:'JavaScript 哪家强'},
```

```
| {type:'JavaScript', content:'JavaScript 哪家强'}  
| ];
```

小白接过数据一看，格式很友好。于是很快完成了经理提出的需求。

```
| for(var i = 6; i >= 0; i--) {  
|   Factory(s[i].type, s[i].content);  
| }
```

“小白，广告那边又来了需求，需要一批 C++ 学科，蓝色字体……”

小白听到，笑了笑……

下章剧透

通过页面投放广告实战我们学会运用工厂方法模式创建不同种类的对象。下一章我们将深入学习工厂模式，看看如果对工厂类抽象会有什么情况。

忆之获

对于创建多类对象，前面学过的简单工厂模式就不太适用了，这是简单工厂模式的应用局限，当然这正是工厂方法模式的价值之所在，通过工厂方法模式我们可以轻松创建多个类的实例对象，这样工厂方法对象在创建对象的方式也避免了使用者与对象类之间的耦合，用户不必关心创建该对象的具体类，只需调用工厂方法即可。

在实践中，理想与现实总有一线之隔的，新来的同事很可能错误地直接调用工厂方法，这样就很有可能通过工厂方法执行中的 this 对象为全局对象添加全局变量，并且得不到期望的实例对象，此时一个安全的工厂对象则让我们吃了一颗定心丸。

我问你答

通过工厂方法模式为页面创建不同功能的按钮。

第5章 出现的都是幻觉——抽象工厂模式

抽象工厂模式 (Abstract Factory): 通过对类的工厂抽象使其业务用于对产品类簇的创建，而不负责创建某一类产品的实例。

前两次的工厂模式讨论依旧让小白荡气回肠，还记得在讨论工厂方法模式时提到的抽象类么？此时抽象类依旧在小白心中荡漾……

5.1 带头模范——抽象类

“小铭，上次你为我介绍工厂方法时你曾提到过抽象类，那么在 JavaScript 中如何创建一个抽象类呀？抽象类有什么用？与抽象类相关的都有哪些模式呀？”

“抽象类？你知道 JavaScript 中 `abstract` 还是一个保留字，所以目前来说还不能像传统面向对象语言那样轻松地创建。抽象类是一种声明但不能使用的类，当你使用时就会报错。不过 JavaScript 是灵活的，所以我们可以在类的方法中手动地抛出错误来模拟抽象类。”

```
// 汽车抽象类，当使用其实例对象的方法时会抛出错误
var Car = function() {};
Car.prototype = {
    getPrice : function(){
        return new Error('抽象方法不能调用');
    },
    getSpeed : function(){
        return new Error('抽象方法不能调用');
    }
};
```

“我们看到我们创建的这个 `car` 类其实什么都不能做，创建时没有任何属性，然而原型 `prototype` 上的方法也不能使用，否则会报错。但在继承上却是很有用的，因为定义了一种类，并定义了该类所必备的方法，如果在子类中没有重写这些方法，那么当调用时能找到这些方法便会报错。这一特点是很有必要的，因为在一些大型应用中，总会有一些子类去继承另一些父类，这些父类经常会定义一些必要的方法，却没有具体的实现，如 `car` 类中的 `getPrice()` 和 `getSpeed()` 方法，那么一旦用子类创建了一个对象，该对象总是应该具备一些必要的方法，但如果这些必要的方法从父类中继承过来而没有具体去重写实现，那么实例化对象便会调用父类

中的这些方法，如果父类能有一个友好提示，那么对于忘记重写子类的这些错误遗漏的避免是很有帮助的。这也是抽象类的一个作用，即定义一个产品簇，并声明一些必备的方法，如果子类中没有去重写就会抛出错误。”

“原来是这样，”小白接着问，“但是对于这种抽象类有没有一套相关的模式呢。”

5.2 幽灵工厂——抽象工厂模式

“当然，面向对象语言里有一种很常见的模式叫抽象工厂模式，一听到工厂你就应该会想到魔术师吧，都是用来创建对象的，不过这个抽象工厂模式可不简单，在JavaScript中一般不用来创建具体对象，你知道这是为什么吗？”

“不能创建对象？嗯……你刚才说了，抽象类中定义的方法只是显性地定义一些功能，但没有具体的实现，而一个对象是要具有一套完整的功能的，所以用抽象类创建的对象当然也是‘抽象的’了，所以我们不能使用它来创建一个真实的对象，对么？”

“很聪明，对，是这样的，所以一般我们用它作为父类来创建一些子类。”

```
// 抽象工厂方法
var VehicleFactory = function(subType, superType) {
    // 判断抽象工厂中是否有该抽象类
    if(typeof VehicleFactory[superType] === 'function') {
        // 缓存类
        function F() {};
        // 继承父类属性和方法
        F.prototype = new VehicleFactory[superType]();
        // 将子类 constructor 指向子类
        subType.constructor = subType;
        // 子类原型继承“父类”
        subType.prototype = new F();
    } else {
        // 不存在该抽象类抛出错误
        throw new Error('未创建该抽象类');
    }
}
// 小汽车抽象类
VehicleFactory.Car = function() {
    this.type = 'car';
};
VehicleFactory.Car.prototype = {
    getPrice : function() {
        return new Error('抽象方法不能调用');
    },
    getSpeed : function() {
        return new Error('抽象方法不能调用');
    }
};
// 公交车抽象类
VehicleFactory.Bus = function() {
    this.type = 'bus';
};
```

```

VehicleFactory.Bus.prototype = {
    getPrice : function(){
        return new Error('抽象方法不能调用');
    },
    getPassengerNum : function(){
        return new Error('抽象方法不能调用');
    }
};
// 货车抽象类
VehicleFactory.Truck = function(){
    this.type = 'truck';
};
VehicleFactory.Truck.prototype = {
    getPrice : function(){
        return new Error('抽象方法不能调用');
    },
    getTrainload : function(){
        return new Error('抽象方法不能调用');
    }
}

```

“你可以看到，抽象工厂其实是一个实现子类继承父类的方法，在这个方法中我们需要通过传递子类以及要继承父类（抽象类）的名称，并且在抽象工厂方法中又增加了一次对抽象类存在性的一次判断，如果存在，则将子类继承父类的方法。然后子类通过寄生式继承。继承父类过程中有一个地方需要注意，就是在对过渡类的原型继承时，我们不是继承父类的原型，而是通过 new 关键字复制的父类的一个实例，这么做是因为过渡类不应仅仅继承父类的原型方法，还要继承父类的对象属性，所以要通过 new 关键字将父类的构造函数执行一遍来复制构造函数中的属性和方法。”小铭歇了歇，喘口气，接着说，“对抽象工厂添加抽象类也很特殊，因为抽象工厂是个方法不需要实例化对象，故只需要一份，因此直接为抽象工厂添加类的属性即可，于是我们就可以通过点语法在抽象工厂上添加我们一会儿需要的三个汽车簇抽象类 Car、Bus、Truck。”

5.3 抽象与实现

“那我该如何使用它们呢？”小白问。

“使用很容易呀，既然抽象工厂是用来创建子类的（而本例中其实是让子类继承父类，是对子类的一个拓展），所以我们需要一些产品子类，然后让子类继承相应的产品簇抽象类，请看下述代码。”

```

// 宝马汽车子类
var BMW = function(price, speed){
    this.price = price;
    this.speed = speed;
}
// 抽象工厂实现对 Car 抽象类的继承
VehicleFactory(BMW, 'Car');
BMW.prototype.getPrice = function() {

```

```

        return this.price;
    }
BMW.prototype.getSpeed = function(){
    return this.speed;
}
// 兰博基尼汽车子类
var Lamborghini = function(price, speed){
    this.price = price;
    this.speed = speed;
}
// 抽象工厂实现对 Car 抽象类的继承
VehicleFactory(Lamborghini, 'Car');
Lamborghini.prototype.getPrice = function(){
    return this.price;
}
Lamborghini.prototype.getSpeed = function(){
    return this.speed;
}
// 宇通汽车子类
var YUTONG = function(price, passenger){
    this.price = price;
    this.passenger = passenger;
}
// 抽象工厂实现对 Bus 抽象类的继承
VehicleFactory(YUTONG, 'Bus');
YUTONG.prototype.getPrice = function(){
    return this.price;
}
YUTONG.prototype.getPassengerNum = function(){
    return this.passenger;
}
// 奔驰汽车子类
var BenzTruck = function(price, trainLoad){
    this.price = price;
    this.trainLoad = trainLoad;
}
// 抽象工厂实现对 Truck 抽象类的继承
VehicleFactory(BenzTruck, 'Truck')
BenzTruck.prototype.getPrice = function(){
    return this.price;
}
BenzTruck.prototype.getTrainload = function(){
    return this.price;
}

```

“小白，现在你能体会抽象工厂在例子中的作用了么？”

“哦，通过抽象工厂，我们就能知道每个子类到底是哪一种类别了，然后他们也具备了该类所必备的属性和方法了。”

“嗯，你自己写个单例测试一下吧。”

“好的。”于是小白写下了测试代码。

```

var truck = new BenzTruck(1000000, 1000);
console.log(truck.getPrice()); // 1000000
console.log(truck.type); // truck

```

“创建的 truck 对象真的可以知道他的类别了，通过重写父类中的 getPrice 方法也可以正确地使用了。”

下章剧透

通过对创建汽车案例的学习，我们认识抽象工厂模式，下一章我们将会学习如何对整体对象局部创建来达到功能复用的最大化。

忆之获

抽象工厂模式是设计模式中最抽象的一种，也是创建模式中唯一一种抽象化创建模式。该模式创建出的结果不是一个真实的对象实例，而是一个类簇，它制定了类的结构，这也就区别于简单工厂模式创建单一对象，工厂方法模式创建多类对象。当然由于 JavaScript 中不支持抽象化创建与虚拟方法，所以导致这种模式不能像其他面向对象语言中应用得那么广泛。

我问你答

谈谈你对抽象工厂模式的理解。

说出抽象工厂模式与工厂方法模式以及简单工厂模式之间的异同点及其关系。

第6章 分即是合——建造者模式

建造者模式（Builder）：将一个复杂对象的构建层与其表示层相互分离，同样的构建过程可采用不同的表示。

小白这几天对工厂模式学习让其认为创建任何对象只需要工厂模式就可以了。这不小铭临时接到在页面发布用户简历的需求，来看看他们的对话吧。

6.1 发布简历

“小铭，还在加班呢？什么事情这么忙？”小白问。

“这不，刚接到任务，有一些找工作的人，想借助咱们的网站发布自己的简历。”

“哦，人很多么？都有什么要求？要不要我帮你选选呀？”小白问。

“不用，不是咱们公司招聘，是将他们简历向外推销。不过接到的简历还真不少，但是这些简历有一个要求，除了可以将他们的兴趣爱好以及一些特长发布在页面里，其他信息，如他们的联系方式，不要发布在网站上。要让需求公司来找咱们。不过话又说回来，他们想找的工作是可以分类的，比如对于喜欢编程的人来说他们要找的职位就是工程师（engineer）了，当然这里可能还有一些描述。比如‘每天沉醉于编程……’”

“听上去还想要分很多部分，这样创建他们要写不少工厂方法吧？”小白问。

“嗯，很多部分需要抽象提取，不过首先要明确创建内容。比如创建用户信息如用户名等要独立处理，因为他们是要隐藏显示的。比如这些应聘者也要独立创建，因为他们代表一个整体。还有这些工作职位也要独立创建，他们是应聘者拥有的一部分，而且种类很多。”

“不过这样一来你需要创建的东西就多了，不仅仅应聘者需要创建，每位应聘者的信息、应聘职位都要创建，不过我想这几天我们讨论的创建模式好像都不太适合你的需求呀。”小白说。

6.2 创建对象的另一种形式

“所以我不准备用任何工厂模式了，我想建造者模式更适合吧。”

“建造者模式？这是一个新模式，以前还从未听说过，它与学过的几类工厂模式之间有区别么？”

“工厂模式主要是为了创建对象实例或者类簇（抽象工厂），关心的是最终产出（创建）的是什么。不关心你创建的整个过程，仅仅需要知道你最终创建的结果。所以通过工厂模式我们得到的都是对象实例或者类簇。然而建造者模式在创建对象时要更为复杂一些，虽然其目的也是为了创建对象，但是它更多关心的是创建这个对象的整个过程，甚至于创建对象的每一个细节，比如创建一个人，我们创建的结果不仅仅要得到人的实例，还要关注创建人的时候，这个人应该穿什么衣服，男的还是女的，兴趣爱好都是什么。所以说建造者模式更注重的是创建的细节，而在本例中我们看到，我们需要的不仅仅是应聘者的一个实例，还要在创建过程中注意一下这位应聘者都有哪些兴趣爱好、他的姓名等信息，他所期望的职位是什么，等等。那么这些关注点都是需要我们创建的。”

```
// 创建一位人类
var Human = function(param) {
    // 技能
    this.skill = param && param.skill || '保密';
    // 兴趣爱好
    this.hobby = param && param.hobby || '保密';
}
// 类人原型方法
Human.prototype = {
    getSkill : function(){
        return this.skill;
    },
    getHobby : function(){
        return this.hobby;
    }
}
// 实例化姓名类
var Named = function(name){
    var that = this;
    // 构造器
    // 构造函数解析姓名的姓与名
    (function(name, that){
        that.wholeName = name;
        if(name.indexOf(' ') > -1){
            that.FirstName = name.slice(0, name.indexOf(' '));
            that.SecondName = name.slice(name.indexOf(' '));
        }
    })(name, that);
}
// 实例化职位类
var Work = function(work){
    var that = this;
    // 构造器
    // 构造函数中通过传入的职位特征来设置相应职位以及描述
    (function(work, that){
        switch(work){
            case 'code':

```

```

        that.work = '工程师';
        that.workDescript = '每天沉醉于编程';
        break;
    case 'UI':
    case 'UE':
        that.work = '设计师';
        that.workDescript = '设计更似一种艺术';
        break;
    case 'teach':
        that.work = '教师';
        that.workDescript = '分享也是一种快乐';
        break;
    default :
        that.work = work;
        that.workDescript = '对不起，我们还不清楚您所选择职位的相关描述';
    }
})(work, that);
}
// 更换期望的职位
Work.prototype.changeWork = function(work) {
    this.work = work;
}
// 添加对职位的描述
Work.prototype.changeDescript = function(setence) {
    this.workDescript = setence;
}

```

6.3 创建一位应聘者

“这样我们就创建了抽象出来的 3 个类——应聘者类、姓名解析类与期望职位类。然而创建应聘者类有些特殊，因为对其类传入的参数做了一个小处理，就是`&&`与`||`的运用，如例子中的`this.skill = param && param.skill || '保密'`;表示，如果存在`param`这个参数，并且`param`拥有`skill`属性，就用这个属性赋值给`this`的`skill`属性，否者将用默认值‘保密’来设置。”小铭接着说，“我们最终的目的是要创建一位应聘者，所以需要上面抽象的 3 个类。这样我们写一个建造者类，在建造者类中我们要通过对这 3 个类组合调用，就可以创建出一个完整的应聘者对象”。

```

*****
 * 应聘者建造者
 * 参数 name : 姓名(全名)
 * 参数 work : 期望职位
 */
var Person = function(name, work) {
    // 创建应聘者缓存对象
    var person = new Human();
    // 创建应聘者姓名解析对象
    person.name = new Named(name);
    // 创建应聘者期望职位
    person.work = new Work(work);
    // 将创建的应聘者对象返回
}

```

```

    return _person;
}

```

“在应聘者建造者中我们分成三个部分来创建一位应聘者对象，首先创建一位应聘者缓存对象，缓存对象需要修饰（添加属性和方法），然后我们向缓存对象添加姓名，添加一个期望职位，最终我们就可得到一位完整的应聘者了。看！”

```
var person = new Person('xiao ming', 'code');
```

“你可以来测试一下看看”

```

console.log(person.skill);           // 保密
console.log(person.name.FirstName); // xiao
console.log(person.work.work);      // 工程师
console.log(person.work.workDescript); // 每一天在编程中度过
person.work.changeDescript('更改一下职位描述！');
console.log(person.work.workDescript); // 更改一下职位描述！

```

“小白，看过之后再回顾一下工厂模式，你感觉最大的感受是什么？”

“正像你说的那样，以前工厂模式创建出来的是一个对象，它追求的是创建的结果，别无他求，所以那仅仅是一个实实在在的创建过程。而建造者模式就有所不同，它不仅仅可得到创建的结果，然而也参与了创建的具体过程，对于创建的具体实现的细节也参与了干涉，可以说创建的对象更复杂，或者说这种模式创建的对象是一个复合对象。”

下章剧透

通过对发布简历这一镜头抓拍实战，我们学习了如何将一个完整的对象局部创建。下一章我们将为页面创建一些焦点图（一种图片切换网页特效），从中我们看看又有什么新模式等着我们去挖掘。

忆之获

回忆我们前面学过的几种工厂模式，他们都有一个共同特点，就是创建的结果都是一个完整的个体，我们对创建过程不得而知，我们只了解得到的创建结果对象。而在建造者模式中我们关心的是对象创建过程，因此我们通常将创建对象的类模块化，这样使被创建的类的每一个模块都可以得到灵活的运用与高质量的复用。当然我们最终的需求是要得到一个完整的个体，因此在拆分创建的整个过程，我们将得到一个统一的结果。

当然这种方式对于整体对象类的拆分无形中增加了结构的复杂性，因此如果对象粒度很小，或者模块间的复用率很低并且变动不大，我们最好还是要创建整体对象。

我问你答

对于一个卡片堆砌页面，想一想如何应用建造者模式实现对每一个页面中结构卡片的创建？

第7章 语言之魂——原型模式

原型模式（Prototype）：用原型实例指向创建对象的类，使用于创建新的对象的类共享原型对象的属性以及方法。

随着 Web 的发展，产品经理对网页特效的追求越来越多。这不，经理让小白加快页面多种焦点图特效的研发呢。

7.1 语言中的原型

“小铭，有个问题我一直没想明白。”小白说。

“怎么了？”小铭问道。

“你看，在 JavaScript 中的继承是靠原型链实现的，那么这就是 JavaScript 中的原型模式吧？”

“原型模式你都知道，看来你知道的知识还真不少。原型模式就是将原型对象指向创建对象的类，使这些类共享原型对象的方法与属性。当然 JavaScript 是基于原型链实现对象之间的继承，这种继承是基于一种对属性或者方法的共享，而不是对属性和方法的复制。”

7.2 创建一个焦点图

“举个例子：假设页面中有很多焦点图（网页中很常见的一种图片轮播，切换效果），那么我们要实现这些焦点图最好的方式就是通过创建对象来一一实现，所以我们就需要有一个焦点图类，比如我们把这个类定义为 LoopImages。”

```
// 图片轮播类
var LoopImages = function(imgArr, container){
    this.imagesArray = imgArr;          // 轮播图片数组
    this.container = container;         // 轮播图片容器
    this.createImage = function(){}     // 创建轮播图片
    this.changeImage = function(){}     // 切换下一张图片
}
```

“如果一个页面中有多个这类焦点图，其切换动画一般是多样化的，有的可能是上下切换，

有的可能是左右切换，有的可能是渐隐切换，有的可能是放缩切换，等等，因此创建的轮播图片结构应该是多样化的，同样切换的效果也应该是多样化的，因此我们应该抽象出一个基类，让不同特效类去继承这个基类，然后对于差异化的需求通过重写这些继承下来的属性或者方法来解决。当然不同的子类之间可能存在不同的结构样式，比如有的包含一个左右切换箭头，于是我们有了下面的例子。”

```
// 上下滑动切换类
var SlideLoopImg = function(imgArr, container) {
    // 构造函数继承图片轮播类
    LoopImages.call(this, imgArr, container);
    // 重写继承的切换下一张图片方法
    this.changeImage = function(){
        console.log('SlideLoopImg changeImage function');
    }
}
// 渐隐切换类
var FadeLoopImg = function(imgArr, container, arrow) {
    LoopImages.call(this, imgArr, container);
    // 切换箭头私有变量
    this.arrow = arrow;
    this.changeImage = function(){
        console.log('FadeLoopImg changeImage function');
    }
}
```

“我们创建一个显隐轮播图片测试实例很容易”

```
// 实例化一个渐隐切换图片类
var fadeImg = new FadeLoopImg([
    '01.jpg',
    '02.jpg',
    '03.jpg',
    '04.jpg'
], 'slide', [
    'left.jpg',
    'right.jpg'
]);
fadeImg.changeImage(); // FadeLoopImg changeImage function
```

“嗯，用这种方式实现这类需求是极好的。”小白说。

7.3 最优的解决方案

“是呀，像你这样刚来的同事，如果能写出这样的代码已经很不错了，不过还是存在一些问题的。首先我们看基类 `LoopImages`，作为基类是要被子类继承的，那么此时将属性和方法都写在基类的构造函数里会有一些问题，比如每次子类继承都要创建一次父类，假如说父类的构造函数中创建时存在很多耗时较长的逻辑，或者说每次初始化都做一些重复性的东西，这样的性能消耗还是蛮大的。为了提高性能，我们需要有一种共享机制，这样每当创建基类时，对

于每次创建的一些简单而又差异化的属性我们可以放在构造函数中，而我们将一些消耗资源比较大的方法放在基类的原型中，这样就会避免很多不必要的消耗，这也就是原型模式的一个雏形。这一模式很像我们之前提到过的类继承，都是基于原型链。也由于这种类型很常见，所以我们经常忽视这种模式。在其他面向对象语言中，之所以这种模式经常被提到，是因为该模式在实现这种‘共享’上要麻烦一些，是通过复制完成的。我们这个例子该如何修改你知道了吧。”

小白思考了一下：“原型模式就是将可复用的、可共享的、耗时大的从基类中提出来然后放在其原型中，然后子类通过组合继承或者寄生组合式继承而将方法和属性继承下来，对于子类中那些需要重写的方法进行重写，这样子类创建的对象既具有子类的属性和方法也共享了基类的原型方法，像下面这样。”

```
// 图片轮播类
var LoopImages = function(imgArr, container){
    this.imagesArray = imgArr;           // 轮播图片数组
    this.container = container;         // 轮播图片容器
}
LoopImages.prototype = {
    // 创建轮播图片
    createImage : function(){
        console.log('LoopImages createImage function');
    },
    // 切换下一张图片
    changeImage : function(){
        console.log('LoopImages changeImage function');
    }
}
// 上下滑动切换类
var SlideLoopImg = function(imgArr, container){
    // 构造函数继承图片轮播类
    LoopImages.call(this, imgArr, container);
}
SlideLoopImg.prototype = new LoopImages();
// 重写继承的切换下一张图片方法
SlideLoopImg.prototype.changeImage = function(){
    console.log('SlideLoopImg changeImage function');
}
// 漫隐切换类
var FadeLoopImg = function(imgArr, container, arrow){
    LoopImages.call(this, imgArr, container);
    // 切换箭头私有变量
    this.arrow = arrow;
}
FadeLoopImg.prototype = new LoopImages();
FadeLoopImg.prototype.changeImage = function(){
    console.log('FadeLoopImg changeImage function');
}
// 测试用例
console.log(fadeImg.container); // slide
fadeImg.changeImage();          // FadeLoopImg changeImage function
```

7.4 原型的拓展

“嗯，很不错，不过小白，你知道原型模式还有一个特点么？”

“给点提示呗，关于子类还是父类的？”小白笑着问道。

“都有，不过确切地说是关于原型对象的。”

“原型对象？”

“原型对象是一个共享的对象，那么不论是父类的实例对象或者是子类的继承，都是对它的一个指向引用，所以原型对象才会被共享。既然被共享，那么对原型对象的拓展，不论是子类或者父类的实例对象都会继承下来。小白你可以试一试。”

于是小白写下测试代码。

```
LoopImages.prototype.getImageLength = function(){
    return this.imagesArray.length;
}
FadeLoopImg.prototype.getContainer = function(){
    return this.container;
}
console.log(fadeImg.getImageLength());      // 4
console.log(fadeImg.getContainer());         //slide
```

“真的是这样呀。”小白兴奋地说。

“所以说原型模式有一个特点就是在任何时候都可以对基类或者子类进行方法的拓展，而且所有被实例化的对象或者类都能获取这些方法，这样给予我们对功能拓展的自由性。但是有一点你要注意，正是由于这种方式太自由了，所以不要随意去做，否则如果修改类的其他属性或者方法很有可能会影响到他人。”

“嗯，不过真的感觉原型继承模式好强大。”小白感叹道。

“那是当然。所以说这种模式是 JavaScript 语言的灵魂，在 JavaScript 中很多面向对象编程思想或者设计模式都是基于原型模式继承实现的。”

“看来原型模式真的很重要。”小白说。

7.5 原型继承

“不过原型模式更多的是用在对对象的创建上。比如创建一个实例对象的构造函数比较复杂，或者耗时比较长，或者通过创建多个对象来实现。此时我们最好不要用 new 关键字去复制这些基类，但可以通过对这些对象属性或者方法进行复制来实现创建，这是原型模式的最初思想。如果涉及多个对象，我们也可以通过原型模式来实现对新对象的创建。那么首先要有一个原型模式的对象复制方法。”

```
*****
```

```

* 基于已经存在的模板对象克隆出新对象的模式
* arguments[0], arguments[1], arguments[2]: 参数 1, 参数 2, 参数 3 表示模板对象
* 注意。这里对模板引用类型的属性实质上进行了浅复制(引用类型属性共享),当然根据需求可以自行深复制(引用类型属性复制)
*****
function prototypeExtend() {
    var F = function() {},           // 缓存类,为实例化返回对象临时创建
        args = arguments,           // 模板对象参数序列
        i = 0,
        len = args.length;
    for(; i < len; i++) {
        // 遍历每个模板对象中的属性
        for(var j in args[i]) {
            // 将这些属性复制到缓存类原型中
            F.prototype[j] = args[i][j];
        }
    }
    // 返回缓存类的一个实例
    return new F();
}

```

比如企鹅游戏中我们创建一个企鹅对象，如果游戏中没有企鹅基类，只是提供了一些动作模板对象，我们就可以通过实现对这些模板对象的继承来创建一个企鹅实例对象。

```

var penguin = prototypeExtend({
    speed : 20,
    swim : function(){
        console.log('游泳速度 ' + this.speed);
    }
}, {
    run : function(speed){
        console.log('奔跑速度 ' + speed);
    }
}, {
    jump : function(){
        console.log('跳跃动作');
    }
})

```

“既然通过 prototypeExtend 创建的是一个对象，我们就无需再用 new 去创建新的实例对象，我们可以直接使用这个对象。”

```

penguin.swim();           // 游泳速度 20
penguin.run(10);          // 奔跑速度 10
penguin.jump();           // 跳跃动作

```

“这又是一种继承的实现方式呀。”小白惊呼。

下章剧透

原型模式实质上也是一种继承，通过学习原型模式，让小白对继承的实现有了新的拓展。

当然原型模式也是一种创建型模式，那么下一章我们将学习创建模式中的最后一种。那是什么模式？我们一起期待吧。

忆之获

原型模式可以让多个对象分享同一个原型对象的属性与方法，这也是一种继承方式，不过这种继承的实现是不需要创建的，而是将原型对象分享给那些继承的对象。当然有时需要让每个继承对象独立拥有一份原型对象，此时我们就需要对原型对象进行复制。

由此我们可以看出，原型对象更适合在创建复杂的对象时，对于那些需求一直在变化而导致对象结构不停地改变时，将那些比较稳定的属性与方法共用而提取的继承的实现。

我问你答

有人说，原型继承的实现是不需要了解创建的过程的。谈谈你的理解。

第8章 一个人的寂寞——单例模式

单例模式 (Singleton): 又被称为单体模式，是只允许实例化一次的对象类。有时我们也用一个对象来规划一个命名空间，并井有条地管理对象上的属性与方法。

这不，小白接到一个任务，公司要开展一个活动，经理让小白做一个宣传页面，这可是小白作为新人接到的第一个完整的项目哦，看看他是怎么做的吧。

8.1 滑动特效

“小白。你在活动页面实现新闻列表中实现的鼠标滑动特效怎么定义了这么多的方法？”小铭走过来对小白说。

```
function g(id){  
    return document.getElementById(id)  
}  
function css(id, key, value){  
    // 简单样式属性设置  
    g(id).style[key] = value;  
}  
function attr(id, key, value){  
    g(id)[key] = value;  
}  
function html(id, value){  
    g(id).innerHTML = value;  
}  
function on(id, type, fn){  
    g(id)['on' + type] = fn;  
}
```

“这么做有什么不妥么？”小白问。

“你在页面中添加了很多变量，比如你定义的绑定事件方法 on，如果日后其他人要为你的页面添加需求，增加代码而定义一个 on 变量或者重写了 on 方法，这样就会和你的代码起冲突了。所以你最好要用单例修改一下你书写的代码。”

8.2 命名空间的管理员

“单例模式？”小白有些疑惑，“你是说要把我定义的这些功能方法放在一个对象里么？”

“单例模式你没有用过么？这可能是 JavaScript 中最常见的一种模式了。这种模式经常为我们提供一个命名空间。如我们使用过的 jQuery 库，单例模式就为它提供了一个命名空间 jQuery。”

“命名空间？”

“没听过么？命名空间就是人们所说的 namespace，有人也叫它名称空间。它解决这么一类问题：为了让代码更易懂，人们常常用单词或者拼音定义变量或者方法，但由于人们可用的单词或者汉字拼音是有限的，所以不同的人定义的变量使用的单词名称很有可能重复，此时就需要用命名空间来约束每个人定义的变量来解决这类问题。比如小张写的代码，他可以定义一个 xiaozhang 命名空间，这样以后使用小张定义的变量时就可以通过 xiaozhang.xx（xx 表示定义的变量名）来使用。同理，小李定义了一个 xiaoli 命名空间，那么以后同样要使用 xiaoli.xx 访问小李定义的变量。”

“啊，这样呀”，小白似乎有些明白，“我们要使用 jQuery 定义 animate 方法我们就要用 jQuery.animate() 来访问，这是同一个道理，对不对？”

“很聪明么，所以说单例模式常用来定义命名空间，所以你知道把你写的代码怎么改了吧。”

“很容易，直接将上面的代码放在定义变量的 {} 里就可以了吧。”

```
var Ming = {
    g : function(id) {
        return document.getElementById(id)
    },
    css : function(id, key, value) {
        // 简单样式属性设置
        g(id).style[key] = value;
    }
    // ...
}
```

“你别忘了，在你的 css 方法中你使用了 g 方法，而我们刚才说过，单例模式要想使用定义的方法一定要加上命名空间 Ming，所以你不要忘记将 css 方法中的 g 方法改成 Ming.g。由于 g 方法和 css 方法都在单例对象 Ming 中，也就是说这两个方法都是单例对象 Ming 的方法，而对象中 this 指代当前对象，所以我们还可以在 css 方法中通过 this.g 来使用 Ming 单例对象中的 g 方法。像下面这样。”

```
var Ming = {
    g : function(id) {
        return document.getElementById(id)
    },
    css : function(id, key, value) {
```

```

    // 通过当前对象 this 来使用 g 方法
    this.g(id).style[key] = value;
}
// ...
}

```

“真是大意了。”小白微笑道。

8.3 模块分明

“其实在 JavaScript 中单例模式除了定义命名空间外，还有一个作用你需要知道，就是通过单例模式来管理代码库的各个模块，比如早期百度 tangram，雅虎的 YUI 都是通过单例模式来控制自己的每个功能模块的，比如 tangram 中定义命名空间为百度，当添加设置元素 class 方法，插入一个元素方法时，他们会放到 dom 模块；当添加事件中阻止事件的冒泡方法，阻止事件的默认行为方法的时候，会放到 event 模块里；当添加去除字符串首尾空白字符方法，将字符串进行 html 编码时，会放到 string 模块中……”

```

baidu.dom.addClass          // 添加元素类
baidu.dom.append            // 插入元素
baidu.event.stopPropagation   // 阻止冒泡
baidu.event.preventDefault  // 阻止默认行为
baidu.event.trim             // 去除字符串收尾空白字符
baidu.string.encodeHTML     // 将字符串进行 html 编码

```

8.4 创建一个小型代码库

“所以我们以后写自己的小型方法库的时候也可以用单例模式来规范我们自己代码库的各个模块，”小铭接着说，“比如我们有一个 A 库，它包含公用模块、工具模块、ajax 模块和其他模块，那么我们就可以自己定制一个如下的小型代码库。

```

var A = {
  Util : {
    util_method1 : function() {},
    util_method2 : function() {}
    // ...
  },
  Tool : {
    tool_method1 : function() {},
    tool_method2 : function() {}
    // ...
  },
  Ajax : {
    get : function() {},
    post : function() {}
    // ...
  },
  others : {
    // ...
  }
}

```

```
// ...
}
// ...
}
```

“那么我们想使用公共模块、工具模块、ajax 模块方法时就像下面这样。”

```
A.Util.util_method2();
A.Tool.tool_method1();
A.Ajax.get();
```

8.5 无法修改的静态变量

“看上去代码库的结构真的很清晰了，大家使用起来更容易了。”小白说。

“嗯，这正是单例模式的好处，不过这是在对代码管理上，其实有一个功能用单例模式实现更适合，就是管理静态变量。”

“静态变量？JavaScript 中不是没有静态变量么？”小白追问。

“你知道，JavaScript 根本没有 static 这类关键字，所以定义任何变量理论上都是可以更改的，所以在 JavaScript 中实现创建静态变量又变得很重要。当然 JavaScript 很灵活，人们根据静态变量只能访问不能修改并且无创建后就能使用这一特点，想出了一个好主意：能访问的变量定义的方式有很多，比如定义在全局空间里，或者定义一个函数内部，并定义一个特权方法访问，等等。既然不能修改，定义在全局空间里就显得很不靠谱了，而如果将变量放在一个函数内部，那必须通过特权方法访问，如果我们不提供赋值变量的方法，只提供获取变量的方法，不就可以做到限制变量的修改并且可以供外界访问的需求了么？但是还有最后一个问题就是目前放在函数内部的变量还能供外界访问，为实现创建后就能使用这一需求，我们就需要让创建的函数执行一次，此时我们创建的对象内保存静态变量通过取值器访问，最后将这个对象作为一个单例放在全局空间里作为静态变量单例对象供他人使用。”

```
var Conf = (function(){
    // 私有变量
    var conf = {
        MAX_NUM : 100,
        MIN_NUM : 1,
        COUNT : 1000
    }
    // 返回取值器对象
    return {
        // 取值器方法
        get : function(name) {
            return conf[name] ? conf[name] : null;
        }
    }
})();
```

“很奇妙呀，没有赋值器我们就不能修改内部定义的变量了。那么我们要想使用创建了的

静态变量，像下面这种方式就可以了吧。”小白说。

```
var count = Conf.get('COUNT')
console.log(count); //1000
```

8.6 惰性单例

“不过为什么静态变量都大写呢？”小白问。

“这是一种定义习惯，在其他编程语言中静态变量都习惯于大写，所以在 JavaScript 中虽然是模拟的静态变量我们也要尊重这一使用习惯。”小铭接着说，“有些时候对于单例对象需要延迟创建，所以在单例中还存在一种延迟创建的形式，有人也称之为‘惰性创建’。”

```
// 惰性载入单例
var LazySingle = (function() {
    // 单例实例引用
    var _instance = null;
    // 单例
    function Single() {
        /*这里定义私有属性和方法*/
        return {
            publicMethod : function() {},
            publicProperty : '1.0'
        }
    }
    // 获取单例对象接口
    return function() {
        // 如果为创建单例将创建单例
        if(!_instance){
            _instance = Single();
        }
        // 返回单例
        return _instance;
    }
})();
```

“我们测试一下可以看出通过 LazySingle 对象可以成功获取内部创建的单例对象了。”

```
console.log(LazySingle().publicProperty); // 1.0
```

下章剧透

一个简简单单的活动页面让小白搞清楚如何管理自己的代码，当然也了解为何很多代码库都提供一个变量（当然有一些还是提供了简单的别名，如 jQuery 中提供了\$别名）管理每个模块的代码。明天小白将学习新一类模式——结构型设计模式，我们看看结构型设计模式中的第一种是什么？

忆之获

单例模式有时也被称为单体模式，它是一个只允许实例化一次的对象类，有时这么做也是为了节省系统资源。当然 JavaScript 中单例模式经常作为命名空间对象来实现，通过单例对象我们可以将各个模块的代码井井有条地梳理在一起。

所以如果你想让系统中只存在一个对象，那么单例模式则是最佳解决方案。

我问你答

实践是检验收获的最好方式，那么我们也来实现一个小型代码库吧，来体会一下单例模式的优越性。

第三篇

结构型设计模式

结构型设计模式关注于如何将类或对象组合成更大、更复杂的结构，以简化设计。

- 第 9 章 套餐服务——外观模式
- 第 10 章 水管弯弯——适配器模式
- 第 11 章 牛郎织女——代理模式
- 第 12 章 房子装修——装饰者模式
- 第 13 章 城市间的公路——桥接模式
- 第 14 章 超值午餐——组合模式
- 第 15 章 城市公交车——享元模式

第9章 套餐服务——外观模式

外观模式（Facade）：为一组复杂的子系统接口提供一个更高级的统一接口，通过这个接口使得对子系统接口的访问更容易。在 JavaScript 中有时也会用于对底层结构兼容性做统一封装来简化用户使用。

前几天的对创建行的设计模式的学习使小白对对象的创建有了深入的理解，今天有了新的挑战，这不小白正在为页面写交互呢。

9.1 添加一个点击事件

“小白，你为页面文档 document 对象绑定了一个 click 事件来实现隐藏提示框的交互功能，直接用 onclick 为 document 绑定的。”小铭过来问小白。

“是这样，不过这有什么问题么？”小白回答说。

“你的功能是完成了，只不过……”小铭犹豫了一下。

“怎么了？”

“我看一下你的代码。”

```
document.onclick = function(e){  
    e.preventDefault();  
    if(e.target !== document.getElementById('myinput')){  
        hidePageAlert();  
    }  
}  
function hidePageAlert(){  
    // 隐藏提示框  
}
```

“首先，你为 document 绑定了 onclick 事件，但是你知道 onclick 是 DOM0 级事件，也就是说这种方式绑定的事件相当于为元素绑定一个事件方法，所以如果我们团队中有人再次通过这种方式为 document 绑定 click 事件时，就相当于重复定义了一个方法，会将你定义的 click 事件方法覆盖，如下列程序。”

```
document.onclick = function(){  
    // 其他开发人员重新为 document 绑定事件会覆盖前面定义的 DOM0 级 click 事件
```

“所以你这种方式是很危险的。因此你应该用 DOM2 级事件处理程序提供的方法 addEventListener 来实现，然而你知道老版本的 IE 浏览器（低于 9）是不支持这个方法的，所以你要用 attachEvent，当然如果有不支持 DOM2 级事件处理程序的浏览器，你只能用 onclick 事件方法绑定事件。”

“如此看来为元素绑定一个事件真不是一件容易的事，我们要做这么多的事情。有没有一个兼容所有浏览器的方式呢？”小白追问。

9.2 兼容方式

小铭微笑着说：“当然有了。我们可以用外观模式来封装他们，这就像每天中午我们急冲冲地去餐厅吃饭，人很多，不论是餐厅点餐员还是作为顾客的我们，大家都想尽量节约时间，以尽可能少的成本完成我们点餐吃饭的整个流程，所以通常的做法是去点套餐，这就很简洁明了。比如鱼香肉丝饭套餐，会为我们提供米饭、菜，甚至饮料等。当然我们就不用再浏览遍历每一种菜，主食要点什么，饮料要买哪种等，因为套餐已经为我们定制好了，那么在 JavaScript 中也可以通过一个‘套餐’来简化复杂的需求。比如我们统一功能接口方法的不统一，我们可以通过外观像点套餐那样定义一个统一接口方法，这样就提供一个更简单的高级接口，简化了我们对复杂的底层接口不统一的使用需求。根据这一思想我们就可以按照如下方式简化我们事件的绑定。”

```
// 外观模式实现
function addEvent(dom, type, fn) {
    // 对于支持 DOM2 级事件处理程序 addEventListener 方法的浏览器
    if(dom.addEventListener){
        dom.addEventListener(type, fn, false);
    }
    // 对于不支持 addEventListener 方法但支持 attachEvent 方法的浏览器
    }else if(dom.attachEvent){
        dom.attachEvent('on' + type, fn);
    }
    // 对于不支持 addEventListener 方法也不支持 attachEvent 方法，但支持 on+‘事件名’的
    // 浏览器
    }else{
        dom['on' + type] = fn;
    }
}
```

“这样我们以后对于支持 addEventListener 或 attachEvent 方法的浏览器就可以放心地绑定多个事件了，如下所示。”

```
var myInput = document.getElementById('myinput');
addEvent(myInput, 'click', function(){
    console.log("绑定第一个事件")
})
addEvent(myInput, 'click', function(){
    console.log("绑定第二个事件")
```

```

    })
addEvent(myInput, 'click', function(){
  console.log("绑定第三个事件")
})
  
```

“如此一来，在团队开发中就能安心地为元素绑定事件了。”小白感叹道。

9.3 除此之外

“不过你之前写的代码问题不只一个，之前说了，外观模式可以简化底层接口复杂性，也可以解决浏览器兼容性问题。而你前面写的代码除了绑定事件问题外，另外两处问题是在其他IE低版本浏览器中不兼容`e.preventDefault()`和`e.target`。你也可以通过外观模式来解决。”

```

// 获取事件对象
var getEvent = function(event){
  // 标准浏览器返回 event, IE 下 window.event
  return event || window.event;
}
// 获取元素
var getTarget = function(event){
  var event = getEvent(event);
  // 标准浏览器下 event.target, IE 下 event.srcElement
  return event.target || event.srcElement;
}
// 阻止默认行为
var preventDefault = function(event){
  var event = getEvent(event);
  // 标准浏览器
  if(event.preventDefault){
    event.preventDefault();
  // IE 浏览器
  }else{
    event.returnValue = false;
  }
}
  
```

“有了上面的方法我们就可以用兼容的简单方式来解决上面的问题了。”

```

document.onclick = function(e){
  // 阻止默认行为
  preventDefault(e);
  // 获取事件源目标对象
  if(getTarget(e) !== document.getElementById('myinput')){
    hideInputSug();
  }
}
  
```

9.4 小型代码库

“小白，外观模式可以将浏览器不兼容的方法变得简单而又兼容各个浏览器，然而这只是

外观模式应用的一部分，很多代码库中都是通过外观模式来封装多个功能，简化底层操作方法，比如我们简单实现获取元素的属性样式的简单方法库。”

```
// 简约版属性样式方法库
var A = {
    // 通过 id 获取元素
    g : function(id) {
        return document.getElementById(id);
    },
    // 设置元素 css 属性
    css : function(id, key, value){
        document.getElementById(id).style[key] = value;
    },
    // 设置元素的属性
    attr : function(id, key, value){
        document.getElementById(id)[key] = value;
    },
    html : function(id, html){
        document.getElementById(id).innerHTML = html;
    },
    // 为元素绑定事件
    on : function(id, type, fn){
        document.getElementById(id)['on' + type] = fn;
    }
};
```

“通过这个代码库，我们再操作元素的属性样式时变得更简单。”

```
A.css('box', 'background', 'red');      // 设置 css 样式
A.attr('box', 'className', 'box');       // 设置 class
A.html('box', '这是新添加的内容');      // 设置内容
A.on('box', 'click', function(){         // 绑定事件
    A.css('box', 'width', '500px');
});
```

下章剧透

为页面添加事件，在 JavaScript 中是一件极其麻烦的事情，尤其在众多表现不一的浏览器中。好在通过外观模式可以简化我们对事件绑定的使用。当然通过外观模式我们也可以将我们的代码库变得更加易用。然而如果自己的代码库不能满足新的需求时，想引进另外一个更先进的代码库该怎么办呢？又会遇到哪些问题？下一章我们将看看小白是如何在一个页面中让不同的代码库兼容的。

忆之获

当一个复杂的系统提供一系列复杂的接口方法时，为系统的管理方便会造成接口方法的使用极其复杂。这在团队的多人开发中，撰写成本是很大的。当然通过外观模式，对接口的二次

封装隐藏其复杂性，并简化其使用是一种很不错的实践。当然这种实践增加了一些资源开销以及程序的复杂度，当然这种开销相对于使用成本来说有时也是可忽略的。

外观模式是对接口方法的外层包装，以供上层代码调用。因此有时外观模式封装的接口方法不需要接口的具体实现，只需要按照接口使用规则使用即可。这也是对系统与客户（使用者）之间的一种松散耦合。使得系统与客户之间不会因结构的变化而相互影响。

我问你答

我们知道事件在不同浏览器中可以有不同的绑定方式，同样获取元素的css样式在不同浏览器中也表现不一，那么就通过外观模式来封装一个获取元素css样式的方法吧。

第 10 章 水管弯弯——适配器模式

适配器模式（Adapter）：将一个类（对象）的接口（方法或者属性）转化成另外一个接口，以满足用户需求，使类（对象）之间接口的不兼容问题通过适配器得以解决。

随着活动页面的功能增加，原生 JavaScript 在一些交互与特效实现上让小白感到力不从心，于是想引入大名鼎鼎的 jQuery……

10.1 引入 jQuery

“小白，咱们的作品活动页面还在用我们公司内部开发的 A 框架，可是很多新来的同事使用 A 框架开发新的功能需求时总是感觉很吃力，而且能用的方法有限，为了让他们尽快融入项目的开发，我想让你引入 jQuery 框架没问题吧。”小铭对小白说。

“没问题，”小白看了一下代码，迟疑一下“可是以前功能所写的代码是不是我还要重新用 jQuery 写一遍呀，比如像这里引入的事件。”

```
A(function() {
  A('button').on('click', function(e) {
    // .....
  });
});
```

“不用呀，咱们公司的 A 框架与 jQuery 框架比较像，所以你简单写个适配器就可以了。”

“适配器？”小白不解“那是什么东西？”

10.2 生活中的适配器

“以前你没有接触过么？”小铭接着说“这可是编程中一种很常见的模式。其实生活中这种模式也很常见，你看公司的水房的两根垂直相交的水管连接处的直角弯管了么？它就是一个适配器，它使得两个不同方向的水管可以疏通流水。再比如我们三角插头手机充电器对于两项插头是不能用的，此时我们就需要一个三项转两项插头电源适配器等等，这些都是适配器。如果你明白这些，那么为页面中的代码写适配器就不难了，其实就是为两个代码库所写的代码兼

容运行而书写的额外代码。有了这样的适配器，你就不需要特意地重写以前的功能代码了。你只需要让用以前的代码库所写的代码适配 jQuery 就可以了。”

“可是我该如何做呢？”小白追问。

10.3 jQuery 适配器

“你看我们公司的 A 框架代码书写格式是不是与 jQuery 代码书写格式很像，所以你需要在加载完 jQuery 框架后写一个适配器。将我们已有的功能适配到 jQuery。比如代码中有两个事件，一个页面加载事件，一个点击事件。不过这两个事件与 jQuery 中的写法很像，所以这里就不用做多少改动了。我们的适配器主要的任务是适配两种代码库中不兼容的代码。那么首当其冲的就是全局对象 A 与 jQuery 了，所以你可以像下面这样轻松实现。”

```
window.A = A = jQuery;
```

“小白，你刷新页面看看是不是运行良好？”小铭笑着问。

小白刷新了一下页面，“真是太神奇了。页面可以正常工作了。这样我们就可以写以前熟悉的 jQuery 代码了。”

10.4 适配异类框架

“嗯，这是因为咱们公司的整个轻量级的 A 框架太像 jQuery 了，我们可以将这两种框架看成是相似框架。但是如果一个框架与 jQuery 血缘远一点，那么对于这种异类框架适配情况就复杂得多了。举个例子吧，还是实现上面两个事件，所以我写了一个这样的框架。”

```
// 定义框架
var A = A || {};
// 通过 ID 获取元素
A.g = function(id){
    return document.getElementById(id)
}
// 为元素绑定事件
A.on = function(id, type, fn){
    // 如果传递参数是字符串则以 id 处理，否则以元素对象处理
    var dom = typeof id === 'string' ? this.g(id) : id;
    // 标准 DOM2 级添加事件方式
    if(dom.addEventListener){
        dom.addEventListener(type, fn, false);
    // IE DOM2 级添加事件方式
    }else if(dom.attachEvent){
        dom.attachEvent('on' + type, fn);
    // 简易添加事件方式
    }else{
        dom['on' + type] = fn;
    }
}
```

“那么要完成上面的需求我们可以这样做。”

```
// 窗口加载完成事件
A.on(window, 'load', function(){
    // 按钮点击事件
    A.on('mybutton', 'click', function(){
        // do something
    })
})
```

“好了，小白，那么我想引入 jQuery 来换原有的 A 库，你知道该如何做么？”

小白思考了一下说：“首先 g 方法是通过 id 获取元素，所以通过\$（jQuery 的简写名称）方法获取 jQuery 对象然后通过 get 获取第一个成员即可，不过 on 方法有些复杂，我们不能直接替换，因为 jQuery 和我们的 A 库在通过 id 获取元素时是有区别的，jQuery 的 id 前面要加#。所以异类框架的适配器应该是这样的吧。”

```
A.g = function(id){
    // 通过 jQuery 获取 jQuery 对象，然后返回第一个成员
    return $(id).get(0);
}
A.on = function(id, type, fn){
    // 如果传递参数是字符串则以 id 处理，否则以元素对象处理
    var dom = typeof id === 'string' ? $('#' + id) : $(id);
    dom.on(type, fn);
}
```

“你还是很聪明的。是这样，通过适配器我们发现如果两种框架的‘血缘’比较相近，那么我们适配起来是比较容易的，如果‘血缘’相差甚远我们的适配器写起来要复杂得多，因此你要记住，日后非到万不得已情况下，尽量引入相似框架。”

“是呀，后一种要写不少代码呀。”小白补充说。

10.5 参数适配器

“除此之外，适配器还有很多用途，比如方法需要传递多个参数，例如……”

```
function doSomeThing(name, title, age, color, size, prize){}
```

“那么我们记住这些参数的顺序是很困难的，所以我们经常是以一个参数对象方式传入的。如下所示。”

```
/**
 * obj.name : name
 * obj.title : title
 * obj.age : age
 * obj.color : color
 * obj.size : size
 * obj.prize : prize
 */
```

```
function doSomeThing(obj){}
```

“然而当调用它的时候又不知道传递的参数是否完整，如有一些必须参数没有传入，一些参数是有默认值的等等，此时我们通常的做法是用适配器来适配传入的这个参数对象。如下所示。”

```
function doSomeThing(obj){
    var _adapter = {
        name : '雨夜清荷',
        title : '设计模式',
        age : 24,
        color : 'pink',
        size : 100,
        prize : 50
    };
    for(var i in _adapter){
        _adapter[i] = obj[i] || _adapter[i];
    }
    // 或者 extend(_adapter, obj) 注：此时可能会多添加属性
    // do things
}
```

“这种方式很常见呀，很多插件对于参数配置都是这么做的。”小白应道。

10.6 数据适配

“没看出你接触过插件开发。”小铭接着说。“对于这类对参数的适配又有衍生，比如对数据的适配，比如这里有一个数组。”

```
var arr = ['JavaScript', 'book', '前端编程语言', '8月1日'];
```

“我们发现数组中的每个成员代表的意义不同，所以这种数据结构语义不好，我们通常会将其适配成对象形式，比如下面这种对象数据结构。”

```
var obj = {
    name : '',
    type : '',
    title : '',
    time : ''
}
```

“我们就可以像下面这样适配。”

```
function arrToObjAdapter(arr){
    return {
        name : arr[0],
        type : arr[1],
        title : arr[2],
        data : arr[3]
    }
}
```

```

    }
var adapterData = arrToObjAdapter(arr);
console.log(adapterData) // {name: "JavaScript", type: "book", title: "前
段编程语言", data: "8月1日"}

```

“这也为数据的灵活使用提供了新思路了。”小白感叹道。

10.7 服务器端数据适配

“是呀！”小铭接着说，“但是，你知道么，最重要的是它解决了前端的数据依赖，前端程序不再为后端传递的数据所束缚，如果后端因为架构改变导致传递的数据结构发生变化，我们只需要写个适配器就可以放心了。比如我们用 `jQuery` 向后端 `someAdress.php` 接口请求数据，然后用 `dosomething` 方法处理接受的数据。如果后端的数据经常变化，比如在某些网站拉取的新闻数据，后端有时无法控制数据的格式，我们在调用 `dosomething` 方法时最好不要直接调用，最好先将传递过来的数据适配成对我们可用的数据再使用，这样将更安全，如下面的例子。”

```

//为简化模型，这里使用 jQuery 的 ajax 方法 理想数据是一个一维数组
function ajaxAdapter(data){
    //处理数据并返回新数据
    return [data['key1'], data['key2'], data['key3']]
}
$.ajax({
    url : 'someAdress.php',
    success : function(data, status){
        if(data){
            //使用适配后的数据——返回的对象
            doSomething(ajaxAdapter(data));
        }
    }
});

```

“像这样，如果日后后端数据有任何变化我们只需相应地更改 `ajaxAdapter` 适配器转换格式就可以了。”

下章剧透

偶然的需求，让小白在页面中引入 `jQuery` 代码库，这就在无形中导致与原始代码库的冲突。这样让小白认识了适配器模式以及应用。下一章小白将遇到跨域访问的问题，看看他是如何解决的吧。

忆之获

传统设计模式中，适配器模式往往是适配两个类接口不兼容的问题，然而在 `JavaScript` 中，

适配器的应用范围更广，比如适配两个代码库，适配前端数据，等等。

JavaScript 中的适配器的应用，更多应用在对象之间，为了使对象可用，通常我们会将对象拆分并重新包装，这样我们就要了解适配对象的内部结构，这也是与外观模式的区别所在，当然适配器模式同样解决了对象之间的耦合度。包装的适配器代码增加了一些资源开销，当然这是微乎其微的。

我问你答

后端传输的数据往往并不是我们所期盼的，比如我们需要一个对象数据，而后端传输的是字符串或者一个 XML 文档，想一想我们又该如何适配呢？

第 11 章 牛郎织女——代理模式

代理模式（Proxy）：由于一个对象不能直接引用另一个对象，所以需要通过代理对象在这两个对象之间起到中介的作用。

由于用户相册模块上传的照片量越来越大，导致服务器端需要将图片上传模块重新部署到另外一个域（可理解为另一台服务器）中，这样对于前端来说，用户上传图片的请求路径发生变化，指向其他服务器，这就导致跨域问题。

11.1 无法获取图片上传模块数据

“小铭，你帮我看看，为什么我向咱们图片上传模块所在的服务器发送的请求，得不到数据呢？”小白问小铭。

```
// 当前域 www.xx.com
$.ajax({
  url : 'http://upload.xx.com/upload.php',
  success : function(res){
    // 无法获取返回的数据
  }
});
```

“打开你的控制台，你发现没有，已经报错了，出现跨域问题了。”

```
// 浏览器控制台报错：XMLHttpRequest cannot load http://upload.xx.com/upload.php.
No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

11.2 一切只因跨域

“哦，为什么会出现，什么是跨域？”小白问。

“由于 JavaScript 对安全访问因素的考虑，不允许跨域调用其他页面，这里的域你可以想象成域名，比如百度的域名 <http://www.baidu.com>，淘宝的域名 <http://www.taobao.com>，不同域名下的页面是不能直接调用的。这样百度域名下的页面是不允许直接调用淘宝页面。这也是一种 JavaScript 中因同源策略所定义的限制，不过仅此一点限制还不够，JavaScript 还对同一

域名不同的端口号、同一域名不同协议、域名和域名对应的IP、主域与子域、子域与子域等做了限制，都不能直接调用。如下所示。”

同一域名不同的端口号，如 <http://www.baidu.com:8001> 与 <http://www.baidu.com:8002>

同一域名不同协议，如 <http://www.baidu.com> 与 <https://www.baidu.com>

域名和域名对应的IP，如 <http://www.baidu.com> 与 <http://61.135.169.125>

主域与子域，如 <http://www.baidu.com> 与 <http://b.a.com>

子域与子域，如 <http://tieba.baidu.com> 与 <http://fanyi.baidu.com>

“这么多限制，原来我刚才遇到的情况正是主域与子域之间的跨域造成的呀。那么纵使这样，在主域下，我的相册页面还能与子域中的图片上传模块所在的服务器之间进行通信么？”小白问。

“这就需要一些技巧了，你看，相册页面与图片上传模块所在的服务器之间你可抽象成两个对象，那么现在的问题是，他们之间被一条河隔开了，就像天河两端的牛郎织女，只能远远观望而不能相聚一见。他们的情感感动万物，所以才有那么多需求为他们搭桥。同样你想让跨域两端的对象之间实现通信，你就需要找个代理对象来实现他们之间的通信。”

“我明白了，虽然他们之间分开了，但是我们可以找一个代理对象来实现相互之间的通信，不过对于这类代理对象又有哪些呢？”小白问。

11.3 站长统计

“当然，代理对象有很多，简单一点的如之类的标签通过src属性可以向其他域下的服务器发送请求。不过这类请求是get请求，并且是单向的，它不会有响应数据，就好比你站在河的一边向另一边发消息，却又不想让别人听见，所以你可以将你的消息写在纸上放在口袋里，然后扔过去，不过河对岸有没有人接收到你的消息就不得而知了。”

“你说的还挺有意思的，不过这类代理对象有什么应用呀？”小白问。

“很多呀，比如一些站长平台会有对于你的页面的统计项，其实现原理就是在你的页面触发一些动作的时候向站长平台发送这类的get请求，然后他们会对你发的请求做统计，然而你并不知道统计的相关消息。”小铭解释道。

```
// 统计代理
var Count = (function() {
    // 缓存图片（参考第二十二章，备忘录模式）
    var img = new Image();
    // 返回统计函数
    return function(param) {
        // 统计请求字符串
        var str = 'http://www.count.com/a.gif?' +
        // 拼接请求字符串
        for(var i in param) {
            str += i + '=' + param[i];
        }
    }
})()
```

```

    // 发送统计请求
    _img.src = str;
}
})();
// 测试用例，统计 num
Count({num : 10});

```

11.4 JSONP

“第二种代理对象形式是通过 script 标签。比如我们在 CDN（内容分发网络，一种更接近用户的网络架构，是用户可以就近获取内容）上更快速地获取 jQuery 文件时，用<script src="http://ajax.

googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js" type="text/JavaScript"></script>来获取，然而这种获取方式获取的 script 内容是不变的。而我们需要的代理对象，是对页面与浏览器间通信的，显然上面的方式还不能满足我们的需求，不过我们知道通过 src 属性可实现 get 请求，因此我们可以在 src 指向的 url(请求地址)上面添加一些字段信息，然后服务器端获取这些字段，再相应地生成一份内容。”

```

// 前端浏览器页面
<script type="text/JavaScript">
// 回调函数，打印出请求数据与响应数据
function jsonpCallBack(res,req){
    console.log(res,req);
}
</script>
<script type="text/JavaScript" src="http://localhost/test/jsonp.php?callback=jsonp_CallBack&data=getData"></script>
// 另外一个域下服务器请求接口
<?php
    /*后端获取请求字段数据，并生成返回内容*/
    $data = $_GET["data"];
    $callback = $_GET["callback"];
    echo $callback."('success', '". $data . "')";
?>

```

“这种方式，你可以想象成河里面的一只小船，通过小船将你的请求发送给对岸，然后对岸的人们将数据放在小船里为你带回来。”

“哦，那这种方式就需要其他域下的服务器端与前端协同工作开发功能了吧。”

11.5 代理模板

“当然，这种方式还要求其他域要有一定可靠性。否则将会攻击到你的网站。当然这种方式也被人称之为 JSONP 方案，有时我们还会通过一个方法来动态生成需要的 JSONP 中的<script>标签”。小铭接着说，“与之类似还有另外一种方案是被称之为代理模板的方案，他

的解决思路是这样的，既然不同域之间相互调用对方的页面是有限制的，那么自己域中的两个页面相互之间的调用是可以的，即代理页面 B 调用被代理的页面 A 中对象的方式是可以的。那么要实现这种方式我们只需要在被访问的域中，请求返回的 Header 重定向到代理页面，并在代理页面中处理被代理的页面 A 就可以了。”

“既然这样，是不是我们在自己的域中要有这样 A、B 两个页面了？”小白问。

“是的。比如我们将自己的域称为 X 域，另外的域称为 Y 域，X 域中要有一个被代理页面，即 A 页面。在 A 页面中应该具备三个部分，第一个部分是发送请求的模块，如 form 表单提交，负责向 Y 域发送请求，并提供额外两组数据，其一是要执行的回调函数名称，其二是 X 域中代理模板所在的路径，并将 target 目标指向内嵌框架。第二个部分是一个内嵌框架，如 iframe，负责提供第一个部分中 form 表单的响应目标 target 的指向，并将嵌入 X 域中的代理页面作为子页面，即 B 页面。第三个部分是一个回调函数，负责处理返回的数据。”

X 域中被代理页面 A

```
<script type="text/JavaScript">
function callback(data){
    console.log('成功接收数据', data);
}
</script>
<iframe name="proxyIframe" id="proxyIframe" src="">

</iframe>
<form action="http://localhost/test/proxy.php" method="post" target=
"proxyIframe">
    <input type="text" name="callback" value="callback">
    <input type="text" name="proxy" value="http://localhost:8080/proxy.html">
    <input type="submit" value="提交">
</form>
```

“其次在 X 域中我们也要有一个代理页面，主要负责将自己页面 URL 中 searcher 部分的数据解析出来，如 http://www.a.com?type=1&title=aa 这个 url 中 searcher 部分指的就是?type=1&title=aa。将数据重新组装好，调用 A 页面里的回调函数，将组装好的数据作为参数传入父页面中定义的回调函数中并执行。”

X 域中代理页面 B

```
<script type="text/JavaScript">
//页面加载后执行
window.onload = function(){
    //如果不在 A 页面中返回，不执行
    if(top == self) return;
    //获取并解析 searcher 中的数据
    var arr = location.search.substr(1).split('&'),
        //预定义函数名称以及参数集
        fn, args;
    for(var i = 0, len = arr.length, item; i < len; i++){
        //解析 searcher 中的每组数据
        item = arr[i].split('=');
```

```

//判断是否为回调函数
if(item[0] == 'callback'){
    //设置回调函数
    fn = item[1];
//判断是否是参数集
}else if(item[0] == 'arg'){
    //设置参数集
    args = item[1];
}
try{
    //执行 A 页面中预设的回调函数
    eval('top.' + fn + '("'+ args +'")');
} catch(e){}
}
</script>

```

“最后是 Y 域中的被请求的接口文件 C，它的主要工作是将从 X 域过来的请求的数据解析并获取回调函数字段与代理模板路径字段数据，并打包返回，并将自己的 Header 重定向为 X 域的代理模板 B 所在路径。”

```

<?php
$proxy = $_POST["proxy"];
$callback = $_POST["callback"];
header("Location: ".$proxy."?callback=".$callback."&arg=success");
?>
测试结果
/*
控制台输出依次是
成功接收数据 success
*/

```

下章剧透

今天小白通过跨域上传图片收获了在 JavaScript 中运用代理模式解决跨域问题的知识。明天项目经理将让小白完成表单提交的一些功能需求，我们一起去看看小白又会遇到哪些问题吧。

忆之获

通过几种代理模式对跨域问题的解决方案，我们可以看到代理对象可以完全解决被代理对象与外界对象之间的耦合，当然从对被代理的页面角度来看是一种保护代理，然而从服务器角度来看又是一种远程代理。除了在跨域问题中有很多应用外，有时对对象的实例化对资源的开销很大，如页面加载初期加载文件有很多，此时能够延迟加载一些图片对页面首屏加载时间收益是很大的；再比如图片预览页面，页面中有很多图片，面对这么多的图片如果一一加载对资源的开销也是很可怕的，所以通常是当用户点击某张图片时加载这张图片。但如果该图片源文

件也很大，此时我们常用的做法是先代理加载一张预览图片，然后再将原图片替换这张预览图片，这种代理有时也称为虚拟代理。

由此可见代理模式可以解决系统之间的耦合度以及系统资源开销大的问题，通过代理对象可保护被代理对象，使被代理对象拓展性不受外界的影响。也可通过代理对象解决某一交互或者某一需求中造成的大量系统开销。

当然无论代理模式在处理系统、对象之间的耦合度问题还是在解决系统资源开销问题，他都将构建出一个复杂的代理对象，增加系统的复杂度，同时也增加了一定的系统开销，当然有时对于这种开销往往是可接受的。

在 JavaScript 中，它的执行常常依托于浏览器，所以代理模式解决问题的思想有时也为我们也提供了一些解决问题的方案。

我问你答

文中提到的动态加载 script 标签的方法，查查资料看看你能否实现。

对于图片预览这种代理模式，新建一个面试试着去实现它。

第 12 章 房子装修——装饰者模式

装饰者模式（Decorator）：在不改变原对象的基础上，通过对对其进行包装拓展（添加属性或者方法）使原有对象可以满足用户的更复杂需求。

静止是相对的，运动是绝对的，所以没有一成不变的需求。这不，为增强用户使用表单的交互体验，项目经理找来小白，正在谈后续需求改进呢。

12.1

为输入框的新需求

“小白，”项目经理走过来，“用户信息表单需求有些变化，以前是当用户点击输入框时，如果输入框输入的内容有限制，那么其后面显示用户输入内容的限制格式的提示文案，现在我们要多加一条，默认输入框上边显示一行提示文案，当用户点击输入框时文案消失。”

小白一听心想：“这很简单，找到对应的代码，然后在后面增加几句就可以了嘛。”于是小白浏览一下前人写过的代码。

```
// 输入框元素
var telInput = document.getElementById('tel_input');
// 输入格式提示文案
var telWarnText = document.getElementById('tel_warn_text');
// 点击输入框显示输入框输入格式提示文案
input.onclick = function(){
    telWarnText.style.display = 'inline-block';
}
于是小白不假思索地修改了这些写过的代码
// 输入框元素
var telInput = document.getElementById('tel_input');
// 输入框输入格式提示文案
var telWarnText = document.getElementById('tel_warn_text');
// 输入框提示输入文案
var telDemoText = document.getElementById('tel_demo_text');
// 点击输入框显示输入框输入格式提示文案并隐藏输入提示文案
input.onclick = function(){
    telWarnText.style.display = 'inline-block';
    telDemoText.style.display = 'none';
}
```

可是悲剧发生了，小白修改了一个电话输入框，后面还有姓名输入框、地址输入框，等等，还要像电话输入框这样在文件中查找功能代码，然后一个一个修改么？小白皱起眉头。

小铭看到后，走过来：“小白，怎么了？”

“项目经理让我给原来用户信息输入框增加一些需求，然而我改一个容易，后面还有这么多输入框，我还要一个一个去文件中查找代码，真不知道该如何是好了。”

“哦，这样呀，试试装饰者模式吧。”

“装饰者模式？以前没有听过，是添加东西的意思么？”

12.2 装饰已有的功能对象

“嗯，很简单，比如你买一个新房子，你想住得更舒适，那么你刚刚买的未装修过的新房子就不能满足你的需求了，所以你要对其装修一番，放置一个床，摆上一个沙发，抬进来一个电视，这样生活更舒适。同样，装饰者模式也是这个道理，原有的功能已经不能满足用户的需求了，此时你要做的就是为原有功能添砖加瓦，设置新功能和属性来满足用户提出的需求。”

“我明白一些了，不过我要如何实现呢？”

“首先明确原有的功能是哪些，”小铭接着说，“看看你已经写过的功能代码，这些就是原有的功能，你要做的就是在这基础上添加一些功能来满足用户提出的需求。看下面的代码。”

```
// 装饰者
var decorator = function(input, fn){
    // 获取事件源
    var input = document.getElementById(input);
    // 若事件源已经绑定事件
    if(typeof input.onclick === 'function'){
        // 缓存事件源原有回调函数
        var oldClickFn = input.onclick;
        // 为事件源定义新的事件
        input.onclick = function(){
            // 事件源原有回调函数
            oldClickFn();
            // 执行事件源新增回调函数
            fn();
        }
    }else{
        // 事件源未绑定事件，直接为事件源添加新增回调函数
        input.onclick = fn;
    }
    // 做其他事情
}
```

12.3 为输入框添砖加瓦

“看看上面的代码，此时装饰者不仅仅可以对绑定过事件的输入框添加新的功能，未绑定

过的输入框同样可以。你可以像下面这样调用。”

```
// 电话输入框功能装饰
decorator('tel_input', function(){
    document.getElementById('tel_demo_text').style.display = 'none';
});
// 姓名输入框功能装饰
decorator('name_input', function(){
    document.getElementById('name_demo_text').style.display = 'none';
});
// 地址输入框功能装饰
decorator('adress_input', function(){
    document.getElementById('adress_demo_text').style.display = 'none';
});
```

“真是太棒了，通过使用装饰者对象方法，无论输入框是否绑定过事件，都可以轻松完成增加隐藏示例框的需求。真的很不错。”小白感叹道。

“装饰者模式很简单，就是对原有对象的属性与方法的添加。”

小白想了想问：“前几天我们学过的适配器模式也是对一个对象的修饰来适配其他对象，那么他与装饰者模式有什么不同呢？”

“适配器方法是对原有对象适配，添加的方法与原有方法功能上大致相似。但是装饰者提供的方法与原来的方法功能项是有一定区别的。再有，使用适配器时我们新增的方法是要调用原来的方法呀。不过在装饰者模式中，不需要了解对象原有的功能，并且对象原有的方法照样可以原封不动地使用。”小铭答道。

“哦，对了，你也提醒了我，这么说既然在适配器模式中增加的方法要调用原有的方法，是不是就要了解原有方法实现的具体细节，而在装饰者中原封不动地使用，我们就不需要知道原有方法实现的具体细节，只有当我们调用方法时才会知道的。”

“嗯，正是这样。”小铭笑了。

下章剧透

一个事件拓展案例使小白学到了装饰者模式，明天小白将会开发用户信息模块，届时我们将去看看小白又将遇到什么问题。

忆之获

通过小白对输入框交互功能的拓展，我们学习了一种可以在不了解原有功能的基础上对功能拓展模式，这是对原有功能的一种增强与拓展。当然同样对原有对象进行拓展的模式还有适配器模式，所不同的是适配器进行拓展很多时候是对对象内部结构的重组，因此了解其自身结构是必需的。而装饰者对对象的拓展是一种良性拓展，不用了解其具体实现，只是在外部进行

了一次封装拓展，这又是对原有功能完整性的一种保护。

我问你答

对输入框添加 focus 与 blur 事件，想一想通过装饰者模式如何实现？

第 13 章 城市间的公路——桥接模式

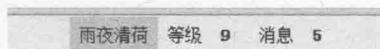
桥接模式（Bridge）：在系统沿着多个维度变化的同时，又不增加其复杂度并已达到解耦。

有时候页面中的一些小小细节改变常常因逻辑相似导致大片臃肿的代码，让页面苦涩不堪。这不，小白为解决这类问题，已经熬了整整一上午了。

13.1 添加事件交互

“小白，什么功能让你写了一上午？”小铭问。

“这不，项目经理让我把页面上部的用户信息部分添加一些鼠标划过特效（如图 13-1 所示）。”



▲图 13-1 用户信息模块

小白接着说：“哎，不过用户信息由很多小部件组成。你看，对于用户名，鼠标划过直接改变背景色，但是像用户等级、用户消息这类部件只能改变里面的数字内容，处理的逻辑不太一样，所以写了不少代码，不过写完时，自己感觉很多是冗余的，却又不知道该如何改善。”

“哦？来让我看看你的代码吧。”

```
var spans = document.getElementsByTagName('span');
// 为用户名绑定特效
spans[0].onmouseover = function(){
    this.style.color = 'red';
    this.style.background = '#ddd';
};
spans[0].onmouseout = function(){
    this.style.color = '#333';
    this.style.background = '#f5f5f5';
};
// 为等级绑定特效
spans[1].onmouseover = function(){
    this.getElementsByTagName('strong')[0].style.color = 'red';
    this.getElementsByTagName('strong')[0].style.background = '#ddd';
};;
```

```

spans[1].onmouseout = function(){
    this.getElementsByTagName('strong')[0].style.color = '#333';
    this.getElementsByTagName('strong')[0].style.background = '#f5f5f5';
};

.....

```

13.2 提取共同点

“看你的代码是有点冗余，不过你知道你的问题出在哪里么？”

“要对事件的回调函数再做处理么？”小白问。

“嗯，你在写代码时一定要注意对相同的逻辑做抽象提取处理，这一点很重要。如果这一点你能做到，那么你的代码将会更简洁，重用率也会更大，当然可读性更高，这也是我推荐你使用面向对象思想编程的一个目的。”小铭接着说，“对于用户信息模块的每一个部分鼠标划过与鼠标离开两个事件的执行函数有很大一部分是相似的，比如他们都处理每个部件中的某个元素，他们都是处理该元素的字体颜色和背景颜色。所以对于这个相似点的抽象提取是很有必要的，因此你可以创建下面这样一个函数，让它解除与事件中的 this 的耦合。”

```

// 抽象
function changeColor(dom, color, bg){
    // 设置元素的字体颜色
    dom.style.color = color;
    // 设置元素的背景颜色
    dom.style.background = bg;
}

```

13.3 事件与业务逻辑之间的桥梁

“剩下你要做的就是对元素绑定事件了，但是有一点你要明白，仅仅知道元素事件绑定与抽象提取的设置样式方法 changeColor 还是不够的，你需要用一个方法将他们链接起来。那么这个方法就是桥接方法，这种模式就是桥接模式。就像你在北京开着车去沈阳旅游，那么你就要找到一条连接北京与沈阳的公路，才能顺利地在两地往返。”小铭接着说，“那么对于事件的桥接方法，我们可以用一个匿名函数来代替，否则直接将 changeColor 作为事件的回调函数，那么我们刚才所做的事情就白做了，因为它们还将耦合在一起。如下面的为用户名绑定事件。”

```

var spans = document.getElementsByTagName('span');
spans[0].onmouseover = function(){
    changeColor(this, 'red', '#ddd');
}

```

“changeColor 方法中的 dom 实质上是事件回调函数中的 this，那么我们想解除它们之间的耦合，我们就需要一个桥接方法——匿名回调函数。通过这个回调函数，我们将获取到的 this 传递到 changeColor 函数中，即可实现需求。同样对于用户名模块的鼠标移开事件用同样的方

式即可。”

```
spans[0].onmouseout = function(){
    changeColor(this, '#333', '#f5f5f5');
}
```

“好了，小白，对于用户名等级这类特殊的需求你知道该怎么做了吧。”小铭问。

“嗯，既然用户名模块的 this 在桥接匿名函数中获取，那么也应该是同样的道理应用在用户等级上，通过桥接函数来获取数字元素，然后传入 changeColor 就可以了吧。像下面这样。”说着小白把余下的代码写了出来。

```
spans[1].onmouseover = function(){
    changeColor(this.getElementsByTagName('strong')[0], 'red', '#ddd');
}
spans[1].onmouseout = function(){
    changeColor(this.getElementsByTagName('strong')[0], '#333', '#f5f5f5');
}
```

“嗯，小白，你再来看看是不是现在与之前相比清晰许多了，如果再想对需求做任何修改我们只需要修改 changeColor 的内容就可以了，而不必去到每个事件回调函数中去修改。当然这种实现方式看起来调理更清晰，但是别忘了它是以新增一个桥接函数为代价实现的。”

“是呀，听你说的，感觉桥接模式只是先抽象提取共用部分，然后将实现与抽象通过桥接方法链接在一起，来实现解耦的作用的吧。”

13.4 多元化对象

“你说的对，不过桥接模式的强大之处不仅仅在此，甚至对于多维的变化也同样适用。比如我们书写一个 canvas 跑步游戏的时候，对于游戏中的人、小精灵、小球等一系列的实物都有动作单元，而他们的每个动作实现起来方式又都是统一的，比如人和精灵和球的运动其实都是位置坐标 x 和 y 的变化，球的颜色与精灵的色彩的绘制方式都相似等，这样我们可以将这些多维变化部分，提取出来作为一个抽象运动单元进行保存，而当我们创建实体时，将需要的每个抽象动作单元通过桥接，链接在一起运作。这样它们之间不会相互影响并且该方式降低了它们之间的耦合。”

```
// 多维变量类
// 运动单元
function Speed(x, y){
    this.x = x;
    this.y = y;
}
Speed.prototype.run = function(){
    console.log('运动起来');
}
// 着色单元
function Color(c1){
```

```

        this.color = cl;
    }
Color.prototype.draw = function(){
    console.log('绘制色彩');
}
// 变形单元
function Shape(sp){
    this.shape = sp;
}
Shape.prototype.change = function(){
    console.log('改变形状');
}
// 说话单元
function Speek(wd){
    this.word = wd;
}
Speek.prototype.say= function(){
    console.log('书写字体');
}

```

“于是我们想创建一个球类，并且它可以运动，可以着色。”

```

function Ball(x,y,c){
    // 实现运动单元
    this.speed = new Speed(x,y);
    // 实现着色单元
    this.color = new Color(c);
}
Ball.prototype.init = function(){
    // 实现运动
    this.speed.run();
    // 实现着色
    this.color.draw();
}

```

“同样我们想创建一个人物类，他可以运动以及说话。”

```

function People(x, y, f){
    this.speed = new Speed(x,y);
    this.font = new Speek(f);
}
People.prototype.init = function(){
    this.speed.run();
    this.font.say();
}

```

“当然我们也可以创建一个精灵类，让它可以运动可以着色可以改变形状。”

```

function Spirite(x, y, c, s){
    this.speed = new Speed(x, y);
    this.color = new Color(c);
    this.shape = new Shape(s);
}
Spirite.prototype.init = function(){
    this.speed.run();
}

```

```
this.color.draw();
this.shape.change();
}
```

“当我们想实现一个人物时，我们直接实例化一个人物对象，这样他就可以有用运动和说话的动作了。”

```
var p = new People(10, 12, 16);
p.init();
```

下章剧透

通过学习桥接模式，我们学会了如何将元素的事件与业务逻辑之间解耦。当然通过桥接模式我们也可以灵活地创建一个对象。那么在下一章我们将学习一种新的模式，来解决复杂而多变页面的创建工作。

忆之获

桥接模式最主要的特点即是将实现层（如元素绑定的事件）与抽象层（如修饰页面 UI 逻辑）解耦分离，使两部分可以独立变化。由此可以看出桥接模式主要是对结构之间的结构。而前面学过的抽象工厂模式与创建者模式主要业务在于创建。通过侨联模式实现的解耦，使实现层与抽象层分开处理，避免需求的改变造成对象内部的修改，体现了面向对象对拓展的开放及对修改的关闭原则，这是很有用的。

当然由于侨联的添加，有时也造成开发成本的增加。有时性能上也会受到影响。

我问你答

创建一个对象桥接 method，实现为对象拓展方法的功能。

第14章 超值午餐——组合模式

组合模式 (Composite): 又称部分-整体模式，将对象组合成树形结构以表示“部分整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

为强化首页用户体验，经理准备在用户首页添加一个新闻模块，当然新闻的内容是根据用户平时关注的内容挖掘的，因此有的人可能会显示文字新闻，有的人会是图片新闻，甚至有的人显示的新闻是一个直播链接，方便用户观看比赛……

14.1 新闻模块十万火急

“小白，新闻模块添加一条文字新闻。”

“小白，新闻模块添加一条带有直播图标的文字新闻。”

“小白，新闻模块添加一条已分类的文字新闻。”

“小白，新闻模块添加一条图片新闻。”

“小白，将图片新闻和文字新闻放在一行。”

.....

项目经理需求的呼喊声频频入耳。

“淡定，淡定。”小白心想，“这么多需求盲目地去写代码一定会把自己累惨，可是我该如何去解决这些需求问题呢？”小白皱起眉头。

身旁小铭见状，拍了拍小白肩膀：“怎么了，什么事情把你难成这样？”

小白一五一十地将项目经理的需求跟小铭说了一遍。

“原来是这样，不过你面对复杂的需求能够冷静思考这是很对的，否者你将陷入代码重构（因已有的代码不能完成当前的需求，以至于对代码重新编写）当中。”小铭接着说，“不过你再浏览一遍需求，你会发现，需求中的这些新闻大致可以分为相互独立的几种类型，而对某类新闻做修改时又不会影响到其他类新闻，所以你完全可以将每一类新闻抽象成面向对象编程中的一个类，这样你就不会担心日后对某类新闻需求修改而影响到其他类新闻，你只需针对于这类新闻做相应的修改即可，如果要完成经理提出的需求，你仅需对在这些类新闻中挑选一些组合成需要的模块，所以说建议你用组合模式来完成经理提出的需求更稳妥。”

“组合模式？听你刚才说的，是不是说在实现上先将新闻模块整体拆分出个体，然后需要时寻找相应的个体组合成新的整体的意思呀？”小白追问一句。

14.2 餐厅里的套餐业务

“嗯，是这样。举个例子，比如一会我们中午出去吃饭，那么我们进入快餐店，如果我们点餐时对于喝的、吃的等一个一个点是很浪费时间的。如果餐厅能提供一个快餐服务，那么我们点餐就简单多了，我们仅提供套餐的名字即可，而且这个结果与我们一个一个点餐的结果是同等的，而快餐店虽然给我们提供套餐服务，但是他们仍专注于组成套餐的每一个菜肴（组成部分），而提供给我们的又仅仅是一个组合结果。每一种菜（部分）之间又是相对独立的，他们可以放心做好每一道菜（部分），而最终他们又能组合出更复杂的套餐（整体）提供给我们。这种实现就是组合。”

“这么说，整体就是对部分的组合，这样就简化了复杂的整体，通过不同的部分组合又丰富了整体。那么对于部分有什么约束要求吗？”

14.3 每个成员要有祖先

“要说要求，我想只有一个，就是接口的统一，在 JavaScript 中我们可以通过继承同一个虚拟类来实现，比如你这个需求中我们可以让所有的新闻都继承一个新闻虚拟父类 News，如下所示。”

```
var News = function() {
    //子组件容器
    this.children = [];
    //当前组件元素
    this.element = null;
}
News.prototype = {
    init : function(){
        throw new Error("请重写你的方法");
    },
    add : function(){
        throw new Error("请重写你的方法");
    },
    getElement : function(){
        throw new Error("请重写你的方法");
    }
}
```

“可是既然是虚拟父类，为何在这个类的构造函数中还要声明一些特权变量呢？”

“嗯，是这样，通常虚拟类是定义而不实现的，但是我们在虚拟类的构造函数中定义两个特权变量时因为后面的所有继承子类都要声明这两个变量，为了简化子类我们也可以将这些共

有的变量提前声明在父类中。”

“那么我们定义了这个接口虚拟父类，下一步我们是不是就可以实现所有子类了？”小白问。

“嗯，不过你要注意一下他们的层次关系，有一点你要知道，组合模式不仅仅是单层次组合，也可以是一个多层次的，比如需求中的将图片新闻和文字新闻放在一行的条件就是说，我们将组合后的整体作为一个部分，继续组合。这样你就应该在拆分整体后还要确定他们的层次关系，比如最顶层是一个新闻模块的容器，再往下面是每一行新闻成员集合，每一行还可能有新闻组合体，当然最后一层组合体里面的成员就是新闻对象了。明白这些你就可以创建每一条新闻对象了。”

“这么说对象的上一层是可以有子成员的，最底层中的对象是没有子成员的。”

14.4 组合要有容器类

“对呀，所以你写代码时就要注意这些层次关系了，最后一点需要说明的是，在组合模式中用到了继承，所以我们要把以前学过的继承相关知识拿出来用在组合模式中，比如应用寄生组合式继承，像下面这样显现新闻模块容器类。”

```
// 容器类构造函数
var Container = function(id, parent){
    // 构造函数继承父类
    News.call(this);
    // 模块 id
    this.id = id;
    // 模块的父容器
    this.parent = parent;
    // 构建方法
    this.init();
}
// 寄生式继承父类原型方法
inheritPrototype(Container, News);
// 构建方法
Container.prototype.init = function(){
    this.element = document.createElement('ul');
    this.element.id = this.id;
    this.element.className = 'new-container';
};
// 添加子元素方法
Container.prototype.add = function(child){
    // 在子元素容器中插入子元素
    this.children.push(child);
    // 插入当前组件元素树中
    this.element.appendChild(child.getElement());
    return this;
}
// 获取当前元素方法
Container.prototype.getElement = function(){
    return this.element;
}
```

```
// 显示方法
Container.prototype.show = function(){
    this.parent.appendChild(this.element);
}
```

“同样下一层级的行成员集合类以及后面的新闻组合体类实现的方式与之类似。”

```
var Item = function(classname){
    News.call(this);
    this.classname = classname || '';
    this.init();
}
inheritPrototype(Item, News);
Item.prototype.init = function(){
    this.element = document.createElement('li');
    this.element.className = this.classname;
}
Item.prototype.add = function(child){
    //在子元素容器中插入子元素
    this.children.push(child);
    //插入当前组件元素树中
    this.element.appendChild(child.getElement());
    return this;
}
Item.prototype.getElement = function(){
    return this.element;
}

var NewsGroup = function(classname) {
    News.call(this);
    this.classname = classname || '';
    this.init();
}
inheritPrototype(NewsGroup, News);
NewsGroup.prototype.init = function(){
    this.element = document.createElement('div');
    this.element.className = this.classname;
}
NewsGroup.prototype.add = function(child){
    //在子元素容器中插入子元素
    this.children.push(child);
    //插入当前组件元素树中
    this.element.appendChild(child.getElement());
    return this;
}
NewsGroup.prototype.getElement = function(){
    return this.element;
}
```

14.5 创建一个新闻类

“好了，小白，上面已经把所有子成员类创建出来了，不过光有这些新闻容器类是不行的，我们还需要有更底层的新闻类，但是注意这些新闻成员类是不能拥有子成员的，但是他们继承

了父类，所以对于 add 方法最好声明一下，比如我们创建图片新闻类。”

```
var ImageNews = function(url, href, classname) {
    News.call(this);
    this.url = url || '';
    this.href = href || '#';
    this.classname = classname || 'normal';
    this.init();
}
inheritPrototype(ImageNews, News);
ImageNews.prototype.init = function() {
    this.element = document.createElement('a');
    var img = new Image();
    img.src = this.url;
    this.element.appendChild(img);
    this.element.className = 'image-news' + this.classname;
    this.element.href = this.href;
}
ImageNews.prototype.add = function(){}
ImageNews.prototype.getElement = function(){
    return this.element;
}
```

“新闻类与上面的容器类很像，也很简单吧，那么下面你把剩下的基类新闻自己动手创建一下吧。”

```
var IconNews = function(text, href, type){
    News.call(this);
    this.text = text || '';
    this.href = href || '#';
    this.type = type || 'video';
    this.init();
}
inheritPrototype(IconNews, News);
IconNews.prototype.init = function(){
    this.element = document.createElement('a');
    this.element.innerHTML = this.text;
    this.element.href = this.href;
    this.element.className = 'icon' + this.type;
}
IconNews.prototype.add = function(){}
IconNews.prototype.getElement = function(){
    return this.element;
}

var EasyNews = function(text, href){
    News.call(this);
    this.text = text || '';
    this.href = href || '#';
    this.init();
}
inheritPrototype(EasyNews, News);
EasyNews.prototype.init = function(){
    this.element = document.createElement('a');
```

```

this.element.innerHTML = this.text
this.element.href = this.href;
this.element.className = 'text';
}
EasyNews.prototype.add = function(){}
EasyNews.prototype.getElement = function(){
    return this.element;
}

var TypeNews = function(text, href, type, pos){
    News.call(this);
    this.text = text || '';
    this.href = href || '#';
    this.type = type || '';
    this.pos = pos || 'left';
    this.init();
}
inheritPrototype(TypeNews, News);
TypeNews.prototype.init = function(){
    this.element = document.createElement('a');
    if(this.pos === 'left'){
        this.element.innerHTML = '[' + this.type + ']' + this.text;
    }else{
        this.element.innerHTML = this.text + '[' + this.type + ']';
    }
    this.element.href = this.href;
    this.element.className = 'text';
}
TypeNews.prototype.add = function(){}
TypeNews.prototype.getElement = function(){
    return this.element;
}

```

14.6 把新闻模块创建出来

“小铭，新闻类都创建了，可是我在使用时，就要通过 add 方法像一棵树一样一层一层创建新闻就可以了吧。”

```

var news1 = new Container('news', document.body);
news1.add(
    new Item('normal').add(
        new IconNews('梅西不拿金球也伟大', '#', 'video')
    )
).add(
    new Item('normal').add(
        new IconNews('保护强国强队用意明显', '#', 'live')
    )
).add(
    new Item('normal').add(
        new NewsGroup('has-img').add(
            new ImageNews('img/1.jpg', '#', 'small')
        ).add(

```

```

        new EasyNews('从 240 斤胖子成功变型男', '#')
    ).add(
        new EasyNews('五大雷人跑步机', '#')
    )
)
).add(
    new Item('normal').add(
        new TypeNews('AK47 不愿为费城打球', '#', 'NBA', 'left')
    )
).add(
    new Item('normal').add(
        new TypeNews('火炮飚 6 三分创新高', '#', 'CBA', 'right')
    )
).show();

```

“对，是这样，这样创建的每一条新闻都是一个独立的个体，互不影响，避免相互间的耦合，也增强了组合后的模块的复杂性。日后不论再有什么样的需求，做相应的组合就可以轻松完成。那么，你刷新一下页面看看你劳作的结果吧（如图 14-1 所示）。”



▲图 14-1 新闻模块

14.7 表单中的应用

“不过，小白，你知道么，其实在页面中，组合模式更常用在创建表单上，比如注册页面可能有不同的表单提交模块。对于这些需求我们只需要有基本的个体，然后通过一定的组合即可实现，比如下面这个页面样式（如图 14-2 所示），你用组合模式实现吧。”

“不妨给你一个小提示，模仿上面创建新闻模块那样创建基类 Base，然后 3 个组合类 FormItem、FieldsetItem、Group，以及成员类 InputItem、LabelItem、SpanItem、TextareaItem，创建完之后像下面这样拼出你的注册页面吧。”

```

var form = new FormItem('FormItem', document.body);
form.add(
    new FieldsetItem('account', '账号').add(
        new Group().add(
            new LabelItem('user_name', '用户名：')
        ).add(

```

```

        new InputItem('user_name')
    ).add(
        new SpanItem('4 到 6 位数字或字母')
    )
).add(
    new Group().add(
        new LabelItem('user_password', '密&emsp;码：')
    ).add(
        new InputItem('user_password')
    ).add(
        new SpanItem('6 到 12 位数字或者密码')
    )
)
).add(
    //.....
).show();

```

账号

用户名: 4到6位数字或字母

密 码: 6到12位数字或者密码

信息

昵称:

状态:

提交

▲图 14-2 表单模块

下章剧透

本章通过对新闻模块的创建，我们学习了组合模式的应用，那么随着用户对新闻访问量的增大，新闻逐渐增多，此时新闻翻页势在必行，那么我们在下一章看看小白又会遇到哪些问题呢？

忆之获

组合模式能够给我们提供一个清晰的组成结构。组合对象类通过继承同一个父类使其具有统一的方法，这样也方便了我们统一管理与使用，当然此时单体成员与组合体成员行为表现就比较一致了。这也就模糊了简单对象与组合对象的区别。有时这也是一种对数据的分级式处理。清晰而又方便我们对数据的管理与使用。

当然组合模式有时在实现需求上给我们带来更多的选择方式，虽然对于单体对象的实现简单而又单一，但是通过对其组合将会给我们带来更多的使用形式。

我问你答

表单组合很常见，那么根据上面提供的使用方式，请你推导出表单成员基类吧。

第 15 章 城市公交车——享元模式

享元模式 (Flyweight): 运用共享技术有效地支持大量的细粒度的对象，避免对象间拥有相同内容造成多余的开销。

新闻模块终于写完了，可是随着内容的增多，单页面已经无法容纳所有的新闻了，所以经理准备采取分页显示所有的新闻。

15.1

翻页需求

“小铭，你能帮我看一下么，我写的代码在 Chrome 浏览器下运行良好，可是到了低版本 IE 浏览器下怎么感觉一卡一卡的呢。”

小铭走过来：“来，我看看你写的代码吧，你要实现一个什么样的功能？”

“哦，很简单的一个新闻翻页功能，点击下一页隐藏当前页面的新闻，然后显示后面的 5 个，这里是我写的代码。”

```
var dom = null, // 缓存创建的新闻标题元素
    paper = 0, // 当前页数
    num = 5, // 每页显示新闻数目
    i = 0, // 创建新闻元素时保存变量
    len = article.length; // 新闻数据长度
for(; i < len; i++){
    dom = document.createElement('div'); // 创建包装新闻标题元素
    dom.innerHTML = article[i]; // 向元素中添加新闻标题
    if(i >= num){ // 默认显示第一页
        dom.style.display = 'none'; // 超出第一页新闻隐藏
    }
    document.getElementById('container').appendChild(dom); // 添加到页面中
}
// 下一页绑定事件
document.getElementById('next_page').onclick = function(){
    var div = document.getElementById('container').getElementsByTagName('div'),
        // 获取所有新闻标题包装元素
        j = k = n = 0; // j, k 循环变量, n 当前页显示的第一个新闻序号
        n = ++paper % Math.ceil(len / num) * num; // 获取当前页显示的第一个新闻序号
        for(; j < len; j++){
            div[j].style.display = 'none'; // 隐藏所有新闻
        }
}
```

```

    }
    for(; k < 5; k++) {
        if(div[n + k])
            div[n + k].style.display = 'block'; // 显示当前页新闻
    }
}

```

“我的思路是，页面加载后，异步请求新闻数据，然后创建所有条新闻并插入页面中，需要显示哪一页就将对应页的新闻显示，其他的新闻隐藏……”

15.2

冗余的结构

还没等小白解释完，小铭接过话来：“你的问题就出在这里了，你看，所有的新闻都有同样的结构，只是内容不同罢了，所以你创建的几百条新闻同时插入页面并操作造成的多余的开销在低版本的IE浏览器中当然会严重影响其性能。那么对于相同结构造成的多余开销问题，你可以用享元模式来解决。”

“享元模式？”小白不解，“你是指对相同的数据结构进行提取么？”

“差不多，不过享元模式主要还是对其数据、方法共享分离，它将数据和方法分成内部数据、内部方法和外部数据、外部方法。内部方法与内部数据指的是相似或者共有的数据和方法，所以将这一部分提取出来减少开销，以提高性能。就像城市里人们每天上班或者学生上学都要走一定的路。他们有的人开私家车，有的坐公交，当然我们是很容易看出坐公交在花费上要比私家车的开销少很多，主要是因为它让更多的人共有这一交通工具。”

15.3

享元对象

“是这样，那么我写的功能里，这些新闻个体都有共同的结构，所以它们应该作为内部的数据，而‘下一页’按钮绑定的事件已经不能再抽象提取了，所以应该就是外部的方法吧。”小白问。

“没错，不过还有一点，既然这些内部的数据提取出来了，为了能使用它们，我们还需要提供一个操作方法，像下面这样。”

```

var Flyweight = function(){
    // 已创建的元素
    var created = [];
    // 创建一个新闻包装容器
    function create(){
        var dom = document.createElement('div');
        // 将容器插入新闻列表容器中
        document.getElementById('container').appendChild(dom);
        // 缓存新创建的元素
        created.push(dom);
        // 返回创建的新元素
        return dom;
    }
}

```

```

    }
    return {
        // 获取创建新闻元素方法
        getDiv : function(){
            // 如果已创建的元素小于当前页元素总个数，则创建
            if(created.length < 5){
                return create();
            }else{
                // 获取第一个元素，并插入最后面
                var div = created.shift();
                created.push(div);
                return div;
            }
        }
    }();
}

```

15.4

实现需求

“首先，创建一个享元类，由于我们每页只能显示 5 条新闻，所以我们创建 5 个元素，保存在享元类的内部，我们可以通过享元类为我们提供的方法 getDiv 来获取创建的元素。内部的数据和内部的方法提取出来之后我们就要实现外部的数据和外部的方法，外部的数据就是我们要显示的所有新闻内容，每个内容都不一样，所以它们不能被共享。当然，为下一页绑定的事件是独立的，它只绑定一次。然后我们要做两件事情，第一件事情，我们要根据新闻的内容实例化页面，第二件事情，对下一页绑定一个点击事件，显示下一页，显示过程中我们可以优化一个策略，就是如果当前页显示的新闻数据不够 5 条，那么我们可以在所有新闻数据中起始处获取，这样，就要求我们新闻数据的总条数大于页面最多显示的条数——5 条。首先我们先初始化页面。”

```

var paper = 0,
    num = 5,
    len = article.length;
// 添加 5 条新闻
for(var i = 0; i < 5; i++){
    if(article[i])
        // 通过享元类获取创建的元素并写入新闻内容
        Flyweight.getDiv().innerHTML = article[i];
}

```

“现在生成新闻界面很简单了，我们还要给‘下一页’元素绑定一个事件，像下面这样。”

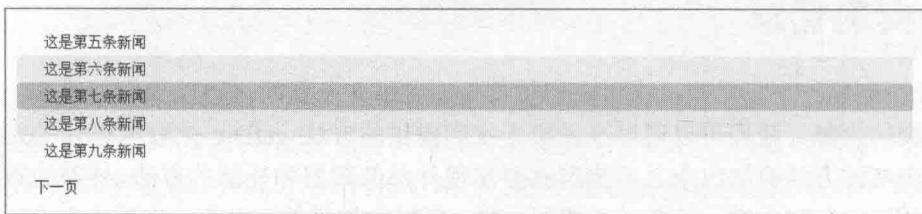
```

// 下一页按钮绑定事件
document.getElementById('next_page').onclick = function(){
    // 如果新闻内容不足 5 条则返回
    if(article.length < 5)
        return;
    var n = ++paper * num % len, // 获取当前页的第一条新闻索引
        j = 0; // 循环变量
    // 插入 5 条新闻
}

```

```
for(; j < 5; j++) {
    // 如果存在第 n + j 条则插入
    if(article[n + j]){
        Flyweight.getDiv().innerHTML = article[n + j];
    }
    // 否则插入起始位置第 n + j - len 条
    }else if(article[n + j - len]){
        Flyweight.getDiv().innerHTML = article[n + j - len];
    }
    // 如果都不存在则插入空字符串
    }else{
        Flyweight.getDiv().innerHTML = "";
    }
}
}
```

“小白，你看我们通过享元模式对页面重构之后每次操作只需要操作那 5 个元素，是不是与之前对几百个元素操作的性能相比，提高了许多，赶紧打开页面测试一下吧。”



▲图 15-1 分页新闻

小白刷新了一下页面，体验一下效果（如图 15-1 所示），“性能果然提高了不少，这么好的模式还有哪些用途呢？”

15.5 享元动作

“你倒是挺机灵的，还想多学点，好吧，其实我们在面向对象编程上还是很常用的，比如说一个游戏中我们可能创建一些人、精灵等角色，那么它们都会有运动这个动作，其实这一动作在所有角色中实现的方式都是相同的。对此我们可以创建一个通用的享元类，让它可以实现横向移动以及纵向移动，举例如下。”

```
var FlyWeight = {
    moveX : function(x) {
        this.x = x;
    },
    moveY : function(y) {
        this.y = y;
    }
}
```

“其他任何角色都可以通过继承的方式来实现这些方法，比如让人继承移动方法。”

```
var Player = function(x, y, c) {
```

```

    this.x = x;
    this.y = y;
    this.color = c;
}
Player.prototype = FlyWeight;
Player.prototype.changeC = function(c) {
    this.color = c;
}

```

“让精灵继承移动的方法。”

```

var Spirit = function(x, y, r) {
    this.x = x;
    this.y = y;
    this.r = r;
}
Spirit.prototype = FlyWeight;
Spirit.prototype.changeR = function(r) {
    this.r = r;
}

```

“那么我们创建一个人。”

```

var player1 = new Player(5, 6, 'red');
console.log(player1); // Player {x: 5, y: 6, color: "red", moveX: function,
moveY: function...}

```

“让这个人移动起来。”

```

player1.moveX(6);
player1.moveY(7);
player1.changeC('pink');
console.log(player1); // Player {x: 6, y: 7, color: "pink", moveX: function,
moveY: function...}

```

“我们创建一个精灵。”

```

var spirit1 = new Spirit(2, 3, 4);
console.log(spirit1); // Spirit {x: 2, y: 3, r: 4, moveX: function, moveY:
function...}

```

“让精灵移动起来。”

```

spirit1.moveX(3);
spirit1.moveY(4);
spirit1.changeR(5); // Spirit {x: 3, y: 4, r: 5, moveX: function, moveY:
function...}

```

“于是我们可以看出，不论是游戏中的人还是游戏中的精灵都拥有了运动这一方法，这样我们就可将本来在人物类以及精灵类中的内部方法（移动方法）提取出来，实现了公用，减少其他类重写时造成的不必要的开销，类似这样的提取在页面中也是很常见的，所以我们要善于观察提取相似可共享的数据与方法来优化我们的应用。”

下章剧透

本章我们通过对翻页功能的优化学习了享元模式。通过享元模式我们可以将共有的数据与方法提取以提高页面效率。不过这么多天来小白跟着团队也开发了不少的页面，每个页面的交互都别具一格，不过在经理眼中却是一种交互不一致的问题，那么对于这类问题小白又会如何解决呢？

忆之获

享元模式的应用目的是为了提高程序的执行效率与系统的性能。因此在大型系统开发中应用是比较广泛的，百分之一的效率提升有时可以发生质的改变。它可以避免程序中的数据重复。有时系统内存在大量对象，会造成大量内存占用，所以应用享元模式来减少内存消耗是很有必要的。不过应用时一定要找准内部状态（数据与方法）与外部状态（数据与方法），这样你才能更合理地提取分离。当然在一些小程序中，性能与内存的消耗对程序的执行影响不大时，强行应用享元模式而引入复杂的代码逻辑，往往回收到负效应。

我问你答

工作中享元模式应用比较多，那么你能否创建几类弹框，然后分析它们中哪些数据结构与方法比较类似？你能否提取出来作为享元对象来优化你的功能？

第四篇

行为型设计模式

行为型设计模式用于不同对象之间职责划分或算法抽象，行为型设计模式不仅仅涉及类和对象，还涉及类或对象之间的交流模式并加以实现。

- 第 16 章 照猫画虎——模板方法模式
- 第 17 章 通信卫星——观察者模式
- 第 18 章 超级玛丽——状态模式
- 第 19 章 活诸葛——策略模式
- 第 20 章 有序车站——职责链模式
- 第 21 章 命令模式
- 第 22 章 驻华大使——访问者模式
- 第 23 章 媒婆——中介者模式
- 第 24 章 做好笔录——备忘录模式
- 第 25 章 点钞机——迭代器模式
- 第 26 章 语言翻译——解释器模式

第 16 章 照猫画虎——模板方法模式

模板方法模式（Template Method）：父类中定义一组操作算法骨架，而将一些实现步骤延迟到子类中，使得子类可以不改变父类的算法结构的同时可重新定义算法中某些实现步骤。

项目经理体验了各个页面中的交互功能，发现每个页面中的弹出框样式都不太一致，有的是高度高一些，有的是字体大了些，有的是按钮歪了些。于是找来小铭，问问是否可以将这些页面中的弹出框归一化。

16.1 提示框归一化

“小铭，”项目经理走过来，“咱们所有页面中的提示框样式能否统一，我发现每个页面的提示框真是千奇百怪，根本不一致，你能不能把他们归一化。”

“没问题，不过我手上还有些事情要做，这样吧，我找小白，让他来完成吧。”小铭于是喊来小白，“小白，你现在是不是手上没什么活？项目经理这边有些需求你来跟进。”

“哦？怎么了？”小白问。

“项目经理说要把咱们所有页面中的提示框统一样式。咱们页面中的弹出框无外乎基本弹出框、包含标题弹出框、左侧确定按钮弹出框、右侧确定按钮弹出框、带有取消按钮弹出框，以及右侧取消按钮弹出框，你去把这些弹出框的样式统一一下吧。”小铭对小白说。

小白听呆了：“可是……我要去一个页面一个页面修改么？”小白迟疑一下。

“当然不是，我的意思是说让你写一个弹出框插件，将这些类弹出框封装好，然后在各个页面重新调用就可以了，这样日后项目经理需求有何变动我们只修改这个插件就可以了。”

“那这个插件中我得要写多少种弹出框呀。”小白嘟囔一句。

“不用写那么多。那么做你就陷入麻烦中了，”小铭听到小白的话说，“你试一试模板方法模式吧。用这个模式解决你的需求那是极好的。”

“模板方法模式？我没用过呀，我该怎么实现呢？”小白不解。

“模板方法模式就是将多个模型抽象化归一，从中抽象提取出来一个最基本的模板，当然这个模块可作为实体对象也可以作为抽象对象，这要看你具体需求了，然后其他模块只需要继承这个模板方法，也可拓展某些方法。”小铭一本正经地解释一遍。

“还是不太明白，你能举个例子么？”

16.2 美味的蛋糕

“很简单的，比如我们生活中用蛋糕模具做蛋糕，做出的蛋糕是外形相同的，因为他们都用的同一个模具，这是最基本的一个蛋糕。当然我们看到商店里卖的蛋糕五花八门，有的涂层奶油，有的抹上巧克力，有的浸入果汁等，这都是对蛋糕的二次加工，也就是说顾客对蛋糕有不同的需求，所以为了满足更多顾客需求，烤制出蛋糕后，我们还要对他们再添加点美味的作料，让他们更美味而满足每个顾客的需求。所以说我们需求解决方案中提到的基本提示框应该就是我们要抽象出来的，因为它是最简单的一个提示框，其他提示框都比这个提示框要多一些功能，也就是说我们要对这个添加一些‘佐料’让其满足用户的需求。比如标题提示框多了一个标题组件，取消按钮提示框多了一个取消按钮组件。但是你首先要实现最基础的提示框。”

16.3 创建基本提示框

“哦，你这么说我有点明白了，我首先要做的就是创建一个基本提示框基类，然后其他提示框类只需要在继承基础上，拓展自己所需即可了吧，这样日后需求再变动我们修改基础类，那么所有提示框类实现的样式都会统一变化了吧？”

“嗯，正是这样。”

“那我先实现基本提示框吧，他有一个提示内容、一个关闭按钮和确定按钮。”

```
// 模板类 基础提示框 data 渲染数据
var Alert = function(data) {
    // 没有数据则返回，防止后面程序执行
    if(!data)
        return;
    // 设置内容
    this.content = data.content;
    // 创建提示框面板
    this.panel = document.createElement('div');
    // 创建提示内容组件
    this.contentNode = document.createElement('p');
    // 创建确定按钮组件
    this.confirmBtn = document.createElement('span');
    // 创建关闭按钮组件
    this.closeBtn = document.createElement('b');
    // 为提示框面板添加类
    this.panel.className = 'alert';
    // 为关闭按钮添加类
    this.closeBtn.className = 'a-close';
    // 为确定按钮添加类
    this.confirmBtn.className = 'a-confirm';
    // 为确定按钮添加文案
    this.confirmBtn.innerHTML = data.confirm || '确认';
}
```

```

    // 为提示内容添加文本
    this.contentNode.innerHTML = this.content;
    // 点击确定按钮执行方法 如果 data 中有 success 方法则为 success 方法，否则为空函数
    this.success = data.success || function() {};
    // 点击关闭按钮执行方法
    this.fail = data.fail || function() {};
}

```

16.4 模板的原型方法

既然这个基本提示框是可创建的，那么它也应该具有一些基本方法，比如应该有 init 方法来组装提示框，bindEvent 方法来绑定点击确定或关闭按钮事件，等等。

```

// 提示框原型方法
Alert.prototype = {
    // 创建方法
    init : function(){
        // 生成提示框
        this.panel.appendChild(this.closeBtn);
        this.panel.appendChild(this.contentNode);
        this.panel.appendChild(this.confirmBtn);
        // 插入页面中
        document.body.appendChild(this.panel);
        // 绑定事件
        this.bindEvent();
        // 显示提示框
        this.show();
    },
    bindEvent : function(){
        var me = this;
        // 关闭按钮点击事件
        this.closeBtn.onclick = function(){
            // 执行关闭取消方法
            me.fail();
            // 隐藏弹层
            me.hide();
        }
        // 确定按钮点击事件
        this.confirmBtn.onclick = function(){
            // 执行关闭确认方法
            me.success();
            // 隐藏弹层
            me.hide();
        }
    },
    // 隐藏弹层方法
    hide : function(){
        this.panel.style.display = 'none';
    },
    // 显示弹层方法
    show : function(){
        this.panel.style.display = 'block';
    }
}

```

16.5

根据模板创建类

“有了这个提示框基类，再想拓展其他类型弹层则容易得多了，比如右侧按钮提示框。”

```
// 右侧按钮提示框
var RightAlert = function(data) {
    // 继承基本提示框构造函数
    Alert.call(this, data);
    // 为确认按钮添加 right 类设置位置居右
    this.confirmBtn.className = this.confirmBtn.className + ' right';
}
// 继承基本提示框方法
RightAlert.prototype = new Alert();
```

“同样道理，实现标题提示框也很简单”

```
// 标题提示框
var TitleAlert = function(data) {
    // 继承基本提示框构造函数
    Alert.call(this, data);
    // 设置标题内容
    this.title = data.title;
    // 创建标题组件
    this.titleNode = document.createElement('h3');
    // 标题组件中写入标题内容
    this.titleNode.innerHTML = this.title;
}
// 继承基本提示框方法
TitleAlert.prototype = new Alert();
// 对基本提示框创建方法拓展
TitleAlert.prototype.init = function() {
    // 插入标题
    this.panel.insertBefore(this.titleNode, this.panel.firstChild);
    // 继承基本提示框 init 方法
    Alert.prototype.init.call(this);
}
```

16.6

继承类也可作为模板类

“在此基础上，如果想创建带有取消按钮的标题提示框也变得很容易，只需要在构造函数中创建一个取消按钮，然后在原型方法的实例化方法 `init` 中加入取消按钮，以及对绑定的事件 `bindEvent` 方法进行拓展即可。这里注意，由于是一种标题提示框，所以带有取消按钮的标题提示框类以标题提示框类作为模板类，继承原型方法更佳。”

```
// 带有取消按钮的弹出框
var CancelAlert = function(data) {
    // 继承标题提示框构造函数
    TitleAlert.call(this, data);
    // 取消按钮文案
```

```

this.cancel = data.cancel;
// 创建取消按钮
this.cancelBtn = document.createElement('span');
// 为取消按钮添加类
this.cancelBtn.className = 'cancel';
// 设置取消按钮内容
this.cancelBtn.innerHTML = this.cancel || '取消';
}
// 继承标题提示框原型方法
CancelAlert.prototype = new Alert();
CancelAlert.prototype.init = function(){
    // 继承标题提示框创建方法
    TitleAlert.prototype.init.call(this);
    // 由于取消按钮要添加在末尾，所以在创建完其他组件后添加
    this.panel.appendChild(this.cancelBtn);
}
CancelAlert.prototype.bindEvent = function(){
    var me = this;
    // 标题提示框绑定事件方法继承
    TitleAlert.prototype.bindEvent.call(me);
    // 取消按钮绑定事件
    this.cancelBtn.onclick = function(){
        // 执行取消回调函数
        me.fail();
        // 隐藏弹层
        me.hide();
    }
}
}

```

16.7 创建一个提示框

“这样实现的提示框不仅功能结构统一，而且日后一旦有什么集体变更的需求，只需要对最基本的提示框类进行修改就可以在所有的提示框中实现。”小铭对小白说，“好了，我们测试一下吧。”

```

new CancelAlert({
    title : '提示标题',
    content : '提示内容',
    success : function(){
        console.log('ok');
    },
    fail : function(){
        console.log('cancel')
    }
}).init();

```

“果然要比我想像中的高级了许多。”小白感叹。

16.8 创建多类导航

“其实模板方法模式不仅仅在我们归一化组件时使用，有时候创建页面时也是很常用的，比如创建三类导航，第一类是基础的，第二类是多了消息提醒功能，第三类多了后面显示网址

功能。我们也可以用模板方法模式实现，此时抽象出来的基本类就是我们这里所说的最简单的基础导航类。”

```
// 格式化字符串方法
function formateString(str, data) {
    return str.replace(/\{#(\w+)\#\}/g, function(match, key){return typeof
data[key] === undefined ? '' : data[key]} );
}

// 基础导航
var Nav = function(data) {
    // 基础导航样式模板
    this.item = '<a href="#href#" title="#title#">#name#</a>';
    // 创建字符串
    this.html = '';
    // 格式化数据
    for(var i = 0, len = data.length; i < len; i++) {
        this.html += formateString(this.item, data[i]);
    }
    // 返回字符串数据
    return this.html;
}
```

“对于消息提醒导航类，我们只需对其额外添加消息提醒组件模板，并用消息提醒组件模板对传入的网址数据进行装饰处理，得到所需要的字符串，再调用从基础导航类中继承的方法处理这些字符串，即可实现消息导航。”

```
// 带有消息提醒信息导航
var NumNav = function(data) {
    // 消息提醒信息组件模板
    var tpl = '<b>#num#</b>';
    // 装饰数据
    for(var i = data.length - 1; i >= 0; i--) {
        data[i].name += data[i].name + formateString(tpl, data[i]);
    }
    // 继承基础导航类，并返回字符串
    return Nav.call(this, data);
}
```

“同样的道理，对于带有网址的导航类来说，对传入的数据通过网址模板装饰，并得到装饰后的模板数据。调用从基础导航类的构造函数中继承的方法处理装饰后的模板数据，即可实现网址导航。”

```
// 带有链接地址的导航
var LinkNav = function(data) {
    // 链接地址模板
    var tpl = '<span>#link#</span>';
    // 装饰数据
    for(var i = data.length - 1; i >= 0; i--) {
        data[i].name += data[i].name + formateString(tpl, data[i]);
    }
    // 继承基础导航类，并返回字符串
    return Nav.call(this, data);
}
```

16.9 创建导航更容易

“比如我们测试带有消息提醒组件的导航模块。”

```
// 获取导航容器
var nav = document.getElementById('content');
// 添加内容
nav.innerHTML = NumNav([
  {
    href : 'http://www.baidu.com/',
    title : '百度一下，你就知道',
    name : '百度',
    num : '10'
  },
  {
    href : 'http://www.taobao.com/',
    title : '淘宝商城',
    name : '淘宝',
    num : '2'
  },
  {
    href : 'http://www.qq.com/',
    title : '腾讯首页',
    name : '腾讯',
    num : '3'
  }
])
```

下章剧透

本章通过对如何更合理地创建弹框进行探讨，而深入学习了模板方法模式，明天小白将与团队中的其他人合作开发项目，看看他又会遇到什么问题呢？

忆之获

模板方法的核心在于对方法的重用，它将核心方法封装在基类中，让子类继承基类的方法，实现基类方法的共享，达到方法共用。当然这种设计模式也将导致基类控制子类必须遵守某些法则。这是一种行为的约束。当然为了让行为的约束更可靠，基类中封装的方法通常是不变的算法，或者具有稳定的调用方式。

子类继承的方法亦是可以扩展的，这就要求对基类继承的方法进行重写。当然为了更好地实践，我们通常要控制这种拓展，这样才能让基类对子类有更稳健的束缚力。然而子类对自身私有行为的拓展还是很有必要的。

我问你答

根据文中的案例，如果让你实现右侧取消按钮提示框，想一想，该怎么实现？

第17章 通信卫星——观察者模式

观察者模式（Observer）：又被称作发布-订阅者模式或消息机制，定义了一种依赖关系，解决了主体对象与观察者之间功能的耦合。

时间过得真快，转眼间小白开始了团队代码开发。这样，需求的研发中经常遇到一个人负责一个模块，可是每个人负责的模块之间该如何进行信息沟通呢？

17.1 团队开发的坎坷

“小铭，”小白急匆匆走过去，“今天写新闻评论模块，它的需求是这样的，当用户发布评论时，会在评论展示模块末尾处追加新的评论，与此同时用户的消息模块的消息数量也会递增。如果用户删除留言区的信息时，用户的消息模块消息数量也会递减，但是今天我浏览了一下这些模块的代码，发现他们是三位不同的工程师写的，都写在自己的独立闭包模块里，现在我要完成我的需求，又不想将他们的模块合并到一起，这样我改动量很大，可是我该如何解决呢？”

“哦，听明白了，你是想实现你的需求而添加一些功能代码，但又不想新添加的代码影响到他人实现的功能。这也就是说，你不想让你自己的模块与他人开发的模块严重耦合在一起吧。对于这类问题，观察者模式是比较理想的解决方案。”

“观察者模式？可以解开我与他们之间的功能耦合么？”小白半信半疑。

17.2 卫星的故事

“嗯，观察者模式也被他人称之为消息机制或者发布-订阅者模式。为了解决主体对象与观察者之间功能的耦合。举个例子，目前每个国家都在研发并发射卫星，那么发射的卫星有什么用呢？”

“是为了监控气象信息，监控城市信息等等吧。”小白回答说。

“很对，为的就是监控。所以发射的卫星就可以看做是一个观察者或者说是一个消息系统。如果让这颗卫星为飞机导航，那么这架飞机就是一个被观察者或者说是一个主体对象。当然主体对象是可以变化的，比如飞机飞翔，就像你的需求中的消息一样，消息的内容每时每刻都可

能变化。因此飞机经常会发出位置消息，比如从沈阳到香港，途中经过北京，那么当经过北京上空时会向卫星发射一则消息来指明自己所处位置。卫星接受这些消息后就能确认这架飞机的具体位置。”小铭接着说，“当然卫星仅仅处理这些事情是不够的，因为只有当前这架飞机和卫星知道该飞机的位置信息，而地面上的中转站目前还是不知道的。为了让地面上的中转站知道飞机的运行情况，并且各地的飞机中转站可以根据接收到的飞机信息而做相应的处理。于是卫星、飞机、中转站的问题模型可简化成这样，中转站与天空中的飞机要想知道某架飞机的运行情况，各地中转站都要在卫星上注册这架飞机的信息，以便接收到这架飞机的信息。于是每当飞机到达一个地方时，都会向卫星发出位置信息，然后卫星又将该信息广播到已经订阅过这架飞机的中转站。这样每个中转站便可以接收飞机的消息并做相应的处理来避免飞机事故发生。”

17.3 创建一个观察者

“听你这么一说我自己明白些。按你说的，把观察者或者消息系统看作一个对象，那么它应该包括两个方法吧，第一个是接收某架飞机发来的消息，第二个是向订阅过该飞机的中转站发送相应的消息吧。”

“嗯，不过，小白别忘了，并不是每个中转站时时都要监控飞机的，比如飞机路过石家庄，那么它已经不再北京了，那么北京中转站就没有必要再继续监控飞机状态了，所以此时北京中转站就应该注销掉飞机之前注册的信息，因此我们还需要有一个取消注册的方法。当然不要忘记对于这些消息，我们还需要有一个保存消息的容器，所以我们还需要一个消息容器。我们分析到这里，观察者对象的雏形就出来了。”

小铭接着说：“首先我们需要把观察者对象创建出来，他有一个消息容器，和三个方法，分别是订阅消息方法、取消订阅的消息方法、发送订阅的消息方法。”

```
// 将观察者放在闭包中，当页面加载就立即执行
var Observer = (function() {
    // 防止消息队列暴露而被篡改故将消息容器作为静态私有变量保存
    var __messages = {};
    return {
        // 注册信息接口
        regist : function(){}
        // 发布信息接口
        fire : function(){}
        // 移除信息接口
        remove : function(){}
    }
})();
```

“好了，观察者对象的雏形出来，我们剩下需要做的事情就是一一实现这三个方法了，我们首先实现消息注册方法，注册方法的作用是将订阅者注册的消息推入到消息队列中，因此我们需要接受两个参数：消息类型以及相应的处理动作，在推入到消息队列时如果此消息不存在则应该创建一个该消息类型并将该消息放入消息队列中，如果此消息存在则应该将消息执行方

法推入该消息对应的执行方法队列中，这么做的目的也是保证多个模块注册同一则消息时能顺利执行。”

```
register : function(type, fn) {
    // 如果此消息不存在则应该创建一个该消息类型
    if(typeof messages[type] === 'undefined') {
        // 将动作推入到该消息对应的动作执行队列中
        messages[type] = [fn];
    } else {
        // 将动作方法推入该消息对应的动作执行序列中
        __messages[type].push(fn);
    }
}
```

“对于发布消息方法，其功能是当观察者发布一个消息时将所有订阅者订阅的消息一次执行。故应接收两个参数，消息类型以及动作执行时需要传递的参数，当然在这里消息类型是必须的。在执行消息动作队列之前校验消息的存在是很有必要的。然后遍历消息执行方法队列，并依次执行。然后将消息类别以及传递的参数打包后依次传入消息执行方法中。”

```
fire : function(type, args) {
    // 如果该消息没有被注册，则返回
    if(!__messages[type])
        return;
    // 定义消息信息
    var events = {
        type : type, // 消息类型
        args : args || {} // 消息携带数据
    },
    i = 0, // 消息动作循环变量
    len = __messages[type].length; // 消息动作长度
    // 遍历消息动作
    for(; i < len; i++) {
        // 依次执行注册的消息对应的动作序列
        __messages[type][i].call(this, events);
    }
}
```

“最后是消息注销方法，其功能是将订阅者注销的消息从消息队列中清除，因此我们也需要两个参数，即消息类型以及执行的某一动作。当然为了避免删除消息动作时消息不存在情况的出现，对消息队列中消息的存在性校验也是很有必要的。”

```
remove : function(type, fn) {
    // 如果消息动作队列存在
    if(__messages[type] instanceof Array) {
        // 从最后一个消息动作遍历
        var i = __messages[type].length - 1;
        for(; i >= 0; i--) {
            // 如果存在该动作则在消息动作序列中移除相应动作
            if(__messages[type][i] === fn && __messages[type].splice(i, 1));
        }
    }
}
```

```

    }
}
```

17.4 拉出来溜溜

“观察者对象或者说消息系统创建成功之后，我们先简单测试一下，首先我们订阅一条消息。”

```

Observer.register('test', function(e) {
  console.log(e.type, e.args.msg);
});
```

“然后我们发布这则消息。”

```
Observer.fire('test', {msg: '传递参数'}); // test 传递参数
```

“测试已经实现了我们预期的效果，”小铭接着说，“小白，不过你可能还不太清楚观察者模式的作用，以及它是如何实现解耦的。我们先穿越回到我们最开始提出问题的时候，我们之前的需求实现出来，来亲身体会一下观察者模式。”

17.5 使用前的思考

“好呀，我也正为不同的工程师将自己的功能代码写在不同的闭包模块中导致无法相互调用的问题所困扰着呢。”

“你说的没有错，同一个模块内的功能理应放在一起，这样管理起来才会更加方便。所以对于追加留言的功能就应放在之前开发过的留言模块里；对于用户消息递增功能则应当放在之前开发过的用户信息模块里。对于留言的提交，理应放在你写的提交模块里。不过，既然我们选择用观察者模式来解决问题，首先就要分析哪些模块应该注册消息，哪些模块应该发布消息，这一点是很重要的。”小铭接着说：“小白，你想一想应该如何分类呢？”

“嗯……，发布留言与删除留言功能需求是用户主动触发，所以应该是观察者发布消息，而评论的追加以及用户消息的增减是被动触发的，所以它们应该是订阅者去注册消息。”小白接着说，“这样的话用户信息模块既是信息的发送者也是信息的接收者，提交模块是信息的发送者，浏览模块是信息的接收者”。

17.6 大显身手

“没有错，根据你的分析，我们一次把它们实现吧。”小铭接着说，“我们先实现消息注册功能的两个模块。首先，用户追加评论的功能的实现应该是这样的。”

```

// 外观模式 简化获取元素
function $(id) {
```

```
        return document.getElementById(id);
    }
    // 工程师 A
    (function() {
        // 追加一则消息
        function addMsgItem(e) {
            var text = e.args.text, // 获取消息中用户添加的文本内容
                ul = $('msg'), // 留言容器元素
                li = document.createElement('li'), // 创建内容容器元素
                span = document.createElement('span');// 删除按钮
            li.innerHTML = text; // 写入评论
            li.onclick = function() {
                ul.removeChild(li); // 移除留言
                // 发布删除留言消息
                Observer.fire('removeCommentMessage', {
                    num : -1
                });
            }
            // 添加删除按钮
            li.appendChild(span);
            // 添加留言节点
            ul.appendChild(li);
        }
        // 注册添加评论信息
        Observer.regist('addCommentMessage', addMsgItem);
    })();
}
```

“实现递增用户信息功能也很容易，只需要在原信息数目基础上加一即可。”

```
// 工程师 B
(function() {
    // 更改用户消息数目
    function changeMsgNum(e) {
        // 获取需要增加的用户消息数目
        var num = e.args.num;
        // 增加用户消息数目并写入页面中
        $('msg_num').innerHTML = parseInt($('msg_num').innerHTML) + num;
    }
    // 注册添加评论信息
    Observer
        .regist('addCommentMessage', changeMsgNum)
        .regist('removeCommentMessage', changeMsgNum);
})();
```

“最后对于一个用户来说，当他提交信息时，就要触发消息发布功能。”

```
// 工程师 C
(function() {
    // 用户点击提交按钮
    $('#user_submit').onclick = function() {
        // 获取用户输入框中输入的信息
        var text = $('#user_input');
        // 如果消息为空则提交失败
        if(text.value === '') {
```

```

        return;
    }
    // 发布一则评论消息
    Observer.fire('addCommentMessage', {
        text : text.value,      // 消息评论内容
        num : 1                // 消息评论数目
    });
    text.value = '';          // 将输入框置为空
}
})();

```

“好了，刷新页面看看我们的成果吧（如图 17-1 所示）。”

雨夜清荷 等级 9 消息 4

最新消息发布

[添加第一条评论](#)

[添加第二条评论](#)

[添加第三条评论](#)

[添加第四条评论](#)

▲图 17-1 消息发布

“这个模式真的太神奇了”。小白惊呼：“各个模块之间耦合问题就这么简单地解决了，属于哪个模块的功能方法都可以写在原模块，一点也不用担心其他模块是怎么实现的，只需要收发消息即可。”此时小白眼睛一转想出一个问题：“小铭，观察者模式能不能解决类或对象之间的耦合呢？”

17.7

对象间解耦

小铭笑了笑说：“当然可以了，给你举一个简单的例子吧，比如你在来公司前还在学校上学，那么在课堂上有学生和老师。我们姑且就课堂老师提问学生的例子说明一下吧。”小铭接着说：“我们首先创建学生类，学生是被提问对象，因此他们是订阅者。同时学生也有对问题的思考结果，以及回答问题的动作。”

```

// 学生类
var Student = function(result){
    var that = this;
    // 学生回答结果
    that.result = result;
    // 学生回答问题动作
    that.say = function(){
        console.log(that.result);
    }
}

```

};

“当然在课堂上学生是可以回答问题的，所以他们有回答问题的方法 answer。”

```
// 回答问题方法
Student.prototype.answer = function(question) {
    // 注册参数问题
    Observer.regist(question, this.say);
}
```

“还有一类学生在课堂上呼呼地睡着了，此时他们就不能回答问题了，所以呢，他们也要有一个睡觉方法 sleep。”

```
// 学生呼呼睡觉，此时不能回答问题
Student.prototype.sleep = function(question) {
    console.log(this.result + ' ' + question + ' 已被注销')
    // 取消对老师问题的监听
    Observer.remove(question, this.say)
}
```

“学生类创建完了，那么我们创建一个教师类，对于教师，他会提问学生，所以他是一个观察者。所以他需要有一个提问问题的方法。”

```
// 教师类
var Teacher = function() {};
// 教师提问问题的方法
Teacher.prototype.ask = function(question) {
    console.log('问题是：' + question);
    // 发布提问消息
    Observer.fire(question)
}
```

17.8 课堂演练

“学生类与教师类创建完成了，接下来我们模拟创建出三位听课的学生吧。”

```
var student1 = new Student('学生 1 回答问题'),
    student2 = new Student('学生 2 回答问题'),
    student3 = new Student('学生 3 回答问题');
```

“然后这三位同学订阅（监听）了老师提问的两个问题。”

```
student1.answer('什么是设计模式');
student1.answer('简述观察者模式');
student2.answer('什么是设计模式');
student3.answer('什么是设计模式');
student3.answer('简述观察者模式');
```

“后来第三位同学睡着了，所以对于订阅的第二个问题‘简述观察者模式’消息就注销了。”

```
student3.sleep('简述观察者模式');
```

“接下来我们创建一个教师类。”

```
var teacher = new Teacher();
```

“提问两个问题。”

```
teacher.ask('什么是设计模式');
teacher.ask('简述观察者模式');
```

“好了，我们观察一下浏览器中输出的结果吧。”

```
// 学生 3 回答问题 简述观察者模式 已被注销
// 问题是：什么是设计模式
// 学生 1 回答问题
// 学生 2 回答问题
// 学生 3 回答问题
// 问题是：简述观察者模式
// 学生 1 回答问题
```

下章剧透

通过对观察者模式的学习，我们可以解决团队开发中的最重要的模块间通信问题，这是模块间解耦的一种可行方案。条件判断语句是一种很常见又很简单的语句，然而开发中即使再简单的语句有时也会隐含问题，下一章我们将看看这种简单的条件判断会存在哪些问题呢？

忆之获

观察者模式最主要的作用是解决类或对象之间的耦合，解耦两个相互依赖的对象，使其依赖于观察者的消息机制。这样对于任意一个订阅者对象来说，其他订阅者对象的改变不会影响到自身。对于每一个订阅者来说，其自身既可以是消息的发出者也可以是消息的执行者，这都依赖于调用观察者对象的三种方法（订阅消息，注销消息，发布消息）中的哪一种。

团队开发中，尤其是大型项目的模块化开发中，一位工程师很难做到熟知项目中的每个模块，此时为完成一个涉及多模块调用的需求，观察者模式的优势就显而易见了，模块间的信息传递不必要相互引用其他模块，只需要通过观察者模式注册或者发布消息即可。通过观察者模式，工程师间对功能的开发只需要按照给定的消息格式开发各自功能即可，而不必去担忧他人的模块。

我问你答

有人说观察者模式也是一种自定义事件，那么对比 DOM 编程中的事件，说一说它们的异同点。

第 18 章 超级玛丽——状态模式

状态模式 (State): 当一个对象的内部状态发生改变时，会导致其行为的改变，这看起来像是改变了对象。

月底公司又要开展月末最美图片评选活动。不过今年的图片评选活动又会有哪些结果呢？

18.1 最美图片

“小白，下周我们要开展一个投票征集活动，让网友投票选出我们本月最美的图片。根据网友的投票，每张图片有以下几种结果……”项目经理对小白说。

小白心想：“实现这样的需求我要做多少分支判断呀，如果我将所有图片的结果展示用一个函数封装，内部也不免有多个分支判断。”

```
// 展示结果
function showResult(result){
    if(result == 0){
        // 处理结果 0
    }else if(result == 1){
        // 处理结果 1
    }else if(result == 2){
        // 处理结果 2
    }else if(result == 3){
        // 处理结果 3
    }
}
```

18.2 分支判断的思考

“如果项目经理哪天心血来潮，想增删结果，我的工作岂不是要悲剧了。还是问问小铭吧，看看他有什么好办法。”于是小白去找小铭。

“小铭，有什么办法可以减少代码中的条件判断语句么？并且使每种判断情况独立存在，这样更方便管理。”小白将上面的案例需求展示给小铭。

“对于这类分支条件内部独立结果的管理，我想状态模式应该会很适合，每一种条件作为

对象内部的一种状态，面对不同判断结果，它其实就是选择对象内的一种状态。”

“结果？对象内部的状态？你都把我说蒙了，你举例说一下吧。”

18.3 状态对象的实现

“对于我们这个简单的例子，我们可以将不同的判断结果封装在状态对象内，然后该状态对象返回一个可被调用的接口方法，用于调用状态对象内部某种方法。比如你可以像下面这样做。”

```
// 投票结果状态对象
var Resul1State = function(){
    // 判断结果保存在内部状态中
    var States = {
        // 每种状态作为一种独立方法保存
        state0 : function(){
            // 处理结果 0
            console.log('这是第一种情况')
        },
        state1 : function(){
            // 处理结果 1
            console.log('这是第二种情况')
        },
        state2 : function(){
            // 处理结果 2
            console.log('这是第三种情况')
        },
        state3 : function(){
            // 处理结果 3
            console.log('这是第四种情况')
        }
    }
    // 获取某一种状态并执行其对应的方法
    function show(result){
        States['state' + result] && States['state' + result]()
    }
    return {
        // 返回调用状态方法接口
        show : show
    }
}();
```

18.4 状态对象演练

“那么我们想调用第三种结果，我们就可以按照下面这种方式来实现。”

```
// 展示结果 3
Resul1State.show(3);
```

“上面的简单案例展示了状态模式的基本雏形”。小铭接着说，“对于状态模式，主要目的就是将条件判断的不同结果转化为状态对象的内部状态，既然是状态对象的内部状态，所以一般

作为状态对象内部的私有变量，然后提供一个能够调用状态对象内部状态的接口方法对象。这样当我们需要增加、修改、调用、删除某种状态方法时就会很容易，也方便了我们对状态对象中内部状态的管理。”

“听起来很不错，不过我要如何实现你说的这些呢”小白好奇地问。

18.5 超级玛丽

“给你举个例子吧，还记得童年时，我们玩的超级玛丽游戏吧，玛丽要吃蘑菇，那么他就会跳起，顶出墙壁里面的蘑菇；玛丽想到悬崖的另一边，它就要跳起；玛丽想避免被前面的乌龟咬到，它就要开枪将其打掉；前方飞过炮弹，玛丽要蹲下躲避；时间不够了，玛丽要加速奔跑……”一口气说了这么多，小铭停顿一下，接着说。

“跳跃，开枪，蹲下，奔跑等，这些都是一个一个状态，如果我们用 if 或者 switch 条件判断语句写的代码一听就不靠谱，而且也是很难维护的，因为增加或者删除一个状态需要改动的地方太多了。此时使用状态模式就再好不过了。对于玛丽，有的时候它需要跳跃开枪，有的时候它需要蹲下开枪，有的时候适合需要奔跑开枪等，如果这些组合状态用 if 或者 switch 分支判断去实现，无形中增加的成本是无法想象的。举例如下。”

```
// 单动作条件判断 每增加一个动作就需要添加一个判断
var lastAction = '';
function changeMarry(action) {
    if(action == 'jump'){
        // 跳跃动作
    }else if(action == 'move'){
        // 移动动作
    }else{
        // 默认情况
    }
    lastAction = action;
}

// 复合动作对条件判断的开销是翻倍的
var lastAction1 = '';
var lastAction2 = '';
function changeMarry(action1, action2) {
    if(action1 == 'shoot'){
        // 射击
    }else if(action1 == 'jump'){
        // 跳跃
    }else if(action1 == 'move' && action2 == 'shoot'){
        // 移动中射击
    }else if(action1 == 'jump' && action2 == 'shoot'){
        // 跳跃中射击
    }
    // 保留上一个动作
    lastAction1 = action1 || '';
    lastAction2 = action2 || '';
}
```

18.6 状态的优化

“小白，你看，即使判断语句实现了我们的需求，其代码结构、代码的可读性以及代码的可维护性都是很糟糕的，这样日后对新动作的添加或者原有动作的修改的成本都是很大的，甚至会影响到其他动作。所以为解决这一类问题我们可以使用状态模式。”小铭接着说，“具体的解决问题的思路是这样的，首先创建一个状态对象，内部保存状态变量，然后内部封装好每种动作对应的状态，最后状态对象返回一个接口对象，它可以对内部的状态修改或者调用。”

```
// 创建超级玛丽状态类
var MarryState = function() {
    // 内部状态私有变量
    var _currentState = {},
        // 动作与状态方法映射
        states = {
            jump : function() {
                // 跳跃
                console.log('jump');
            },
            move : function() {
                // 移动
                console.log('move');
            },
            shoot : function() {
                // 射击
                console.log('shoot');
            },
            squat : function() {
                // 蹲下
                console.log('squat');
            }
        };
    // 动作控制类
    var Action = {
        // 改变状态方法
        changeState : function(){
            // 组合动作通过传递多个参数实现
            var arg = arguments;
            // 重置内部状态
            currentState = {};
            // 如果有动作则添加动作
            if(arg.length){
                // 遍历动作
                for(var i = 0, len = arg.length; i < len; i++){
                    // 向内部状态中添加动作
                    _currentState[arg[i]] = true;
                }
            }
            // 返回动作控制类
            return this;
        },
    };
}
```

```

    // 执行动作
    goes : function(){
        console.log('触发一次动作');
        // 遍历内部状态保存的动作
        for(var i in currentState){
            // 如果该动作存在则执行
            states[i] && states[i]();
        }
        return this;
    }
}
// 返回接口方法 change、gose
return {
    change : Action.changeState,
    goes : Action.goes
}
}

```

18.7 两种使用方式

“我们的超级玛丽状态类顺利创建出来，当我们想使用它的时候很容易。我们有两种方式，如果你喜欢函数方式，我们可以直接执行这个状态类，但是这只能由你自己使用，如果别人使用的时候就可能会修改状态类内部状态。”

```

MarryState()
.change('jump', 'shoot')           // 添加跳跃与设计动作
.goes()                            // 执行动作
.goes()                            // 执行动作
.change('shoot')                  // 添加射击动作
.goes();                           // 执行动作

```

“当然为了更安全我们还是实例化一下这个状态类，这样我们使用的是对状态类的一个复制品，这样无论你怎么使用，都可以放心了。”

```

// 创建一个超级玛丽
var marry = new MarryState();
marry
.change('jump', 'shoot')           // 添加跳跃与设计动作
.goes()                            // 执行动作
.goes()                            // 执行动作
.change('shoot')                  // 添加射击动作
.goes();                           // 执行动作
输出结果:
// 触发一次动作
// jump
// shoot
// 触发一次动作
// jump
// shoot
// 触发一次动作
// shoot

```

“思路很清晰呀”，小白惊叹，“我们改变了状态类的一个状态，就改变了状态对象的执行结果，看起来好像变了一个对象似的。”

下章剧透

通过对状态模式的学习，我们可以对条件判断中的每一种情况独立管理，解决条件分支之间的耦合问题。下一章我们将学习一种与状态模式很相似的模式，究竟是什么模式？我们拭目以待。

忆之获

状态模式既是解决程序中臃肿的分支判断语句问题，将每个分支转化为一种状态独立出来，方便每种状态的管理又不至于每次执行时遍历所有分支。在程序中到底产出哪种行为结果，决定于选择哪种状态，而选择何种状态又是在程序运行时决定的。当然状态模式最终的目的即是简化分支判断流程。

我问你答

为何说状态模式简化了分支判断遍历逻辑。

第 19 章 活诸葛——策略模式

策略模式 (Strategy): 将定义的一组算法封装起来，使其相互之间可以替换。封装的算法具有一定独立性，不会随客户端变化而变化。

每年年底，公司商品展销页都要开展大促销活动，这不项目经理又在策划一次商品喷血打折大促销活动呢……

19.1 商品促销

“小白，快到年底了，咱们的商品拍卖页要准备办一些活动来减轻库存压力。在圣诞节，一部分商品 5 折出售，一部分商品 8 折出售，一部分商品 9 折出售，等到元旦，我们要搞个幸运反馈活动，普通用户满 100 返 30，高级 VIP 用户满 100 返 50……”

“促销就促销呗，干嘛弄这么多情况，我这要一个一个写，得工作到啥时候？”小白心想，“难道我还要一个一个地实现每一种促销策略？”

```
// 100 返 30
function return30(price){}
// 100 返 50
function return50(price){}
// 9 折
function percent90(price){}
// .....
```

忽然间小白脑海中闪现上次学习的状态模式，它不就是用来处理多种分支判断的么？可又一想，对于圣诞节或者元旦，当天的一种商品只会有一种促销策略，而不用去关心其他促销状态，如果采用状态模式那么就会为每一个商品创建一个状态对象，这么做是不是就冗余了呢？于是找到小铭，将自己遇到的问题以及想法诉说一遍。

“你说得很对。对于一种商品的促销策略只用一种情况，而不需要其他促销策略，此时采用策略模式会更合理。”小铭说。

19.2 活诸葛

“策略模式？也是一种处理多种分支判断的模式么？”小白问。

“结构上看，它与状态模式很像，也是在内部封装一个对象，然后通过返回的接口对象实现对内部对象的调用，不同点是，策略模式不需要管理状态、状态间没有依赖关系、策略之间可以相互替换、在策略对象内部保存的是相互独立的一些算法。它就像一位活诸葛，对于一件事情的处理，总有千万种计谋，每次都可以随心所欲地选择一种计谋来达到不同种结果。所以你也可以将你的促销策略放在活诸葛的心中，让他来帮你解决问题。”

“为实现对每种商品的策略调用。你首先要将这些算法封装在一个策略对象内，然后对每种商品的策略调用时，直接对策略对象中的算法调用即可，而策略算法又独立地分装在策略对象内。为方便我们的管理与使用，我们需要返回一个调用接口对象来实现对策略算法的调用。”

19.3 策略对象

```
// 价格策略对象
var PriceStrategy = function() {
    // 内部算法对象
    var stragtegy = {
        // 100 返 30
        return30 : function(price) {
            // parseInt 可通过~~、|等运算符替换，要注意此时 price 要在 [-2147483648,
            // 2147483647] 之间
            // +price 转化为数字类型
            return +price + parseInt(price / 100) * 30;
        },
        // 100 返 50
        return50 : function(price) {
            return +price + parseInt(price / 100) * 50;
        },
        // 9 折
        percent90 : function(price) {
            // JavaScript 在处理小数乘除法有 bug，故运算前转化为整数
            return price * 100 * 90 / 10000;
        },
        // 8 折
        parcent80 : function(price) {
            return price * 100 * 80 / 10000;
        },
        // 5 折
        percent50 : function() {
            return price * 100 * 50 / 10000;
        }
    }
    // 策略算法调用接口
    return function(algorithm, price) {
        // 如果算法存在，则调用算法，否则返回 false
        return stragtegy[algorithm] && stragtegy[algorithm](price)
    }
}();
```

19.4 诸葛亮奇谋

“我们的活诸葛（策略对象）已经创建出来。接下来让活诸葛为我们奉献一种奇谋吧。当

然我们也要告诉这位活诸葛我们需要的那类算法以及要处理的是哪些东西。”

```
var price = PriceStrategy('return50', '314.67');
console.log(price); // 464.67
```

“策略模式使我们在外部看不到算法的具体实现。”小白接着说，“这是不是就是说我们只关注算法的实现结果，不需要了解算法的实现过程呢？如果真是这样，日后除夕的活动促销策略就不用操心了，我只需要简简单单通过策略对象的接口方法直接调用内部封装的某种策略算法就可以了。简直方便极了。”

19.5 缓冲函数

“嗯，是这样的。小白。这种模式是很常见的，猜一猜你在使用 jQuery 中哪个模块里也用到了？”小铭问。

“策略算法？嗯，又可以独立替换的，哦，有了。在写 jQuery 动画时的缓冲函数就是运用策略模式实现的吧。例如让一个 div 运动起来，通过对 jQuery 的 animate 动画传入不同运动算法就可以实现两种不同的运动曲线。”

```
$('.div').animate( {width: '200px'}, 1000, 'linear' );
$('.div').animate( {width: '200px'}, 1000, 'swing' );
```

“没错，jQuery 中的缓冲函数就是用策略模式实现的，比如它为我们提供的 linear、swing 的两种曲线就是两种策略算法，当然 jQuery 提供的这类缓冲函数是有限的，不过给你推荐一个 easing.js，它就是利用策略模式实现的一个 jQuery 缓冲动画算法插件，为我们提供了 30 种缓冲函数算法。举例如下。”

```
$('.div').animate( {width: '200px'}, 1000, 'easeOutQuart' );
$('.div').animate( {width: '200px'}, 1000, 'easeOutBounce' );
```

“这太强了，有了这些美妙的算法就能写出更梦幻的动画了。”小白笑了笑。

19.6 表单验证

“除此之外，还有很多地方用到了策略模式。比如过两天我要写的表单验证模块，验证某一种表单内容。当然验证方法就是一组正则算法。”

```
// 表单正则验证策略对象
var InputStrategy = function() {
    var strategy = {
        // 是否为空
        notNull : function(value) {
            return /\s+/.test(value) ? '请输入内容' : '';
        }
    };
}
```

```

// 是否是一个数字
number : function(value){
    return /^[0-9]+(\.[0-9]+)?$/ .test(value) ? '' : '请输入数字';
},
// 是否是本地电话
phone : function(value){
    return /^(\d{3})-(\d{8})$|^\d{4}-\d{7}$/.test(value) ? '' : '请输入正确的电话号码格式，如：010-12345678 或 0418-1234567';
}
},
return {
    // 验证接口 type 算法 value 表单值
    check : function(type, value) {
        // 去除收尾空白符
        value = value.replace(/^\s+|\s+$/g, '');
        return strategy[type] ? strategy[type](value) : '没有该类型的检测方法';
    },
    // 添加策略
    addStrategy : function(type, fn) {
        strategy[type] = fn;
    }
}();

```

19.7 算法拓展

“不知道你有没有发现，这个例子与上面的竞价活动的例子有一个小小的区别，它在最后返回的接口中添加了一个添加策略接口。因为已有的策略即使再多，有时候也是不能满足其他工程师的需求的。此时我们为增强策略对象的拓展性，我们增加了一个为策略对象添加策略算法的接口，这样我们如果有新的策略算法只需要通过 `addStrategy` 方法即可实现策略算法的添加，而不用修改策略对象内部的代码。”小铭接着说，“那么我们想增加一个验证‘2014-12-12’日期的策略算法怎么办？”

```

// 拓展 可以延伸算法
InputStrategy.addStrategy('nickname', function(value){
    return /^[a-zA-Z]\w{3,7}$/.test(value) ? '' : '请输入 4-8 位昵称，如：YYQH';
});

```

19.8 算法调用

“如此一来我们就将新的日期验证算法添加进来了，当我们想验证某个表单填写的某类信息是否正确。是很容易的。”

```

// 外观模式简化元素的获取
function $tag(tag, context) {
    context = context || document;
    return context.getElementsByTagName(tag);
}

```

```
// 提交按钮点击
$tag('input')[1].onclick = function(){
    // 获取输入框内的内容
    var value = $tag('input')[0].value;
    // 获取日期格式验证结果
    $tag('span')[0].innerHTML = InputStrategy.check('nickname', value);
}
```

下章剧透

策略模式使得算法脱离于模块逻辑而独立管理，使我们可以专心研发算法，而不必受模块逻辑所约束。对于一个复杂的模块开发往往会受到很多限制，下一章我们将学习一种分解复杂模块的模式来解决这些限制。

忆之获

策略模式最主要的特色是创建一系列策略算法，每组算法处理的业务都是相同的，只是处理的过程或者处理的结果不一样，所以它们又是可以相互替换的，这样就解决了算法与使用者之间的耦合。在测试层面上讲，由于每组算法相互之间的独立性，该模式更方便于对每组算法进行单元测试，保证算法的质量。

对于策略模式的优点可以归纳为3点，第一，策略模式封装了一组代码簇，并且封装的代码相互之间独立，便于对算法的重复引用，提高了算法的复用率。第二，策略模式与继承相比，在类的继承中继承的方法是被封装在类中，因此当需求很多算法时，就不得不创建出多种类，这样会导致算法与算法的使用者耦合在一起，不利于算法的独立演化，并且在类的外部改变类的算法难度也是极大的。第三，同状态模式一样，策略模式也是一种优化分支判断语句的模式，采用策略模式对算法封装使得算法更利于维护。

当然策略模式也有其自身的缺点。由于选择哪种算法的决定权在用户，所以对用户来说就必须了解每种算法的实现。这就增加了用户对策略对象的使用成本。其次，由于每种算法间相互独立，这样对于一些复杂的算法处理相同逻辑的部分无法实现共享，这就会造成一些资源的浪费。当然你可以通过享元模式（第十三章）来解决。

对于分支语句的优化，目前为止我们已经学习了3种模式，分别为工厂方法模式，状态模式与策略模式。对于工厂方法模式来说，它是一种创建型模式，他的最终目的是创建对象。而状态模式与策略模式都是行为性模式，不过在状态模式中，其核心是对状态的控制来决定表现行为，所以状态之间通常是不能相互替代的，否则将产生不同的行为结果。而策略模式核心是算法，由于每种算法要处理的业务逻辑相同，因此他们可以相互替换，当然策略模式并不关心使用者环境，因为同一种策略模式最终产出的结果是一定的。

我问你答

你的实际项目中有时候也会遇到表单验证吧？那么使用策略模式完成你的表单验证，体验一下它的独特之处吧。

第20章 有序车站——职责链模式

职责链模式（Chain of Responsibility）：解决请求的发送者与请求的接受者之间的耦合，通过职责链上的多个对象对分解请求流程，实现请求在多个对象之间的传递，直到最后一个对象完成请求的处理。

项目经理准备改善页面中的输入验证与输入提示交互体验。如用户在输入框输入信息后，在输入框的下面提示出一些备选项，当用户输入完成后，则要对用户输入的信息进行验证等等，页面中很多模块需要用户提交信息，为增强用户体验，这些输入框大部分需要具备上面两种功能。

20.1 “半成品”需求

“小白，这有一些用户信息提交表单需求，你来完成吧。”项目经理走过来对小明说。

“好嘞，没问题。”小白应道。

“不过，根据我们产品部门的讨论会议，以后很可能要对原有表单交互体验做一些修改”。项目经理补充一句。

小白心想：“这样呀，对于不确定的需求，还是不做了，否则还要改来改去的。即使完成这些表单交互需求，日后还是要改的。”

小铭见状走过来：“小白，怎么了，项目经理交代的任务怎么不去做呢？”

“她给了我一个‘半成品’需求，即使现在做日后还要改，还不如不做。”小白念念有词。

“怎么还是个‘半成品’需求呢？”

“项目经理让我给表单输入框添加事件，做输入提示和输入验证处理，完成这类需求要向服务器端发送请求还要在原有页面中创建其他组件，但是具体输入框有哪些还不确定，现在做也只是白做，日后还是要修改的。”

小铭笑了笑：“小白，你知道么。造成这种原因是因为你把事件源、异步请求、创建组件的逻辑混在一起，你想一气做完，但是需求往往是会变化的，而且你却想把各个模块耦合在一起完成，日后想让其他模块复用代码也很难呀。”

“那你感觉要如何做呢？”小白反问。

20.2 分解需求

“既然你完成一个需求要做这么多的事情，那就把每件事情独立出一个模块对象去处理，这样完整的需求就被分解成一部分一部分相互独立的模块需求，通过这些对象的分工协作，每个对象只做与自己分内的事，无关的事情传到下一个对象中去做，直到需求完成。”小铭接着说，“而且这么做还有一个好处，即便前一个对象不确定也不会影响到其他对象，这样你就可以把自己有限的精力去处理那些已确定的对象。通过这种方式创建的对象还可以进行单元测试，测试每个模块对象对各种情况的处理情况，保证每个组件对象的处理逻辑的安全性。”

“听你这么一说，豁然开朗。分解需求，让整个需求模块化，然后让模块直接分工协作来完成完整的需求。那么我现在首要的任务就是分解需求流程吧。”

“没错，你的粒度越细，那么你单元测试时越有把握越轻松。”

“哦，对于我的需求，有的输入框需要绑定 keyup 事件，有的输入框需要绑定 change 事件，所以绑定事件应当是第一部分，第二部分就是创建 XHR 进行异步请求获取数据，接下来第三部分就是对适配响应数据，将接收到的数据格式化成可处理的形式，最后一部分是向组件创建器中传入相应数据生成组件。”

“嗯，这很好，你已经将需求颗粒化，现在即使项目经理那边的某一模块的需求不确定，你完全可以完成其他几个模块了。”

20.3 第一站——请求模块

“是呀，我先创建一个异步请求模块对象，它应该只做向服务器端发送请求的事情。”

```
/*
 * 异步请求对象（简化版本）
 * 参数 data      请求数据
 * 参数 dealType  响应数据处理对象
 * 参数 dom       事件源
 */
var sendData = function(data, dealType, dom) {
    // XHR 对象 简化版本 IE 另行处理
    var xhr = new XMLHttpRequest(),
        // 请求路径
        url = 'getData.php?mod=userInfo';
    // 请求返回事件
    xhr.onload = function(event) {
        // 请求成功
        if ((xhr.status >= 200 && xhr.status < 300) ||
            xhr.status == 304) {
            dealData(xhr.responseText, dealType, dom);
        } else {
            // 请求失败
        }
    }
}
```

```

};

// 拼接请求字符串
for(var i in data){
    url += '&' + i + '=' + data[i];
}
// 发送异步请求
xhr.open("get", url, true);
xhr.send(null);
}

```

20.4 下一站——响应数据适配模块

“在这个模块对象中没有对返回的结果做任何逻辑处理，直接将得到的结果传入响应数据适配模块对象方法中，所以对于响应数据适配模块也应该处理一件事——适配响应数据（Ajax请求返回的数据）。”小白接着说。

```

/*
 * 处理响应数据
 * 参数 data      响应数据
 * 参数 dealType   响应数据处理对象
 * 参数 dom        事件源
 */
var dealData = function(data, dealType, dom) {
    // 对象 toString 方法简化引用
    var dataType = Object.prototype.toString.call(data);
    // 判断相应数据处理对象
    switch(dealType) {
        // 输入框提示功能
        case 'sug':
            // 如果数据为数组
            if(dataType === "[object Array]") {
                // 创建提示框组件
                return createSug(data, dom);
            }
            // 将响应的对象数据转化为数组
            if(dataType === "[object Object]") {
                var newData = [];
                for(var i in data)
                    newData.push(data[i]);
                // 创建提示框组件
                return createSug(newData, dom);
            }
            // 将响应的其他数据转化为数组
            return createSug([data], dom);
            break;
        case 'validate':
            // 创建校验组件
            return createValidataResult(data, dom);
            break;
    }
}

```

20.5 终点站——创建组件模块

“在响应数据适配模块中，适配了响应的数据，并将适配后的数据传入下一级创建组件模块中。所以创建组件模块应该做的一件事是——根据响应数据创建组件。”

```
/*
 * 创建提示框组件
 * 参数 data 响应适配数据
 * 参数 dom 事件源
 */
var createSug = function(data, dom) {
    var i = 0,
        len = data.length,
        html = '';
    // 拼接每一条提示语句
    for(; i < len; i++) {
        html += '<li>' + data[i] + '</li>';
    }
    // 显示提示框
    dom.parentNode.getElementsByTagName('ul')[0].innerHTML = html;
}
/*
 * 创建校验组件
 * 参数 data 响应适配数据
 * 参数 dom 事件源
 */
var createValidataResult = function(data, dom) {
    // 显示校验结果
    dom.parentNode.getElementsByTagName('span')[0].innerHTML = data;
}
```

20.6 站点检测——单元测试

“小白，没想到这么快你就把需求职责链搭建出来了，不过现在项目经理的需求还没定下来，现在你可以对你写的模块进行单元测试了。”

“单元测试？这需要怎么做？”小白问。

“单元测试就是对你所写程序中的每个独立单元在不同种运行环境下进行的逻辑测试，比如对 dealData 模块进行单元测试，你首先要分析这个对象方法都可能接受哪些类数据，然后你要对传入这些类数据运行的结果进行预测，如果每一次运行的结果跟你预期结果一致，那么你创建的对象方法是安全的。那么现在，你分析 dealData 运行情况，然后模拟数据，测试看看吧。”小铭说。

“好，对于 dealData 这个对象方法它的参数 dealType 可接收两种可能情况，参数 data 可接受 3 种情况，但这 3 种情况是针对 dealType 为 sug 的，所以对于单侧分类只需要 4 组数据即可。”

```
| dealData('用户名不正确', 'validate', input[0]);
```

```

dealData(123, 'sug', input[1]);
dealData(['爱奇艺', '阿里巴巴', '爱漫画'], 'sug', input[1]);
dealData({
  'iqy' : '爱奇艺',
  'albb' : '阿里巴巴',
  'imh' : '爱漫画'
}, 'sug', input[1]);

```

“不过这么测试，将直接调用相应组件的创建方法 `createSug` 和 `createValidataResult`，为了简化测试结果，你可模拟一下测试方法”

```

var createSug = function(data, dom) {
  console.log(data, dom, 'createSug');
}
var createValidataResult = function(data, dom) {
  console.log(data, dom, 'createValidataResult');
}

```

此时小白打开一下浏览器看的测试结果：

```

// 用户名不正确 <input type="text"> createValidataResult
// [123] <input type="text"> "createSug"
// ["爱奇艺", "阿里巴巴", "爱漫画"] <input type="text"> "createSug"
// ["爱奇艺", "阿里巴巴", "爱漫画"] <input type="text"> "createSug"

```

“嗯，结果跟预期的一样，这样的方法在运行时就放心了。”小白对自己说。

20.7 方案确定

“小白，讨论结果确定了。你需要创建两个输入框，第一个要具有验证功能，第二个要具有下拉框提示功能（输入框 `sug` 组件）。”项目经理对小白说。

小白心中暗喜，“这很简单”。

```

var input = document.getElementsByTagName('input');
// 监听内容改变事件做内容校验
input[0].onchange = function(e) {
  sendData({value : input[0].value}, 'validate', input[0]);
}
// 监听键盘事件对内容做提示处理
input[1].onkeydown = function(e){
  sendData({value : input[1].value}, 'sug', input[1]);
}

```

“现在只需要对事件源绑定事件，有了单元测试保证，后面的处理逻辑就不用再操心了。”

下章剧透

对复杂模块分解在一定程度上可以减少开发过程中收到的制约。有时也能提高开发效率。下一章我们将学习一种将分散化的小粒度功能整合成复杂的对象，并通过提供更简单而统一的

API（接口）来管理并简化功能的使用。

忆之获

职责链模式定义了请求的传递方向，通过多个对象对请求的传递，实现一个复杂的逻辑操作。因此职责链模式将负责的需求颗粒化逐一实现每个对象分内的需求，并将请求顺序地传递。对于职责链上的每一个对象来说，它都可能是请求的发起者也可能是请求的接收者。通过这样的方式不仅仅简化原对象的复杂度，而且解决原请求的发起者与原请求的接收者之间的耦合。当然也方便对每个阶段对象进行单元测试。同时对于中途插入的请求，此模式依然使用，并可顺利对请求执行并产出结果。

对于职责链上的每一个对象不一定都能参与请求的传递，有时会造成一丝资源的浪费，并且多个对象参与请求的传递，这在代码调试时增加了调试成本。

我问你答

试着用职责链模式实现如下流程：从服务器端拉取新闻数据，适配并解析新闻数据，创建新闻模块，为新闻模块添加交互，在页面中展现新闻模块。



第 21 章 命令模式

命令模式 (Command): 将请求与实现解耦并封装成独立对象，从而使不同的请求对客户端的实现参数化。

项目经理让小白做个活动页面，平铺式的结构，不过页面的每个模块都有些相似的地方，比如每个预览产品图片区域，都有一行标题，然后标题的下面是产品图片，只是图片的数量与排列不同。

21.1 自由化创建视图

“小铭，有什么方式可以自由地创建视图模块呀。”小白问。

“自由创建的方式？”

“嗯，就是有时候在模块里面想创建一个图片，有时候想创建多张。”

“试试命令模式，通过执行命令语句自由创建图片，这种方式应该就可以满足你的需求了。”

“命令模式，”小白追问，“它的工作原理是什么样的，如何实现呢？”

“命令模式就是将请求模块与实现模块解耦，听到这句话你可能有些朦胧。没关系，详细解释一下你就明白了。命令模式是将创建模块的逻辑封装在一个对象里，这个对象提供一个参数化的请求接口，通过调用这个接口并传递一些参数实现调用命令对象内部中的一些方法。”

小铭接着说，“请求部分很简单，只需要按照给定的参数格式书写指令即可，所以实现部分的封装才是重点，因为它要为请求部分提供所需方法。”

“这样是不是首先我要明确我需要哪些命令呀”小白问。

“明确命令固然重要，但其实质还是说要实现你的哪些需求，以及哪些需求可以命令化，而这些需求往往又是动态的，比如你要创建的图片，可以有一张也可以有多张。所以你需要在命令对象内部合理封装这些处理方法，但是你还要提供一个命令接口，来合理化接受并处理你的命令。好了，你分析一下你的需求，看看哪些是变动的，哪些可以命令化，然后搭出命令对象吧。”

21.2 命令对象

“嗯，既然动态展示不同模块，所以创建元素这一需求就是变化的，因此创建元素方法、展示方法应该被命令化。”

```
// 模块实现模块
var viewCommand = (function() {
    // 方法集合
    var Action = {
        // 创建方法
        create : function(){}
        // 展示方法
        display : function(){}
    }
    // 命令接口
    return function execute(){}
})();
```

21.3 视图创建

“小白，你的命令对象框架既然搭建出来了。那么你就要命令对象中的每一个方法一一实现吧。不过提醒你一下，创建视图过程中如果单纯用 DOM 操作拼凑页面的开销实在有些大，索性格式化字符串模板来创建你的页面吧。不过要注意的是，在实现创建视图方法之前就应该给出页面中每个模块的字符串模板。并且需要有一个格式化字符串模板的方法 `formatString`。”

```
// 模块实现模块
var viewCommand = (function() {
    var tpl = {
        // 展示图片结构模板
        product : [
            '<div>',
            '',
            '<p>#text#</p>',
            '</div>'
        ].join(),
        // 展示标题结构模板
        title : [
            '<div class="title">',
            '<div class="main">',
            '<h2>#title#</h2>',
            '<p>#tips#</p>',
            '</div>',
            '</div>'
        ].join()
    },
    // 格式化字符串缓存字符串
    html = '';
    // 格式化字符串 如: '<div>#content#</div>' 用 {content:'demo'} 替换后可得到字符
```

```

串:'<div>demo</div>'
function formateString(str, obj) {
    // 替换'#{'与'}#'之间的字符串
    return str.replace(/\{\#(\w+)\#\}/g, function(match, key) {
        return obj[key];
    })
}
// 方法集合
var Action = {}
// 命令接口
return function execute() {}
})();

```

“创建模块视图就可通过对模块视图数据进行字符串模板格式化来获取，并封装在 create 方法里。”

```

create : function(data, view) {
    // 解析数据 如果数据是一个数组
    if(data.length) {
        // 遍历数组
        for(var i = 0, len = data.length; i < len; i++) {
            // 将格式化之后的字符串缓存到 html 中
            html += formateString.tpl[view], data[i]);
        }
    }else{
        // 直接格式化字符串缓存到 html 中
        html += formateString.tpl[view], data);
    }
}

```

“在 create 方法 data 参数指定了模块视图数据，view 参数指定了视图模板，用数据去格式化视图模板即可获得视图字符串。不过注意的是传入的 data 参数可能是对象也可能是数组，所以解析并适配传入 data 的数据是很有必要的，如果是数组，一一格式化每个数组成员数据并将结果缓存在 html 中，否则直接格式化传入的 data 数据。”

21.4 视图展示

“嗯，小白，还有一个视图模块展示方法 display 你思考一下，看看如何实现。”

```

// 展示方法
display : function(container, data, view) {
    // 如果传入数据
    if(data) {
        // 根据给定数据创建视图
        this.create(data, view);
    }
    // 展示模块
    document.getElementById(container).innerHTML = html;
    // 展示后清空缓存的字符串
    html = '';
}

```

}

“对于视图模块展示方法 `display`，首先要判断是否传入了数据，如果传入了数据则解析这组数据，生成并缓存视图模块对应的字符串，然后将缓存的模块字符串插入页面中显示模块，最后将展示后的视图模块缓存字符串清空。”

21.5 命令接口

“嗯，创建视图模块与展示视图模块的两个方法你很好地实现了，那么接下来就需要实现命令接口了。对于这个接口的参数应该包括两部分，第一部分是命令对象内部的方法名称，第二部分是命令对象内部方法对应的参数，通过这两个参数你就可以实现自由地对视图模块创建或者展示了。”

```
// 命令接口
return function execute(msg) {
    // 解析命令，如果 msg.param 不是数组则将其转化为数组（apply 方法要求第二个参数为数组）
    msg.param = Object.prototype.toString.call(msg.param) === "[object Array]" ?
        msg.param : [msg.param];
    // Action 内部调用的方法引用 this，所以此处为保证作用域 this 执行传入 Action
    Action[msg.command].apply(Action, msg.param)
}
```

21.6 大功告成

“现在大功告成了，你有没有测试数据，拿过来测试测试我们的命令对象吧。”

“好嘞。”小白拿出项目经理提供的后端传输的数据样品。

```
// 产品展示数据
var productData = [
    {
        src : 'command/02.jpg',
        text : '绽放的桃花'
    },
    {
        src : 'command/03.jpg',
        text : '阳光下的温馨'
    },
    {
        src : 'command/04.jpg',
        text : '镜头前的绿色'
    }
],
// 模块标题数据
titleData = {
    title : '夏日里的一片温馨',
    tips : '暖暖的温情带给人们家的感受。'
};
```

“那么我该如何用呢，小铭。”

“很简单，首先要知道你的指令格式，然后就可以像写一条指令一样，很简单地完成每个模块视图的创建或展示，比如展示一个标题模块”。

```
viewCommand({  
    // 参数说明 方法 display  
    command : 'display',  
    // 参数说明 param1 元素容器 param2 标题数据 param3 元素模板 详见 display 方法  
    param : ['title', titleData, 'title']  
});
```

“你可以创建一个图片。”

```
viewCommand({  
    command : 'create',  
    // 详见 create 方法参数  
    param : [{  
        src : 'command/01.jpg',  
        text : '迎着朝阳的野菊花',  
    }, 'product']  
})
```

“当然你想创建多张图片也很简单。”

```
viewCommand({  
    command : 'display',  
    param : ['product', productData, 'product']  
})
```

“好了刷新你的浏览器，看看你的劳动成果吧（如图 21-1 所示）。”



▲图 21-1 展示效果图

“有了命令模式以后想创建任何页面视图都是一件很简单的事情。”

21.7 绘图命令

“其实命令模式也常用于解耦，”小铭接着说，“比如我们在使用 canvas 的时候经常会调用一些内置方法，但是你就需要不停地使用 canvas 元素的上下文引用，这在多人项目开发中耦合度是比较高的，如果一个人不小心篡改了 canvas 元素的上下文引用，那么后果是无法估计

的。我们通常的做法是将上下文引用对象安全地封装在一个命令对象的内部，如果他人想绘图。直接通过命令对象书写一条命令，即可调用命令对象内部封装的方法来完成需求任务。因此首先你要创建一个命令来实现对象。”

```
// 实现对象
var CanvasCommand = (function(){
    // 获取 canvas
    var canvas = document.getElementById('canvas'),
        // canvas 元素的上下文引用对象缓存在命令对象的内部
        ctx = canvas.getContext('2d');
    // 内部方法对象
    var Action = {
        // 填充色彩
        fillStyle : function(c){
            ctx.fillStyle = c;
        },
        // 填充矩形
        fillRect : function(x, y, width, height){
            ctx.fillRect(x, y, width, height);
        },
        // 描边色彩
        strokeStyle : function(c){
            ctx.strokeStyle = c;
        },
        // 描边矩形
        strokeRect : function(x, y, width, height){
            ctx.strokeRect(x, y, width, height);
        },
        // 填充字体
        fillText : function(text, x, y){
            ctx.fillText(text, x, y)
        },
        // 开启路径
        beginPath : function(){
            ctx.beginPath();
        },
        // 移动画笔触电
        moveTo : function(x, y){
            ctx.moveTo(x, y)
        },
        // 画笔连线
        lineTo : function(x, y){
            ctx.lineTo(x, y);
        },
        // 绘制弧线
        arc : function(x, y, r, begin, end, dir){
            ctx.arc(x, y, r, begin, end, dir);
        },
        // 填充
        fill : function(){
            ctx.fill();
        },
        // 描边
        stroke : function(){}
```

```
        ctx.stroke();
    }
}
return {
    // 命令接口
    excute : function(msg) {
        // 如果没有指令返回
        if(!msg)
            return;
        // 如果命令是一个数组
        if(msg.length){
            // 遍历执行多个命令
            for(var i = 0, len = msg.length; i < len; i++)
                arguments.callee(msg[i]);
        // 执行一个命令
        }else{
            // 如果 msg.param 不是一个数组，将其转化为数组，apply 第二个参数要求格式
            msg.param = Object.prototype.toString.call(msg.param) ===
            "[object Array]" ? msg.param : [msg.param];
            // Action 内部调用的方法可能引用 this，为保证作用域中 this 指向正确，故传
            // 入 Action
            Action[msg.command].apply(Action, msg.param);
        }
    }
})();
}
```

21.8 写一条命令

“有了这个对象，任何人想绘制图形都不需要依赖 canvas 了，只要是按照命令对象结构给出的命令格式，写一条命令即可，比如”

```
// 设置填充色彩为红色，并绘制一个矩形
CanvasCommand.excute([
    {command : 'fillStyle', param : 'red'},
    {command : 'fillRect', param : [20, 20, 100, 100]}
]);
```

“这种格式真的很简洁而且直观明了”小白接着说，“甚至我们不用关心每种方法的具体实现了，即使在不同浏览器实现不一致我们也不用关心，因为那些不兼容的问题都已经在命令对象内部实现了，所以我们根本不用担心了。”

下章剧透

封装功能，并提供简单而高效的 API 是提高团队开发效率的可行方案。命令模式即是如此。有时我们想将创建的函数执行在另一个对象作用域内，以获取并处理该对象内部的数据，那就赶快翻开下一章，看看有没有可行方案。

忆之获

命令模式是将执行的命令封装，解决命令的发起者与命令的执行者之间的耦合。每一条命令实质上是一个操作。命令的使用者不必要了解命令的执行者（命令对象）的命令接口是如何实现的、命令是如何接受的、命令是如何执行的。所有的命令都被存储在命令对象中。

命令模式的优点自然是解决命令使用者之间的耦合。新的命令很容易加入到命令系统中，供使用者使用。命令的使用具有一致性，多数的命令在一定程度上是简化操作方法的使用的。

命令模式是对一些操作的封装，这就造成每执行一次操作都要调用一次命令对象，增加了系统的复杂度。

我问你答

试着使用文中给出的 CanvasCommand 命令对象在页面中绘制一个时钟，体会一下命令模式的特点。

第 22 章 驻华大使——访问者模式

访问者模式（Visitor）：针对于对象结构中的元素，定义在不改变该对象的前提下访问结构中元素的新方法。

随着浏览器的更新换代，Dom 操作中实现的一些操作也焕然一新，这不小白正试着用原生 JavaScript 中 Dom2 级事件为元素添加行为呢。

22.1 设置样式

“小铭，今天我用 DOM2 级事件为页面中的元素绑定了一些事件，在事件中为该元素设置了一些 css 样式，可是在标准浏览器下可以成功，在低版本 IE 下面不成功呀。你知道是什么原因么。”

```
var bindEvent = function(dom, type, fn) {
    if(dom.addEventListener){
        dom.addEventListener(type, fn, false);
    }else if(dom.attachEvent){
        dom.attachEvent('on' + type, fn);
    }else{
        dom['on' + type] = fn;
    }
}
var demo = document.getElementById('demo');
bindEvent(demo, 'click', function(){
    this.style.background = 'red';
});
```

“你看看，低版本 IE 是不是报错了”小铭对小白说。

小白打开浏览器，“真是呀，提示说 ‘this.style 为空或不为对象’。可是这里的 this 指代的不是这个元素么”小白不解。

22.2 自娱自乐的 IE

“W3C 是这么规定的，可是 IE 浏览器总喜欢自娱自乐。他们可不按照标准来。比如 IE 提

供的方法 `attachEvent` 就不是按照 W3C 标准实现的。所以你遇到的问题很简单，在你的事件内部添加一条测试语句 ‘`alert(this === window)`’ 你会发现 IE 中提示的是 `true`，这也就是说我们在 `attachEvent` 事件中 `this` 指向的不是这个元素而是 `window`，所以如果你想获取事件对象，应该通过 `window.e` 来获取的。”

“我的天，IE 怎么可以这样”。

“IE 脾气很倔强，这也是因为当初 IE 浏览器在世界浏览器市场范围内的垄断，所以微软（IE 浏览器厂商）开发什么功能都很任性，从来不顾虑 W3C 的感受。所以也就苦了我们程序员。”小铭接着说，“不过我们可以借用访问者模式的思想来解决这个问题。”

“你是说访问者模式？它的思想可以帮助我们解决事件回调函数中对该元素的访问问题吗？”

22.3 访问操作元素

“正是，访问者模式的思想就是说我们在不改变操作对象的同时，为它添加新的操作方法，来实现对操作对象的访问。就像国家间相互访问，不会让全国人民都去，而且这种成本还是蛮大的，所以派遣一名大使访问便可解决问题。此处我们要为操作元素添加的新操作方法就是事件。可能听上去很复杂，不过没关系，当你看到它的实现时，你会有种恍然大悟的感觉。来看看 IE 的实现方式吧。”

```
function bindIEEvent(dom, type, fn, data) {
    var data = data || {};
    dom.attachEvent('on' + type, function(e) {
        fn.call(dom, e, data);
    });
}
```

“小白，你看，其实实现的核心就是调用了一次 `call` 方法。我们知道 `call` 和 `apply` 的作用就是更改函数执行时的作用域，这正是访问者模式的精髓，通过这两种方法我们就可以让某个对象在其他作用域中运行，比如我们这里让我们的事件源元素对象在我们的事件回调函数中的作用域中运行，那么我们在回调函数访问的 `this` 当然指代的就是我们的事件源元素对象了。”

22.4 事件自定义数据

“哦，不过这里怎么又多了一个参数 `data` 呢？它表示什么意思呀？”

“这是通过访问者模式思想对操作元素绑定的事件而进行的一次拓展，目的是为了向事件回调函数中传入自定义数据。我们知道，W3C 给我们定义事件绑定的回调函数内部只能有一个参数对象供我们访问，它就是例子中的参数 `e`——事件对象，通过它我们能访问到事件相关的一些信息，然而有些时候，这并不能满足我们的需求，比如当用户触发一个点击事件时，想

再给事件回调函数传递一些信息是很困难的。为解决这类问题，有时候我们将数据存储在外面，有时候会将数据绑定在回调函数上等，然而这些方案中存在的存在可被篡改风险，有的做不到对数据的实时更新。所以在事件执行时，将一些必要数据传入到事件回调函数中是很必要的。”

“的确，前几天写评论模块时，点击去评论按钮，要获取以前评论的相关数据，如总评论数等等，总需要做额外的工作。为了能在事件中访问到，将数据保存在页面各处”。

“你知道，call 和 apply 方法在向对象添加访问的方法时是允许我们添加参数的。所以我们顺便将数据也一起传进来，不过你注意到没有，我在使用 call 方法时，是在回调函数内部调用的，所以不要忘记在 call 方法中携带事件对象，这样在回调函数中才能顺利访问到这个事件对象。”

“哦，这么说以后我再为 IE 添加事件时，事件源对象 this、事件对象 e、自定义数据等我们都可以在回调函数中访问到了吧”。

“嗯，你可以测试一下，不过在回调函数中不要忘记声明事件对象和自定义数据。”

```
function $(id){return document.getElementById(id);};
bindIEEvent($('btn'), 'click', function(e, d){
    $('test').innerHTML = e.type + d.text + this.tagName;
}, {text : 'test demo'});
```

小白点击了一下 id 为 btn 的按钮元素，发现 p 元素增加了一行内容：

```
click
test demo
BUTTON
```

“果然是这样。这样绑定的事件真的很实用。”

22.5 原生对象构造器

“其实 JavaScript 中的源生对象构造器就设计成一个访问者，比如在判断某种对象的具体类型时我们可以通过 Object.prototype.toString.call 的方式，判断返回的字符串来确定该数据的类型，如 ‘[object Array]’ 代表的就是一个数组对象。那么我们对这种思想发散一下：我们为对象添加属性数据通常是没有次序的，所以很难找到我们最后一次添加的属性数据，所以如果我们能像处理数组的方式一样来处理一个对象就好了，这样我们创建的类数组对象，可以通过 push 添加数据成员，通过 pop 删除最后一次添加的成员等。”

“还有这么神奇的事？”，小白惊讶，“那是不是我们创建的类数组对象在每次执行的时候都要调用数组的方法呀。”

22.6 对象访问器

“嗯，不过这里我们最好要创建一个访问器，然后将必要的方法封装在内部，这样使用和

管理起来就会更方便。”

```
// 访问器
var Visitor = (function(){
    return {
        // 截取方法
        splice : function(){
            // splice 方法参数，从原参数的第二个参数开始算起
            var args = Array.prototype.splice.call(arguments, 1);
            // 对第一个参数对象执行 splice 方法
            return Array.prototype.splice.apply(arguments[0], args);
        },
        // 追加数据方法
        push : function(){
            // 强化类数组对象，使他拥有 length 属性
            var len = arguments[0].length || 0;
            // 添加的数据从原参数的第二个参数算起
            var args = this.splice(arguments, 1);
            // 校正 length 属性
            arguments[0].length = len + arguments.length - 1;
            // 对第一个参数对象执行 push 方法
            return Array.prototype.push.apply(arguments[0], args);
        },
        // 弹出最后一次添加的元素
        pop : function(){
            // 对第一个参数对象执行 pop 方法
            return Array.prototype.pop.apply(arguments[0]);
        }
    }
})();
```

“对于对象访问器我们简简单单添加了三个方法 splice、push 和 pop。由于他们的原理都是通过对数组对象的源生方法的访问来实现，所以这三个方法都有很多相似之处，比如他们的第一个参数表示要被处理的对象，从第二个参数算起是为数组源生方法需要的参数准备的。而我们将他们封装在访问器 Visitor 中，这样我们在使用时就可以直接通过 Visitor 来使用每一个方法，如果想链式访问（参考第二十五章链模式）只需要对这个访问器每一个方法返回（return）this 即可，如果想在每一个方法中添加一些额外功能，只需要找到这个方法添加，在调用时即可统一实现。”

22.7 操作类数组

“有了这个访问器我就可以像操作数组一样操作一个对象了吧，我去试一下。”

```
var a = new Object();
console.log(a.length);      // undefined
Visitor.push(a, 1, 2, 3, 4);
console.log(a.length);      // 4
Visitor.push(a, 4, 5, 6);
console.log(a);             // Object {0: 1, 1: 2, 2: 3, 3: 4, 4: 4, 5: 5, 6: }
```

```
6, length: 7}
console.log(a.length)           // 7
Visitor.pop(a);
console.log(a);                // Object {0: 1, 1: 2, 2: 3, 3: 4, 4: 4, 5: 5, length:
6}
console.log(a.length);          // 6
Visitor.splice(a, 2);
console.log(a);                // Object {0: 1, 1: 2, length: 2}
```

“创建了一个 a 对象，它是没有 length 属性的，当调用了 push 方法，他就拥有了 length 属性，并且调用每个数组方法 pop、splice 时又会以数组内部的实现方式修改了 length 属性。这样的对象更像是一个数组了，不过类数组在工作中还真是很常见的，比如函数的参数对象 arguments 可以看成一个类数组，它可以像数组那样访问成员，也有 length 属性，还有 jQuery 对象等。”

下章剧透

通过本章的学习我们知道函数对象不仅仅可以工作在自己作用域内，还可以在其他作用域内执行，这种访问模式为我们解决了许多工作中的问题。下一章我们将学习一种更强大的对象，它可以像将军一样，在战场上控制着自己手上的军队。

忆之获

访问者模式解决数据与数据的操作方法之间的耦合，将数据的操作方法独立于数据，使其可以自由化演变。因此访问者更适合于那些数据稳定，但是数据的操作方法易变的环境下。因此当操作环境改变时，可以自由修改操作方法以适应操作环境，而不用修改原数据，实现操作方法的拓展。同时对于同一个数据，它可以被多个访问对象所访问，这极大增加了操作数据的灵活性。

我问你答

借用对象的 `toString` 方法，运用访问者思想，封装一个校验对象类型的方法。

第23章 媒婆——中介者模式

中介者模式（Mediator）：通过中介者对象封装一系列对象之间的交互，使对象之间不再相互引用，降低他们之间的耦合。有时中介者对象也可改变对象之间的交互。

项目经理准备在用户首页上的导航模块添加一个设置层，让用户可以通过设置层，来设置导航展现样式，于是找来小白谈谈新需求。

23.1 导航设置层

“小白，近日一些用户反馈说，咱们首页中为导航添加消息提醒影响他们的视觉体验，说咱们的页面有些乱，甚至有的人认为这是广告入侵。还有一些用户不想在导航中出现链接地址，这也会让他们感觉页面混乱……我们准备要对导航模块添加一个设置层，让用户自由设置导航样式，这里是需求文档，你仔细看看，尽快实现，也好赶上这个月月末同我们的开发新功能一同上线。”项目经理交代了小白一些新需求。

“这有些麻烦了”，小白心想，“好多模块都有导航，我改从何做起？”小白皱起眉头。

“怎么了小白”小铭接水回来路过看见小白若有所思的样子，走向前问。

“这不，项目经理让我为用户首页的导航模块添加一个设置层，来设置各个导航模块里导航的样式，我正发愁呢。”

“这有何难呀。”

“模块太多无从下手呗”。小白答道。“对了，上次说观察者模式可以解决模块之间的耦合，你感觉观察者模式可以么？”

小铭想了一下问：“观察者模式？导航模块里的内容有需要向设置层发送请求的需求？”

“没有，设置层只是单向的控制导航模块内导航的样式”小白解释道。

“要是这样单向通信的，你可以试试中介者模式，我感觉这个模式是比较适合你的需求的。”

“中介者模式？这是一个怎样的模式，为何观察者模式不可以呢？”小白好奇地问。

“首先他们都是通过消息的收发机制实现的，不过在观察者模式中，一个对象既可以是消息的发送者也可以是消息的接收者，他们之间信息交流依托于消息系统实现的解耦。而中介者模式中消息的发送方只有一个，就是中介者对象，而且中介者对象不能订阅消息，只有那些活

跃对象（订阅者）才可订阅中介者的消息，当然你也可以看作是将消息系统封装在中介者对象内部，所以中介者对象只能是消息的发送者。这就像过去男女婚假，总要找媒婆介绍，通过媒婆的沟通，男女才会相识，走到一起，但是没听说过哪位媒婆天天找青年男女帮他介绍对象的”，小铭开个玩笑继续说，“而观察者模式你需要写一个消息系统，那样增加了你的开发成本。所以我建议你用中介者模式。”小铭解释说。

“如果用中介者模式来解决我的需求问题，那么设置层模块对象就应该是一个中介者对象了吧。他负责向各个导航模块对象发送用户设置的消息，而各个导航模块则应该作为消息的订阅者存在吧”。小白思考着说。

“没错，既然你都明白这其中的角色，不妨将其实现，试试看可不可以解决你的需求问题。”小铭建议说。

23.2 创建中介者对象

“好的，记得当初写观察者模式的时候首先实现的是消息系统，那么在这里，消息系统融为中介者对象，那么我应该先实现这个中介者对象吧。”

```
//中介者对象
var Mediator = function(){
    // 消息对象
    var _msg = {};
    return {
        /**
         * 订阅消息方法
         * 参数 type    消息名称
         * 参数 action   消息回调函数
        ****/
        register : function(type, action){
            // 如果该消息存在
            if(_msg[type])
                // 存入回调函数
                _msg[type].push(action);
            else{
                // 不存在 则建立该消息容器
                _msg[type] = [];
                // 存入新消息回调函数
                _msg[type].push(action);
            }
        },
        /**
         * 发布消息方法
         * 参数 type 消息名称
        ****/
        send : function(type){
            // 如果该消息已经被订阅
            if(_msg[type]){
                // 遍历已存储的消息回调函数
                for(var i = 0, len = _msg[type].length; i < len; i++){
                    // 执行回调函数
                    _msg[type][i]();
                }
            }
        }
    };
}
```

```

        // 执行该回调函数
        _msg[type][i] && _msg[type][i]();
    }
}
}();

```

“不过，小白，记住，你虽然创建出中介者了，但是还不能使用，应先进行单元测试，看看你实现的功能是否可用”小铭对小白建议说。

23.3 试试看，可否一用

“嗯。我只需要订阅两个消息，然后让中介者发布这个消息看看是否成功就可以了吧。”

```

// 单元测试
// 订阅 demo 消息 执行回调函数--输出 first
Mediator.register('demo', function() {
    console.log('first');
});
// 订阅 demo 消息 执行回调函数--输出 second
Mediator.register('demo', function() {
    console.log('second');
});
// 发布 demo 消息
Mediator.send('demo');
// 输出结果依次为
// first
// second

```

“小铭，看到没，我的中介者模式成功了！”小白欢喜。

“嗯，很不错，下面你可以实现你的需求了，为你的那些导航模块注册消息吧。”

23.4 攻克需求

小白正要写代码，忽然间想起小铭曾说的“写前多思考”的五字箴言，于是看了看这些导航模块中每条导航元素结构，发现每条导航的消息提醒都是一个****标签，每个导航后面的网址都是标签，这让小白有了些想法：“既然这些导航结构如此的相似是不是可以将设置导航的方法归一化？这样每个模块调用同一个方法就可以了，而且消息提醒导航中的消息元素与网址导航中的网址元素也只是标签名称不一样而已。这样对于一条导航网址元素来说仅需要设置 4 个变量，即导航消息（b）、导航网址（span）、导航显示、导航隐藏。所以只要将这些变量提取出来不就可以实现设置导航的功能了么”。

小白豁然开朗，对自己需求的实现充满希望。“不过我对每个导航模块的样式结构不太了解，对导航隐藏设置 `display:none` 会不会影响到导航后面元素的样式呢”小白又一想：“css 在

显隐上不仅仅给我们提供了 display，还有一个属性 visibility 是占位隐藏的，这样不就可以影响不到其他元素了么。”

23.5 订阅消息

```
/*
 * 显隐导航小组件
 * 参数 mod 模块
 * 参数 tag 处理的标签（消息提醒 b，网址 span）
 * 参数 showOrHide 显示还是隐藏
 */
var showHideNavWidget = function(mod, tag, showOrHide) {
    // 获取导航模块
    var mod = document.getElementById(mod),
        // 获取下面的标签名为 tag 的元素
        tag = mod.getElementsByTagName(tag),
        // 如果设置为 false 或者为 hide 则值为 hidden，否则为 visible
        showOrHide = (!showOrHide || showOrHide == 'hide') ? 'hidden' : 'visible';
    // 占位隐藏这些标签
    for(var i = tag.length - 1; i >= 0; i--) {
        tag.style.visibility = showOrHide;
    }
};
```

“真是太好了”小白心想，“有了这个神奇的中介者模式以及这个神奇的 showHideNavWidget 方法，后面各个模块只需要订阅这个方法就可以了”。

用户收藏导航模块里面的导航有消息提醒与导航网址功能。

```
// 用户收藏导航模块
(function(){
    // ...其他交互逻辑
    // 订阅隐藏用户收藏导航消息提醒消息
    Mediator.register('hideAllNavNum', function(){
        showHideNavWidget('collection_nav', 'b', false);
    });
    // 订阅显示用户收藏导航消息提醒消息
    Mediator.register('showAllNavNum', function(){
        showHideNavWidget('collection_nav', 'b', true);
    });
    // 订阅隐藏用户收藏导航网址消息
    Mediator.register('hideAllNavUrl', function(){
        showHideNavWidget('collection_nav', 'span', false);
    });
    // 订阅显示用户收藏导航网址消息
    Mediator.register('showAllNavUrl', function(){
        showHideNavWidget('collection_nav', 'span', true);
    });
})();
```

推荐用户导航内的导航有消息提醒功能

```
// 推荐用户导航
(function() {
    // ...其他交互逻辑
    // 订阅隐藏推荐用户导航消息提醒消息
    Mediator.register('hideAllNavNum', function(){
        showHideNavWidget('recommend_nav', 'b', false);
    });
    // 订阅显示推荐用户导航消息提醒消息
    Mediator.register('showAllNavNum', function(){
        showHideNavWidget('recommend_nav', 'b', true);
    });
})();
```

最近常用导航内的导航有导航网址功能。

```
// 最近常用导航
(function() {
    // ...其他交互逻辑
    // 订阅显示最近常用导航网址消息
    Mediator.register('hideAllNavController', function(){
        showHideNavWidget('recently_nav', 'span', 'hide');
    });
    // 订阅显示最近常用导航网址消息
    Mediator.register('showAllNavController', function(){
        showHideNavWidget('recently_nav', 'span', 'show');
    });
})();
```

23.6 发布消息

这些模块订阅中介者提供的消息，剩下的只需要监听每位用户对设置层内的导航展现样式的设置，并发送样式消息就可以了。

```
// 设置层模块
(function() {
    // 消息提醒选框
    var hideNum = document.getElementById('hide_num'),
        // 网址选框
        hideUrl = document.getElementById('hide_url');
    // 消息提醒选框事件
    hideNum.onchange = function() {
        // 如果勾选
        if(hideNum.checked) {
            // 中介者发布隐藏消息提醒功能消息
            Mediator.send('hideAllNavNum');
        } else {
            // 中介者发布显示消息提醒功能消息
            Mediator.send('showAllNavNum');
        }
    }
    // 网址选框事件
    hideUrl.onchange = function() {
        // 如果勾选
        if(hideUrl.checked) {
            // 中介者发布隐藏所有网址功能消息
        }
    }
})
```

```
        Mediator.send('hideAllNavUrl');
    }else{
        // 中介者发布显示所有网址功能消息
        Mediator.send('showAllNavUrl');
    }
})();
```

下章剧透

通过中介者对象，我们可以自由调度其他模块，这也是一种解决模块间解耦问题的可行方案。有时候程序做过的事情就该记住，不要等到再次发生时从头执行，那么有没有一种帮助程序记忆的模式呢？翻开下一章去探索吧。

忆之获

同观察者模式一样，中介者模式的主要业务也是通过模块间或者对象间的复杂通信，来解决模块间或对象间的耦合。对于中介者对象的本质是分装多个对象的交互，并且这些对象的交互一般都是在中介者内部实现的。

与外观模式的封装特性相比，中介者模式对多个对象交互地封装，且这些对象一般处于同一层面上，并且封装的交互在中介者内部，而外观模式封装的目的是为了提供更简单的易用接口，而不会添加其他功能。

与观察者模式相比，虽然两种模式都是通过消息传递实现对象间或模块间的解耦。观察者模式中的订阅者是双向的，既可以是消息的发布者，也可以是消息的订阅者。而在中介者模式中，订阅者是单向的，只能是消息的订阅者。而消息统一由中介者对象发布，所有的订阅者对象间接地被中介者管理。

我问你答

通过中介者模式来实现点击键盘方向键控制页面中盒子的移动。

第 24 章 做好笔录——备忘录模式

备忘录模式（Memento）：在不破坏对象的封装性的前提下，在对象之外捕获并保存该对象内部的状态以便日后对象使用或者对象恢复到以前的某个状态。

今日新闻模块要上线了，为保证产品质量，小铭要检查（review）一下新人写的代码，当浏览到显示某一页新闻逻辑代码的时候感觉代码有些不妥，于是叫来小白。

24.1 新闻展示

“小白，显示某一页新闻逻辑代码是你写的么？”

“嗯，有什么问题么？”小白问

“按你目前的逻辑来看是可以走通的，不过对于资源的请求，在你的代码逻辑中倒是有一些浪费。比如，你用 jQuery 代码库对上一页和下一页按钮定义了两个事件，请求上一篇文章或下一篇文章，并将响应得到的新闻内容数据显示在内容区域内。”

```
// 下一页按钮点击事件
$('#next_page').click(function(){
    // 获取新闻内容元素
    var $news = $('#news_content'),
        // 获取新闻内容元素当前页数据
        page = $news.data('page');
    // 获取并显示新闻
    getPageData(page, function(){
        // 修正新闻内容元素当前页数据
        $news.data('page', page + 1);
    })
});
// 上一页按钮点击事件
$('#pre_page').click(function(){
    // 显示上一页
})
```

“那么 getPageData 是你获取新闻内容的逻辑方法，你在该方法里面发起了对新闻内容的异步请求”。

```
// 请求某一页新闻 page: 当前页 fn: 成功回调函数
```

```

function getPageData(page, fn) {
    // post 请求数据
    $.post('./data/getNewsData.php', {
        // 数据: 页码
        page : page
    }, function(res) {
        // 正常返回数据
        if(res.errNo == 0) {
            // 显示当前页
            showPage(page, res.data);
            // 执行回调函数
            fn && fn();
        }
    })
}

// 显示某页逻辑
function showPage(page, data) {
    //...
}

```

“你这么处理是无可厚非的，不过，对于用户点击下一页后，又点击上一页的这种浏览行为你不感觉第二次请求是多余的吗？因为第一次你已经获取了数据，不需要再多发送多余的请求了，如果网速不好，那对用户来说，会照成糟糕的用户体验，如果在手机端也会造成不必要的流量浪费。”

“哦，这便如何是好？”小白顽皮地问。

24.2 缓存数据

“你可以通过备忘录模式来缓存你请求过的数据。也就是说每次发送请求的时候对当前状态做一次记录，将你请求下的数据以及对应的页码缓存下来等，如果将来的某一时刻想返回到某一浏览过的新闻页，直接在缓存中查询即可。直接恢复记录过的状态而不必触发新的请求行为，这是很高效的。”

“还是不太明白，你的意思是说让我把每次请求的数据进行一次缓存吗？然后当用的时候直接在缓存的数据中提取么？”小白追问。

24.3 新闻缓存器

“正是这个意思。你可以像下面这样创建一个 Page 类。”

```

// Page 备忘录类
var Page = function() {
    // 信息缓存对象
    var cache = {};
    /**
     * 主函数
     * 参数 page 页码
    */
}
```

```

* 参数 fn 成功回调函数
*/
return function(page, fn) {
    // 判断该页数据是否在缓存中
    if (cache[page]) {
        // 恢复到该页状态，显示该页内容
        showPage(page, cache[page]);
        // 执行成功回调函数
        fn && fn();
    } else {
        // 若缓存 Cache 中无该页数据
        $.post('../data/getNewsData.php', {
            // 请求携带数据 page 页码
            page : page
        }, function(res) {
            // 成功返回
            if(res.errNo == 0) {
                // 显示该页数据
                showPage(page, res.data);
                // 将该页数据种入缓存中
                cache[page] = res.data;
                // 执行成功回调函数
                fn && fn();
            } else {
                // 处理异常
            }
        })
    }
}
}

```

“小白你看，Page 缓存器与 getPageData 方法不同之处就是在 Page 类的内部缓存了每次请求回来的新闻数据，这样以后如果用户想回看某页新闻数据，就不需要发送不必要的请求了，而且用户也不必要因等待请求响应而造成时间延迟了。”

“简简单单的改动却能优化出这么理想的效果，看来以后我还得多思考了。”小白叹服。

“对于 Page 对象类的使用可以直接替换上一页与下一页绑定实践中的 getPageData 方法即可”。

```

// 下一页按钮点击事件
$('#next_page').click(function() {
    // 获取新闻内容元素
    var $news = $('#news_content'),
        // 获取新闻内容元素当前页数据
        page = $news.data('page');
    // 获取并显示新闻
    Page(page, function() {
        // 修正新闻内容元素当前页数据
        $news.data('page', page + 1);
    })
});

```

24.4 工作中的备忘录

“其实备忘录模式的应用还挺多的，比如你打开页面中的换肤的设置层，第一次打开是要向服务器端发送请求来获取响应数据的，但是第二次就不需要再发送了，我们可以将第一次获取的数据缓存下来即可。再有 MVC 架构中的 M (model) 部分。其实很多时候它都会缓存一些数据，供视图或者控制器模块使用。这些都是应用了备忘录模式的思想而实现的。”

下章剧透

书写备忘录的确对于“记忆力”不是很好的程序来说，是一种不错的缓存方案。在程序语言中循环语句是很重要的一种语句，然而有时程序中过多地出现循环语句又显得臃肿不堪，可读性差，下一章将为我们展示一种循环语句替代者，那是什么？我们拭目以待。

忆之获

备忘录模式最主要的任务是对现有的数据或状态做缓存，为将来某个时刻使用或恢复做准备。在 JavaScript 编程中，备忘录模式常常运用于对数据的缓存备份，浏览器端获取的数据往往是从服务器端请求获取到的，而请求流程往往是以时间与流量为代价的。因此对重复性数据反复请求不仅增加了服务器端的压力，而且造成浏览器端对请求数据的等待进而影响用户体验。

在备忘录模式中，数据常常存储在备忘录对象的缓存器中，这样对于数据的读取必定要通过调用备忘录提供的方法，因此备忘录对象也是对数据缓存器的一次保护性封装，防止外界的直接访问，方便数据的管理，规范化外界对象对数据的使用。一旦备忘录对象发现请求的数据或状态在缓存器中已存在，将直接从缓存器中读取，从而降低对数据的获取成本。

当数据量过大时，会严重占用系统提供的资源，这会极大降低系统性能。此时对缓存器的缓存策略优化是很有必要的，复用率低的数据缓存下来是不值得的。因此资源空间的限制是影响备忘录模式应用的一大障碍。不过随着硬件水平的提高以及浏览器的不断优化，相信资源空间的限制在不久的将来也会得到改善。

我问你答

通过备忘录模式获取的数据的优越性体现在哪里？

第 25 章 点钞机——迭代器模式

迭代器模式（Iterator）：在不暴露对象内部结构的同时，可以顺序地访问聚合对象内部的元素。

程序中的循环是一种利器，循环语句也使我们程序开发更简洁高效，然后有时一遍又一遍的重复性循环却让代码显得臃肿不堪。

25.1 简化循环遍历

“小铭，页面中的焦点图（页面中一种常见特效，通过切换图片实现在一固定区域内浏览多张图片的组件）种类太多了，而且每次创建一个焦点图对象时都要重复写一遍元素遍历循环，哎，这些重复的东西写起来或者看起来真是太痛苦了，有什么好的解决方法么”小白走到小铭面前诉苦。

“哦，一个焦点图对象是由一个焦点图对象类创建的，你只需要在焦点图对象类中创建一次循环就可以了，怎么可能像你说的那么多循环呢。”小铭问道。

“是这样，页面中的焦点图交互特效不是一种，有的要缩放，有的要淡入/淡出，有的要滑动门……所以我只能创建多个类了。”

“你有没有观察到这些类之间有何共同点呀？”小铭问。

“共同点？我想想。啊，他们遍历的内部对象（每一张图片）都是在一个类数组容器里，上一页是获取的上一张图片，下一页是获取的下一张图片，显示某一张图片时，首先要对每张图片做某些处理，最后再处理将要显示的这张图片。”

“你能分析到这里你就应该会想到将他们提取出来了吧，然后创建一个基类让所有焦点图类去继承他。其实除此之外迭代器模式更适合你去解决重复循环迭代的问题，你可以试着创建一个迭代器类。”

25.2 迭代器

“迭代器？”

“嗯，迭代器就是用来顺序地访问一个聚合对象内部的元素的，它可以简化我们遍历操作，就像银行里的点钞机，有了它降低了我们点钞成本，安全而可靠。在我们的焦点图例子中的那些图片元素对象其实就是一组聚合对象，然后你要做的就是创建一个迭代器，专门用来访问这些图片数据的，比如前一张、后一张、第一张、最后一张，对每一张图片的处理，对当前图片的处理等等”

```
// 迭代器
var Iterator = function(items, container){
    // 获取父容器，若 container 参数存在，并且可以获取该元素则获取，否则获取 document
    var container = container && document.getElementById(container) || document,
        // 获取元素
        items = container.getElementsByTagName(items),
        // 获取元素长度
        length = items.length,
        // 当前索引值，默认：0
        index = 0;
    // 缓存源生数组 splice 方法
    var splice = [].splice;
    return {
        // 获取第一个元素
        first : function() {},
        // 获取最后一个元素
        second : function() {},
        // 获取前一个元素
        pre : function() {},
        // 获取后一个元素
        next : function() {},
        // 获取某一个元素
        get : function() {},
        // 对每一个元素执行某一个方法
        dealEach : function() {},
        // 对某一个元素执行某一个方法
        dealItem : function() {},
        // 排他方式处理某一个元素
        exclusive : function() {}
    }
}
```

25.3 实现迭代器

“对于获取第一个元素与获取最后一个元素实现的方式很简单，首先设置当前索引并缓存，然后返回索引对应的元素。”

```
// 获取第一个元素
first : function(){
    index = 0;          // 校正当前索引
    return items[index]; // 获取第一个元素
},
// 获取最后一个元素
second : function(){
    index = length - 1; // 校正当前索引
    return items[index]; // 获取最后一个元素
}
```

},

“对于前一个与后一个元素的获取方法则稍复杂，对于获取前一个元素你要判断这个元素是否是第一个，是的话则返回空元素，并设置索引为 0。对于获取后一个元素来说，你要判断这个元素是否是最后一个，是的话则返回空元素，并设置索引值为 length-1”。

```
// 获取前一个元素
pre : function(){
    if(--index > 0){           // 如果索引值大于 0
        return items[index];   // 获取索引值对应的元素
    }else{
        index = 0;             // 索引值为 0
        return null;            // 返回空
    }
},
// 获取后一个元素
next : function(){
    if(++index < length){    // 如果索引值小于长度
        return items[index];   // 获取索引值对应的元素
    }else{
        index = length - 1;   // 索引值为 length - 1
        return null;            // 返回空
    }
},
```

“对于获取某一个元素方法，需要注意的是，如果参数大于等于 0 则直接对 length 取模获取对应的元素并设置索引值，否则对 length 取模后还要加上 length 才能获取对应的索引值，此时是从后向前搜索元素。”

```
// 获取某一个元素
get : function(num){
    // 如果 num 大于等于 0 再获取正向获取，否则逆向获取
    index = num >= 0 ? num % length : num % length + length;
    // 返回对应元素
    return items[index];
},
```

“对于为每一个元素执行某一个方法既是通过访问者模式（参考二十章访问者模式）使回调函数在每一个元素的作用域中执行一次。此时如果传入的参数大于一个，则将多余参数作为回调函数的参数传递。”

```
// 对每一个元素执行某一个方法
dealEach : function(fn){
    // 第二个参数开始为回调函数中参数
    var args = splice.call(arguments, 1);
    // 遍历元素
    for(var i = 0; i < length; i++){
        // 对元素执行回调函数
        fn.apply(items[i], args);
    }
},
```

“对于处理某一个元素的方法的实现很简单，只需要将回调函数执行时的作用域变为该元素即可。如果传入的参数个数大于两个，则将多余参数作为回调函数的参数传递”。

```
// 对某一个元素执行某一个方法
dealItem : function(num, fn) {
    // 对元素执行回调函数，注：1 第三个参数开始为回调函数中参数 2 通过 this.get 方法设置
    // index 索引值
    fn.apply(this.get(num), splice.call(arguments, 2))
},
```

“排他方法处理某一个元素的思想是综合运用 dealEach 方法与 dealItem 方法。但要注意，如果传入的参数为数组是表示处理多个元素，这次要执行一次遍历”。

```
// 排他方式处理某一个元素
exclusive : function(num, allFn, numFn) {
    // 对所有元素执行回调函数
    this.dealEach(allFn);
    // 如果 num 类型为数组
    if(Object.prototype.toString.call(num) === "[object Array]"){
        // 遍历 num 数组
        for(var i = 0, len = num.length; i < len; i++){
            // 分别处理数组中每一个元素
            this.dealItem(num[i], numFn);
        }
    }else{
        // 处理第 num 个元素
        this.dealItem(num, numFn);
    }
}
```

“小白，有了这个迭代器，以后你再创建任何焦点图类时，只需要用迭代器创建内部图片数据对象即可，如果想处理某一张图片只需要调用迭代器提供的接口方法即可。”

25.4 小试牛刀

“你能举例说明一下应该如何使用么？”小白问。

“比如我们获取我们页面中 id 为 container 的 ul 元素中的 4 个 li 元素我们就可以创建一个迭代器对象”。

```
var demo = new Iterator('li', 'container');
```

“然后如果我们想为每一个元素处理一些逻辑我们就可以这样。”

```
console.log(demo.first());           // <li>1</li>
console.log(demo.pre());             // null
console.log(demo.next());            // <li>2</li>
console.log(demo.get(2000));          // <li>1</li>
// 处理所有元素
demo.dealEach(function(text, color){
```

```

this.innerHTML = text;           // 设置内容
this.style.background = color;  // 设置背景色
}, 'test', 'pink');
// 排他思想处理第3个与第4个元素
demo.exclusive([2, 3], function() {
    this.innerHTML = '被排除的';
    this.style.background = 'green';
}, function() {
    this.innerHTML = '选中的';
    this.style.background = 'red';
})

```

“通过迭代器提供的接口方法就能轻易地访问聚合对象中的每一个元素。甚至不需要知道聚合对象的内部的具体结构，这种思想真是不错的。”小白感叹。

25.5 数组迭代器

“嗯，其实 JavaScript 中很多方法都运用了迭代器思想，比如我们常见的自定义的 each 方法，虽然有些浏览器对数组或者对象都提供了该方法，但是一些低版本的 IE 浏览器还是没有实现，所以有时候我们也需要人工定义他们，比如数组的迭代器方法，依次对数组中每一个元素遍历，并将该元素的索引与索引值传入回调函数中。”

```

// 数组迭代器
var eachArray = function(arr, fn) {
    var i = 0,
        len = arr.length;
    // 遍历数组
    for(; i < len; i++) {
        // 依次执行回调函数，注意回调函数中传入的参数第一个为索引，第二个为该索引对应的值
        if(fn.call(arr[i], i, arr[i]) === false) {
            break;
        }
    }
}

```

25.6 对象迭代器

“对象迭代器与数组迭代器比较类似，但传入回调函数中的为对象的属性与对象的属性值。”

```

// 对象迭代器
var eachObject = function(obj, fn) {
    // 遍历对象中的每一个属性
    for(var i in obj) {
        // 依次执行回调函数，注意回调函数中传入的参数第一个为属性，第二个为该属性对应的值
        if(fn.call(obj[i], i, obj[i]) === false) {
            break;
        }
    }
}

```

```

    }
}
```

25.7 试用迭代器

“好了，小白，如果我们想对一个数组执行一次数组迭代器可以这样。”

```

// 创建一个数组
for(var arr = [], i = 0; i < 5; arr[i++] = i);
eachArray(arr, function(i, data){
    console.log(i, data);
});
// 测试结果
// 0 1
// 1 2
// 2 3
// 3 4
// 4 5
```

“如果想使用对象迭代器处理一个对象是不是就可以像下面这样呀”小白应道。

```

var obj = {
    a : 23,
    b : 56,
    c : 67
}
eachObject(obj, function(i, data){
    console.log(i, data);
});
// 测试结果
// a 23
// b 56
// c 67
```

25.8 同步变量迭代器

“迭代器是很常用的，有时我们操作（获取或者设置）页面中的同步变量（页面加载时打印到页面中的变量）内的某些属性值时，我们不知道服务器端是否将该属性或者该属性的上级属性正确地打印到页面中，此时直接通过点语法或者[]语法直接获取会导致报错，因此每次操作都要一层一层地做安全校验，来屏蔽这些错误，这样使得我们的代码臃肿不堪，例如：”

对象 A: var A = {};

获取 A 属性 b 下面的 c 属性: var c = A.b.c (这种直接获取方式是不允许的，会报错)

正确方式: var c = A && A.b && A.b.c。

“通过迭代器我们即可减少编写这类校验代码。假如页面中有下面一列同步数据”。

```

// 同步变量
var A = {
```

```
// 所有用户共有
common : {},
// 客户端数据
client : {
    user : {
        username : '雨夜清河',
        uid : '123'
    }
},
// 服务器端数据
server : {}
};
```

“那么我们想获取客户端（client）的用户名（userName）数据可以通过一个同步变量迭代取值器来实现”。

```
// 同步变量迭代取值器
AGetter = function(key) {
    // 如果不存在 A 则返回未定义
    if(!A)
        return undefined;
    var result = A;          // 获取同步变量 A 对象
    key = key.split('.');    // 解析属性层次序列
    // 迭代同步变量 A 对象属性
    for(var i = 0, len = key.length; i < len; i++) {
        // 如果第 i 层属性存在对应的值则迭代该属性值
        if(result[key[i]] !== undefined){
            result = result[key[i]];
        } else{
            return undefined;
        }
    }
    // 返回获取的结果
    return result;
}
// 获取用户名数据
console.log(AGetter('client.user.username')); // 雨夜清河
// 获取本地语言数据
console.log(AGetter('server.lang.local')); // undefined
```

“有时在交互中会修改或者增加一些同步变量属性数据，比如我们缓存用户在主页中添加体育新闻模块这一动作数据，我们可以通过同步变量迭代赋值器来实现。”

```
// 同步变量迭代赋值器
ASetter = function(key, val) {
    // 如果不存在 A 则返回未定义
    if(!A)
        return false;
    var result = A;          // 获取同步变量 A 对象
    key = key.split('.');    // 解析属性层次序列
    // 迭代同步变量 A 对象属性
    for(var i = 0, len = key.length; i < len - 1; i++) {
        // 如果第 i 层属性对应的值不存在，则定义为对象
```

```

        if(result[key[i]] === undefined){
            result[key[i]] = {};
        }
        // 如果第 i 层属性对应的值不是对象 (Object) 的一个实例，则抛出错误
        if(!(result[key[i]] instanceof Object)){
            throw new Error('A.' + key.splice(0,i+1).join('.') + ' is not Object');
            return false;
        }
        // 迭代该层属性值
        result = result[key[i]];
    }
    // 返回设置成功的属性值
    return result[key[i]] = val;
}
// 缓存添加体育新闻模块数据
console.log(ASetter('client.module.news.sports', 'on'));           // on

“当然对于一些值类型属性的数据进行子属性赋值操作是不允许的。”
// 为值类型数据添加属性是不允许的
console.log(ASetter('client.user.username.sports', 'on'));
Uncaught Error: A.client.user.username is not Object

```

“有时为了操作对象方便，取值器与赋值器也通常直接赋值给对象的基类。”

25.9 分支循环嵌套问题

当然有时候迭代器也可以解决分支循环嵌套，比如我们用 canvas 来处理图片像素数据。绘制如下效果图（如图 25-1 所示）。



▲图 25-1 canvas 图像处理

```

window.onload = function(){
    var canvas = document.getElementsByTagName('canvas')[0], // 获取画布
        img = document.images[0],                         // 获取图片
        width = (canvas.width = img.width * 2) / 2,       // 获取并设置宽度
        height = canvas.height = img.height,              // 获取并设置高度
        ctx = canvas.getContext('2d');                   // 获取渲染上下文
    ctx.drawImage(img, 0, 0);                          // 绘制图片
    // 绘制特效图片
}

```

```

function dealImage() {
// 为图片添加特效
dealImage('gray', 0, 0, width, height, 255);
dealImage('gray', 100, 50, 300, 200, 100);
dealImage('gray', 150, 100, 200, 100, 255);
}

```

“对于绘制特效图片方法 dealImage 在处理图片数据时需要两个步骤，第一步是迭代每一个图片像素数据，第二步是根据给定特效类型选择不同算法处理像素数据。”

```

/*
 * 绘制特效图片
 * param t    特效类型
 * param x    x 坐标
 * param y    y 坐标
 * param w    宽度
 * param h    高度
 * param a    透明度
 */
function dealImage(t, x, y, w, h, a) {
// 获取画布图片数据
var canvasData = ctx.getImageData(x, y, w, h);
// 获取像素数据
var data = canvasData.data;
// 遍历每组像素数据 (4个数据表示一个像素点数据，分别代表红色、绿色、蓝色、透明度)
for(var i = 0, len = data.length; i < len; i += 4) {
switch(t) {
// 红色滤镜 将绿色与蓝色取值为 0
case 'red' :
    data[i + 1] = 0;
    data[i + 2] = 0;
    data[i + 3] = a;
    break;
// 绿色滤镜 将红色和蓝色取值为 0
case 'green' :
    data[i] = 0;
    data[i + 2] = 0;
    data[i + 3] = a;
    break;
// 蓝色滤镜 将红色和绿色取值为 0
case 'blue' :
    data[i] = 0;
    data[i + 1] = 0;
    data[i + 3] = a;
    break;
// 平均值灰色滤镜 取三色平均值
case 'gray':
    var num = parseInt((data[i] + data[i + 1] + data[i + 2]) / 3);
    data[i] = num;
    data[i + 1] = num;
    data[i + 2] = num;
    data[i + 3] = a;
    break;
// 其他方案
}
}

```

```

    }
    // 绘制处理后的图片
    ctx.putImageData(canvasData, width + x, y);
}

```

“然而我们发现，这种处理逻辑并不是最优的，因为每一次遍历都需要进行一次分支判断，假设有 1000000 个像素点（相当于一张 1000*1000 的图片），并且有 200 种特效，最糟糕的情况下（每处理一个像素点分支判断了 200 次），我们就要做 200000000 次这种无用的分支判断。所以我们要想办法减少这样不必要的消耗，我们可以将循环遍历抽象出来作为一个迭代器存在，每次循环都执行传入迭代器中的某一固定算法，而对于特效算法我们可以设置在策略对象中实现，通过策略模式与迭代器模式的综合运用即可解决上述分支判断问题。”

25.10

解决方案

```

function dealImage(t, x, y, w, h, a){
    var canvasData = ctx.getImageData(x, y, w, h),
        data = canvasData.data;
    // 状态模式封装算法
    var Deal = function(){
        var method = {
            // 默认类型——平均灰度特效
            'default' : function(i){
                return method['gray'](i);
            },
            // 红色特效
            'red' : function(i){
                data[i + 1] = 0;
                data[i + 2] = 0;
                data[i + 3] = a;
            },
            // 平均灰度特效
            'gray' : function(i){
                // 将红、绿、蓝色取平均值
                data[i] = data[i + 1] = parseInt(data[i + 2] = (data[i] + data[i + 1] + data[i + 2]) / 3);
                data[i + 3] = a;
            }
        };
        // 主函数，通过给定类型返回对应滤镜算法
        return function(type){
            return method[type] || method['default'];
        }
    }();
    // 迭代器处理数据
    function eachData(fn){
        for(var i = 0, len = data.length; i < len; i += 4){
            // 处理一组像素数据
            fn(i);
        }
    }
    // 处理数据
}

```

```
eachData(Deal(t));  
ctx.putImageData(canvasData, width + x, y);  
}
```

下章剧透

迭代器是优化循环语句的一种可行方案，它使得程序清晰易读。工作中常常听到经理为你描述一些事情让你去实现，然而这一类需求到底是个什么模型？有没有一套更好的解决方案。下一章将会告诉你。

忆之获

通过迭代器我们可以顺序地访问一个聚合对象中的每一个元素。在开发中，迭代器极大简化了代码中的循环语句，使代码结构清晰紧凑，然而这些简化了的循环语句实质上隐形地移到了迭代器中。当然用迭代器去处理一个对象时，我们只需提供处理的方法，而不必去关心对象的内部结构，这也解决了对象的使用者与对象内部结构之间的耦合。当然迭代器的存在也为我们提供了操作对象的一个统一接口。

我问你答

如何将对象迭代器与数组迭代器融为一体？

第 26 章 语言翻译——解释器模式

解释器模式 (Interpreter): 对于一种语言，给出其文法表示形式，并定义一种解释器，通过使用这种解释器来解释语言中定义的句子。

一个页面中的某些功能好坏有时是靠一定的数据依据支撑的。这不，经理想看看用户对最近新增的功能使用情况，于是前后端要给出统计数据，然而前端交互统计项中要给出交互元素路径。

26.1 统计元素路径

“小铭，后端同事让我们帮忙统计一下页面中点击事件触发元素在页面中所处的路径，我不知道该如何实现它。”小白问。

“哦，这是个什么样的需求呢？”小铭反问。

“是这样的，比如，我们点击页面中的一个 button 元素，就要知道这个元素的 Xpath（元素在页面中所处的路径），感觉这件事情与事件冒泡类似，只不过，在这个路径中他们还要关心一下同一层级中当前元素前面的兄弟元素。如下面的结构”

```
<div class="wrap">
    <div class="link-inner">
        <a href="#">link</a>
    </div>
    <div class="button-inner">
        <button>text</button>
    </div>
</div>
```

“要获取 button 相对于 class 为 wrap 的 div 元素的 Xpath 路径，那么可以表示成 DIV>DIV2>SPAN，然后发送给后端以作统计。再比如”

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
```

```
</head>
<body>
<button>text</button>
</body>
</html>
```

“获取 button 相对于整个页面文档的 XPath 路径为 HTML>BODY>HEAD>BUTTON。”

26.2 描述文法

小铭思考了一下说：“小白，你知道你现在在说什么么？”

“需求？定义？给的条件？”小白疑惑。

“有点贴边，你描述的其实是一种文法，想更好地解决这个问题解释器模式是比较理想的。”小铭解释道。

“文法？这个概念太抽象了，你能解释一下么？”小白问。

“文法就是一种语言中的语法描述的工具，比如我们汉语中的疑问句‘什么什么吗？’感叹句‘什么什么啊！’。也就是说文法是用来定义一组语言规则的。像你刚才描述的那些其实就是一组规则。而我们要做的就是写一个实现这种规则的解释器。所以为完成你的需求建议你使用解释器模式来实现。”

26.3 解释器

“哦，解释器模式是用解释器来完成我刚刚定义的那些文法么？”小白问。

“嗯，所以说你现在最关键的任务是创建一个解释器，有了这个解释器你的任务就基本完成了。”小铭说。

“哦，可是在写解释器之前我应该考虑哪些问题呢？如何实现一个解释器呢？”小白有些犹豫。

“首先你要分析你给出的文法，查找他们的相似点，你会发现他们的相似点是，最右边第一个元素都是目标元素，而最左边第一个元素都是最外层容器元素。而我们从右向左观察，这很像我们熟悉的事件流中的冒泡阶段。所以我们可以试着按照事件冒泡流程把我们的 XPath 路径提取出来。”

小铭歇了歇接着说：“要完成这个过程首先我们要通过元素的 parentNode 来获取当前元素的父元素，直到我们找到元素的终结点位置。但是在这个过程中我们要注意，每冒泡到上一层我们还要遍历该元素前面的兄弟元素 previousSibling，这里需要注意的是如果这个兄弟元素名字与他后面的元素（或者当前元素）的名字相同，则在原元素名上加一，否者添加该元素的元素名称。为使流程简化，我们先将每一层获取到的元素名作为数组成员保存在数组中。首先实现遍历同级兄弟元素方法 getSublingName。”

26.4 同级兄弟元素遍历

```
// 获取兄弟元素名称
function getSublingName(node) {
    // 如果存在兄弟元素
    if(node.previousSibling) {
        var name = '', // 返回的兄弟元素名称字符串
            count = 1, // 紧邻兄弟元素中相同名称元素个数
            nodeName = node.nodeName, // 原始节点名称
            sibling = node.previousSibling; // 前一个兄弟元素
        // 如果存在前一个兄弟元素
        while(sibling) {
            // 如果节点为元素 并且节点类型与前一个兄弟元素类型相同，并且前一个兄弟元素名称存在
            if(sibling.nodeType == 1 && sibling.nodeType === node.nodeType &&
                sibling.nodeName) {
                // 如果节点名称和前一个兄弟元素名称相同
                if(nodeName == sibling.nodeName) {
                    // 节点名称后面添加计数
                    name += ++count;
                } else {
                    // 重置相同紧邻节点名称节点个数
                    count = 1;
                    // 追加新的节点名称
                    name += ' | ' + sibling.nodeName.toUpperCase();
                }
            }
            // 向前获取前一个兄弟元素
            sibling = sibling.previousSibling;
        }
        return name;
    } else {
        return '';
    }
}
```

26.5 遍历文档树

“有了这个方法我们在实现冒泡遍历整个文档树的时候，处理每一层的元素节点就方便多了，下面我们实现冒泡遍历文档树功能，并实现我们的解释器。”

```
//XPath 解释器
var Interpreter = (function() {
    // 获取兄弟元素名称
    function getSublingName(node) {
        // .....
    }
    // 参数1 node: 目标节点 参数二 wrap: 容器节点
    return function(node, wrap) {
        // 路径数组
        var path = [],
            // 如果不存在容器节点，默认为 document
            wrap = wrap || document;
```

```

// 如果当前(目标)节点等于容器节点
if(node === wrap){
    // 容器节点为元素
    if(wrap.nodeType == 1){
        // 路径数组中输入容器节点名称
        path.push(wrap.nodeName.toUpperCase());
    }
    // 返回最终路径数组结果
    return path;
}
// 如果当前节点的父节点不等于容器节点
if(node.parentNode !== wrap){
    // 对当前节点的父节点执行遍历操作
    path = argumentscallee(node.parentNode, wrap);
}
// 如果当前节点的父元素节点与容器节点相等
else{
    // 容器节点为元素
    if(wrap.nodeType == 1){
        // 路径数组中输入容器节点名称
        path.push(wrap.nodeName.toUpperCase());
    }
}
// 获取元素的兄弟元素名称统计
var sublingsNames = getSublingName(node);
// 如果节点为元素
if(node.nodeType == 1){
    // 输入当前节点元素名称及其前面兄弟元素名称统计
    path.push(node.nodeName.toUpperCase() + sublingsNames);
}
// 返回最终路径数组结果
return path;
}
// 立即执行方法
})();

```

26.6 小试牛刀

“小白，有了这个 XPath 解析器，对于你的需求的实现就容易多了，比如页面有个这样的结构：”

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
<div></div>
<div id="container">
    <div>
        <div>
            <ul>
                <li><span id="span1"></span></li>
                <li><span id="span2"></span></li>

```

```

        </ul>
    </div>
</div>
<div>
    <div>
        <ul>
            <li><span id="span6"></span></li>
            <li><span id="span7"></span></li>
        </ul>
    </div>
</div>
</body>
</html>

```

“我们如果想获取 id 为 span6 元素相对于文档的路径怎么办”小铭问。

“直接调用 Interpreter 方法就可以了吧”小白答道。

```
var path = Interpreter(document.getElementById('span7'));
```

“哈哈，不要忘记，我们的解析器 Interpreter 返回的是数组呀，所以我们还要执行一步操作——用 ‘>’ 拼接数组就可以了。”

```
console.log(path.join('>')); // HTML>BODY|HEAD>DIV2>DIV2>DIV>UL>LI2>SPAN
```

“这样我们的任务就能轻松地完成了。”

下章剧透

一些描述性的语句，几次功能的提取抽象，形成了一套语言法则，这就是解释器模式要处理的事情。还曾记得为页面流畅地书写简单而高效的 jQuery 代码么？一个小圆点，一个方法名，一对圆括号，将他们一组组地拼连即可完成诸多任务，但是它又是如何实现的呢？下一章我们将解密 jQuery 实现原理。

忆之获

解释器即是对客户提出的需求，经过解析而形成的一个抽象解释程序。而是否可以应用解释器模式的一条重要准则是否能根据需求解析出一套完成的语法规则，不论该语法规则简单或是复杂都是必须的。因为解释器要按照这套规则才能实现相应功能。

我问你答

用解释器描述单击事件中目标对象到事件源对象的层次结构。如为 div 绑定 click 事件，当点击 div 中的 button 元素时，则应表示成 button>div。

第五篇

技巧型设计模式

技巧型设计模式是通过一些特定技巧来解决组件的某些方面的问题，这类技巧一般通过实践经验总结得到。

- 第 27 章 永无尽头——链模式
- 第 28 章 未来预言家——委托模式
- 第 29 章 数据管理器——数据访问对象模式
- 第 30 章 执行控制——节流模式
- 第 31 章 卡片拼图——简单模板模式
- 第 32 章 机器学习——惰性模式
- 第 33 章 异国战场——参与者模式
- 第 34 章 入场仪式——等待者模式

第 27 章 永无尽头——链模式

链模式（Operate of Responsibility）：通过在对象方法中将当前对象返回，实现对同一个对象多个方法的链式调用。从而简化对该对象的多个方法的多次调用时，对该对象的多次引用。

近日小白研习 jQuery，深深陶醉于 jQuery 简洁的链式操作，可看了半天也不知道如何实现的，想看看了源码却又感觉无从看起，感到学习的坎坷与前途的迷惑，于是跑来问小铭……

27.1 深究 jQuery

“小铭，我看你们写的 jQuery 代码，通过点语法就可以使用多个方法，你知道这是如何实现的么？看上去很精炼而且做了不少的事情。”小白好奇地问。

“jQuery 么？这可是一个高端而不失奢华的框架。奇特的设计思想使其通过链式调用而显得更加简洁。不过这种链模式是基于原型继承的，并且在每一个原型方法的实现上都返回当前对象 this，使当前对象一直处于原型链作用域的顶端。这样即可实现链式调用。我们先来看看 jQuery 是如何架构的。那么我们就要了解 JavaScript 的原型式继承。比如”

27.2 原型式继承

```
var A = function(){}
A.prototype = {
  length : 2,
  size : function(){
    return this.length;
  }
}
```

“我们创建一个对象 A，并且 A 的原型对象上拥有一个属性 length 和一个 size 方法，那么我们想访问 size 方法该怎么做呢？”

“方法在 A 的原型上，所以我们应该通过 new 关键字创建新对象访问吧。”小白说。

```
var a = new A();
console.log(a.size()); // 2
```

“嗯，对，但如果要是以下面两种方式访问程序就会报错的。”

```
console.log(A.size())
console.log(A().size())
```

“第一种方式报错的原因是因为 size 绑定在 A 的原型上没有绑定在其自身上。第二种方式报错的原因是因为 A 函数对象执行的结果是没有返回值的所以找不到 size 方法。两种方式都是因为 size 方法绑定在 A 类的原型上导致的。但是你回想一下 jQuery 中的方法是如何访问到的呢？”

“通过\$()的方式呀。啊，这就是说\$()在函数执行结束后返回了一个带有很多方法的对象吧！”小白若似恍然大悟。

27.3 找位助手

“对，既然自己访问不到，找人帮忙试试，于是我们可以借助另外一个对象来实现。”

```
var A = function(){
    return B;
}
var B = A.prototype = {
    length : 2,
    size : function(){
        return this.length;
    }
}
```

“此时若想访问 size 方法我们就可以像下面这样访问。”

```
console.log(A().size()); // 2
```

“而在 jQuery 中，为了减少变量的创建，索性直接将 B 对象看做是 A 的一个属性设置。”

```
var A = function(){
    return A.fn;
}
A.fn = A.prototype = {}
```

27.4 获取元素

“到这里，就遇到新问题了，我们知道 jQuery 的目的是为了获取元素，返回的是一组元素簇（元素的聚合对象），但现在返回的却是一个 A.fn 对象，显然达不到我们的需求，所以，如果 A.fn 能提供给我们一个获取元素的方法 init 就好了。所以我们可以将 init 方法获取到的元素在 A 方法中返回。”

```
var A = function(selector){
    return A.fn.init(selector);
```

```

    }
A.fn = A.prototype = {
    init : function(selector){
        return document.getElementById(selector)
    },
    length : 2,
    size : function(){
        return this.length;
    }
}
console.log(A('demo'));//<div id="demo"></div>

```

27.5 一个大问题

“但是别忘了，我们还想让 A 对象返回的结果还能够拥有 A.fn 中的方法，如 size 等，现在的设计是无法达到的。不过好在我们有个神器——对象中的 this 指向的就是当前对象，而在 init 方法中的当前对象就是 A.fn，所以我们直接在 init 中将 this 返回即可。”

“但是我们还想要获取到元素呀，鱼和熊掌不可兼得怎么办？”小白追问。

“这是一个好问题，我们想一下，既然 this 指向的是一个对象，那么对象就可以设置属性，我们可以通过 this 对象将元素设置成当前对象的一个属性不就可以了么，而且如果你想像数组那样访问你就可以将他们的属性值顺序地设置为数字索引。为了更像数组，我们还可以校正一下他的 length 属性。”

```

var A = function(selector){
    return A.fn.init(selector);
}
A.fn = A.prototype = {
    init : function(selector){
        this[0] = document.getElementById(selector); // 作为当前对象的属性值保存
        this.length = 1; // 校正 length 属性
        return this; // 返回当前对象
    },
    length : 2,
    size : function(){
        return this.length;
    }
}
var demo = A('demo');
console.log(demo); //Object {0: div#demo, init: function, length: 1, size: ...}
console.log(A('demo').size()); // 1

```

小白也在后面写了一行测试代码。对比一下发现奇怪的现象，后获取的 test 元素将先获取的 demo 元素覆盖了。

```

var test = A('test');
console.log(demo); // Object {0: div#test, init: function, length: 1, size: ...}

```

27.6 覆盖获取

小白思考了一下说，“小铭，后获取的 id 为 test 元素会将先获取的 id 为 demo 的元素进行覆盖，是不是因为我们每次在 A 的构造函数中返回的 A.fn.init(selector); 对象指向同一个对象造成的。”

“是这样，其实 size 这个方法也被共用的。当然解决这个问题也很容易，直接使用 new 关键字复制创建即可，不过此时调用 size 方法就会报错了。”

```
var A = function(selector){
    return new A.fn.init(selector);
}
console.log(A('demo'))           // {0: div#demo, length: 1}
console.log(A('test'))           // {0: div#test, length: 1}
console.log(A('demo').size())     // Uncaught TypeError
```

27.7 方法丢失

“这是因为在 A 的构造函数中返回的 A.fn.init(selector) 与 new A.fn.init(selector) 的实现的差别造成的，先说 A.fn.init(selector)，init 返回的 this 是指向的当前对象，也就是 A.fn 和 A.prototype，而 new A.fn.init(selector)，由于进行了 new 对对象内的属性复制，所以 this 指向的就不是 A.fn 和 A.prototype 了，我们可以测试一下这两种情况，如”

```
init : function(selector){
    this[0] = document.getElementById(selector)
    this.length = 1;
    console.log(this === A.fn, this === A.prototype, this);
    return this;
}
```

A.fn.init(selector) 的测试结果：true true Object {0: div#demo, init: function, length: 1, size: ...}

new A.fn.init(selector) 的测试结果：false false A.fn.A.init {0: div#demo, length: 1}

“小白，这次测试你能明白他们之间的不同了吧”。小铭问小白。

“啊，原来是这样，” 小白再仔细一看发现一个小差别：“A.fn.init(selector) 方式中 init 方法中的 this 是一个对象，这个我可以理解，毕竟 A.fn = A.prototype = {} 写法造成的，后面的 ‘{}’ 表示的是一个对象，但是 new A.fn.init(selector) 方式中 init 方法中的 this 为什么是 A.fn.A.init 呢？” 小白不解。

“你挺仔细的，要明白这件事情我们要从构造函数说起，我们知道 new 关键字执行的实质是对构造函数的属性的一次复制。那么问你一个问题，new A.fn.init(selector) 方式两次获取到的元素可以独立保存而没有被覆盖，但是你知道 new A.fn.init(selector) 的构造函数怎么表示吗？”

小白看了看 return new A.fn.init(selector);，想了想，有些疑问地写下两行：

“A.fn.init 这样？还是 A.init（new 关键字创建并通过 A.prototype 原型找到的）这样？”

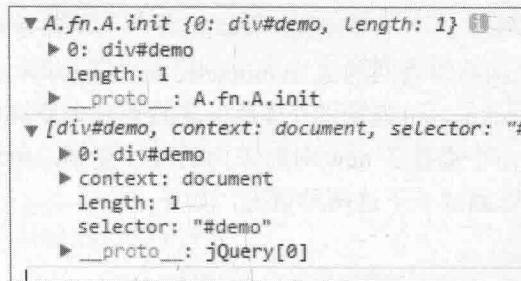
“两种方式都对，所以他们可以这么表示 A.fn.init = A.init，我们将 A.init 带入 A.fn.init 中的 init，就可以得到 A.fn.A.init 结果了，这正是上面测试中所看到的。不过你要知道原型对象中的方法是在构造函数通过 new 关键字执行时才能被构造函数获取到，所以平时构造函数是获取不到原型对象中的方法的。”

27.8 对比 jQuery

“原来是这样。”小白想了想，打开控制台看了看 A 框架获取的结果与 jQuery 获取元素的对比结果。

```
console.log(A('demo'))
console.log($('#demo'))
```

发现他们的原型有些不同（如图 27-1 所示）。



▲图 27-1 原型对比

第一个__proto__为 A.fn.A.init

第二个__proto__为 jQuery[0]

小白惊讶。“小铭，你看 jQuery 中 A.fn.A.init 这个问题已经解决了，jQuery 看上去更像是一个 jQuery 对象呀，我们这里如何才能实现呢？”

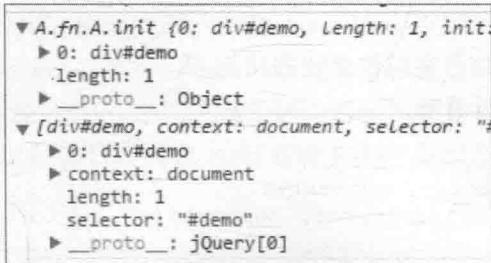
“其实你看过 jQuery 源代码，你会发现解决这个问题的方式很简单却也很巧妙，只要将构造函数的原型指向一个已存在的对象即可，比如：”

```
A.fn.init.prototype = A.fn;
```

“此时你再看看结果吧（如图 27-2 所示）。”

“果然是 Object 了，不过 jQuery 中确实是 jQuery 呀”小白继续问。

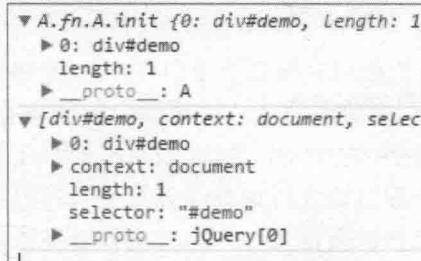
“这是因为实例化的对象是在构造函数执行时创建的，所以 constructor 指向的就是 A.fn.A.init 构造函数，但是这个对象在执行完毕之后就‘不存在’了，所以我们为了强化 constructor，我们想 jQuery 一样，为了让__proto__指向 A 对象，我们可以这么做：”



▲图 27-2 原型对比

```

var A = function(selector){
  return new A.fn.init(selector);
}
A.fn = A.prototype = {
  // 强化构造器
  constructor : A,
  init : function(selector){
    console.log(this.constructor);
    /*
     * 此处输出的测试结果
     */
    *function (selector){
      *  return new A.fn.init(selector);
    *}
    */
    ...
  }
  ...
}
A.fn.init.prototype = A.fn;
  
```



▲图 27-3 原型对比

“此时我们的 A 框架的 `__proto__` 为 A 了（如图 27-3 所示），并且 `size` 方法我们也可以正常使用了。”

```
console.log(A('demo').size())           // 1
```

27.9

丰富元素获取

“哦，小铭，但是我们现在不仅仅想获取带有 id 的元素了，我们还想获取某类元素，我只

需要修改 init 方法就可以吧。”

“嗯，是这样，你可以自己实现你想要的功能。”

“哦，那我写一下，你看看呀。”

```
// selector 选择符, context 上下文
var A = function(selector, context) {
    return new A.fn.init(selector, context);
}
A.fn = A.prototype = {
    constructor : A,
    init : function(selector, context) {
        // 获取元素长度
        this.length = 0,
        // 默认获取元素的上下文为 document
        context = context || document;
        // 如果是 id 选择符 按位非将-1 转化为 0, 转化为布尔值 false
        if(~selector.indexOf('#')){
            // 截取 id 并选择
            this[0] = document.getElementById(selector.slice(1));
            this.length = 1;
        } // 如果是元素名称
        else{
            // 在上下文中选择元素
            var doms = context.getElementsByTagName(selector),
                i = 0,                      // 从第一个开始筛选
                len = doms.length;          // 获取元素长度
            for(; i < len; i++){
                // 压入 this 中
                this[i] = doms[i];
            }
            // 校正长度
            this.length = len;
        }
        // 保存上下文
        this.context = context;
        // 保存选择符
        this.selector = selector;
        // 返回对象
        return this;
    },
    .....
}
```

27.10 数组与对象

“小白，你测试时候有没有发现控制台中出现下面这种情况：jQuery 获取的元素更像一个数组，而我们的 A 框架返回的更像是一个对象的情况呢（如图 27-4 所示）。”

“是呀，这是怎么一回事呢？”

“由于 JavaScript 的弱类型语言，并且数组、对象、函数都被看成是对象的实例，所以

JavaScript 中并没有一个纯粹的数组类型。而且 JavaScript 引擎的实现也没有做严格的校验，也是基于对象实现的。一些浏览器解析引擎在判断对象是否是数组的时候不仅仅判断其有没有 length 属性，可否通过‘[索引值]’方式访问元素，还会判断其是否具有数组方法来确定是否要用数组的形式展现，所以我们只需要在 A.fn 中添加几个数组常用的方法来增强数组特性就可以解决问题了。如”

```
► A.fn.A.init {0: div#demo, 1: div#test, 2: div, length: 3}
► [div#demo, div#test, div, prevObject: jQuery.fn.jQuery.init[1], c
```

▲图 27-4 结果对比

```
A.fn = A.prototype = {
    ...
    // 增强数组
    push: [].push,
    sort: [].sort,
    splice: [].splice
}
```

测试结果（如图 27-5 所示）：

```
► [div#demo, div#test, div, context: body, selector: "div", construc
► [div#demo, div#test, div, prevObject: jQuery.fn.jQuery.init[1], c
```

▲图 27-5 结果对比

27.11 方法拓展

“原来如此。对了，我看到 jQuery 中很多方法都可以通过点语法链式使用，这是重点呀。我们应该如何添加这些方法才是链式调用呢？”小白继续问。

“是这样，jQuery 中定义了一个 extend 方法，比如我们经常使用的 jQuery 插件，jQueryUI 等等都是通过 extend 方法拓展的，不过我们有时对对象拓展中也会用到它，所以我们基本断定 extend 有两个作用，一是对外部对象拓展，二是对内部对象拓展。根据这个原理我们可以简单实现 extend 方法：如果只有一个参数我们就定义为对 A 对象或者 A.fn 对象的拓展，对 A.fn 对象的拓展是因为我们使用 A() 返回对象中的方法是从 A.fn 对象上获取的。多个参数表示对第一个对象的拓展。”

```
// 对象拓展
A.extend = A.fn.extend = function() {
    // 拓展对象从第二个参数算起
    var i = 1,
        // 获取参数长度
        len = arguments.length,
        // 第一个参数为源对象
        target = arguments[0],
```

```

    // 拓展对象中属性
    j;
    // 如果只传一个参数
    if(i == len){
        // 源对象为当前对象
        target = this;
        // i 从 0 计数
        i--;
    }
    // 遍历参数中拓展对象
    for(; i < len; i++){
        // 遍历拓展对象中的属性
        for(j in arguments[i]){
            // 拓展源对象
            target[j] = arguments[i][j];
        }
    }
    // 返回源对象
    return target;
}

```

测试用例：

```

// 拓展一个对象
var demo = A.extend({first : 1}, {second : 2}, {third : 3});
console.log(demo) // {first: 1, second: 2, third: 3}
// 拓展 A.fn 方式一
A.extend(A.fn, {version : '1.0'});           // 1.0
console.log(A('demo').version);
// 拓展 A.fn 方式二
A.fn.extend({getVersion: function(){return this.version}})
console.log(A('demo').getVersion());          // 1.0
// 拓展 A 方式一
A.extend(A, {author:'张容铭'});
console.log(A.author);           //张容铭
// 拓展 A 方式二
A.extend({nickname:'雨夜清河'});
console.log(A.nickname);      //雨夜清河

```

27.12

添加方法

“小白，有了这个方法我们就可以拓展那些可以链式调用的方法了，因为我们向 A.fn 对象中添加的方法间接地添加到了 A.fn.init 的原型对象上了，不过不要忘记我们为 A.fn 添加的方法在结尾处要返回当前对象 this 呀，这样我们就能实现链式调用了。我们现在就简单实现为元素添加绑定事件方法 on，设置 CSS 样式方法 css，设置元素属性方法 attr，设置元素内容方法 html。”

```

A.fn.extend({
    // 添加事件
    on : (function(){
        // 标准浏览器 DOM2 级事件
    })
}

```

```

if(document.addEventListener){
    return function(type, fn){
        var i = this.length - 1;
        // 遍历所有元素添加事件
        for(; i >= 0; i--){
            this[i].addEventListener(type, fn, false);
        }
        // 返回源对象
        return this;
    }
}
// IE 浏览器 DOM2 级事件
} else if(document.attachEvent){
    return function(type, fn){
        var i = this.length - 1;
        for(; i >= 0; i--){
            this[i].addEvent('on' + type, fn);
        }
        return this;
    }
}
// 不支持 DOM2 级事件浏览器添加事件
} else{
    return function(type, fn){
        var i = this.length - 1;
        for(; i >= 0; i--){
            this[i]['on' + type] = fn;
        }
        return this;
    }
}
})()
})

```

“对于绑定事件方法，我们针对不同浏览器采用不同的绑定方式，当然为了日后绑定事件时减少浏览器支持功能校验开销，我们在第一次加载时创建出适用该浏览器的事件绑定方法。”

```

A.extend({
    // 将'-'分割线转化为驼峰式，如:'border-color' -> 'borderColor'
    camelCase : function(str){
        return str.replace(/-(\w)/g, function(all, letter){
            return letter.toUpperCase();
        });
    }
});
A.extend({
    // 设置 css 样式
    css : function(){
        var arg = arguments,
            len = arg.length;
        if(this.length < 1){
            return this;
        }
        // 只有一个参数时
        if(len === 1){
            // 如果为字符串则为获取第一个元素 CSS 样式
            if(typeof arg[0] === 'string'){

```

```

        // IE
        if(this[0].currentStyle){
            return this[0].currentStyle[name];
        }else{
            return getComputedStyle(this[0],false)[name];
        }
    // 为对象时则设置多个样式
    }else if(typeof arg[0] === 'object'){
        for(var i in arg[0]){
            // 遍历每个样式
            for(var j = this.length - 1; j >= 0; j--){
                // 调用拓展方法 camelCase 将'-'分割线转化为驼峰式
                this[j].style[A.camelCase(i)] = arg[0][i];
            }
        }
    }
    // 两个参数则设置一个样式
}else if(len === 2){
    for(var j = this.length - 1; j >= 0; j--){
        this[j].style[A.camelCase(arg[0])] = arg[1];
    }
}
return this;
}
})
)

```

“获取或设置 CSS 样式方法中，如果只传递一个参数，如果参数是字符串则返回第一个元素 CSS 样式值，此时不能进行链式调用。如果是对象则为每一个元素设置多个 CSS 样式，如果是两个参数则为每一个元素设置样式。”

```

A.fn.extend({
    // 设置属性
    attr : function(){
        var arg = arguments,
            len = arg.length;
        if(this.length < 1){
            return this;
        }
        // 如果一个参数
        if(len === 1){
            // 为字符串则获取第一个元素属性
            if(typeof arg[0] === 'string'){
                return this[0].getAttribute(arg[0]);
            // 为对象设置每个元素的多个属性
            }else if(typeof arg[0] === 'object'){
                for(var i in arg[0]){
                    // 遍历属性
                    for(var j = this.length - 1; j >= 0; j--){
                        this[j].setAttribute(i, arg[0][i]);
                    }
                }
            }
        }
        // 两个参数则设置每个元素单个属性
    }else if(len === 2){
        for(var j = this.length - 1; j >= 0; j--){

```

```

        this[j].setAttribute(arg[0], arg[1]);
    }
}
return this;
})
})

```

“获取或设置元素属性方法与设置元素 css 样式方法思路一样，如果只传递一个参数，如果参数为字符串则返回第一个元素属性值，此时不能再继续链式调用。如果参数是对象则设置每一个元素的多个属性值，如果传递两个参数，则第一个参数为属性名，第二个参数为属性值，设置每个元素的属性。”

```

A.fn.extend({
// 获取或者设置元素的内容
html : function(){
    var arg = arguments,
        len = arg.length;
// 无参数则获取第一个元素的内容
if(len === 0){
    return this[0] && this[0].innerHTML;
// 一个参数则设置每一个元素的内容
} else{
    for(var i = this.length - 1; i >= 0; i--){
        this[i].innerHTML = arg[0];
    }
}
return this;
}
})

```

“获取或者设置元素内容的方法参数跟前两个有些变化，如果无参数则返回第一个元素的内容，此时亦不能链式调用。如果有参数，则将第一个参数作为内容来设置各个元素的内容。”

27.13 大功告成

“好了，小白。我们的工作完成了，现在我们体验链式操作的乐趣吧。”

```

A('div')
.css({
    height : '30px',
    border : '1px solid #000',
    'background-color' : 'red'
})
.attr('class', 'demo')
.html('add demo text')
.on('click', function(){
    console.log('clicked');
});

```

下章剧透

链模式可以提高功能的开发效率，降低开发成本，其简洁明了的风格深受开发者喜爱。事件绑定在 Web 编程中是很重要的一部分，它是实现页面交互的纽带，然后很多开发者为实现某些功能，肆无忌惮地绑定事件也会为页面造成很多负面作用。通过下章我们将学习如何更好地解决这类问题。

忆之获

JavaScript 中的链模式的核心思想就是通过在对象中的每个方法调用执行完毕后返回当前对象 this 来实现的。由于链模式使得代码紧凑简洁而高效，在工作中已经得到很广泛的应用。当然当前主流代码库都以该模式作为自己的一种风格。例如 jQuery 等等。所以说熟知链模式的工作原理，在我们实际工作中是很有帮助的。

我问你答

运用链模式实现一套属于你自己的代码库吧。

第 28 章 未来预言家——委托模式

委托模式（Entrust）：多个对象接收并处理同一请求，他们将请求委托给另一个对象统一处理请求。

很多用户反馈说要在用户页面添加一个日历功能，这不项目经理接到小白的任务正在紧锣密鼓地开发呢。

28.1 点击日历交互

“小白，日历模块是你写的么？”小铭问。

“是呀，有什么问题么？”

“项目经理交代需求时说，当用户点击每个日期格子的时候将格子的背景色变成灰色，可是你怎么为每一个日期元素都绑定了一个事件呀。”

```
var ul = document.getElementById('container'),  
    li = document.getElementsByTagName('li'),  
    i = li.length - 1;  
for(; i >= 0; i--){  
    li[i].onclick = function(){  
        this.style.backgroundColor = 'grey';  
    }  
}
```

“直接用 for 循环就可将所有原始对象绑定点击事件，简单而实用。这有什么问题吗？”

“当然呀，你看你这么做无形之中增加了多少个事件，事件多了，内存消耗就大了，尤其在一些老版本浏览器中如 IE6、IE7、IE8，事件过多会影响用户体验的。这里你完全可以通过代理模式，绑定一个事件完成需求的。”

“代理模式？还有这么神奇的模式？”小白好奇地问。

28.2 委托父元素

“当然了，你知道完整的事件流，是从事件捕获开始，到触发该事件，再到事件冒泡三个

阶段，所以你可以将子元素的事件委托给更高层面上的父元素去绑定执行。所以说你写的事件逻辑就可以改写成下面的方式去实现。”

```
ul.onclick = function(e) {
    var e = e || window.event,
        tar = e.target || e.srcElement;
    if(tar.nodeName.toLowerCase() === 'li'){
        tar.style.backgroundColor = 'grey';
    }
}
```

“我们看到，这些个 li 元素共有一个父元素 ul，所以如果将点击事件绑定给父元素 ul，通过事件冒泡就实现事件的传递，这样在子元素 li 上点击的事件就能传递给 ul，这样我们只需要监听 ul 的点击事件，然后判断目标元素的名称是不是我们寻找的元素，如果是则执行相应操作。这样我们为父元素绑定一个事件，通过委托模式就实现了所有子元素的点击事件需求。这样就可起到对页面优化的作用。”

“哦，原来事件委托是将子元素的事件委托给父元素，然后通过事件冒泡传递的，再通过判断事件源的某种特性来执行某一业务逻辑，果然更高明一些。”

“这是当然，正如你所说的判断子元素某种特性，我们这里判断的元素名称，当然你还可以判断 class、id，甚至是自定义的属性。不过，事件委托不仅可以优化页面中事件的数量，还有另外一个作用就是对‘预言未来’。”小铭笑着说。

28.3 预言未来

“预言未来？听起来倒是很神秘的，又是一个大魔法师么？”小白笑了笑，好奇地问。

“预言未来当然指的是未来的事情，也就是当前页面所不存在的，比如我们在未来某个时刻会添加某些元素，要是想对未来的元素绑定事件，以我们当前的技术是很难做到的。不过委托模式巧妙地将未来元素的事件委托给现有的父元素，便可以实现对未来元素的间接事件绑定。比如页面中有一行文字内容”。

```
<div id="article">
    <p>第一段文字</p>
</div>
```

“对于这个 p 元素我们很容易为其添加一个 click 事件，因为目前这个元素是存在的，是可以获取的。但是将来某个时刻，在这行文字下面又添加了一行文字，此时我们想对新添加的这行文字绑定 click 事件确实是困难的。不过我们可以转换一下思路，将事件绑定给 id 为 article 的 div 父元素。这是因为目前或者将来，这个 div 元素都是存在的，在未来通过事件冒泡判断触发点击的元素是不是将来添加的元素来执行某一业务逻辑是可以实现的。比如”

```
var article = document.getElementById('article');
article.onclick = function(){}
```

```

var e = e || window.event,
    tar = e.target || e.srcElement;
if(tar.nodeName.toLowerCase() === 'p'){
    tar.innerHTML = '我要更改这段内容!';
}
var p = document.createElement('p');
p.innerHTML = '新增一段内容';
article.appendChild(p);

```

“我们在为父元素绑定事件后，创建一个 p 元素并插入父元素中，这样当点击 p 元素时依旧可以触发相应的逻辑”。

“这边是极好的。”小白要了个鬼脸以表钦佩。

28.4 内存外泄

“其实，小白你知道吗？有时候委托模式还能处理一些内存外泄问题，这又要提到糟糕的老版本 IE 浏览器了，由于它的引用计数式垃圾回收机制，使得那些对 Dom 元素的引用没有显性清除的数据会遗留在内存中，除非关闭浏览器，否则无法清除，比如”

```

<div id="btn_container">
    <button id="btn">demo</button>
</div>
var g = function(id){return document.getElementById(id)}
g('btn').onclick = function(){
    g('btn_container').innerHTML = '触发了事件';
}

```

“为 id 为 btn 的元素绑定点击事件，在触发时，在其父元素中重置内容，这样将会将按钮自身覆盖掉，然而 g 变量保存的元素绑定的 click 事件没有清除，所以这个事件就会泄露到内存中。为了解决这个问题，我们可以在父元素设置内容前显性地清除事件，比如”

```

g('btn').onclick = function(){
    g('btn') = null;
    g('btn_container').innerHTML = '触发了事件';
}

```

“但是 g('btn') = null; 这个清除事件语句在一些标准浏览器中是不需要的，因为他们采用标记清除方式管理着自身的内存。所以更好的解决办法便是委托模式了，比如”

```

g('btn_container').onclick = function(e){
    // 获取触发事件元素
    var target = e && e.target || window.event.srcElement;
    // 判断触发事件元素是否是 id 为 btn 的元素
    if(target.id === 'btn'){
        // 重置父元素内容
        g('btn_container').innerHTML = '触发了事件';
    }
}

```

“哦，原来委托模式在事件中有这么广泛的应用，可是在其他方面有什么应用么？”小白问。

28.5

数据分发

这是当然，比如我们在与后端处理一些数据请求时，比如我们页面的 banner、aside、article、member、message 模块都需要向后端发起请求以获取数据时，通常做法是：

```
$ .get("./deal.php?q=banner", function(res) {
    // 处理 banner 模块逻辑
})
$.get("./deal.php?q=aside", function(res) {
    // 处理 aside 模块逻辑
})
$.get("./deal.php?q=article", function(res) {
    // 处理 article 模块数据
})
$.get("./deal.php?q=member", function(res) {
    // 处理 member 模块数据
})
$.get("./deal.php?q=message", function(res) {
    // 处理 message 模块数据
})
```

“这样做，我们就会向服务器发送 5 次请求，既是对资源的浪费，又会造成漫长的等待。然而我们更好的方式是将这些请求打包，委托以另外一个对象发送，当得到响应数据时再通过委托对象拆包数据分发给各个模块，比如”

```
var Deal = {
    banner : function() {
        // 处理 banner 模块逻辑
    },
    aside : function(res) {
        // 处理 aside 模块逻辑
    },
    article : function(res) {
        // 处理 article 模块数据
    },
    member : function(res) {
        // 处理 member 模块数据
    },
    message : function(res) {
        // 处理 message 模块数据
    }
}
$.get('./deal.php?', function(res) {
    // 数据拆包分发
    for(var i in res) {
        Deal[i] && Deal[i](res[i]);
    }
});
```

“这样我们通过一个请求便可将 5 次请求业务一次性做完，这既节省流量又节约了多次请求对时间的开销。”

下章剧透

委托模式优化了页面中的事件绑定，对于未来元素的事件绑定是现有技术所做不到的。Web 开发中也常常遇到数据存储问题，对数据的存储又有哪些问题呢？下一章我们一见分晓。

忆之获

委托模式是通过委托者将请求委托给被委托者去处理实现的。因此委托模式解决了请求与委托者之间的耦合。通过被委托者对接收到的请求的处理后，分发给相应的委托者去处理。

在 JavaScript 中，委托模式已经得到很广泛的应用，尤其在处理事件上，当然委托模式是一种基础技巧，因此也同样在其他设计模式中被引用，如状态模式中状态对象对接收的状态处理，策略模式中策略对象对接收到的算法处理，命令模式中命令对象对接收到的命令处理等。

我问你答

用委托模式封装一个事件委托方法，事件委托方法使用如下：

```
delegate(document.body, 'button', 'click', function(){
    console.log('委托成功! ');
})
```

第 29 章 数据管理器——数据访问对象模式

数据访问对象模式（Data access object-DAO）：抽象和封装对数据源的访问与存储，DAO通过对数据源链接的管理方便对数据的访问与存储。

新品上线，用户总免不了对其有陌生感，人性化的用户引导（一种指引用户使用页面功能的说明，常见于页面中）势在必行，而用户引导又往往涉及数据存储，然而前端对于数据的存储又会有哪些问题呢？

29.1 用户引导

“小白，咱们新版活动页面的用户引导要上一周，不过后端同学近期比较忙，没办法支持我们这个小需求，所以你把用户引导展现情况记录在本地吧，用户看过就看不出了”项目经理语重心长地对小白说。

于是小白开始忙碌项目经理提供的需求，准备将相关数据进行本地存储。可是没过一会儿发现了问题，每次添加一组数据总担心会不会覆盖他人数据，或被他人所覆盖，添加一条数据为兼容所有浏览器总要处理好多事情……总之一堆问题压了过来。

“小铭，创建一个使用本地存储有没有什么方式呀。能不能介绍一个？”小白对小铭说。

“你是不是感到与以往在后端操作数据库的习惯不一样了？”小铭问。

“可不，而且还不能像服务器端数据库那样得到对本地存储使用的结果，比如我添加一个数据，不知道有没有成功，还有我与他人添加数据时都在一个数据库，也不知道会不会相互影响。”小白答道。

“这样，你可以建立一个数据访问对象类 DAO，这样以后每个人在自己模块就可以通过创建这个 DAO 类的实例来使用了，这样做你不仅可以方便管理自己本地存储库还可以方便大家呀”小铭建议说。

29.2 数据访问对象类

“数据访问对象类？是说让我封装本地存储数据库么？我该怎么做？”小白问。

“数据访问对象类的确是对本地存储的一次封装，要创建 DAO 类，首先你要了解当前你的需求是什么。当然最好也要顾虑一下以后他人的所需，这样你创建的 DAO 类将更加健全。”

“我只需要对数据进行增、删、改、查这些操作。有了这些操作方法，我想我的需求基本可以实现了。”小白答道。

“哦，可是当别人再想使用本地存储呢，他们会知道你保存的数据吗？”

“应该不会的。”

“所以你应该关注这里，不要只顾着自己。然而本地存储数据库不同于服务器端关系型数据库，它是将数据保存在 `localStorage` 这个对象里。`localStorage` 相当于一个大容器，对于同一个站点，它里面根本没有分割库，所以别人使用 `localStorage` 的时候和你用的是一个，所以你应该将每次存储的数据字段前面添加前缀标识来‘分割’`localStorage` 存储。本地存储对数据的保存实际上是 `localStorage` 的一个字符串属性。有时对于本地存储来说，了解他的存储时间是很有必要的，它能方便日后对数据的管理（如定期清除），因此你可以添加一个时间戳，但对于每个人存储的数据内容不同，时间戳与将要存储的数据都是属性字符串，所以他们之间要设置一个拼接符。下面我们一起创建一个 DAO 类。”

```
/*
 * 本地存储类
 * 参数 preId      本地存储数据库前缀
 * 参数 timeSign    时间戳与存储数据之间的拼接符
 *
 */
var BaseLocalStorage = function(preId, timeSign) {
  // 定义本地存储数据库前缀
  this.preId = preId;
  // 定义时间戳与存储数据之间的拼接符
  this.timeSign = timeSign || '-|';
}
```

29.3 数据操作状态

“之前你说想知道数据的操作状态，那么我们可以在 DAO 类的内部保存操作返回状态供日后使用时的调用。我们还需将本地存储服务的引用保存在 DAO 类的内部以方便我们使用。当然 DAO 还要提供对数据库的增删改查操作接口方法。”

```
// 本地存储类原型方法
BaseLocalStorage.prototype = {
  // 操作状态
  status : {
```

```

        SUCCESS : 0,      // 成功
        FAILURE : 1,     // 失败
        OVERFLOW : 2,    // 溢出
        TIMEOUT : 3      // 过期
    },
    // 保存本地存储链接
    storage : localStorage || window.localStorage,
    // 获取本地存储数据库数据真实字段
    getKey : function(key){
        return this.preId + key;
    },
    // 添加(修改)数据
    set : function(key, value, callback, time){
        // 省略操作
    },
    // 获取数据
    get : function(key, callback){
        // 省略操作
    },
    // 删除数据
    remove : function(key, callback){
        // 省略操作
    }
}

```

“小铭，我们只需要一一实现 set、get、remove 3 个方法，我们的 DAO 对象功能就完整了吧。”

29.4 增添数据

“嗯。首先我们实现 set 方法，不过需要注意的是，在 set 方法中，我们要注意在字段中添加时间戳，并且我们向本地存储中添加数据实质上是调用 localStorage 的 setItem 方法。最后我们还要执行回调函数并将操作的结果传入回调函数中。”

```

*****
 * 添加(修改)数据
 * 参数 key    : 数据字段标识
 * 参数 value  : 数据值
 * 参数 callback: 回调函数
 * 参数 time   : 添加时间
 ****/
set : function(key, value, callback, time){
    // 默认操作状态时成功
    var status = this.status.SUCCESS,
        // 获取真实字段
        key = this.getKey(key);
    try{
        // 参数时间参数时获取时间戳
        time = new Date(time).getTime() || time.getTime();
    }catch(e){
        // 为传入时间参数或者时间参数有误获取默认时间：一个月
    }
}

```

```

        time = new Date().getTime() + 1000 * 60 * 60 * 24 * 31;
    }
    try{
        // 向数据库中添加数据
        this.storage.setItem(key, time + this.timeSign + value);
    }catch(e){
        // 溢出失败，返回溢出状态
        status = this.status.OVERFLOW;
    }
    // 有回调函数则执行回调函数并传入参数操作状态，真实数据字段标识以及存储数据值
    callback && callback.call(this, status, key, value);
}

```

29.5 查找数据

“有了设置数据方法，我们通过存储字段获取数据的方法也就简单了，还是调用localStorage的getItem方法获取数据。不过需要注意的是，这里兼容了4种查询数据情况，第一种情况是该字段的数据本来就不存在，这样应该返回失败状态；第二种情况操作成功但是没有获取到值，此时也应该返回失败状态；第三种情况是获取到值，但是时间已经过期，此时应该删除该数据；最后一种情况是成功获取数据并可成功返回。”

```

*****
 * 获取数据
 * 参数 key      : 数据字段标识
 * 参数 callback: 回调函数
 ****/
get : function(key, callback){
    // 默认操作状态时成功
    var status = this.status.SUCCESS,
        // 获取
        key = this.getKey(key),
        // 默认值为空
        value = null,
        // 时间戳与存储数据之间的拼接符长度
        timeSignLen = this.timeSign.length,
        // 缓存当前对象
        that = this,
        // 时间戳与存储数据之间的拼接符起始位置
        index,
        // 时间戳
        time,
        // 最终获取的数据
        result;
    try{
        // 获取字段对应的数据字符串
        value = that.storage.getItem(key);
    }catch(e){
        // 获取失败则返回失败状态，数据结果为null
        result = {
            status : that.status.FAILURE,
            value : null
        }
    }
    // 将结果放入回调函数
    callback(result);
}

```

```

    };
    // 执行回调并返回
    callback && callback.call(this, result.status, result.value);
    return result;
}
// 如果成功获取数据字符串
if(value){
    // 获取时间戳与存储数据之间的拼接符起始位置
    index = value.indexOf(that.timeSign);
    // 获取时间戳
    time = +value.slice(0, index);
    // 如果时间为过期
    if(new Date(time).getTime() > new Date().getTime() || time == 0){
        // 获取数据结果 (拼接符后面的字符串)
        value = value.slice(index + timeSignLen);
    }else{
        // 过期则结果为 null
        value = null,
        // 设置状态为过期状态
        status = that.status.TIMEOUT;
        // 删除该字段
        that.remove(key);
    }
}else{
    // 未获取数据字符串状态为失败状态
    status = that.status.FAILURE;
}
// 设置结果
result = {
    status : status,
    value : value
};
// 执行回调函数
callback && callback.call(this, result.status, result.value);
// 返回结果
return result;
}

```

29.6 删 除 数据

“小白，设置和获取方法我都说完了，这回你来想一想删除方法 remove 该如何实现呢？”

“嘿嘿，有了前面两个方法做参考，删除数据方法就变得简单了，首先删除应该用 localStorage 的 removeItem 方法。然后执行时应该有 3 种情况吧，第一种情况是没有改字段，这样操作结果应该是失败状态，第二种情况是删除（调用 removeItem 方法）时发生异常，这样应该也是失败状态，最后一种情况当然就是成功完成了。没问题吧？”小白答道。

“嗯，不过这里对于 3 种情况可做优化，即所有情况的回调函数放在最后统一执行，像下面这样”

```

*****
* 删除数据

```

```

* 参数 key    : 数据字段标识
* 参数 callback: 回调函数
****/
remove : function(key, callback) {
    // 设置默认操作状态为失败
    var status = this.status.FAILURE,
        // 获取实际数据字段名称
        key = this.getKey(key),
        // 设置默认数据结果为空
        value = null;
    try{
        // 获取字段对应的数据
        value = this.storage.getItem(key);
    }catch(e){}
    // 如果数据存在
    if(value){
        try{
            // 删除数据
            this.storage.removeItem(key);
            // 设置操作成功
            status = this.status.SUCCESS;
        }catch(e){}
    }
    // 执行毁掉 注意传入回调函数中的数据值: 如果操作状态成功则返回真实的数据结果, 否则返回空
    callback && callback.call(this, status, status > 0 ? null :
value.slice(value.indexOf(this.timeSign) + this.timeSign.length))
}

```

“这样我们的整个 DAO 类就完成了，下面你测试使用一下吧。”

29.7 检验 DAO

“小铭，刚刚再次浏览了一遍我们写的 set、get、remove 三个方法，发现为什么有的方法默认操作状态是成功，如 set 和 get，有的方法默认操作状态是失败呢，如 remove”小白问。

“通常我们在设置默认状态的时候应该保证该默认状态尽可能多地出现，这样我们的代码执行效率也会更高一些，这也体现一个工程师对自己设计的代码的一个全局把控能力。”

“哦，这样，好吧，我来测试一下我们创建的这个 DAO 类吧，首先还是创建一个本地存储库。”

```
var LS = new BaseLocalStorage('LS__');
```

“然后创建变量 a，然后获取它，然后删除两次看看结果，再获取这个变量 a。看看结果吧。”

```

LS.set('a', 'xiao ming', function(){console.log(arguments)}); // [0,
"LS_a", "xiao ming"]
LS.get('a', function(){console.log(arguments)});           // [0, "xiao ming"]
LS.remove('a', function(){console.log(arguments)});         // [0, "xiao ming"]
LS.remove('a', function(){console.log(arguments)});         // [1, null]
LS.get('a', function(){console.log(arguments)});           // [1, null]

```

29.8 MongoDB

“小白，其实数据访问对象模式更适合与服务器端的数据库操作，比如我们在 nodejs 中操作 MongoDB 时，便可创建 MongoDB 数据库的数据访问对象。”

“听起来很不错，可是我还没用过你说的这些技术，你能简单举个例子吗？”

“当然，比如我们简单封装对某个数据库的增删改查操作，体验一下这类模式的优越性你就会深深理解它。”

“太好了，赶紧教教我吧。”小白求知若渴。

29.9 在 nodejs 中写入配置项

“在创建 MongoDB 数据库的数据访问对象之前我们先做些准备工作，比如我们将应用的数据库名、主机名、端口号等写在配置文件中方便日后使用。”

```
/* config.js */
// 将配置数据输出
module.exports = {
  // 数据库相关配置数据
  DB : {
    db : 'demo',          // 数据库名称
    host : 'localhost',   // 主机名
    port : 27017          // 端口号
  }
}
```

29.10 连接 MongoDB

“有了配置信息我们就可以连接对应的数据库。”

```
/* db.js */
// 引用 mongodb 模块
var mongodb = require('mongodb');
// 引用配置模块的数据库配置信息
var config = require('../config').DB;
// 创建数据库对象
var d = new mongodb.Db(
  config.db,                  // 数据库名称
  new mongodb.Server(
    config.host,               // 主机名
    config.port,               // 端口号
    {auto_reconnect : true}    // 自动连接
  ),
  {safe : true}                // 安全模式
);
// 输出数据访问对象
exports.DB = function() {}
```

29.11 操作集合

“现在我们就可以通过操作数据库对象（d）来完成数据访问对象（DB）的增删改查操作了。由于每次执行增删改查操作时，都要打开数据库操作集合（可看作关系数据库中的表），所以我们抽象出一个打开数据库操作集合的方法 connect。”

```
/**
 * 打开数据库，操作集合
 * @param col 集合名
 * @param fn 操作方法
 */
function connect(col, fn) {
    // 打开数据库
    d.open(function(err, db) {
        // 打开数据库报错则抛出错误
        if(err)
            throw err;
        // 操作集合
        else
            db.collection(col, function(err, col) {
                // 操作集合报错则抛出错误
                if(err)
                    throw err;
                // 执行操作
                else
                    fn && fn(col, db);
            });
    });
}
```

“我们要执行的增删改查操作要在 connect 的 fn 中实现。”

```
exports.DB = function(col) {
    return {
        // 插入数据
        insert : function(){}
        // 删除数据
        remove : function(){}
        // 更新数据
        update : function(){}
        // 查找数据
        find : function(){}
    }
}
```

29.12 插入操作

“下面我们要依次实现这 4 个操作，为了能让你看懂，这里我简化这些操作，当然你学习的主要目的是要理解数据访问对象模式的思想。首先对于插入数据，我们需要有插入的数据项，以及操作成功与失败回调函数，并在操作成功后将其关闭。”

```
/**
 * 插入数据
```

```

* @param data      插入数据项
* @param success  操作成功回调函数
* @param fail     操作失败回调函数
*/
insert : function(data, success, fail){
    // 打开数据库操作 col 集合
    connect(col, function(col, db){
        // 向集合中插入数据
        col.insert(data, function(err, docs){
            // 失败，抛出插入错误
            if(err)
                fail && fail(err);
            // 成功，执行成功回调函数
            else
                success && success(docs);
            // 关闭数据库
            db.close();
        });
    });
}

```

“小白，现在你可以新建一个文件，然后引用该数据访问对象，向数据库中添加数据了。”

```

/* test.js */
var DB = require('./db').DB;      // 引用数据访问对象模块
var user = DB('user');          // 操作 user 集合
// 向集合中插入一条数据
user.insert({name : '小白', nick : '雨夜清荷'}, function(docs){
    console.log(docs); // [{name : '小白', nick : '雨夜清荷', _id : 54e956410017a3fc06195be9}] (_id 为数据项的索引值)
});

```

29.13 删除操作

“接下来我们要实现 remove 操作方法，通过传入删除数据项，在集合中删除该数据，当然，操作成功我们可以得到删除数据项的长度。”

```

/****
 * 删除数据
* @param data 删除数据项
* @param success 成功回调
* @param fail 失败回调
*/
remove : function(data, success, fail){
    // 打开数据库操作 col 集合
    connect(col, function(col, db){
        // 在集合中删除数据项
        col.remove(data, function(err, len){
            if(err)
                fail && fail(err);
            else
                success && success(len);
            db.close();
        });
    });
}

```

“此时我们如果想删除刚才添加的数据我们可通过下面方法来完成”。

```
user.remove({name : '小白'}, function(len){
    console.log(len); // 1 (删除数据项长度)
})
```

29.14 更新操作

“对于更新操作，我们首先要通过给定条件筛选出具备这些条件的数据，然后对这些数据执行更新操作。当然执行成功后我们获取到的依旧是更新数据项长度。”

```
/*
 * 更新数据
 * @param con      筛选条件
 * @param doc      更新数据项
 * @param success  成功回调
 * @param fail     失败回调
 */
update : function(con, doc, success, fail){
    connect(col, function(col, db){
        // 在集合中更新数据项
        col.update(con, doc, function(err, len){
            if(err)
                fail && fail(err);
            else
                success && success(len);
            db.close();
        });
    });
}
```

“比如将刚才在数据库中添加的数据中 name 为'小白'的数据项内容更新为{name : '小白', nick : '雨夜'}”

```
user.update({name : '小白'}, {name : '小白', nick : '雨夜'}, function(len){
    console.log(len); // 1
})
```

29.15 查找操作

“有了上面的经验，查找操作的实现容易许多，为了简化操作，我们只提供筛选条件参数进行数据查找，这样即可获取所有满足条件的数据项。当操作执行成功后我们可以获取这些数据。”

```
/*
 * 查找数据
 * @param con      筛选条件
 * @param success  成功回调
 * @param fail     失败回调
 */
find : function(con, success, fail){
    connect(col, function(col, db){
        // 在集合中查找数据
        col.find(con).toArray(function(err, docs) {
```

```

        if(err)
            fail && fail(err);
        else
            success && success(docs);
        db.close();
    });
});
}

```

“最后我们找一找我们向集合中添加的 name 值为'小白'的数据项。”

```

user.find({name : '小白'}, function(doc){
    console.log(doc)// [{name : '小白', nick : '雨夜清荷', _id : 54e956410017a3fc06195be9}]
});

```

29.16 操作其他集合

“有了数据访问对象 DB，我们在操作其他集合时变得容易，比如我们操作 book 时，直接在 test.js 文件中添加如下代码即可。”

```

var book = DB('book');
book.insert({title : 'JavaScript 设计模式', type : 'JavaScript'})

```

下章剧透

数据访问对象模式可以方便我们对前端存储的管理，统一我们的使用方式，并且使用时更方便。事件的触发往往是一瞬间的，所以页面中的某些交互往往会造成事件重复触发，如何解决这类问题来优化用户体验呢？赶紧浏览下一章的内容吧。

忆之获

数据访问对象（DAO）模式即是对数据库的操作（如简单的 CRUD 创建、读取、更新、删除）进行封装，用户不必为操作数据库感到苦恼，DAO 已经为我们提供了简单而统一的操作接口。并且对于使用者来说，不必了解 DAO 内部操作是如何实现的，有时甚至不必了解数据库是如何操作的。对于后端数据库来说（如 MongoDB），DAO 对象甚至会保留对数据库的链接，这样我们每次操作数据库时不必一次次地向数据库发送链接请求。

DAO 是一个对象，因此它封装了属性和方法，并通过这些属性与方法管理着数据库。因此有时为了实现需求我们还可以对 DAO 对象进行拓展。但是更佳实践是对 DAO 做一层试用于你自己的封装，这样在团队开发中不会影响到他人的使用。

我问你答

对于不支持本地存储的浏览器，我们有时会通过 cookie 或者浏览器 userData 存储区存储数据，那么你能否对我们的本地存储 DAO 重构，使其兼容更多的浏览器呢？

第30章 执行控制——节流模式

节流模式（Throttler）：对重复的业务逻辑进行节流控制，执行最后一次操作并取消其他操作，以提高性能。

重复的业务逻辑真的让人很厌恶的，但其中往往蕴含着可被优化的空间。在 JavaScript 的编程世界里，由于功能的实现常常寄托于浏览器，所以任何性能的优化都是很有价值的。

30.1 返回顶部

“小铭，你看我为返回顶部按钮添加的动画，每次拖动页面滚动条时他都在不停地抖动。这是什么原因呀？该如何解决呢？”小白问。

“你是监听 scroll 事件而为返回顶部按钮添加的动画吧。由于拖动页面滚动条时不停地触发 scroll 事件所以返回顶部按钮在不停地执行动画。解决这个问题很容易，你写个节流就可以了”小铭直截了当地回答。

“节流？我还是头一次听说呢。”小白追问。

“节流很简单，你可以简单理解为屏蔽重复的事情（业务逻辑）只执行最后一次。比如你做的返回顶部按钮，每当你拖动滚动条都会多次触发浏览器中的 scroll 事件，这正是造成你为返回顶部按钮添加的动画执行多次的原因，才会返回顶部按钮不停地抖动。”

“原来是这样，我说怎么一直在动，原来是浏览器多次触发了 scroll 事件而执行了多次该事件回调函数。可是我该如何用节流模式解决这个问题呢。”小白问。

30.2 节流器

“节流模式我们通常又称之为函数的节流。所以要解决你的问题我们首先要实现一个节流器。”

```
// 节流器
var throttle = function() {
    // 获取第一个参数
    var isClear = arguments[0], fn;
```

```

// 如果第一个参数是 boolean 类型那么第一个参数则表示是否清除计时器
if(typeof isClear === 'boolean'){
    // 第二个参数则为函数
    fn = arguments[1];
    // 函数的计时器句柄存在，这清除该计时器
    fn._throttleID && clearTimeout(fn._throttleID);
    // 通过计时器延迟函数的执行
} else{
    // 第一个参数为函数
    fn = isClear;
    // 第二个参数为函数执行时的参数
    param = arguments[1];
    // 对执行时的参数适配默认值，这里我们用到以前学过的 extend 方法
    var p = extend({
        context : null,    // 执行函数执行时的作用域
        args : [],         // 执行函数执行时的相关参数（IE 下要为数组）
        time : 300         // 执行函数延迟执行的时间
    }, param);
    // 清除执行函数计时器句柄
    arguments.callee(true, fn);
    // 为函数绑定计时器句柄，延迟执行函数
    fn._throttleID = setTimeout(function() {
        // 执行函数
        fn.apply(p.context, p.args)
    }, p.time)
}
}

```

小铭接着说：“构造节流器的思路是这样的：首先节流器应该能做两件事情，第一件事情就是清除将要执行的函数，此时要对节流器传递两个参数（是否清除， 执行函数），如果第一个参数为 true，则表示清除将要执行的函数。同时会判断第二个参数（执行函数）有没有计时器句柄，有则清除计时器。节流器能做的第二件事情就是延迟执行函数。此时要传递两个参数（执行函数，相关参数）。在节流器内部首先要为执行函数绑定一个计时器句柄，来保存该执行函数的计时器，对于第二个参数——相关参数来说，大致包括 3 个部分，执行函数在执行时的作用域、执行函数的参数、执行函数延迟执行的时间。”

“像你说的那样使用就可以了吗？”小白问。

事件节流

“嗯，比如对于你创建的返回顶部按钮，我们可以像下面这样优化”。

```

// 首先引入 jquery.js 与 easing.js 方便返回顶部动画实现
// 返回顶部按钮动画
function moveScroll(){
    var top = $(document).scrollTop();
    $('#back').animate({top : top + 300}, 400, 'easeOutCubic')
}
// 监听页面滚动条事件
$(window).on('scroll', function(){
    // 节流执行返回顶部按钮动画
    throttle(moveScroll);
})

```

30.3

优化浮层

“拖动浏览器滚动条时，按钮果然不抖动了，实在太完美了，这么好用的节流器是不是有很广阔的应用市场呀。”小白惊叹。

“的确，比如优化一些鼠标滑过元素而展现浮层的交互。按照以前的交互方式，当鼠标一不小心移出浮层时，浮层就会消失。有时候无意间移入元素上面就会导致浮层展现。这两种交互体验都是很糟糕的体验，所以为了优化这种体验我们也可以使用节流模式。”小铭说。

“正好我这有一个手机扫二维码的需求，当鼠标移动微信或者新浪微博的 icon 上时显示出对应的二维码浮层。”

“借此机会你可以实践一下新学到的函数节流器呀。”

“是呀，首先创建一个如下结构的视图，容器内包含两个 icon，icon 后面是一个 div 容器，容器里面包含着两个二维码图片”。

```
<div id="icon" class="icon">
  <ul class="icon">
    <li class="weixin"></li>
    <li class="weibo"></li>
  </ul>
  <div class="">
    
    
    <span class="arrow"><em></em></span>
  </div>
</div>
```

“小白，接下来要分析你的交互流程，想一想哪些交互需要用到节流器帮助你增强用户体验”。小铭补一句建议。

30.4

创建浮层类

“不过我想先把整个框架打出来，然后再分析交互的逻辑吧。”

```
// 外观模式封装获取元素方法
function $(id){return document.getElementById(id)}
function $tag(tag, container){
  container = container || document;
  return container.getElementsByTagName(tag)
}
// 浮层类
var Layer = function(id){
  // 获取容器
  this.container = $(id);
  // 获取容器中的浮层容器
  this.layer = $tag('div', this.container)[0];
```

```

    // 获取 icon 容器
    this.lis = $tag('li', this.container);
    // 获取二维码图片
    this.imgs = $tag('img', this.container);
    // 绑定事件
    this.bindEvent();
}
Layer.prototype = {
    // 绑定交互事件
    bindEvent : function(){
        // .....
    },
    // 事件绑定方法
    on : function(ele, type, fn){
        ele.addEventListener ? ele.addEventListener(type, fn, false) :
        ele.attachEvent('on' + type, fn);
        return this;
    }
}

```

30.5 添加节流器

“对于交互部分，当鼠标光标划入容器，为解除用户误划入操作，浮层不应立即展现，此时应该使用节流器延迟浮层展现方法的执行。而鼠标光标移出容器的时间比较短的时候，可能是用户不小心移出的，此时应该使用节流器延迟隐藏浮层。”

```

// 绑定交互事件
bindEvent : function(){
    // 缓存当前对象
    var that = this;
    // 隐藏浮层
    function hideLayer(){
        that.layer.className = '';
    }
    // 显示浮层
    function showLayer(){
        that.layer.className = 'show';
    }
    // 鼠标光标移入事件
    that.on(that.container, 'mouseenter', function(){
        // 清除隐藏浮层方法计时器
        throttle(true, hideLayer);
        // 延迟显示浮层方法
        throttle(showLayer);
    })
    // 鼠标光标移出事件
    }).on(that.container, 'mouseleave', function(){
        // 延迟浮层隐藏方法
        throttle(hideLayer);
        // 清除显示浮层方法计时器
        throttle(true, showLayer);
    });
    // 遍历 icon 绑定事件
}

```

```

for(var i = 0; i < that.lis.length; i++) {
    // 自定义属性 index
    that.lis[i].index = i;
    // 为每一个 li 元素绑定鼠标移入事件
    that.on(that.lis[i], 'mouseenter', function() {
        // 获取自定义属性 index
        var index = this.index;
        // 排除所有 img 的 show 类
        for(var i = 0; i < that.imgs.length; i++) {
            that.imgs[i].className = '';
        }
        // 为目标图片设置 show 类
        that.imgs[index].className = 'show';
        // 从新定义浮层位置
        that.layer.style.left = -22 + 60 * index + 'px';
    })
}
}
}

```

30.6 图片的延迟加载

“小白，节流模式除了适用于处理浮层显隐之外，有时也可用于对图片的延迟加载优化。我们知道之所以对页面图片进行延迟加载，是由于浏览器加载线程有限造成的。当页面首屏加载过多图片时，会严重影响其他必要文件的加载（如 css 文件，js 文件等），这会造成糟糕的用户体验，同时也会使页面 load 事件处理逻辑推迟执行。同样道理，当页面篇幅很长，并且图片较多的时候，用户快速将页面拉直底部时，由于上面的图片会优先加载造成底部图片加载推迟，这种体验也是很不好的，此时我们就可以用节流模式来处理这些加载逻辑，使可视范围内的图片优先加载。”

“哦，听上去很高端大气上档次呀！”小白惊叹。

“这种想法的实现也很容易，首先同图片延迟加载的流程不变，都是监听页面的 scroll 与 resize 事件，只不过要对这两个事件节流处理，执行一次交互中最后一次执行的事件回调函数即可。那么这里的‘最后一次执行的事件’的捕获就要依靠节流器了。当然对图片的处理流程基本一样，将所有图片做缓存。检测每张图片是否处在页面的可视范围内，如果在可视范围，则将此图片加载，并将此张图片从图片的缓存中清除”。

30.7 延迟加载图片类

```

/**
 * 节流延迟加载图片类
 * param id 延迟加载图片的容器 id
 * 注：图片格式如下 
 ****/
function LazyLoad(id) {
    // 获取需要节流延迟加载图片的容器

```

```

this.container = document.getElementById(id);
// 缓存图片
this.imgs = this.getImg();
// 执行逻辑
this.init();
}
// 节流延迟加载图片类原型方法
LazyLoad.prototype = {
  // 起始执行逻辑
  init : function() {},
  // 获取延迟加载图片
  getImg : function() {},
  // 加载图片
  update : function() {},
  // 判断图片是否在可视范围内
  shouldShow : function(i) {},
  // 获取元素页面中的纵坐标位置
  pageY : function(element) {},
  // 绑定事件 (简化版)
  on : function(element, type, fn) {},
  // 为窗口绑定 resize 事件与 scroll 事件
  bindEvent : function(){}
}

```

“对于节流延迟加载图片类的原型方法 init 应该做两件事，初始化图片加载（即执行 update 方法）和为窗口绑定事件（bindEvent 方法）。”

```

// 起始执行逻辑
init : function(){
  // 加载当前视图图片
  this.update();
  // 绑定事件
  this.bindEvent();
}

```

30.8 获取容器内的图片

“对于获取容器内图片的 getImg 方法需要注意的是，为了方便操作获取的图片元素集合（类数组），需要将其转化成数组，由于在 IE 中对获取到的元素集合直接执行数组方法 slice 会报错，故通过遍历每一个元素，并将其加入新数组中返回，来显性创建数组。”

```

// 获取延迟加载图片
getImg : function() {
  // 新数组容器
  var arr = [];
  // 获取图片
  var imgs = this.container.getElementsByTagName('img');
  // 将获取的图片转化为数组(IE 下通过 Array.prototype.slice 会报错)
  for(var i = 0, len = imgs.length; i < len; i++){
    arr.push(imgs[i]);
  }
  return arr;
}

```

30.9**加载图片**

“对于加载图片方法 `update`，需要遍历每一个图片元素，如果处在可视区域内则加载并将其在图片缓存中清除。”

```
// 加载图片
update : function(){
    // 如果图片都加载完成，返回
    if(!this.imgs.length){
        return;
    }
    // 获取图片长度
    var i = this.imgs.length;
    // 遍历图片
    for(--i; i >= 0; i--){
        // 如果图片在可视范围内
        if(this.shouldShow(i)){
            // 加载图片
            this.imgs[i].src = this.imgs[i].getAttribute('data-src');
            // 清除缓存中的此图片
            this.imgs.splice(i, 1);
        }
    }
}
```

30.10**筛选需加载的图片**

“对于判断图片是否在可视范围内 `shouldShow` 方法，是判断图片的上下边坐标位置是否符合下列条件（由于页面左边关系，对于 y 坐标，由上至下一次增大，对于 x 坐标，由左至右一次增大）：图片底部高度大于可视视图顶部高度并且图片底部高度小于可视视图底部高度，或者图片顶部高度大于可视视图顶部高度并且图片顶部高度小于可视视图底部高度。”

```
// 判断图片是否在可视范围内
shouldShow : function(i){
    // 获取当前图片
    var img = this.imgs[i],
        // 可视范围内顶部高度(页面滚动条 top 值)
        scrollTop = document.documentElement.scrollTop || document.body.scrollTop,
        // 可视范围内底部高度
        scrollBottom = scrollTop + document.documentElement.clientHeight;
    // 图片的顶部位置
    imgTop = this.pageY(img),
    // 图片的底部位置
    imgBottom = imgTop + img.offsetHeight;
    // 判断图片是否在可视范围内：图片底部高度大于可视视图顶部高度并且图片底部高度小于可视视图底部高度，或者图片顶部高度大于可视视图顶部高度并且图片顶部高度小于可视视图底部高度
    if(imgBottom > scrollTop && imgBottom < scrollBottom || (imgTop > scrollTop && imgTop < scrollBottom))
        return true;
}
```

```
// 不满足上面条件则返回 false
return false;
}
```

30.11 获取纵坐标

“对于获取图片元素纵坐标方法 pageY 这是通过元素一级一级遍历其父元素，并累加每一级元素 offsetTop 值获取的。”

```
// 获取元素页面中的纵坐标位置
pageY : function(element) {
    // 如果元素有父元素
    if(element.offsetParent){
        // 返回元素+父元素高度
        return element.offsetTop + this.pageY(element.offsetParent);
    }else{
        //否则返回元素高度
        return element.offsetTop;
    }
}
```

“在这里为了简化代码，我们先简单实现 on 绑定事件方法。”

```
// 绑定事件（简化版）
on : function(element, type, fn) {
    if(element.addEventListener){
        addEventListener(type, fn, false);
    }else{
        element.attachEvent('on' + type, fn, false);
    }
}
```

30.12 节流器优化加载

“最后一个方法 bindEvent 绑定事件则是对页面的 scroll 与 resize 事件的监听，为检测每一次交互中事件的最后一次执行，故需要对事件的回调函数做节流处理。”

```
// 为窗口绑定 resize 事件与 scroll 事件
bindEvent : function(){
    var that = this;
    this.on(window, 'resize', function(){
        // 节流处理更新图片逻辑
        throttle(that.update, {context : that});
    });
    this.on(window, 'scroll', function(){
        // 节流处理更新图片逻辑
        throttle(that.update, {context : that});
    })
}
```

30.13 大功告成

“最终我们实例化 LazyLoad 类来延迟加载 container 容器内的所有图片，并通过节流器实现加载优化。”

```
// 延迟加载 container 容器内的图片
new LazyLoad('container');
```

30.14 统计打包

“节流模式优化了图片加载时的用户体验，对于一些前后端的数据请求有时也可通过节流模式打包来优化请求次数。比如在统计中（尤其对于鼠标移入移出等频发性事件），我们经常监听事件触发次数，当触发次数达到某一值时才发送请求。”

```
// 打包统计对象
var LogPack = function() {
    var data = [], // 请求缓存数组
        MaxNum = 10, // 请求缓存最大值
        itemSplitStr = '|', // 统计项统计参数间隔符
        keyValueSplitStr = '**', // 统计项统计参数键值对间隔符
        img = new Image(); // 请求触发器，通过图片 src 属性实现简单的 get 请求
    // 发送请求方法
    function sendLog(){}
    // 统计方法
    return function(param){
        // 如果无参数则发送统计
        if(!param){
            sendLog();
            return;
        }
        // 添加统计项
        data.push(param);
        // 如果统计项大于请求缓存最大值则发送统计请求包
        data.length >= MaxNum && sendLog();
    }
}();
```

30.15 组装统计

“你可以看到简单的 img 图片即可作为请求触发器，因为当为 img 赋值 src 属性时即向服务器发送了 1 次 get 请求。然而每次调用 LogPack 方法时不会发送统计，而是当缓存的统计数组长度大于 MaxNum 数值时，才会发送统计。下面我们要实现 sendLog 方法，在该方法中我们要做 3 件事，首先我们要从统计缓存中截取 MaxNum 统计项，然后将截取的统计项打包成 1 个字符串，最后通过请求触发器发送请求。”

```
// 发送请求方法
function sendLog() {
    // 请求参数
    var logStr = '';
    // 截取 MaxNum 个统计项发送
    fireData = data.splice(0, MaxNum);
    // 遍历统计项
    for(var i = 0, len = fireData.length; i < len; i++) {
        // 添加统计项顺序索引
        logStr += 'log' + i + '=';
        // 遍历统计项内的统计参数
        for(var j in fireData[i]){
            // 添加统计项参数键 + 间隔符 + 值
            logStr += j + keyValueSplitStr + fireData[i][j];
            // 添加统计项统计参数间隔符
            logStr += itemSplitStr;
        }
        // &符拼接多个统计项
        logStr = logStr.replace(/\|$/, '') + '&';
    }
    // 添加统计项打包长度
    logStr += 'logLength=' + len;
    // 请求触发器发送统计
    img.src = 'a.gif?' + logStr;
}
}
```

测试统计

“假设我们的页面中有个 button 元素<button id="btn">test</button>, 我们为 btn (在某些浏览器中我们可以直接通过 id 名称使用该元素)添加事件, 在事件的回调函数中我们发送统计。”

```
// 点击统计
btn.onclick = function(){
    LogPack({
        btnId : this.id,
        context : this.innerHTML,
        type : 'click'
    });
};

// 点击统计
btn.onmouseover = function(){
    LogPack({
        btnId : this.id,
        context : this.innerHTML,
        type : 'mouseover'
    });
};
```

“打开浏览器的网络我们即可看到当多次触发事件后才会发送一个打包后的统计, 这样即可避免多次请求对流量的浪费, 当然最重要的是减轻了服务器对统计请求的承载压力。”

下章剧透

节流模式解决了页面中的一些简单交互造成事件重复触发的问题。有时也可屏蔽掉一些无意触发的事件。JavaScript 中的 DOM 操作为我们创建视图提供了大量的方法，但往往是操作大量节点的。不仅可读性差而且消耗资源。下一章将为我们展示一种创建视图的优质的模式，会是什么？敬请期待吧。

忆之获

对于 DOM 的操作，常常会占用大量的内存资源和 cpu 处理时间。甚至大量的 DOM 操作在一些浏览器中也很可能导致浏览器的崩溃。由于 JavaScript 的单线程处理机制，导致 DOM 操作占用大量资源时会严重堵塞后面重要程序的执行。

节流模式的核心思想是创建计时器，延迟程序的执行。这也使得计时器中回调函数的操作异步执行（这里的异步执行并不是说 JavaScript 是多线程语言，JavaScript 从设计之初就是单线程语言，异步只是说脱离原来程序执行的顺序，看上去，异步程序像是在同时执行。但是某一时刻，当前执行的程序一定是所有异步程序（包括原程序）中的某一个）。

由此可看出节流模式主要有两点优势：第一，程序能否执行是可控的。执行前的某一时刻是否清除计时器来决定程序是否可以继续执行。第二，程序是异步的。由于计时器机制，使得程序脱离原程序而异步执行（当然随着 worker 技术的兴起，也可开启多线程模式实现），因此不会影响后面的程序的正常执行。在其他方面，比如对异步请求（ajax）应用节流，此时可以优化请求次数来节省资源。

最后，对于在节流器中对计时器的设置，有的人可能感觉直接绑定在原函数会暴露计时器句柄，这使得外部可修改。当然你也可以将节流器改造成类的形式，将计时器句柄作为私有变量存放在类内部。

我问你答

在许多搜索页面中，为提高搜索速度，经常会检测搜索框的 change 事件，向服务器发送请求，当用户会搜索一句话时，每添加一个字便会发送一次请求并渲染页面。对于某些浏览器来说，页面过于复杂时，重新渲染页面会使得页面产生“卡顿”现象，这是一种糟糕的体验，那么你能否通过节流模式来优化这一体验呢？

第31章 卡片拼图——简单模板模式

简单模板模式 (Simple template): 通过格式化字符串拼凑出视图避免创建视图时大量节点操作。优化内存开销。

快到年底了，公司准备举办一个节日活动，借机宣传一下公司的产品，所以要在活动页的主体部分展示新产品的介绍、图片等一些信息，商品很多，不过展现的模板很固定，比如带有文字描述的图片、文字列表描述、一句标新立异的宣传语等，此次重担又压在了小白的头上。

31.1 展示模板

“小铭，”小白走到小铭面前：“咱们年底的活动知道吧，领导让我做，你感觉主体展示模块如何处理比较好呢？”

“你可以在创建的视图（展示区域）上做文章，毕竟这里可能会出现多种情况，所以灵活度会高一些。”

“我也想不出什么好办法，不过我是不是应该将创建视图的这些方法用策略模式封装在一起呢？”小白问。

“嗯，这样当然好了，并且你可以根据后端传过来的数据来选择某一种方式来渲染视图。”

“对呀，这样将封装后的策略对象可作为一个单体对象一个属性存在，这样日后再想创建哪类视图直接通过单体对象引用策略对象中的创建算法就可以实现了。”

31.2 实现方案

```
// 命名空间 单体对象
var A = A || {};
// 主体展示区容器
A.root = document.getElementById('container');
// 创建视图方法集合
A.strategy = {
  'listPart' : function(){},
  'codePart' : function(){},
  'onlyTitle' : function(){},
  'guide' : function(){}
}
```

```

    // .....
}

// 创建视图入口
A.init = function(data) {
    // 根据传输的视图类型创建视图
    this.strategy[data.type](data);
}

```

“小铭，下面的任务是实现 strategy 中的每一种创建算法吧。”

“嗯，所以赶快把它们一一实现吧。”

31.3 创建文字列表视图

```

// 文字列表展示
'listPart' : function(data) {
    var s = document.createElement("div") ,           // 模块容器
        h = document.createElement("h2") ,             // 标题容器
        p = document.createElement("p") ,              // 描述容器
        ht = document.createTextNode(data.data.h2),   // 标题内容
        pt = document.createTextNode(data.data.p),    // 描述内容
        ul = document.createElement("ul") ,            // 列表容器
        ldata = data.data.li,                         // 列表数据
        li, strong, span, t, c;
    // 有 id 设置模块 id
    data.id && (s.id = data.id);
    s.className = "part"; // 设置模块类名
    h.appendChild(ht); // 将标题内容放入标题容器中
    p.appendChild(pt); // 将描述内容放入描述容器中
    s.appendChild(h); // 将标题容器插入模块容器中
    s.appendChild(p); // 将描述容器插入模块容器中
    // 遍历列表数据
    for(var i = 0, len = ldata.length; i < len; i++) {
        li = document.createElement("li");           // 创建列表项容器
        strong = document.createElement("strong");   // 创建列表项标题容器
        span = document.createElement("span");        // 创建列表项解释容器
        t = document.createTextNode(ldata[i].strong); // 创建列表项标题内容
        c = document.createTextNode(ldata[i].span);   // 创建列表项解释内容
        strong.appendChild(t); // 向列表项标题容器中插入标题内容
        span.appendChild(c); // 向列表项解释容器中插入解释内容
        li.appendChild(strong); // 向列表项中插入列表项标题
        li.appendChild(span); // 向列表项中插入列表项解释
        ul.appendChild(li); // 向列表容器中插入列表项
    }
    s.appendChild(ul); // 向模块中插入列表
    A.root.appendChild(s); // 展现模块
}

```

“怎么样，小铭？”小白自信地问。

“创建这么简单的一个视图，你的操作也太多了吧。”

小白像碰壁一样，滚烫的小心脏瞬间结成冰点：“怎么会？我倒是感觉实现得挺好的。”

“创建一个简单的视图就要操作这么多大节点，如果让你创建一个比较复杂的页面，那么很难想象你会操作每个节点多少次。你看看你创建一个 p 元素要做至少 4 件事情，首先创建标签 p 节点，然后创建文本节点，然后将文本节点插入 p 节点中，最后将 p 节点插入父容器中，当然这过程中还可能会遇到像添加类、绑定事件什么的，这样你的开销就大了。”

“那么你有什么好办法吗？”小白问。

31.4 新方案

“你可以试着创建模板呀，用数据去格式化字符串来渲染视图并插入到容器里，这样实现的方案性能上会提高许多。就像我们以前玩过的卡片拼图游戏一样，我们将一张张卡片放到相应的位置，得到我们最终的展板。在这里你首先要有一个渲染方法，就像拼凑卡片的玩家一样，想渲染模板就要有一个渲染器。比如你想用数据对象 ‘{demo:"this is a demo"}’ 去格式化 ‘<a>{#demo#} +’ 字符串模板，得到 ‘<a>this is a demo’ 渲染后的模板，便可插入到页面中。不过你首先要有一个渲染模板引擎方法，如 `formatString` 方法，将字符串模板中的 {#demo#} 替换成数据对象中 demo 的属性值。”

```
// 模板渲染方法
A.formatString = function(str, data) {
    return str.replace(/\{\#(\w+)\#\}/g, function(match, key){return typeof
data[key] === undefined ? '' : data[key]});
}
```

“有了这个方法，我们就可以通过简单模板来渲染出我们需求的视图了。”

```
// 文字列表展示
'listPart' : function(data){
    var s = document.createElement("div"),      // 模块容器
        ul = '',                                // 列表字符串
        ldata = data.data.li,                     // 列表数据
        // 模块模板
        tpl = [
            '<h2>{#h2#}</h2>',
            '<p>{#p#}</p>',
            '<ul>{#ul#}</ul>'
        ].join(''),
        // 列表项模板
        liTpl = [
            '<li>',
            '  <strong>{#strong#}</strong>',
            '  <span>{#span#}</span>',
            '</li>'
        ].join('');
    // 有 id 设置模块 id
    data.id && (s.id = data.id);
    // 遍历列表数据
```

```

for(var i = 0, len = ldata.length; i < len; i++) {
    // 如果有列表项数据
    if(ldata[i].em || ldata[i].span) {
        // 列表字符串追加一项列表项
        ul += A.formateString(liTpl, ldata[i]);
    }
}
// 装饰列表数据
data.data.ul = ul;
// 渲染模块并插入模块中
s.innerHTML = A.formateString(tp1, data.data);
// 渲染模块
A.root.appendChild(s);
}

```

“小白你看看如何？是不是优化了许多？减少了对页面中节点的操作。”小铭拿给小白看看。

“嗯，原来模板渲染引擎 `formateString` 是其中的灵魂，一行渲染工作，便完成了上面那么多的节点操作，实在太棒了！”小白感叹。

31.5 再次优化

“不过这还不是最优的版本，你观察了没有，`tp1` 这个模板内部每一行是不是还有相似之处？”

“你是说标签名不一样么？”

“对，只有标签名不一样，所以我们可以对他们进行提取，这样我们可以创建一个模板生成器，既然只有标签名不一样，我们只需要传入标签名便可返回一行字符串模板，那样岂不是更好。对于特殊的模板我们传入特殊的标识来从模板生成器里读取，以后我们如果需要模板时，就不用再去花时间设计模板了，只需要到模板生成器里读取即可。所以我们要创建出一个模板生成器。”

31.6 模板生成器

```

// 模板生成器 name: 标识
A.view = function(name) {
    // 模板库
    var v = {
        // 代码模板
        code : '<pre><code>{#code#}</code></pre>',
        // 图片模板
        img : '',
        // 带有 id 和类的模块模板
        part : '<div id="{#id#}" class="{#class#}">{#part#}</div>',
        // 组合模板
        theme : [

```

```

        '<div>',
        '<h1>{#title#}</h1>',
        '{#content#}',
        '</div>',
    ].join('')
}

// 如果参数是一个数组，则返回多行模板
if(Object.prototype.toString.call(name) === '[object Array]'){
    // 模板缓存器
    var tpl = '';
    // 遍历标识
    for(var i = 0, len = name.length; i < len; i++){
        // 模板缓存器追加模板
        tpl += arguments.callee(name[i]);
    }
    // 返回最终模板
    return tpl;
} else{
    // 如果模板库中有该模板则返回该模板，否则返回简易模板
    return v[name] ? v[name] : ('<' + name + '>{#' + name + '#}</' + name +
'>');
}
}
}

```

31.7 最佳方案

“有了这个小型的模板生成器，我们想获取一个模板就简单多了，比如我们想获取一个‘{#span#}’就可以通过 A.view('span')方式来获取。有了它，要想完成我们的需求就容易多了”。

```

// 文字列表展示
'listPart' : function(data){
//.....
    // 模块模板
    tpl = A.view(['h2', 'p', 'ul']),
    // 列表项模板
    liTpl = A.formateString(A.view('li'), {li : A.view(['strong', 'span'])}),
    // .....
}

```

下章剧透

简单模板模式使我们创建视图更灵活高效，这是一种挣脱传统的新方式。由于 Web 世界中，各种浏览器割据纷争，造成同一功能实现的方法五花八门。这造成了很多页面功能不停地做着功能校验，然而在每一个浏览器使用的功能方法是不变的，那么如何才能屏蔽掉每次调用时都去做的那些功能校验呢，那就赶快翻开下一章吧。

忆之获

简单模板模式意在解决运用 DOM 操作创建视图时造成资源消耗大、性能低下、操作复杂等问题。用正则匹配方式去格式化字符串的执行的性能要远高于 DOM 操作拼接视图的执行性能，因此这种方式常备用于大型框架（如 MVC 等）创建视图操作中。

简单模板模式主要包含三部分，字符串模板库，格式化方法，字符串拼接操作。然而前者，在不同需求的实现中，视图往往是不一致的，因此字符串模板常常是多变的，如何更好地创建模板，给了我们极大的灵活性，对于字符串格式化方法在一个项目中通常是不变的，团队中所有成员都应该以同种方式去格式化模板才能使模板更易读。对于字符串拼接操作，常常是随需求中的视图变化而变化，这里对拼接的灵活的运用可以使你创建的视图过程更高效，模板复用率更高。

我问你答

运用简单模板模式创建一个页面中的导航条。

第32章 机器学习——惰性模式

惰性模式 (layier): 减少每次代码执行时的重复性的分支判断，通过对对象重定义来屏蔽原对象中的分支判断。

今日小白整理着自己消遣时写的“微型代码库”，可是当看到事件模块时感觉每次为元素添加事件时总是重复着做些什么，于是皱起眉头捉摸着如何优化代码。

32.1 对事件的思考

“小白怎么了。”小铭见小白若有所思的样子，走过来问。

“有没有什么办法能解决函数执行时的重复性的分支判断？”小白问。

“给我看看具体的例子吧，帮你分析分析。”

说着小白将自己写的代码展示给了小铭。

```
// 单体模式定义命名空间
var A = {};
// 添加绑定事件方法 on
A.on = function(dom, type, fn){
    if(dom.addEventListener){
        dom.addEventListener(type, fn, false);
    }else if(dom.attachEvent){
        dom.attachEvent('on' + type, fn);
    }else{
        dom['on' + type] = fn;
    }
}
```

“小铭你看，我每次为元素添加事件时，感觉总会走一遍能力检测。这是不是有点多余呀，因为在同一个浏览器中两次执行方法，能力检测时是不可能走两个不同分支的”小白问。

“还真是，不过想解决这个问题也不难，惰性模式可以帮你解决，它的作用就是减少这类函数执行时的重复性分支判断的。”

32.2 机器学习

“你每次都说得这么专业，我还是一名菜鸟码农，还不太理解你说的模式呢。”

“很容易理解，就像你在大学上学时，学到的机器学习一样，既然第一次执行时已经判断过了，而且以后再执行是不必要的，那么在第一次执行后就重新定义它吧。这就是惰性模式的精髓。不过工作中这种模式思想有两种实现方式，第一种就是在文件加载进来时通过闭包执行该方法对其重新定义。不过这样会使页面加载时占用一定资源。第二种方式是在第一种方式基础上做一次延迟执行，在函数第一次调用的时候对其重定义。这么做的好处就是减少文件加载时的资源消耗，但是却在第一次执行时有一定的资源消耗，所以你可以看出这种方式实质上是对消耗的一种惰性推移，有时候是很有必要的，尤其在重定义时消耗资源比较大而且想尽量早地预览到页面时。”小铭继续说，“回到你这代码中来，对于第一种解决方案你可以这么做。”

32.3 加载即执行

```
// 添加绑定事件方法 on
A.on = function(dom, type, fn) {
    // 如果支持 addEventListener 方法
    if (document.addEventListener) {
        // 返回新定义方法
        return function(dom, type, fn) {
            dom.addEventListener(type, fn, false);
        }
    }
    // 如果支持 attachEvent 方法 (IE)
} else if (document.attachEvent) {
    // 返回新定义方法
    return function(dom, type, fn) {
        dom.attachEvent('on' + type, fn);
    }
}
// 定义 on 方法
} else {
    // 返回新定义方法
    return function(dom, type, fn) {
        dom['on' + type] = fn;
    }
}
}();
```

“小白，对于第一种解决办法，首先对 `document` 做能力检测，通过闭包在页面加载时执行它，这样达到重写 `A.on` 方法的目的。我们将 `A.on` 打印出来。”

```
// chrome 浏览器中
console.log(A.on)
// function (dom, type, fn){
//     dom.addEventListener(type, fn, false);
// }
```

“我们发现此时 A.on 方法已经被我们重新定义了。”

“是呀，页面加载时就去执行可能真的要消耗一些资源了，不过第二种方式是如何实现的呢”小白问。

“不用着急，让我慢慢写给你看”。

32.4 惰性执行

```
// 添加绑定事件方法 on
A.on = function(dom, type, fn){
    // 如果支持 addEventListener 方法
    if(dom.addEventListener){
        // 显示重定义 on 方法
        A.on = function(dom, type, fn){
            dom.addEventListener(type, fn, false);
        }
    }
    // 如果支持 attachEvent 方法 (IE)
    }else if(dom.attachEvent){
        // 显示重定义 on 方法
        A.on = function(dom, type, fn){
            dom.attachEvent('on' + type, fn);
        }
    }
    // 如果支持 DOM0 级事件绑定
}else{
    // 显示重定义 on 方法
    A.on = function(dom, type, fn){
        dom['on' + type] = fn;
    }
}
// 执行重定义 on 方法
A.on(dom, type, fn);
};
```

“第二种实现方式与第一种实现方式的不同之处在于，首先，内部对元素 dom 执行能力检测并显示重写，其次，原始函数在函数的最末尾重新执行一遍来绑定事件。不过在文件加载后 A.on 方法还没能重新被定义。所以我们还需等到某一元素绑定事件时，A.on 才能被重定义。”

```
A.on(document.body, 'click', function(){
    alert(11);
})
```

“其实这两种惰性方式都使得以前不必要的分支判断得到剥离，也是对原方法的一种重写，我默默地感觉到，这种模式是不是有很多应用前景呀。”小白问。

32.5 创建 XHR 对象

“是呀，正如你所知道的，当今浏览器繁多，尤其在国内更是复杂，很多功能在不同浏览器中实现不一，所以很多代码为兼容各个浏览器写的分支特别臃肿，有了这种模式，就能使臃

肿的代码执行效率得到提高了。比如我们创建 XHR 对象：“

```
// 创建 XHR 对象
function createXHR() {
    // 标准浏览器
    if(typeof XMLHttpRequest != "undefined"){
        return new XMLHttpRequest();
    }
    // IE 浏览器
    }else if(typeof ActiveXObject != "undefined"){
        if(typeof arguments.callee.activeXString != "string"){
            var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0", "MSXML2.XMLHttp"];
            i = 0,
            len = versions.length;
            //遍历并设置版本
            for(; i < len; i++){
                try{
                    new ActiveXObject(versions[i]);
                    arguments.callee.activeXString = versions[i];
                    break;
                }catch(ex){
                }
            }
        }
        return new ActiveXObject(arguments.callee.activeXString);
    }
    // 对不支持的浏览器抛出错误提示
}else{
    throw new Error("您的浏览器并不支持 Ajax.");
}
}
```

“这是为了兼容低版本 IE 浏览器和标准浏览器，所以代码执行了很多分支，但是有些浏览器中一些分支是不必要的，所以我们要用惰性模式进行代码优化。”

32.6 第一种方案

```
// 第一种 加载时损失性能，但是第一次调用时不损失性能
var createXHR = (function(){
    if (typeof XMLHttpRequest != "undefined"){
        return function(){
            return new XMLHttpRequest();
        };
    } else if (typeof ActiveXObject != "undefined"){
        return function(){
            // 省略代码
        };
    } else {
        return function(){
            throw new Error("No XHR object available.");
        };
    }
})();
```

32.1 第二种方案

```
// 第二种 加载时不损失性能，但是第一次调用时损失性能
function createXHR() {
    if(typeof XMLHttpRequest != "undefined") {
        createXHR = function(){
            return new XMLHttpRequest();
        };
    } else if(typeof ActiveXObject != "undefined") {
        createXHR = function(){
            // 省略代码
        };
    } else{
        createXHR = function(){
            throw new Error("No XHR object available.");
        };
    }
    return createXHR();
}
```

下章剧透

通过惰性模式，我们解决了每次使用浏览器提供的功能方法时对浏览器重复性的功能校验问题。下一章我们将学习如何向执行的对象方法中传入自定义数据，来优化事件的回调函数。

忆之获

惰性模式是一种拖延模式，由于对象的创建或者数据的计算会花费高昂的代价（如页面刚加载时无法辨别是该浏览器支持某个功能，此时创建对象是不够安全的），因此页面之处会延迟对这一类对象的创建。惰性模式又分为两种：第一种是文件加载后立即执行对象方法来重定义对象。第二种是当第一次使用方法对象时重定义对象。对于第一种方式，由于文件加载时执行，因此会占用一些资源。对于第二种方式由于在第一次使用时重定义对象，以致第一次执行时间增加。有时候两种方式对资源的开销都是可接受的，因此到底使用哪种方式，要看具体需求而定。

我问你答

对于区域元素的 `css` 样式，不同浏览器中实现的方式不一样，比如 IE 中通过元素的 `currentStyle` 属性获取，而标准浏览器（世界五大浏览器除 IE 以外其他 4 种，Chrome、Safari、Opera、Firefox 这 4 种浏览器通常是符合 w3c 规范的。国内浏览器除外，国内浏览器一般是双核（IE 内核与标准浏览器内核）浏览器）中通过 `getComputedStyle` 方法获取。那么请你运用惰性模式来封装一个获取元素 `css` 样式的方法 `getStyle`。

第33章 异国战场——参与者模式

参与者 (participator): 在特定的作用域中执行给定的函数，并将参数原封不动地传递。

天气模块的后端数据可以成功获取了，按照项目经理对天气模块提出的需求，打开页面后要定时从后端拉取数据，缓存下来，一旦用户点击查看天气按钮，就要展现天气模块并显示天气信息。

33.1 传递数据

“小铭，有没有什么好办法来实现点击一个按钮时将额外的数据传入这个事件的回调函数中？”小白问。

“封装一个事件框架吧。”

“事件框架？可是我封装的事件框架只是可以兼容各个浏览器，但是还不能实现传递额外的数据需求呀”，小白给小铭展示了自己写的事件框架代码。

```
// 事件绑定方法
A.event.on = function(dom, type, fn) {
    // w3c 标准事件绑定
    if(dom.addEventListener){
        dom.addEventListener(type, fn, false);
    // ie 事件绑定
    }else if(dom.attachEvent){
        dom.attachEvent('on' + type, fn);
    // DOM0 级事件绑定
    }else{
        dom['on' + type] = fn;
    }
}
```

“首先你想在回调函数中传递自定义数据，至少你还没有向 on 方法中传递自定义数据参数呀”小铭挖苦着小白，幽默地一笑。

“可是 addEventListener 方法也不能向回调函数中传递自定义数据参数呀。”

“嗯，你说的对，既然 addEventListener 不能，所以你就要在回调函数中做文章。”

“在回调函数中做文章？那该如何？”小白问。

“给你写个案例你先看看。”说着小铭写下示例代码。

```
A.event.on = function(dom, type, fn, data) {
    // w3c 标准事件绑定
    if(dom.addEventListener){
        dom.addEventListener(type, function(e){
            // 在 dom 环境中调用 fn，并传入事件对象与 data 数据参数
            fn.call(dom, e, data);
        }, false);
    }
    // ie 事件绑定
    // .....
}
```

“JavaScript 中的 call 与 apply 方法很神奇，它可以使我们在特定作用域中执行某个函数，并传入参数，所以，在回调函数中，我们借助 call 函数就实现了你提出的需求了”，小铭解释说。

“可是，添加的事件回调函数就不能移除了？”小白问。

“你想的还挺深入的，这正是这种方式存在的问题，因为此时事件的回调函数其实是匿名函数，所以要想解决这种问题就要借助参与者模式了。”小铭接着说，“即可让更多的对象参与执行时的函数。”

“哦，你这一句把我说糊涂了，你还是举个例子吧”小白急切地说。

33.2 函数绑定

“你知道函数的绑定方法么？”小铭问。

“你指的是 bind 方法么？这个在低版本浏览器中未被支持呀。”

“对，就是 bind，正如你说的该方法支持性不好，所以在很多框架中都添加了这个方法。它实现的思想很简单，就是让一个函数在一个作用域中执行，根据这一条原理我们实现它就简单得多了，只需要一个闭包即可。”

```
// 函数绑定 bind
function bind(fn, context) {
    // 闭包返回新函数
    return function() {
        // 对 fn 装饰并返回
        return fn.apply(context, arguments);
    }
}
```

“我们可以测试一下，新建一个对象 demoObj 和一个函数 demoFn，然后让 demoObj 对象参与 demoFn 的执行，并保存在 bindFn 变量中，我们来观察一下 demoFn 和 bindFn 执行的结果。”

```
// 测试对象
```

```

var demoObj = {
    title : '这是一个例子'
}
// 测试方法
function demoFn(){
    console.log(this.title);
}
// 让 demoObj 参与 demoFn 的执行
var bindFn = bind(demoFn, demoObj);
demoFn();      // undefined
bindFn();      // 这是一个例子

```

“bindFn 函数返回结果了”小白惊呼。

“对呀, bindFn 返回了结果, 这说明 demoFn 在执行时 demoObj 参与进来并提供了作用域, 不过注意 bindFn 是让 demoFn 寄生其中, 并在执行时才让 demoObj 加入的, 所以说 bindFn 和 demoFn 是两个不同的函数。”

33.3 应用于事件

“哦, 这就像是古代异国战争, 作战环境是在别的国家一样。不过这种思想能否应用在我的事件框架中呢”说着, 小白动手对事件进行了测试。

```

var btn = document.getElementsByTagName('button')[0];
var p = document.getElementsByTagName('p')[0];
// 对 demoFn 改进, 在控制台输出参数与 this 对象
function demoFn(){
    console.log(arguments, this);
}
// 未设置提供参与对象
var bindFn = bind(demoFn);
// 绑定事件
btn.addEventListener('click', bindFn);
// chrome 输出: [MouseEvent] Window

// 提供 btn 元素参与对象
var bindFn = bind(demoFn, btn);
// chrome 输出: [MouseEvent] <button>按钮</button>

// 提供 p 元素参与对象
var bindFn = bind(demoFn, p);
// chrome 输出: [MouseEvent] <p>hello</p>

// 移除事件
btn.removeEventListener('click', bindFn);

```

“当未提供参与对象时, 执行的结果是返回 this 对象指向全局对象 Window, 说明此时是在全局作用域中执行的, 当提供 btn 元素参与对象时, 返回的 this 对象是元素自身, 说明此时是在 btn 元素作用域中执行的, 当提供 p 元素参与对象时, 返回的是 p 元素, 说明此时是在 p 元素作用域中执行的。并且我们发现函数的参数中还为我们传递了事件对象。并且我们这种方

式添加的事件还可以通过 `removeEventListener` 来移除。”小白跟小铭诉说着自己的发现。

33.4 原生 bind 方法

“你还挺爱动脑的，没错，你说的对，不过在一些标准浏览器中如高版本的 firefox、chrome、Safari 等中还是为我们提供了原生 `bind` 方法，所以你还可以像下面这种方式使用原生 `bind` 方法。”

```
// 提供 p 元素参与对象
var bindFn = demoFn.bind(p);
```

“不过这只是我们要实现你所说的需求的第一步，将添加的事件成功移除。下面我们要完成第二步，即为运行的函数添加额外的自定义数据参数。要实现这个功能，我想就要借助于函数的柯里化了”小铭继续说。

33.5 函数柯里化

“函数的柯里化？又听见一个新词，真是长见识了”，小白开个玩笑。

“函数柯里化的思想是对函数的参数分割，这有点像其他面向语言中的类的多态，就是根据传递的参数不同，可以让一个函数存在多种状态，只不过函数柯里化处理的是函数，因此要实现函数的柯里化是要以函数为基础的，借助柯里化器伪造其他函数，让这些伪造的函数在执行时调用这个基函数完成不同的功能。还是跟函数绑定一样，首先创建出一个函数柯里化器。”

```
// 函数柯里化
function curry(fn) {
  // 缓存数组 slice 方法 Array.prototype.slice
  var Slice = [].slice
  // 从第二个参数开始截取参数
  var args = Slice.call(arguments, 1);
  // 闭包返回新函数
  return function() {
    // 将参数（类数组）转化为数组
    var addArgs = Slice.call(arguments),
      // 拼接参数
      allArgs = args.concat(addArgs);
    // 返回新函数
    return fn.apply(null, allArgs);
  }
}
```

“我们还是先测试一下我们这个函数柯里化器。创建一个加法器，然后对加法器进行拓展。当传递一个参数时相加一个固定数字；当不传递参数时相加两个固定数字。”

```
// 加法器
function add(num1, num2) {
  return num1 + num2;
```

```

}
// 加 5 加法器
function add5(num) {
    return add(5, num);
}
// 测试 add 加法器
console.log(add(1,2));      // 3
// 测试加 5 加法器
console.log(add5(6));       // 11
// 函数柯里化创建加 5 加法器
var add5 = curry(add, 5);
console.log(add5(7));       // 12
// 7+8
var add7and8 = curry(add, 7, 8);
console.log(add7and8());     // 15

```

“小白，看到没，通过函数柯里化器对 add 方法实现的多态拓展且不需要像以前那样明确声明函数了，因为函数的创建过程已经在函数柯里化器中完成了。”

“嗯。你上面的案例我看明白了，不过，柯里化器要如何融入我们的事件框架中来解决我的需求呢？”

33.6 重构 bind

“当然有帮助呀，你想，你的需求的第二步是传递额外的自定义数据参数，所以我们需要用函数柯里化的思想来拓展函数执行的参数就可以了。你看我重写一下 bind 函数，然后你看看就明白了”说着小铭写下了新版 bind 函数。

```

// 重写 bind
function bind(fn, context){
    // 缓存数组 slice 方法
    var Slice = Array.prototype.slice,
        // 从第三个参数开始截取参数（包括第三个参数）
        args = Slice.call(arguments, 2);
    // 返回新方法
    return function(){
        // 将参数转化为数组
        var addArgs = Slice.call(arguments),
            // 拼接参数
            allArgs = addArgs.concat(args);
        // 对 fn 装饰并返回
        return fn.apply(context, allArgs);
    }
}

```

“现在我们创建两个数据对象 demoData1 和 demoData2 然后传入事件的回调函数中，我们在控制台中看看输出的结果。”

```

var demoData1 = {
    text : '这是第一组数据'
},

```

```

demoData2 = {
    text : '这是第二组数据'
};

// 提供 btn 元素、demoData1 参与对象
// var bindFn = bind(demoFn, btn, demoData1);
// chrome 输出: [MouseEvent, Object] <button>按钮</button>

// 提供 btn 元素、demoData1、demoData2 参与对象
// var bindFn = bind(demoFn, btn, demoData1, demoData2);
// chrome 输出: [MouseEvent, Object, Object] <button>按钮</button>

// 提供 p 元素、demoData1 参与对象
var bindFn = bind(demoFn, p, demoData1);
// [MouseEvent, Object] <p>hello</p>

```

“在回调函数中果然可以访问到传入的自定义数据对象了。不过，小铭，是不是浏览器内置的 bind 方法也可以这么用呢？”小白写下代码试了试。

```

var bindFn = demoFn.bind(p, demoData1);
// chrome 输出: [Object, MouseEvent] <p>hello</p>

```

“不过，小铭，回调函数中的参数顺序变了，它把事件对象放在了最后面了。”小白说。

33.7 兼容版本

“嗯，是这样的，当然有时候为了浏览器统一实现，我们常常会对未提供 bind 方法的浏览器的原生 Function 对象添加 bind 方法的，这样在各个浏览器中就可以兼容了。比如像下面这样。”

```

// 兼容各个浏览器
if(Function.prototype.bind === undefined) {
    Function.prototype.bind = function(context) {
        // 缓存数组 slice 方法
        var Slice = Array.prototype.slice,
            // 从第二个参数截取参数
            args = Slice.call(arguments, 1),
            // 保存当前函数引用
            that = this;
        // 返回新函数
        return function(){
            // 将参数数组化
            var addArgs = Slice.call(arguments),
                // 拼接参数，注意：传入的参数放在了后面
                allArgs = args.concat(addArgs);
            // 对当前函数装饰并返回
            return that.apply(context, allArgs);
        }
    }
}

```

下章剧透

通过运用参与者模式，使我们的事件绑定功能更加完善。然而 JavaScript 语言中有很多异步功能逻辑，对于异步逻辑执行完成的检测又是一件很头疼的问题，下一章我们将学习如何实现对多个异步逻辑进行检测其执行完成的模式。

忆之获

参与者模式实质上是两种技术的结晶，函数绑定和函数柯里化。早期浏览器中并未提供 bind 方法，因此聪明的工程师们为了使添加的事件能够移除，事件回调函数中能够访问到事件源，并且可以向事件回调函数中传入自定义数据，才发明了函数绑定与函数柯里化技术。

对于函数绑定，它将函数以函数指针（函数名）的形式传递，使函数在被绑定的对象作用域中执行，因此函数的执行中可以顺利地访问到对象内部的数据，由于函数绑定构造复杂，执行时需消耗更多的内存，因此执行速度上要稍慢一些。不过相对于解决的问题来说这种消耗还是值得的，因此它常用于事件、setTimeout 或 setInterval 等异步逻辑中的回调函数。

对于函数柯里化即是将接受多个参数的函数转化为接受一部分参数的新函数，余下的参数保存下来，当函数调用时，返回传入的参数与保存的参数共同执行的结果。通常保存下来的参数保存于闭包内，因此函数柯里化的实现要消耗一定的资源。函数的柯里化有点类似类的重载，不同点是类的重载是同一个类对象，函数的柯里化是两个不同的函数。随着函数柯里化的发展，现在又衍生出一种反柯里化的函数，其目的是方便我们对方法的调用，它的实现如下。

```
// 反柯里化
Function.prototype.uncurry = function(){
    // 保存当前对象
    var that = this;
    return function(){
        return Function.prototype.call.apply(that, arguments);
    }
}
```

当用 Object.prototype.toString 校验对象类型时：

```
// 获取校验方法
var toString = Object.prototype.toString.uncurry();
// 测试对象数据类型
console.log(toString(function(){})); // chrome: [object Function]
console.log(toString([])); // chrome: [object Array]
```

用数组的 push 方法为对象添加数据成员：

```
// 保存数组 push 方法
var push = [].push.uncurry();
```

```
// 创建一个对象
var demoArr = {};
// 通过 push 方法为对象添加数据成员
push(demoArr, '第一个成员', '第二个成员');
console.log(demoArr); // chrome: Object {0: "第一个成员", 1: "第二个成员", length: 2}
```

我问你答

思考一下，参与者模式能否用于自定义事件（观察者模式）中？

第34章 入场仪式——等待者模式

等待者模式（waiter）：通过对多个异步进程监听，来触发未来发生动作。

由于新闻访问数量增加，造成服务器端压力过大，所以服务器端决定重新部署，原来的新闻搜索接口拆分成新闻搜索结果接口和新闻推荐结果接口，并且都部署在独立的服务器上。

34.1 接口拆分

“小白，咱们公司服务器端新进一批服务器，听说服务器端接口重新部署了，你知道么？”小铭问小白。

“是呀，那咱们的新闻搜索页面也要重构吧”小白问。

“为何要重构？以前的逻辑不是写完上线了么”。

“是呀，不过据说原来的新闻搜索接口拆分了，拆分成新闻搜索结果接口和新闻推荐结果接口。这样以前通过对单一接口请求回来的数据对两个模块分发数据的处理逻辑现在走不通了。因为拆分出来的两个独立接口要对搜索结果模块和推荐模块分别处理了。”

“啊，原来你是在纠结这件事情呀，你也不用重构，你用等待者模式监听两个异步请求的结束，然后按照以前的分发逻辑执行就可以了。”小铭解释道。

“等待者模式？他是如何实现的？更何况我根本不知道这两个异步接口哪个最后完成呀。”

34.2 入场仪式

“等待者模式或者说等待者对象是用来解决那些不确定先后完成的异步逻辑的。就像运动会的入场仪式，你不确定请哪只队伍先入场，但有一点你很确定，就是会议开始必须等到所有的队伍入场完毕。而这里的会议开始就相当于等待这模式的逻辑执行。所以说等待者模式监听的是所有异步逻辑的完成，这样才会自动执行成功回调函数，当然有一个异步逻辑执行失败了，它便会执行失败回调函数。”

“哦，那是不是要在等待者对象内部建立状态迭代器呀，一直监听着所有请求的完成状态。”小白问。

“差不多，只不过等待者对象不用实时监听异步逻辑的完成，它只需要对注册监听的异步逻辑发生状态改变时（请求成功或者请求失败）对所有异步逻辑的状态做一次确认迭代。”小铭接着说，“说了这么多可能你也糊涂了，我们还是简单实现一个等待者吧”。

34.3 等待者对象

```
// 等待对象
var Waiter = function() {
    // 注册了的等待对象容器
    var dfd = [],
        // 成功回调方法容器
        doneArr = [],
        // 失败回调方法容器
        failArr = [],
        // 缓存 Array 方法 slice
        slice = Array.prototype.slice,
        // 保存当前等待者对象
        that = this;

    // 监控对象类
    var Promise = function() {
        // 监控对象是否解决成功状态
        this.resolved = false;
        // 监控对象是否解决失败状态
        this.rejected = false;
    }
    // 监控对象类原型方法
    Promise.prototype = {
        // 解决成功
        resolve : function(){ },
        // 解决失败
        reject : function(){}
    }

    // 创建监控对象
    that.Deferred = function(){
        return new Promise();
    }

    // 回调执行方法
    function _exec(arr){}

    // 监控异步方法 参数：监控对象
    that.when = function(){};

    // 解决成功回调函数添加方法
    that.done = function(){};

    // 解决失败回调函数添加方法
    that.fail = function(){}
}
```

“对于等待者对象你可以看出，其内部定义了三个数组，分别是监控对象容器，以及成功与失败回调函数容器；一个类-监控对象，该对象有两个属性，即监控解决成功状态与监控解决失败状态，两个方法，解决成功方法与解决失败方法；一个私有方法_exec来处理成功失败回调函数的方法；3个共有方法接口：when方法监控异步逻辑，done方法添加成功回调函数，fail方法添加失败回调函数。”小铭喘口气，歇了一下继续说，“等待者对象内部的属性我们已经定义好了，下面我们一一实现这些方法吧，首先我们还是完成监控对象类的原型方法 resolve 和 reject。”

34.4 监控对象

“这两个方法是对异步逻辑的监控方法，不过实现它们需要注意哪些点呢”小白问。

“首先他们都是因异步逻辑状态的改变而执行相应操作的，不同的是 resolve 方法是要执行成功回调函数，所以要对所有被监控的异步逻辑进行状态校验。而 reject 方法是要执行失败回调函数，所以只要有一个被监控的异步逻辑状态改变成失败状态，就要执行失败回调函数。”

```
// 监控对象原型方法
Promise.prototype = {
    // 解决成功
    resolve : function(){
        // 设置当前监控对象解决成功
        this.resolved = true;
        // 如果没有监控对象则取消执行
        if(!dfd.length)
            return;
        // 遍历所有注册了的监控对象
        for(var i = dfd.length - 1; i >= 0; i--){
            // 如果有任何一个监控对象没有被解决或者解决失败则返回
            if(dfd[i] && !dfd[i].resolved || dfd[i].rejected){
                return;
            }
            // 清除监控对象
            dfd.splice(i,1);
        }
        // 执行解决成功回调方法
        _exec(doneArr);
    },
    // 解决失败
    reject : function(){
        // 设置当前监控对象解决失败
        this.rejected = true;
        // 如果没有监控对象则取消执行
        if(!dfd.length)
            return;
        // 清除所有监控对象
        dfd.splice(0);
        // 执行解决失败回调方法
        _exec(failArr);
    }
}
```

}

“对于回调执行方法要做的事情很简单，就是遍历成功或者失败回调函数容器，然后依次执行内部的方法。”

```
// 回调执行方法
function _exec(arr) {
  var i = 0,
    len = arr.length;
  // 遍历回调数组执行回调
  for(; i < len; i++) {
    try{
      // 执行回调函数
      arr[i] && arr[i]();
    }catch(e){}
  }
}
```

34.5 完善接口方法

“等待者对象还定义了三个共有接口方法 when、done 和 fail，对于 when 方法是要监测已注册过的监控对象的异步逻辑（请求： ajax 请求等；方法： setTimeout 方法等），所以 when 方法就要将监测对象放入监测对象容器中，当然还要判断监测对象是否存在、是否解决、是否是监测对象类的实例。最后还要返回该等待对象便于链式调用。”

```
// 监控异步方法 参数： 监控对象
that.when = function(){
  // 设置监控对象
  dfd = slice.call(arguments);
  // 获取监控对象数组长度
  var i = dfd.length;
  // 向前遍历监控对象，最后一个监控对象的索引值为 length-1
  for(--i; i >= 0; i--) {
    // 如果不存在监控对象，或者监控对象已经解决，或者不是监控对象
    if(!dfd[i] || dfd[i].resolved || dfd[i].rejected || !dfd[i] instanceof
      Primitive) {
      // 清理内存 清除当前监控对象
      dfd.splice(i,1);
    }
  }
  // 返回等待者对象
  return that;
};
```

“对于 done 方法与 fail 方法的功能则简单得多，只需要向对应的回调函数容器中添加相应回调函数即可，并最终将等待者对象返回便于链式调用”。

```
// 解决成功回调函数添加方法
that.done = function(){
  // 向成功回调函数容器中添加回调方法
```

```

doneArr = doneArr.concat(slice.call(arguments));
// 返回等待者对象
return that;
};

// 解决失败回调函数添加方法
that.fail = function(){
    // 向失败回调函数容器中添加回调方法
    failArr = failArr.concat(slice.call(arguments));
    // 返回等待者对象
    return that;
}

```

34.6 学以致用

“好了，小白，有了这个等待对象，我们就可以对异步逻辑进行监控了。下面我们先简单测试一下吧。”

“那我们是不是还要为异步请求（ajax）搭建一个测试服务器呢？”小白问。

“先不用，因为在JavaScript中异步逻辑的实现有很多种，比如需求中的 ajax 请求等，不过除此之外你常用的 setTimeout 单次定时器、setInterval 循环定时器也是一种异步机制实现的方法。”

34.7 异步方法

“异步方法？你是怎么看出来的？”小白问。

“让你理解透彻，我先写个例子”说着小铭写下例子。

```

setTimeout(function(){
    console.log('first');
}, 30)
console.log('second');
// 控制台输出结果
// second
// first

```

“首先说一下同步机制，同步就像公司外面的同向单车道马路，汽车只能一辆一辆通过，后面的小汽车即使跑得再快也不可能超越前面的货车。这就是同步你能明白吧”。

“嗯，正是因为这些慢悠悠的货车堵塞交通所以要开辟多车道吧。”小白说。

“对，你很聪明嘛。这些货车跑的太慢了，如果能让他们在一边行驶，把这条马路让出来，我们的交通不就很畅通了么，于是有了异步-多车道，目的就是将那些耗时的程序剥离出去单独运行，这也是为什么上面的程序在控制台中先输出的是‘second’，后输出‘first’的原因，如果 setTimeout 是同步方法，按照程序从上至下执行原则，那么控制台首先输出的一定是‘first’。也是因为定时器是一种耗时方法，浏览器是不会让这类耗时方法堵塞页面程序的执行的。”

“既然 Ajax 和定时器都是异步逻辑，那么我们使用 setTimeout 定时器测试就可以了吧。”小白问。

34.8 结果如何

“没有错，所以我们这里就用定时器来测试我们的等待者对象。我们假设一种场景，活动页面中有几个随机运动的彩蛋，当每个彩蛋运动结束后我们要展示一个欢迎页面。这种需求很常见，但是要检测几个运动彩蛋全部结束是一件很头疼的事情。好在我们可以用等待者模式来解决，那么我们首先要创建一个等待者对象”。

```
var waiter = new Waiter();
```

“这里我们简化彩蛋，将其抽象成一个方法，而且在某一时刻会结束。我们要在彩蛋方法执行内部创建监听对象。”

```
// 第一个彩蛋，5 秒后停止
var first = function(){
    // 创建监听对象
    var dtd = waiter.Deferred();
    setTimeout(function(){
        console.log('first finish');
        // 发布解决成功消息
        dtd.resolve();
    },5000);
    // 返回监听对象
    return dtd;
}();
// 第二个彩蛋，10 秒后停止
var second = function(){
    // 创建监听对象
    var dtd = waiter.Deferred();
    setTimeout(function(){
        console.log('second finish');
        // 发布解决成功消息
        dtd.resolve();
    },10000);
    // 返回监听对象
    return dtd;
}();
```

“最后我们要用等待者对象监听两个彩蛋的工作状态，并执行相应的成功回调函数与失败回调函数。”

```
waiter
    // 监听两个彩蛋
    .when(first, second)
    // 添加成功回调函数
    .done(function(){
        console.log('success');
```

```

    }, function(){
      console.log('success again')
    })
    // 添加失败回调函数
    .fail(function(){
      console.log('fail');
    });
  });
  // 输出结果
  // first
  // second
  // success
  // success again

```

“我们看到十秒后当第二个彩蛋结束后，我们注册的两个成功回调函数成功执行，当然如果第一个执行失败，那么我们就将执行失败回调函数。”

```

var first = function(){
  var dtd = waiter.Deferred();
  setTimeout(function(){
    console.log('first finish');
    // 发布解决失败消息
    dtd.reject();
  }, 5000);
  return dtd;
}();
// 输出结果
// first
// fail
// second finish

```

34.9 框架中的等待者

“小白，其实等待者模式在很多框架中都有实现，比如jQuery中的Deferred对象，并且它还用等待者思想对 ajax 方法进行了封装。”

```

$.ajax("test.php")
  .done(function(){ console.log("成功"); })
  .fail(function(){ console.log("失败"); });

```

34.10 封装异步请求

“当然我们自己也可以将原生的 ajax 方法封装成等待者模式，将 resolve 方法放在请求成功回调函数内调用，将 reject 方法放在请求失败回调函数中调用。”

```

// 封装 get 请求
var ajaxGet = function(url, success, fail){
  var xhr = new XMLHttpRequest();
  // 创建检测对象
  var dtd = waiter.Deferred();
  xhr.onload = function(event){

```

```

// 请求成功
if((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
    success && success()
    dtd.resolve();
// 请求失败
} else{
    dtd.reject();
    fail && fail();
}
};

xhr.open("get", url, true);
xhr.send(null);
};

```

“有了等待者对象，以后就不愁后端的部署了，我是需要修改一下接口就可以了，省去了很多工作”小白喜悦地说。

34.11 轮询

“除此之外，我们的轮询（定期向后端发送请求）实现的机制也有些类似等待者模式，只不过是在其自身内部对未来动作进行了监听。”

```

// 长轮询
(function getAjaxData(){
    // 保存当前函数
    var fn = arguments.callee;
    setTimeout(function(){
        $.get('./test.php', function(){
            console.log('轮询一次');
            // 再一次执行轮询
            fn();
        })
    }, 5000)
})();

```

下章剧透

通过等待者模式我们可以监听多个异步逻辑执行成功与否，并可执行相应的回调函数处理。下一章，多人开发的复杂系统中又会有哪些问题呢？

忆之获

等待者模式意在处理耗时比较长的操作，比如 canvas 中遍历并操作一张大图片中的每一个像素点、定时器操作、异步请求等。等待者模式为我们提供了一个抽象的非阻塞的解决方案，通过创建 Promise 对象，对耗时逻辑的未来状态变化返回一个响应，通过在等待者对象内部捕获这些响应信息，为耗时较长的操作提供了回调方案，使我们可以捕获耗时操作完成时或中断

时的状态并执行相应的回调方案。

我问你答

搭建一个服务器，并创建两个 Ajax 请求，运用等待者模式监听两个请求的成功响应，并在控制台中输出获取到的数据。

第六篇

架构型设计模式

架构型设计模式是一类框架结构，通过提供一些子系统，指定他们的职责，并将它们条理清晰地组织在一起。

第 35 章 死心眼——同步模块模式

第 36 章 大心脏——异步模块模式

第 37 章 分而治之——Widget 模式

第 38 章 三人行——MVC 模式

第 39 章 三军统帅——MVP 模式

第 40 章 视图的逆袭——MVVM 模式

第35章 死心眼——同步模块模式

模块化：将复杂的系统分解成高内聚、低耦合的模块，使系统开发变得可控、可维护、可拓展，提高模块的复用率。

同步模块模式——SMD (Synchronous Module Definition)：请求发出后，无论模块是否存在，立即执行后续的逻辑，实现模块开发中对模块的立即引用。

随着页面功能的增加，系统的业务逻辑越来越复杂。多人开发的功能经常耦合在一起。有时项目经理提出的需求，分配给多人实现时，常常因为某一处功能耦合了多人的实现代码，出现“排队”修改的现象，这很不利于团队开发，就像下面的例子。

35.1 排队开发

“小铭，项目经理让我为导航添加新消息提示引导，可是别人还在修改导航条，我该怎么办？我提交的代码（团队开发中常常会将开发完的代码提交到版本库类以备后续上线）很可能与他人冲突呀，这该怎么办？我是不是要等到他们开发完毕以后再开始开发？可是项目经理那边……”小白问小铭。

“怎么又要排队开发？这说明你们页面中代码架构得很不合理呀，我看你们是如何实现的”说着，小铭看了看页面中的代码。

```
// A 工程师 获取数据并创建导航模块 (沿用 A 框架)
// 导航数据
var data = null,
    // 获取导航容器元素
    dom = A('#nav'),
    // 创建导航逻辑
    createNav = function(){
        // A 工程师完成导航创建逻辑

        ***** C 工程师加入 为导航添加引导图片 *****/
        var li = A('li', dom);
        for(var i = 0, len = data.length){
            if(data[i].hasGuide){
                $(li[i]).addClass('has-guide')
            }
        }
    }
}
```

```

    }
/***** B 工程师加入 完成对导航添加事件需求 *****/
// 在导航容器中获取每条导航容器
var li = A('li', dom);
li.on('mouseover', function(){
    // 显示下拉框逻辑
}).on('mouseout', function(){
    // 隐藏下拉框逻辑
});
/***** */
};

// 获取导航数据
A.ajax('data/nav', function(res){
    if(res.errorNo == 0){
        // 保存数据
        data = res.data;
        // 创建的导航模块
        createNav();
    }
});

```

“你们怎么都把代码写在一起了”小铭感觉很奇怪。

“方便呀，你看 A 工程师已经把数据和导航元素容器都保存在那里了，我们直接用就可以了”。

“不过你们有没有想过，如果 A 工程师正在修改导航，那么 B、C 工程师对于导航的新需求就无法执行了”小铭问。

“可是……我们又该如何做呢？”

35.2 模块化开发

“小白。不知道你是否了解模块化开发？模块化开发的思想是将复杂的系统分解成高内聚低耦合的模块，然后每位工程师独立地去开发自己的模块实现复杂的系统可控、可维护、可拓展，当然模块间也可相互调用，来提升模块的功能复用。”

“好厉害，那么我该如何分解复杂的系统，模块间又是如何调用的呢？”小白问。

“这次你算是问到重点了。要想实现模块化开发，首先要有一个模块管理器，它管理着模块的创建与调度。对于模块的调用分为两类，第一类同步模块调度的实现比较简单，不需要考虑模块间的异步加载。第二类异步模块调度的实现就比较繁琐。它可实现对模块的加载调度。”

“听上去好高端，赶快教教我如何实现吧”小白很着急。

35.3 模块管理器与创建方法

“别着急。你这个需求很简单，今天我们先学习最简单的同步模块模式来解决你的需求问

题吧，对于同步模块调度首先要创建模块，如何没有模块那么谈何调度呢。我们首先定义一个模块管理器对象 F，然后为其创建一个模块定义方法 define。”

```
// 定义模块管理器单体对象
var F = F || {};
/**/
 * 定义模块方法（理论上，模块方法应放在闭包中实现，可以隐蔽内部信息，这里我们为让读者能够看
明白，我们忽略此步骤）
* @param str 模块路由
* @param fn 模块方法
*/
F.define = function(str, fn){
    // 解析模块路由
    var parts = str.split('.'),
        // old 当前模块的祖父模块，parent 当前模块父模块
        // 如果在闭包中，为了屏蔽对模块直接访问，建议将模块添加给闭包内部私有变量
        old = parent = this,
        // i 模块层级，len 模块层级长度
        i = len = 0;
    // 如果第一个模式是模块管理器单体对象，则移除
    if(parts[0] === 'F'){
        parts = parts.slice(1);
    }
    // 屏蔽对 define 与 module 模块方法的重写
    if(parts[0] === 'define' || parts[0] === 'module'){
        return;
    }
    // 遍历路由模块并定义每层模块
    for(len = parts.length; i < len; i++){
        // 如果父模块中不存在当前模块
        if(typeof parent[parts[i]] === 'undefined'){
            // 声明当前模块
            parent[parts[i]] = {};
        }
        // 缓存下一层级的祖父模块
        old = parent;
        // 缓存下一层级父模块
        parent = parent[parts[i]];
    }
    // 如果给定模块方法则定义该模块方法
    if(fn){
        // 此时 i 等于 parts.length，故减一
        old[parts[--i]] = fn();
    }
    // 返回模块管理器单体对象
    return this;
}
```

35.4 创建模块

“创建模块的方法 define 我们实现了，下面我们创建一些模块，首先我们创建 String 模块，对于 String 模块，要为我们提供 trim 方法。”

```
// F.string 模块
F.define('string', function(){
    // 接口方法
    return {
        // 清楚字符串两边空白
        trim : function(str){
            return str.replace(/^\s+|\s$/g, '');
        }
    };
});
```

“我们测试一下。”

// 注意：在真正的模块开发中，是不允许这样直接调用的，有两点原因，技术上，模块通常保存在闭包内部的私有变量里，而不会保存在 F 上，因此是获取不到的，而我们这里简化了闭包，也为测试方便，因此直接引用。其次，类似如下调用是不符合模块化开发规范的。
F.string.trim('测试用例。') // "测试用例。"

“对于模块的回调函数，我们也可以以构造函数的形式返回接口，比如我们创建 DOM 模块，其中包括 dom() 获取元素方法、html() 获取或者设置元素 innerHTML 内容方法等等。”

```
F.define('dom', function(){
    // 简化获取元素方法(重复获取可被替代，此设计用于演示模块添加)
    var $ = function(id){
        $.dom = document.getElementById(id);
        // 返回构造函数对象
        return $;
    }
    // 获取或者设置元素内容
    $.html = function(html){
        // 如果传参则设置元素内容，否则获取元素内容
        if(html){
            this.dom.innerHTML = html;
            return this;
        }else{
            return this.dom.innerHTML;
        }
    }
    // 返回构造函数
    return $;
});
// 测试用例(页面元素: <div id="test"> test </div> )
F.dom('test').html(); // "test"
```

“对于模块的创建，我们也可以先声明后创建，如添加 addClass() 为元素添加 class 方法。”

```
// 为 dom 模块添加 addClass 方法
// 注意，此种添加模式之所以可行，是因为将模块添加到 F 对象上，模块化开发中只允许上面的添加方式
F.define('dom.addClass');
F.dom.addClass = function(type, fn) {
    return function(className){
        // 如果不存在该类
        if(!~this.dom.className.indexOf(className)) {
```

```

    // 简单添加类
    this.dom.className += ' ' + className;
}
}

}();
// 测试用例
F.dom('test').addClass('test')

```

35.5 模块调用方法

“小白，现在你已经可以熟练创建模块了，那么下面你要做的事情就是，用这些已有的模块去完成你的需求。既然要使用模块，那么我们需要创建一个‘使用’模块方法——module。”

```

// 模块调用方法
F.module = function(){
    // 将参数转化为数组
    var args = [].slice.call(arguments),
        // 获取回调执行函数
        fn = args.pop(),
        // 获取依赖模块，如果 args[0] 是数组，则依赖模块为 args[0]。否则依赖模块为 arg
        parts = args[0] && args[0] instanceof Array ? args[0] : args,
        // 依赖模块列表
        modules = [],
        // 模块路由
        modIDs = '',
        // 依赖模块索引
        i = 0,
        // 依赖模块长度
        ilen = parts.length,
        // 父模块，模块路由层级索引，模块路由层级长度
        parent, j, jlen;
    // 遍历依赖模块
    while(i < ilen){
        // 如果是模块路由
        if(typeof parts[i] === 'string'){
            // 设置当前模块父对象(F)
            parent = this;
            // 解析模块路由，并屏蔽掉模块父对象
            modIDs = parts[i].replace(/^F\.\./, '').split('.');
            // 遍历模块路由层级
            for(j = 0, jlen = modIDs.length; j < jlen; j++){
                // 重置父模块
                parent = parent[modIDs[j]] || false;
            }
            // 将模块添加到依赖模块列表中
            modules.push(parent);
        } else{
            // 直接加入依赖模块列表中
            modules.push(parts[i]);
        }
        // 取下一个依赖模块
        i++;
    }
}

```

```

    }
    // 执行回调执行函数
    fn.apply(null, modules);
}

```

35.6 调用模块

“对于模块调用方法，参数可分为两部分，依赖模块与回调执行函数（最后一个参数），它的实现原理是，首先遍历并获取所有的依赖模块，并一次保存在依赖模块列表中，然后将这些依赖模块作为参数传入执行函数中执行。”

小白激动不已，“我都等不及了，赶紧让我调用一个模块吧。”

```

// 引用 dom 模块与 document 对象（注意，依赖模块对象通常为已创建的模块对象）
F.module(['dom', 'document'], function(dom, doc) {
    // 通过 dom 模块设置元素内容
    dom('test').html('new add!');
    // 通过 document 设置 body 元素背景色
    doc.body.style.background = 'red';
});

```

“小白，你通过数组来声明了依赖模块，其实你看我们定义 module 方法时，依赖模块还可以以字符串形式传入，比如”

```

// 依赖引用 dom 模块，string.trim 方法
F.module('dom', 'string.trim', function(dom, trim) {
    // 测试元素 <div id="test"> test </div>
    var html = dom('test').html();      // 获取元素内容
    var str = trim(html);              // 去除字符串两边空白符
    console.log("**" + html + "**", "**" + str + "**"); // * test * *test*
});

```

“这种变成方式真不错，写在 module 回调函数里面的功能安全可靠，模块间的组织条理清晰，这样以后我们自己负责自己的模块就可以了。”

“嗯，所以以后你们再开发就可以一个人将导航模块获取到的服务器端数据放在一个模块里，然后其他工程师在完成需求时直接引用该模块数据即可，而且可以独立开发，互不影响。”

下章剧透

本章我们与小白一起学习了模块化开发，以及如何实现同步模块开发。但是调用模块时，这些模块必然是已经创建好的，对于未加载文件中的模块我们该如何使用呢？下一章我们将一同看看如何实现对未加载文件中的模块依赖进行调用。

忆之获

模块化开发即是以分而治之的思想，实现对复杂系统的分解，使系统随着其功能的增加而变得可控、可拓展、可维护。这就要求我们对模块细化。随着系统功能的增加模块的数量也随之增加。模块开发的成本随之减少，但是模块的接口数量却随之增加，接口的使用成本和开发与维护成本也随之增加，所以合理的模块分割显得尤为重要。

模块化开发是一种对系统的分解，但使用时又像是以组合模式对模块的组合。因此这也使得系统中的问题一般出现在局部，使得开发人员处理相应模块即可，而不用顾虑整个系统。因此相对于整个复杂的系统，对于局部模块的改造、优化甚至替换所需成本要小得多。组合的灵活性也使得我们可以实现更复杂、多样化的功能。

在 Web 前段，实现的模块化开发往往创建了大量的闭包，这会在内存中占用大量的资源得不到释放，这是一种资源的浪费，但相对于解决的问题来说，这种开销是值得的。

同步模块模式是模块化开发的一种最简单的形式，这种模式使得依赖的模块无论加载，无论有无，模块创建即执行，这就要求依赖的模块必然是创建过的。同步模块模式无法处理异步加载的模块，因此浏览器端异步加载文件的环境模式限制了同步模块模式的应用。不过对于服务器端如 nodejs 等，他们的文件都存储在本地，因此同步模块模式更适用。

我问你答

本节课我们学习并实践了同步模块化开发，请你将模块管理器 F 封装在闭包内，隐蔽模块的直接调用。

第36章 大心脏——异步模块模式

模块化：将复杂的系统分解成高内聚、低耦合的模块，使系统开发变得可控、可维护、可拓展，提高模块的复用率。

异步模块模式——AMD (Asynchronous Module Definition)：请求发出后，继续其他业务逻辑，知道模块加载完成执行后续的逻辑，实现模块开发中对模块加载完成后的引用。

浏览器环境不同于服务器环境，在浏览器中对文件的加载是异步的。因此要使用未加载文件中的某些模块方法时必然经历文件加载过程。因此对未加载文件中的模块引用，同步模块模式是无能为力的，为解决这类问题，看看小白与他的团队都做了些什么。

36.1 异步加载文件中的模块

“小铭，昨天学的同步模块模式重构我们的项目，发现对已经加载的文件中的模块依赖调用是没有问题的。不过对于加载的文件中模块调用，即使加载这个文件还是调用不到”小白问小铭。

“你当真加载了么？”小铭问。

“加载了，不信你看看”说着小白给小铭看了自己写的代码。

```
// 加载脚本文件
var loadScript = function(src){
  var _script = document.createElement('script'); // 创建脚本元素
  _script.type = 'text/JavaScript'; // 设置类型
  _script.src = src; // 设置加载路径
  document.getElementsByTagName('head')[0].appendChild(_script);
} // 将元素插入到页面中
// 加载 localStorage 文件
loadScript('localStorage.js');
// 使用 localStorage 模块
F.module('localStorage', function(ls){
  // do something
});
```

“问题就出在 `loadScript` 方法里”，小铭说。

“怎么了？有什么问题吗”，小白不解。

“你知道浏览器中的文件是异步加载吗？异步加载是说，虽然现在开始加载 localstorage.js 文件，不过在文件没有加载完之前你可以继续做其他的事情，并且你写的方法，对于文件什么时候加载完成，你是无法获知的，而同步模块模式会立即引用该模块，此时文件加载尚未加载完成，因此你是引用不到该模块的。”

“这样的话，如何才能解决呢？”小白问。

36.2 异步模块

“既然同步模块模式不行你可以改成异步模块模式，这更适合于浏览器环境。”小铭建议说。

“异步模块？你是说等到文件加载完成时再对模块引用吗？这实现起来难度会很大吧？使用方式和同步模块模式差别大么？”

“有了同步模块思想再使用异步模块就简单多了，不过它的模块创建和调用都通过一个方法，如创建 Event 模块，但要涉及 DOM，你可以这样使用 F.module('lib/event', ['lib/dom'], function(dom){})。不过要注意的是，第一个参数 (lib/event) 是模块 id，一定要对应你的文件路径，这样实现起来成本会更小一些。如果说同步模块死心眼的话，只要有模块依赖不管该模块是否存在，立即执行。那么异步模块就像有个大心脏，即便给定依赖模块，也会耐心等待所有模块加载完成再执行。因此异步模块模式会涉及到模块依赖，根据依赖模块加载文件，加载文件成功后执行引用模块时声明的回调函数等一些技术的实现。”

“不太明白，你能说一说它的实现流程么？”小白问。

36.3 闭包环境

“当然可以，首先你要创建一个闭包，目的是封闭已创建的模块，防止外界对其直接访问，并在闭包中创建模块管理器 F，并作为接口保存在全局作用域中。”

```
// 向闭包中传入模块管理器对象 F (~屏蔽压缩文件时，前面漏写；报错)
~(function(F){
    // 模块缓存器。存储已创建模块
    var moduleCache = {};
})((function(){
    // 创建模块管理器对象 F，并保存在全局作用域中
    return window.F = {};
}))();
```

“有了安全的闭包环境，下面要做的就是为模块管理器对象提供一个 module 方法。”

“跟同步模块模式中的 module 方法一样么？都是调用模块并执行回调函数么？”小白问？

36.4 创建与调度模块

“你只说对了一部分，这里的 module 方法集模块创建方法于一身。在这个方法中要遍历所有依赖模块，并判断所有模块都存在才可执行回调函数，否则加载相应文件，直到文件加载完成才执行回调函数。”

```
/**
 * 创建或调用模块方法
 * @param url          参数为模块 url
 * @param deps         参数为依赖模块
 * @param callback     参数为模块主函数
 */
F.module = function(url, modDeps, modCallback) {
    // 将参数转化为数组
    var args = [].slice.call(arguments),
        // 获取模块构造函数（参数数组中最后一个参数成员）
        callback = args.pop(),
        // 获取依赖模块（紧邻回调函数参数，且数据类型为数组）
        deps = (args.length && args[args.length - 1] instanceof Array) ? args.
    pop() : [],
        // 该模块 url（模块 ID）
        url = args.length ? args.pop() : null,
        // 依赖模块序列
        params = [],
        // 未加载的依赖模块数量统计
        depsCount = 0,
        // 依赖模块序列中索引值
        i = 0,
        // 依赖模块序列长度
        len;
    // 获取依赖模块长度
    if(len = deps.length) {
        // 遍历依赖模块
        while(i < len) {
            // 闭包保存 i
            (function(i) {
                // 增加未加载依赖模块数量统计
                depsCount++;
                // 异步加载依赖模块
                loadModule(deps[i], function(mod) {
                    // 依赖模块序列中添加依赖模块接口引用
                    params[i] = mod;
                    // 依赖模块加载完成，依赖模块数量统计减一
                    depsCount--;
                    // 如果依赖模块全部加载
                    if(depsCount === 0) {
                        // 在模块缓存器中矫正该模块，并执行构造函数
                        setModule(url, params, callback);
                    }
                });
            });
        }
    }
}
```

```

        }) (i);
        // 遍历下一依赖模块
        i++;
    }
    // 无依赖模块，直接执行回调函数
} else {
    // 在模块缓存器中矫正该模块，并执行构造函数
    setModule(url, [], callback);
}
}

```

36.5 加载模块

“在 module 方法中有两个方法我们还没有定义，loadModule 方法与 setModule 方法，我们先实现 loadModule 方法，loadModule 方法目的是加载依赖模块对应的文件并执行回调函数。对此我们可分三种情况处理。第一种情况，如果文件已经被要求加载过，我们要区分文件已经加载完成或是正在加载中，如果已经加载完成，我们异步执行该模块的加载完成回调函数（相见 F.module 方法中对 loadModule 方法调用部分）。第二种情况，如果文件未加载完成，我们要将加载完成回调函数缓存入模块加载完成回调函数容器中（该模块的 onload 数组容器）。第三种情况，如果依赖模块对应的文件未被要求加载过，那么我们要加载该文件，并将该依赖模块的初始化信息写入模块缓存器中。”

```

var moduleCache = {};
setModule = function(moduleName, params, callback) {},
/**
 * 异步加载依赖模块所在文件
 * @param moduleName 模块路径 (id)
 * @param callback 模块加载完成回调函数
 */
loadModule = function(moduleName, callback) {
    // 依赖模块
    var _module;
    // 如果依赖模块被要求加载过
    if(moduleCache[moduleName]){
        // 获取该模块信息
        _module = moduleCache[moduleName];
        // 如果模块加载完成
        if(_module.status === 'loaded'){
            // 执行模块加载完成回调函数
            setTimeout(callback(_module.exports), 0);
        }else{
            // 缓存该模块所处文件加载完成回调函数
            _module.onload.push(callback);
        }
    }
    // 模块第一次被依赖引用
} else{
    // 缓存该模块初始化信息
    moduleCache[moduleName] = {
        moduleName : moduleName, // 模块 Id
}
}

```

```

        status : 'loading',           // 模块对应文件加载状态（默认加载中）
        exports : null,              // 模块接口
        onload : [callback]         // 模块对应文件加载完成回调函数缓冲器
    };
    // 加载模块对应文件
    loadScript(getUrl(moduleName));
}

},
getUrl = function(){},
loadScript = function(){}

```

“小铭，你的思路是说，执行模块对应文件加载完成回调函数是为了使 module 修改内部的依赖模块统计变量 depsCount，直到所有依赖模块全部加载后顺利执行 setModule 吧，有些懂了，你的思路很巧妙呀。”小白惊叹。

“你说的很对，不过加载模块对应文件时需要引用 loadScript 加载脚本方法与 getUrl 获取文件路径方法，我们顺便将这两个方法实现了吧。”

```

// 获取文件路径
getUrl = function(moduleName) {
    // 拼接完整的文件路径字符串，如'lib/ajax' => 'lib/ajax.js'
    return String(moduleName).replace(/\.\w+$/g, '') + '.js';
},
// 加载脚本文件
loadScript = function(src) {
    // 创建 script 元素
    var _script = document.createElement('script');
    _script.type = 'text/JavaScript';          // 文件类型
    _script.charset = 'UTF-8';                  // 确认编码
    _script.async = true;                      // 异步加载
    _script.src = src;                         // 文件路径
    document.getElementsByTagName('head')[0].appendChild(_script); // 插入页面中
};

```

“前面说过要将模块的 id 和文件的路径对应的原因现在你明白了吧，良好的对应关系使我们加载该模块对应的文件更简单。有时同页面一起加载的模块，并且不会被其他模块引用，对于这些匿名模块我们有时候也可以不用添加模块 id。”

“上面打的伏笔原来埋藏得这么深”小白笑了笑。

36.6 设置模块

“现在我们把注意力放在最后一个核心方法 setModule 方法上。这个方法的实现很简单，却很巧妙。表面上看，该方法就是执行模块回调函数的。但实质上它做了 3 件事，第一，对创建的模块来说，当我的所有依赖模块加载完成时，我要使用该方法。第二，对于被依赖的模块来说，其所在的文件加载后要执行该依赖模块（即创建该模块过程）又间接地使用该方法。第

三，对于一个匿名模块来说（F.module 方法中无 url 参数数据），执行过程中也会使用该方法。”

```
/*
 * 设置模块并执行模块构造函数
 * @param moduleName 模块 id 名称
 * @param params 依赖模块
 * @param callback 模块构造函数
 */
setModule = function(moduleName, params, callback) {
    // 模块容器，模块文件加载完成回调函数
    var _module, fn;
    // 如果模块被调用过
    if(moduleCache[moduleName]){
        // 获取模块
        module = moduleCache[moduleName];
        // 设置模块已经加载完成
        module.status = 'loaded';
        // 纠正模块接口
        module.exports = callback ? callback.apply(_module, params) : null;
        // 执行模块文件加载完成回调函数
        while(fn = _module.onload.shift()){
            fn(_module.exports);
        }
    }else{
        // 模块不存在（匿名模块），则直接执行构造函数
        callback && callback.apply(null, params);
    }
}
```

36.7 学以致用

“到这里，我们的异步模块管理器就完成了，现在我们来体验一下异步模块开发的乐趣。首先我们在 lib/dom.js 中定义 dom 模块。对于 dom 模块来说，它不依赖任何其他模块。”

```
F.module('lib/dom', function(){
    return {
        // 获取元素方法
        g : function(id){
            return document.getElementById(id);
        },
        // 获取或者设置元素内容方法
        html : function(id, html){
            if(html)
                this.g(id).innerHTML = html;
            else
                return this.g(id).innerHTML;
        }
    });
});
```

“接下来我们要定义一个 event 模块，保存在 lib/event.js 文件中，不过由于为元素绑定事件要通过 id 获取元素，因此要引用 dom 模块。”

```

F.module('lib/event', ['lib/dom'], function(dom) {
    var events = {
        // 绑定事件
        on : function(id, type, fn) {
            dom.g(id)['on' + type] = fn;
        }
    }
    return events;
});

```

36.8 实现交互

“最后我们在页面中引用 dom 模块与 event 模块，为页面中的 button 绑定事件，添加交互。”

```

//index.html 页面中
F.module(['lib/event', 'lib/dom'], function(events, dom) {
    events.on('demo', 'click', function() {
        dom.html('demo', 'success');
    })
});

```

下章剧透

模块化开发是对系统功能的分解。本章我们一起学习了异步模块模式，它解决了异步加载的文件中的模块依赖问题。下一章我们将学习一种对页面分解开发模式，那是什么？我们拭目以待。

忆之获

模块化开发不仅解决了系统的复杂性问题，而且减少了多人开发中变量、方法名被覆盖的问题。通过其强大的命名空间管理，使模块的结构更合理。通过对模块的引用，提高了模块代码复用率。异步模块模式在此基础上增加了模块依赖，使开发者不必担心某些方法尚未加载或未加载完全造成的无法使用问题。异步加载部分功能也可将更多首屏不必要的功能剥离出去，减少首屏加载成本。

我问你答

开发中，常常会在为页面添加某些功能时，设置页面中某些模块的样式。那么在模块化开发中如何实现对样式文件（css 文件）的加载依赖呢。

第37章 分而治之——Widget模式

Widget: (Web Widget 指的是一块可以在任意页面中执行的代码块) Widget模式是指借用 Web Widget 思想将页面分解成部件，针对部件开发，最终组合成完整的页面。

模块化开发使页面的功能细化，逐一实现每个微功能，完成系统需求，这是一种很好的编程实践，不过对于页面视图的开发这种思想该如何实现呢？

37.1 视图模块化

“小铭，这几天学的模块化开发使页面中的功能管理有了很大改善，我们同事之间开发项目时再也不用担心相互影响了。可是我们在构造页面视图时还是常常会掺和在一起，有没有什么好办法解决呢？”小白问。

“对于页面视图的开发有一种模式叫 Widget 你听说过么？”

“难道是我孤陋寡闻了，竟不知道你说的这个模式”。

“这种模式是借助 Web Widget 思想予以实践的，它将页面粒度化，分解成一个个组件，当然一个组件也对应着一个模块，一个完整的组件包含该模块的完整的视图和一套完整的能力。”

“听你这么一说，我倒是有点感觉了，是不是像我们前几天探讨的对功能分割成一个个组件作为一个模块一样，分解视图开发呢。如果这样，我们就要对视图进行分割，将功能融合形成一套完整的组件，来解决视图开发中的耦合问题吧”。

“没错，所以 Widget 模式开发中一个组件就要对应一个文件了，而不是某个功能或者某个视图，因此在这个组件文件中你要做两件事，第一创建视图，第二添加相应的能力。不过对于添加功能我们之前已经讨论过，那么现在我们学习如何分解与创建视图。”小铭接着说，“既然创建视图就要有个模板渲染方法，才能使我们开发更高效。你还记得我们学习的简单模板模式中的那个 `formatString` 模板渲染方法吗？”

“记得呀，不就是那个格式化字符串模板的方法么，怎么了？”小白问。

37.2 模板引擎

“创建视图就要借用到简单模板模式的思想，用服务器端请求来的数据格式化我们视图模板实现对视图的创建，不过之前的方法太简单了不适合复杂视图创建，我们对它拓展一下，封装成一个模板功能组件 template。让他可以对页面元素模板、script 模板，表单模板，字符串模板格式化，甚至可以编译并执行 JavaScript 语句。”

“还有这样的方法？听上去好厉害，这不就是一个类似服务器端的模板引擎么（如 smarty 模板）”小白感觉不可思议。

“哈哈，心动了吧，那么我们赶紧把 template 组件实现吧。”

“不过，你能说说 template 的大体的实现思路么？让我理解下。”

37.3 实现原理

“很简单，分 4 步，第 2 步处理数据，第 3 步获取模板，第 4 步处理模板，第 4 步编译执行。所以我们按照这 4 个步做下去就可以了。不过在开始实现模板引擎之前我们要明白一件事，就是我们要做出什么效果，我们可参考下面的例子。”

模板：`{{=text%}}`

数据：`{is_selected:true, value:'zh', text:'zh-text'}`

» 输出结果：`zh-text`

37.4 模板引擎模块

“有了上面的模板渲染样品做参考，我们就可以按部就班地完成模板引擎中渲染的 4 个步骤了。不过，首先我们还是把我们组件的环境搭起来。”

```
// 模板引擎模块
F.module('lib/template', function() {
    // 模板引擎 处理数据与编译模板入口
    var _TplEngine = function() {},
        // 获取模板
        _getTpl = function() {},
        // 处理模板
        _dealTpl = function() {},
        // 编译执行
        _compileTpl = function() {};
    return _TplEngine;
});
```

“好熟悉的代码，这就是我们之前学过的模块化开发么。小铭，你这是要把模板引擎包装

成一个模块吧。”

“没错，这样以后使用时只需引用模板引擎模块依赖即可。”

```
F.module(['lib/template'], function(template) {
    // do something
})
```

37.5 处理数据

“下面我们要将模板引擎设计的4个步骤实现4个方法。第1步的处理数据方法要做两件事，如果传入的数据是对象，则直接渲染，如果数据是数组，则要遍历数组，逐一渲染，并将返回的字符串拼接在一起。”

```
/*
 * 模板引擎 处理数据与编译模板入口
 * @param str 模板容器 id 或者模板字符串
 * @param data 渲染数据
 */
TplEngine = function(str, data) {
    // 如果数据是数组
    if(data instanceof Array) {
        // 缓存渲染模板结果
        var html = '',
            // 数据索引
            i = 0,
            // 数据长度
            len = data.length;
        // 遍历数据
        for(; i < len; i++) {
            // 缓存模板渲染结果，也可写成 html += arguments.callee(str, data[i]);
            html += _getTpl(str)(data[i]);
        }
        // 返回模板渲染最终结果
        return html;
    } else{
        // 返回模板渲染结果
        return _getTpl(str)(data);
    }
}
```

37.6 获取模板

“第2步，获取模板方法_getTpl实现起来更为容易，如果str是一个id，并且可获得该id对应的元素，那么我们获取元素的内容或值（针对于表单元素），否则将str看作模板字符串直接处理。”

```
/*
 * 获得模板
*/
```

```

* @param str 模板容器 id, 或者模板字符串
*/
_getTpl = function(str){
    // 获取元素
    var ele = document.getElementById(str);
    // 如果元素存在
    if(ele){
        // 如果是 input 或者 textarea 表单元素, 则获取该元素的 value 值, 否则获取元素的内容
        var html = /^(textarea|input)$/.test(ele.nodeName) ? ele.value :
ele.innerHTML;
        // 编译模板
        return _compileTpl(html);
    }else{
        // 编译模板
        return _compileTpl(str);
    }
}

```

37.7 处理模板

“第3步，处理模板方法_dealTpl 要做的事情如下，首先要明确哪些内容是要被替换的，确定替换内容的左右分隔符。接下来要对模板字符串处理，将模板字符串分割并传入编译环境中的 template_array 数组中。由于处理的流程比较复杂，这里举例说明一下。”

例如：模板：`<a>{%=test%}`

处理后的形式为 `template_array.push('<a>',typeof(test)=='undefined'?'':test,'');`

“首先显性地将传入的内容转化为字符串，这是容错处理。然后将 html 常用标签内的‘<’和‘>’分别转义成‘<’和‘>’，将三类空白符（回车符\r，制表符\t，换行符\n）过滤掉。然后将‘{%=text%}’转化成‘',typeof(\$1)=='undefined'?'':\$1,'’形式。最后将‘%’替换成‘);’，将‘%’替换成‘template_array.push("")’。

```

_dealTpl = function(str){
    var _left = '{%',      // 左分隔符
        _right = '%}';    // 右分隔符
    // 显式转化为字符串
    return String(str)
        // 转义标签内的< 如: <div>{if(a<b)}</div> -> <div>%if(a<b)%</div>
        .replace(/&lt;/g, '<')
        // 转义标签内的>
        .replace(/&gt;/g, '>')
        // 过滤回车符, 制表符, 回车符
        .replace(/[\r\t\n]/g, '')
        // 替换内容
        .replace(new RegExp(_left+'=(.*?)+_right, 'g'), "", typeof($1) ==
'undefined'?'': $1,'')
        // 替换左分隔符
        .replace(new RegExp(_left, 'g'), "");"
        // 替换右分隔符
        .replace(new RegExp(_right, 'g'), "template_array.push('')");
}

```

37.8 编译执行

“第4步，编译执行是方法要将模板处理方法_dealTpl得到的模板字符串编译成最终的模板，所以我们要实现模板编译方法_compileTpl，这个方法是我们模板引擎的核心，也是最复杂的，它应用了函数声明技巧，通过new Function将我们模板字符串转化成函数体执行的语句，编译原理是先声明数据变量，然后用数据变量替换template_array内的变量，并得到结果。”

```
/*
 * 编译执行
 * @param str 模板数据
 */
_compileTpl = function(str) {
    // 编译函数体
    var fnBody = "var template_array=[];\nvar fn=(function(data){\nvar template_
key='';\nfor(key in data){\ntemplate_key+=('var '+key+'=data['+"'"+key+"']+');\n}\n}\n)\n(templateData);\nnf = null;\nreturn template_array.join('');");
    // 编译函数
    return new Function("templateData", fnBody);
},
```

“真是看晕了。fnBody 函数体也太复杂了，能不能帮我分析一下？”

“当然，其实编译的过程不难，我们只要了解编译的流程即可。”

```
"// 声明 template_array 模板容器组
var template_array=[];
// 闭包，模板容器组添加成员
var fn=(function(data){
    // 渲染数据变量的执行函数体
    var template_key='';
    // 遍历渲染数据
    for(key in data){
        // 为渲染数据变量的执行函数体添加赋值语句
        template_key+=('var '+key+'=data['+"'"+key+"']+');\n
    }
    // 执行渲染数据变量函数
    eval(template_key);
    // 为模板容器组添加成员（注意，此时渲染数据将替换容器中的变量）
    template_array.push(''+_dealTpl(str)+');\n
    // 释放渲染数据变量函数
    template_key=null;\n
    // 为闭包传入数据
}) (templateData);\n
// 释放闭包
fn = null;\n
// 返回渲染后的模板容器组，并拼接成字符串
return template_array.join('');
```

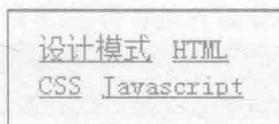
“小白，这回明白了吧？其实编译函数就是执行一遍变量赋值语句，并用渲染数据替换

template_array 容器内的变量，最终得到渲染后的模板字符串”。

37.9 几种模板

“还好，但依然感觉好复杂。现在可以使用我们辛辛苦苦创建的模板引擎了吧”。

“嗯，之前说了，你可以编译页面标签内的内容，也可以是表单元素内的内容，script 模板中的内容，甚至是手写的内容都可以。当然，模板引擎很强大，你也可以添加一些 JavaScript 语句。回到我们 Widget 模式中，比如我们页面中有个简单的标签云模块（如图 37-1 所示），在模块容器（div 元素）中有一些标签（a 元素），后端传输的数据控制我们前端标签云模块的展现内容，我们通过 Widget 模式实现它。首先我们要创建模板，有如下几种方式：”



▲图 37-1 标签云效果图

```
// 页面元素内容
<div id="demo_tag" class="template">
    <div id="tag_cloud">
        {%
            for(var i = 0, len = tagCloud.length; i < len; i++) {
                var ctx = tagCloud[i];
                <a href="#" class="tag_item"
                    {%
                        if(ctx['is_selected']){ %
                            selected
                        {%
                        } %
                    }
                    " title="{%=ctx["title"]%}">{%=ctx["text"]%}</a>
                {%
                } %
            }
        </div>
    </div>
// 表单元素内的内容
<textarea id="demo_textarea" class="template">
    <div id="tag_cloud">
        {%
            for(var i = 0, len = tagCloud.length; i < len; i++) {
                var ctx = tagCloud[i];
                <a href="#" class="tag_item"
                    {%
                        if(ctx["is_selected"]){ %
                            selected
                        {%
                        } %
                    }
                    " title="{%=ctx["title"]%}">{%=ctx["text"]%}</a>
                {%
                } %
            }
        </div>
    </div>
// script 模板内容
<script type="text/template" id="demo_script">
    <div id="tag_cloud">
        {%
            for(var i = 0, len = tagCloud.length; i < len; i++) {
                var ctx = tagCloud[i];
                <a href="#" class="tag_item"
                    {%
                        if(ctx["is_selected"]){ %

```

```

        selected
    { % } %
    " title="<%={ctx["title"]}%>">{=%=ctx["text"]%}</a>
    { % } %
    </div>
</script>
// 自定义模板
var demo_tpl = ['<div id="tag_cloud">',
    '{% for(var i = 0, len = tagCloud.length; i < len; i++){',
        'var ctx = tagCloud[i];%',
        '<a href="#" class="tag_item"',
        '{% if(ctx["is_selected"]){ %}',
            'selected',
        '{ % } %}',
        '" title="<%={ctx["title"]}%>">{=%=ctx["text"]%}</a>',
        '{ % } %',
    '</div>'].join('');

```

37.10 实现组件

“假设我们已经通过异步请求从服务器端获取到格式良好的数据：”

```

var data = {
    tagCloud : [
        {is_selected:true, title:'这是一本设计模式书', text:'设计模式'},
        {is_selected:false, title:'这是一本HTML书', text:'HTML'},
        {is_selected:null, title:'这是一本CSS书', text:'CSS'},
        {is_selected:'', title:'这是一本JavaScript书', text:'JavaScript'}
    ]
};

```

“完成我们的标签云组件模块很容易。”

```

/*widget/tag_cloud.js*/
F.module(['lib/template', 'lib/dom'], function(template, dom) {
    // 服务器端获取到 data 数据逻辑
    // 创建组件视图逻辑
    var str = template('demo_script', data);
    dom.html('test', str);
    // 组件其他交互逻辑
});

```

“这样就完成了一个组件。如果页面还有其他组件我们只需要为组件创建一个页面即可。”

下章剧透

本章通过学习 Widget 我们更深入地认识了模板引擎，并且可以将页面的视图与交互逻辑模块化，但也因此将视图与创建视图所需的数据以及组件的交互逻辑耦合在一起，如何更好地解决管理这类问题？下一章 MVC 模式将为我们解开答案。

忆之获

Widget 架构模式是页面开发模块化，不仅仅是页面功能，甚至页面的每个组件模块都可以独立地开发，这更适合团队中多人开发。并且降低相互之间因功能或者视图创建的耦合影响概率。一个组件即是一个文件，也让我们更好地管理一个页面，当然组件的多样化也会组建一个更丰富的页面，同样也会让组件的复用率提高，这是很有必要的。这也是组件开发的价值所在。

由于在前端开发中很少直接操作数据库（有时会操作一些前端数据库，如 localstorage 等），因此同步数据（页面加载时获得）与异步数据（ajax 等异步请求获取）是组件数据的主要来源，如果页面交互复杂，异步数据是主要来源。而数据的操作主要由服务器端负责，因此组件获取到的数据通常是可以满足前端需求格式的，除非服务器端对请求接口的重构需要前端适配新数据，否则通常获取的数据都能满足前端需求。

我问你答

运用 Widget 模式实现页面中的商品展示组件模块。

第38章 三人行——MVC模式

MVC 即模型（model）—视图（view）—控制器（controller），用一种将业务逻辑、数据、视图分离的方式组织架构代码。

组件式架构开发，常常将视图、数据、业务逻辑等写在一个模块内，如果组件内容很多，常常造成层次的混乱，增加开发与日后维护的成本。为让页面系统中的视图、数据、业务逻辑层次分明，这不小白正与他的小伙伴们研究 MVC 模式开发呢。

38.1 小白的顾虑

“小白，最近使用组件式（Widget）开发，有何感受呀。”

“模式很不错，不过当别人的组件很繁琐时，很难对内部的数据进行追踪，注释很少时，不知道代码块是处理数据，或者创建视图，还是其他交互逻辑，这导致同事间每次引用别人组件时都要详细咨询该组件的负责人。”

“哦，不过你知道刚才你说的现象是一种什么问题吗”小铭问。

“是功能的耦合吗？”

“层次的混乱，在页面开发中我们常常将，页面分成3个层次。视图层，像你刚才说的创建视图等。数据层，像你刚才提到的处理数据；业务逻辑层，比如你刚才说的交互逻辑等。是由于你在组件开发中没有分清3个层次，而在开发组件中随意书写，没有很好地管理3个层次内容，才会造成你所遇到的问题。”

“那有什么能解决这类问题的办法吗？”

“当然了，经典的MVC就是专门处理这类问题的。”

38.2 一个传说——MVC

“MVC？这我在上学时候可听说过，大名鼎鼎的MVC框架做的就是分离模型(model)层、视图（view）层、控制器（controller）层的吧？可是这也只是在学校里耳闻，要是在页面中实践我可一窍不通了。”

“你知道 MVC 是用来分层的就可以了，我们对页面处理也是指依据 MVC 的思想，将页面分成 3 层部分，数据层部分、视图层部分、控制器层部分。而我们知道，视图层可以调用数据层创建视图，控制器层可以调用数据层数据与视图层内的视图创建页面增添逻辑。因此我们就可以将页面简化成 3 个部分：”

```
// 为简化页面操作逻辑，这里引用链模式中实现的 A 框架，具体方法参考附录 A
// 页面加载后创建 MVC 对象
$(function() {
    // 初始化 MVC 对象
    var MVC = MVC || {};
    // 初始化 MVC 数据模型层
    MVC.model = function(){}();
    // 初始化 MVC 视图层
    MVC.view = function(){}();
    // 初始化 MVC 控制器层
    MVC.ctrl = function(){}();
});
});
```

“首先我们通过单体模式创建 MVC 对象，由于 MVC 对象要分 3 个层次，因此我们创建 3 个层次对象属性，分别是模型对象 model、视图对象 view、控制器对象 ctrl。剩下的事情就是要依次实现 3 个层次对象。”

“你写的 MVC 雏形我看明白了，就是将页面所有逻辑分成 3 个层次分别书写在 3 个对象中是吧。”

38.3 数据层

“当然了，不过你要注意到每个对象是一个自动执行的函数 `function(function(){})()`，你知道这是为什么吗？”小铭问。

“你之前说 3 个层次对象可被调用，而声明的函数在执行前是不能被调用的，比如模型对象要被视图和控制器调用，因此执行一遍是不是为其他对象调用提供接口方法呀！”

“很聪明，比如数据模型层内部的服务器端获取数据（data）与组件配置数据（config），为了能被视图层调用，我们要让数据模型对象返回操作这两类数据的接口方法。”

```
// MVC 数据模型层
MVC.model = function() {
    // 内部数据对象
    var M = {};
    // 服务器端获取的数据，通常通过 Ajax 获取并存储，后面的案例为简化实现，直接作为同步数据
    // 写在页面中，减少服务器端异步请求操作
    M.data = {};
    // 配置数据，页面加载时即提供
    M.conf = {};
    // 返回数据模型层对象操作方法
    return {
        // 获取服务器端数据
        getData : function(m) {
```

```

        // 根据数据字段获取数据
        return M.data[m];
    },
    // 获取配置数据
    getConf : function(c) {
        // 根据配置数据字段获取配置数据
        return M.conf[c]
    },
    // 设置服务器端数据（通常将服务器端异步获取到的数据，更新该数据）
    setData : function(m, v) {
        // 设置数据字段m对应的数据v
        M.data[m] = v;
        return this;
    }
    // 设置配置数据（通常在页面中执行某些操作，为做记录而更新配置数据）
    setConf : function(c, v) {
        // 设置配置数据字段c对应的配置数据v
        M.conf[c] = v;
        return this;
    }
}
}();

```

“真不错，通过 model 接口对象返回的 4 个操作方法即可对 model 内部的服务器端数据与配置数据做增删改查。这样我们可以在视图对象以及控制器对象内部轻松操作模型数据对象内部的数据了。”小白感叹。

38.4 视图层

“嗯，根据模型数据层对象的设计思想我们可以创建出视图对象，当然视图对象保存着对应组件内部的视图，为了创建这些视图我们还要在视图对象内部引用模型数据对象以操作模型数据对象内部的数据，最后为了让控制器可操作视图层内的视图，我们则需要返回一些操作接口方法。”

```

// MVC 视图层
MVC.view = function() {
    // 模型数据层对象操作方法引用
    var M = MVC.model;
    // 内部视图创建方法对象
    var V = {}
    // 获取视图接口方法
    return function(v) {
        // 根据视图名称返回视图(由于获取的是一个方法，这里需要将该方法执行一遍以获取相应视图)
        V[v]();
    }
}();

```

“视图层要比想象的简单多了，只有 3 个部分，一是操作模型数据对象方法的引用（M），二是内部视图创建方法对象，三是外部获取视图创建方法接口。”小白总结道。

38.5 控制器

“所以说 MVC 实现起来并不难，只是注意对页面层次的区分即可。最后一个层次对象是控制器对象，由于控制器对象要操作视图层对象与模型数据层对象，参考数据模型层与视图层的实现思想，将控制器层对象创建出来。”

```
// MVC 控制器层
MVC.ctrl = function() {
    // 模型数据层对象操作方法引用
    var M = MVC.model;
    // 视图数据层对象操作方法引用
    var V = MVC.view;
    // 控制器创建方法对象
    var C = {};
}
}();
```

38.6 侧边导航栏

“小白，MVC 的 3 个层次对象我们就创建完成了，下面我们要做的就是根据这 3 个层次来实现每一个组件模块，比如我们实现页面左侧的导航栏（如图 38-1 所示）。”



▲图 38-1 侧边导航栏

“当鼠标移到一个导航图标上显示该模块浮层，鼠标离开后浮层消失；当鼠标点击下面的箭头图标，导航浮层按照箭头方向隐藏或者显示。对于这个需求我们可以拆分出3个模块对象层次，第一显示这些导航模块的数据（文案以及图标）是可变的，因此需要从服务器端获取数据，属于模型数据层内部的数据，由于在导航显隐过程中需要屏蔽箭头按钮的操作功能，因此需要在页面中配置一个记录数据，而该数据应属于模型数据层内的配置数据。”

38.7 侧边导航栏数据模型层

```
MVC.model = function(){
    var M = {};
    M.data = [
        // 左侧侧边栏导航服务器端请求得到的响应数据
        slideBar : [
            {
                text : '萌妹子',
                icon : 'left_meng.png',
                title : '喵耳萝莉的千本樱',
                content : '自古幼女有三好~',
                img : 'left_meng_img.png',
                href : 'http://moe.hao123.com'
            },
            {
                text : '动漫',
                icon : 'left_comic.png',
                title : '喵耳萝莉的千本樱',
                content : '自古幼女有三好~',
                img : 'left_comic_img.png',
                href : 'http://v.hao123.com/dongman/'
            },
            {
                text : 'lol 直播',
                icon : 'left_lol.png',
                title : '喵耳萝莉的千本樱',
                content : '自古幼女有三好~',
                img : 'left_lol_img.png',
                href : 'http://www.hao123.com/video/lol'
            },
            {
                text : '网络剧',
                icon : 'left_tv.png',
                title : '喵耳萝莉的千本樱',
                content : '自古幼女有三好~',
                img : 'left_tv_img.png',
                href : 'http://www.hao123.com/video/yuanchuang'
            },
            {
                text : '热帖',
                icon : 'left_tie.png',
                title : '喵耳萝莉的千本樱',
                content : '自古幼女有三好~',
                img : 'left_tie_img.gif',
            }
        ]
    ]
}
```

```

        href : 'http://www.hao123.com/gaoxiao/retie'
    }
}
M.conf = {
    // 侧边导航动画配置数据
    slideBarCloseAnimate : false
}
return /*接口方法*/
}();

```

38.8 侧边导航栏视图层

“对于每个导航图标模块结构（包括浮层）都是类似的，因此他们有相同的视图模板，将侧边导航模块视图模块分为两部分：一部分是模块容器模板，另一部分是导航图标模块模板。为了得到侧边导航视图，我们需要用该模块对应的数据，通过模板渲染引擎 `formatString` 来（为简化案例实现，我们引用简单模板模式中定义的方法 `formatString`）渲染整个模块视图。”

```

MVC.view = function(){
    var M = MVC.model;
    var V = {
        // 创建侧边导航模块视图
        createSlideBar : function(){
            // 导航图标内容
            var html = '',
                // 视图渲染数据
                data = M.getData('slideBar');
            // 屏蔽无效数据
            if(!data || !data.length){
                return;
            }
            // 创建视图容器（参考附录 A 中，A 框架中创建元素方法 create）
            var dom = $.create('div', {
                'class' : 'slidebar',
                'id' : 'slidebar'
            });
            // 视图容器模板
            var tpl = {
                container : [
                    '<div class="slidebar-inner"><ul>{#content#}</ul></div>',
                    '<a href="#" class="slidebar-close" title="收起"/>',
                ].join(''),
                // 导航图标模块模板
                item : [
                    '<li>',
                    '<a class="icon" href="{#href#}">',
                    '',
                    '<span>{#text#}</span>',
                    '</a>',
                    '<div class="box">',
                    '<div>',
                    '<a class="title" href="{#href#}">{#title#}</a>',
                ]
            };
            // 将视图容器和图标模块模板拼接
            var result = '';
            for(var key in V) {
                if(V[key] != undefined) {
                    result += V[key];
                }
            }
            // 渲染视图容器
            dom.innerHTML = formatString(dom, html);
            // 渲染图标模块
            var items = data.items;
            for(var i = 0; i < items.length; i++) {
                var item = items[i];
                var itemTpl = formatString(tpl.item, item);
                dom.innerHTML += itemTpl;
            }
            // 渲染视图容器
            dom.innerHTML = formatString(dom, result);
            // 将视图容器插入到页面中
            $(dom).insertAfter('#header');
        }
    }
}

```

```

        '<a href="#{#href#}">{#content#}</a>',
        '</div>',
        '<a class="image" href="#{#href#}"></a>',
        '</div>',
        '</li>'
    ].join('')
};

// 渲染全部导航图片模块
for(var i = 0, len = data.length; i < len; i++) {
    html += $.formatString(tpl.item, data[i]);
}

// 在页面中创建侧边导航视图
dom
    // 向侧边导航模块容器中插入侧边导航视图
    .html(
        // 渲染导航视图 (content 为导航图片内容)
        $.formatString(tpl.container, {content : html})
    )
    // 将侧边导航模块容器插入页面中
    .appendTo('body');
}

return function(v) {
    V[v]();
}
}();

```

38.9 侧边导航栏控制器层

“通过调用视图层对象中某组件的视图方法，我们即可在页面中创建侧边导航栏视图。接下来我们要做的就是为视图添加交互方法，这一部分应当放在控制器对象中。获取视图元素，并为视图中的元素绑定事件交互，以及添加动画特效。”

```

MVC.ctrl = function(){
    var V = MVC.view;
    var M = MVC.model;
    var C = {
        // 侧边导航栏模块
        initSlideBar : function(){
            // 渲染导航栏模块视图
            V('createSlideBar');
            // 为每一个导航图标添加鼠标光标滑过与鼠标光标离开交互事件 (具体方法参考附录 A 中
            // A 框架)
            $('li', 'slidebar')
            // 鼠标移入导航 icon (图标) 显示导航浮层
            .on('mouseover', function(e){
                $(this).addClass('show');
            })
            // 鼠标移出导航 icon (图标) 隐藏导航浮层
            .on('mouseout', function(e){
                $(this).removeClass('show');
            });
            // 箭头 icon (图标) 动画交互
        }
    }
}

```

```
$('.slidebar-close', 'slidebar')
// 点击箭头 icon 时
.on('click', function(e){
    // 如果正在执行动画
    if(M.getConf('slideBarCloseAnimate')){
        // 终止操作
        return false;
    }
    // 设置侧边导航模块动画配置数据开关为打开状态
    M.setConf('slideBarCloseAnimate', true);
    // 获取当前元素（箭头 icon）
    var $this = $(this);
    // 如果箭头 icon 是关闭状态（含有 is-close 类）
    if($this.hasClass('is-close')){
        // 为侧边导航模块添加显示动画
        $('.slidebar-inner', 'slidebar')
            .animate({
                // 动画时间
                duration : 800,
                // 动画类型
                type : 'easeOutQuart',
                // 动画主函数
                main : function(dom){
                    // 每一帧改变导航模块容器 left 值
                    dom.css('left', -50 + this.tween * 50 + 'px');
                },
                // 动画结束时回调函数
                end : function(){
                    // 设置箭头 icon 为打开状态（删除 is-close）类
                    $this.removeClass('is-close');
                    // 设置侧边导航模块动画配置数据开关为关闭状态（此时可继续进行模块
                    // 显隐动画交互）
                    M.setConf('slideBarCloseAnimate', false);
                }
            });
        // 如果箭头 icon 是打开状态（不含 is-close 类）
    }else{
        // 为侧边导航模块添加显示动画
        $('.slidebar-inner', 'slidebar')
            .animate({
                // 动画时间
                duration : 800,
                // 动画类型
                type : 'easeOutQuart',
                // 动画主函数
                main : function(dom){
                    // 每一帧改变导航模块容器 left 值
                    dom.css('left', this.tween * -50 + 'px');
                },
                // 动画结束时回调函数
                end : function(){
                    // 设置箭头 icon 为打开状态（删除 is-close）类
                    $this.addClass('is-close');
                    // 设置侧边导航模块动画配置数据开关为关闭状态（此时可继续进行模块
                    // 显隐动画交互）
                    M.setConf('slideBarCloseAnimate', false);
                }
            });
    }
});
```

```
// 为侧边导航模块添加交互与动画特效
C.initSlideBar();
}();
```

38.10 执行控制器

“控制器要做的事情其实很简单，第一创建视图页面，第二添加交互与动画特效。而在MVC的控制器中，由于创建视图的主要逻辑在视图层对象中，因此也就弱化了控制器中创建对象的功能，我们只需一行搞定（直接调用视图层对象接口方法渲染），而控制器对象也将自己主要的精力放在交互与特效上。最后我们看到，为了实现控制器中的功能，我们显性地调用了C.initSlideBar();方法，但是如果模块很多，那么一次一次地调用会造成控制器内部混乱，对此有两种解决办法，第一种，可以将对象中的方法改为创建即执行，具体如下所示”

```
var C = {
    // 侧边导航栏模块
    initSlideBar : function(){}()
};

“第二种，可以在对象末尾处遍历内部对象 C 中的每一个方法并执行”
for(var i in C){
    // 如果模块方法存在则执行
    C[i] && C[i]();
}
```

38.11 增加一个模块

“这样设计程序，果然使结构层次清晰，用数据时可以专心处理数据，创建视图时也可责无旁贷，于是可以将更多经历专心于交互与特效上。”小白感叹。

“是呀，比如我们想添加一个导航模块，我们只需要为3个对象添加代码即可，比如我们添加一个新闻模块”。

```
MVC.model = function(){
    var M = {};
    M.data = {
        // 左侧侧边栏导航服务器端请求得到的响应数据
        slideBar : [
            // ...
        ],
        /* 新增模块追加代码 */
        newsMod : [
            // ...
        ]
    }
    M.conf = {
        // 侧边导航动画配置数据
        slideBarCloseAnimate : false
        /* 新增模块追加代码 */
        // newModConf...
    }
}
```

```

    }
    return {
        // ...
    }
}();
MVC.view = function(){
    var M = MVC.model;
    var V = {
        // 创建侧边导航模块视图
        createSlideBar : function(){
            // ...
        },
        /* 新增模块追加代码 */
        createNewMod : function(){
            // ...
        }
    }
    return function(v){
        V[v]();
    }
}();
MVC.ctrl = function(){
    var V = MVC.view;
    var M = MVC.model;
    var C = {
        // 侧边导航栏模块
        initSlideBar : function(){
            // ...
        },
        /* 新增模块追加代码 */
        initNewMod : function(){
            // ...
        }
    };
    // 为侧边导航模块添加交互与动画特效
    for(var i in C){
        C[i] && C[i]();
    }
}();

```

下章剧透

本章通过学习 MVC，我们学会了管理页面系统中的数据、视图、控制器（业务逻辑）3个层次，以及他们实现需求的工作原理。可是在视图层创建界面时常常会用到数据层内的数据，使视图层与模型层耦合在一起。这样降低了视图创建的灵活性与复用性，有什么解决办法吗？赶紧开始我们下一章之旅吧。

忆之获

MVC 架构模式很好地解决了页面中数据层、视图层、业务逻辑层（控制器）之间的耦合

关系，使它们得到显性的区分，这也使得层次之间的耦合度降低。我们在开发中可以不用顾忌所有需求而专注于某一层次开发，降低了开发与维护成本，提升了开发效率。如果页面系统足够复杂，某些视图要共享同一组数据，或者某些需求的实现引用类似视图，此时 MVC 模式便可提高某些视图与数据的复用率。

因此对于大型页面系统的开发，三个层次各司其职。每一层次专注于自己的事情，有利于工程化、模式化开发并管理代码；便于大型页面系统的可持续开发与维护；也是降低层次耦合、提升代码复用的良好实践。

在复杂组件的开发中，运用 MVC 思想管理组件内部的层次也是一种不错的选择。

我问你答

通过运用 MVC 模式，在页面中创建导航模块并实现导航二级菜单下拉与收起的交互功能。

第39章 三军统帅——MVP模式

MVP 即模型（Model）—视图（View）—管理器（Presenter）：View 层不直接引用 Model 层内的数据，而是通过 Presenter 层实现对 Model 层内的数据访问。即所有层次的交互都发生在 Presenter 层中。

MVC 模式开发中，视图层常常因渲染页面而直接引用数据层内的数据，对于发生的这一切，控制器常常不得而知。因此数据层内的数据修改，常常在控制器不知情的情况下影响到视图层的呈现。人的创造力是无穷的，为了解决这类问题，他们又是如何做的。

39.1 数据模型层与视图层联姻的代价

“小铭。上次我做的侧边栏导航想添加一个新消息提醒功能，数据层从服务器端获取的并保存的数据将改变，而完成这类需求我还要修改视图层侧边导航模块的展现，还要修改控制器层，为侧边导航添加消息提醒功能……可是如果需求变化很大，这样修改的成本很高呀”小白咨询小铭，有何解决办法。

“这是因为 MVC 模式中视图层要与数据层耦合在一起才出现你遇到的问题。解决这类问题很简单，将视图层与数据层解耦，统一交由控制器层管理就好了。”小铭答道。

“交由控制器处理？那控制器岂不是承包了之前视图层的业务逻辑？”

“当然，不过这种模式应该称之为 MVP 模式，这里的 P 不是控制层，而是管理层（presenter），他负责管理数据、UI 视图创建、交互逻辑、动画特效等等一切事务。因此管理层也就强大起来。这样数据层只负责存储数据，视图层只负责创建视图模板，他们的业务独立而又单一，因此我们添加或修改模块，只需要对管理层做处理就足够了。”

“听起来很不错，可是我该如何做才能实现你所说的呢？”

39.2 MVP 模式

“MVP 是在 MVC 模式中演变过来的，因此数据层不需要太多变化。但是视图层则不一样了，既然要与数据层解耦，并且可以独立创建视图模板，我们就要大刀阔斧地修改视图层了。

首先我们创建一个 MVP 单体对象。”

```
// MVP 模块
~(function(window) {
    // MVP 构造函数
    var MVP = function() {};
    // 数据层
    MVP.model = function() {};
    // 视图层
    MVP.view = function() {};
    // 管理层
    MVP.presenter = function() {};
    // MVP 入口
    MVP.init = function() {}
    // 暴露 MVP 对象，这样即可在外部访问 MVP
    window.MVP = MVP;
}) (window)
```

39.3 数据层的填补

“相对于 MVC 中的数据层对象结构，在 MVP 模式中数据层对象的结构变化不大”

```
// 数据层 与 MVC 模式中的数据层相似
MVP.model = function() {
    var M = {};
    M.data = {}
    M.conf = {}
    return {
        getData : function(m) {
            return M.data[m];
        },
        /**
         * 设置数据
         * @param m 模块名称
         * @param v 模块数据
         */
        setData : function(m, v) {
            M.data[m] = v;
            return v;
        },
        getConf : function(c) {
            return M.conf[c];
        },
        /**
         * 设置配置
         * @param c 配置项名称
         * @param v 配置项值
         */
        setConf : function(c, v) {
            M.conf[c] = v;
            return v;
        }
    }
}
```

```
    }();
```

“果然与 MVC 模式中的数据层结构类似，这是因为在 MVC 模式中数据层在职责金字塔中最底层（属于被视图层与控制器层操作的角色）的缘故吧。”小白问。

39.4 视图层的大刀阔斧

“的确，在 MVP 中对数据层改动不大，不过对于视图层可要改朝换代了。比如我们要创建一个导航（如图 39-1 所示）。”



▲图 39-1 导航

“为了在管理层中渲染并创建视图，渲染每一个导航都需要视图层提供一个导航视图模板。”

```
var tpl = [
  '<li class="#mode# {#choose#} {#last#}" data-mode="#mode#">',
  '  <a id="nav_{#mode#}" class="nav-{#mode#}" href="#url#" title="',
  '{#text#}">',
  '    <i class="nav-icon-{#mode#}"></i>',
  '    <span>{#text#}</span>',
  '  </a>',
  '</li>',
].join('');
```

“但是这种模板书写成本太高了，如果可以像 Zen Coding 那样创建模板该多好呀，比如：'li.@mode @choose @last[data-mode=@mode]>a#@mode.nav-@mode[href=@url title=@text]>i.nav-icon-@mode+span{@text}'字符串通过视图层处理便可得到上面的模板。为了实现上面的功能，我们要对视图层做如下处理”。

```
// 视图层
MVP.view = MVP.view = function(){
  return function(str){
    // 将参数字符串转换成期望模板
    return html;
  }
}();
```

39.5 模板创建的分层处理

“接下来我们要解析字符串并创建视图。对于参数字符串 str，我们做的第一步是分层，也就是说要确认每一个元素之间的层级关系，我们发现 ‘>’ 表示后面的元素是前面的元素的子

元素，而+表示前面元素与后面元素是兄弟元素。由于兄弟元素处在元素树中的同一层级上，因此我们要先做‘>’不同层级处理后作‘+’同一层级处理，最后针对每一个元素做处理并按层级顺序拼接成期望模板字符串。”

```

MVP.view = function(){
    // 子元素或者兄弟元素替换模板
    var REPLACEKEY = '__REPLACEKEY__';
    // 获取完整元素模板
    function getHTML(str, replacePos) {}
    /**
     * 数组迭代器
     * @param arr    数组
     * @param fn     回调函数
     */
    function eachArray(arr, fn){
        // 遍历数组
        for(var i = 0, len = arr.length; i < len; i++){
            // 将索引值、索引对应值、数组长度传入回调函数中并执行
            fn(i, arr[i], len);
        }
    }
    /**
     * 替换兄弟元素模板或者子元素模板
     * @param str    原始字符串
     * @param rep    兄弟元素模板或者子元素模板
     */
    function formateItem(str, rep){
        // 用对应元素字符串替换兄弟元素模板或者子元素模板
        return str.replace(new RegExp(REPLACEKEY, 'g'), rep);
    }
    // 模板解析器
    return function(str){
        // 模板层级数组
        var part = str
        // 去除首尾空白符
        .replace(/^\s+|\s+$/g, '')
        // 去除>两端空白符
        .replace(/^\s+(>)\s+/g, '$1')
        // 以>分组
        .split('>'),
        // 模块视图根模板
        html = REPLACEKEY,
        // 同层元素
        item,
        // 同级元素模板
        nodeTpl;
        // 遍历每组元素
        eachArray(part, function(partIndex, partValue, partLen){
            // 为同级元素分组
            item = partValue.split('+');
            // 设置同级元素初始模板
            nodeTpl = REPLACEKEY;
            // 遍历同级每一个元素
            eachArray(item, function(itemIndex, itemValue, itemLen) {

```

```

    // 用渲染元素得到的模板去渲染同级元素模板，此处简化逻辑处理
    // 如果 itemIndex ( 同级元素索引 ) 对应元素不是最后一个 则作为兄弟元素处理
    // 否则，如果 partIndex ( 层级索引 ) 对应的层级不是最后一层 则作为父层级处理
    // ( 该层级有子
    // 层级，即该元素是父元素 )
    // 否则，该元素无兄弟元素，无子元素
    nodeTpl = formatItem(nodeTpl, getHTML(itemValue, itemIndex ===
        itemLen - 1 ? (partIndex === partLen - 1 ? '' : 'in') : 'add'));
    });
    // 用渲染子层级得到的模板去渲染父层级模板
    html = formatItem(html, nodeTpl);
}
// 返回期望视图模板
return html;
}
}();

```

39.6 处理一个元素

“最后我们要做的就是对一个元素模板的渲染，即 `getHTML` 方法。`getHTML` 方法比较复杂，我们首先要分清该元素是否拥有子元素，或是拥有兄弟元素，或是最后一个叶子元素（既无子元素，后面也无兄弟元素）3 种情况。并针对 3 种情况做特殊处理。紧接着要将元素填补成完整元素，如 `div` 要转化成`<div></div>`。接下来要对元素的特殊属性 `id` (#标识) 或 `class` (.标识) 做处理。然后对元素的其他属性进行处理 ([]内的用空格分隔的属性组)。最后要将可替换内容标识 (@标识) 替换成代码库中模板渲染方法中可嗅探内容标识形式（比如我们引用的 A 框架中 `formatString` 方法的可嗅探内容标识为`{##}`）。”

```

/**
 * 获取完整元素模板
 * @param str      元素字符串
 * @param type     元素类型
 */
function getHTML(str, type) {
    // 简化实现，只处理字符串中第一个{}里面的内容
    return str
        .replace(/^(\\w+)([^\\{\\}]*?)?({{([@\\w]+)\\}})?(.*)$/ , function(match, $1,
        $2, $3, $4, $5) {
            $2 = $2 || '';      // 元素属性参数容错处理
            $3 = $3 || '';      // {元素内容}参数容错处理
            $4 = $4 || '';      // 元素内容参数容错处理
            $5 = $5.replace(/\{{([@\\w]+)\\}\}/g, ''); // 去除元素内容后面添加的元素属性
            // 中的{}内容
        })
    // 以 str=div 举例：如果 div 元素有子元素则表示成<div>_REPLACEKEY_</div>, 如果 div 有
    // /兄弟元素则表示成<div></div>_REPLACEKEY_, 否则表示成<div></div>
    return type === 'in' ?
        '<' + $1 + $2 + $5 + '>' + $4 + REPLACEKEY + '</' + $1 + '>' :
    type === 'add' ?
        '<' + $1 + $2 + $5 + '>' + $4 + '</' + $1 + '>' + REPLACEKEY :
        '<' + $1 + $2 + $5 + '>' + $4 + '</' + $1 + '>';
}

```

```

        })
        // 处理特殊标识#--id 属性
        .replace(/#([@\-\w]+)/g, ' id="$1"')
        // 处理特殊标识.--class 属性
        .replace(/\.\.([@\-\s\w]+)/g, ' class="$1"')
        // 处理其他属性组
        .replace(/\[(.+)\]/g, function(match, key){
            // 元素属性组
            var a = key
            // 过滤其中引号
            .replace(/^\|/g, '')
            // 以空格分组
            .split(' ')
            // 属性模板字符串
            h = '';
            // 遍历属性组
            for(var j = 0, len = a.length; j < len; j++) {
                // 处理并拼接每一个属性
                h += ' ' + a[j].replace(/=(.*)/g, '="$1"');
            }
            // 返回属性组模板字符串
            return h;
        })
        // 处理可替换内容，可根据不同模板渲染引擎自由处理
        .replace(/\@(\w+)/g, '{$$1}');
    }
}

```

39.7 改头换面的管理器

“有了模板引擎，我们在管理器（管理层 P）中实现就容易多了。为了使管理器更适合我们的 MVP 模式，我们对管理器稍加改动，添加管理器执行方法 init，这样方便在任何时候执行我们的管理器。但总的来说还是和控制器（MVC 模式中的控制层）类似。”

```

// 管理器层
MVP.presenter = function(){
    var V = MVP.view;
    var M = MVP.model;
    var C = {};
    return {
        // 执行方法
        init : function(){
            // 遍历内部管理器
            for(var i in C){
                // 执行所有管理器内部逻辑
                C[i] && C[i](M, V, i);
            }
        },
    };
}();

```

39.8 一个案例

“完整的 MVP 对象创建出来了，接下来我们创建一个导航，我们为管理器添加导航管理器逻辑。”

```
var C = {
    /**
     * 导航管理器
     * @param M    数据层对象
     * @param V    视图层对象
     */
    nav : function(M, V) {
        // 获取导航渲染数据
        var data = M.getData('nav');
        // 处理导航渲染数据
        data[0].choose = 'choose';
        data[data.length - 1].last = 'last';
        // 获取导航渲染模板
        var tpl = V('li.@mode @choose @last[data-mode=@mode]>a#nav_@mode.nav-@mode
[href=@url title=@text]>i.nav-icon-@mode+span{@text}');
        $(
            // 创建导航容器
            .create('ul', {
                'class' : 'navigation',
                'id' : 'nav'
            })
            // 插入导航视图
            .html(
                // 渲染导航视图
                A.formateString(tpl, data)
            )
            // 导航模块添加到页面中
            .appendTo('#container');

            // 其他交互逻辑与动画逻辑
            // .....
        );
    }
};
```

39.9 用数据装扮导航

“假设我们现在可以从后端获取导航模块数据并已经通过 setData 方法设置在数据层中。”

```
M.data = {
    // 导航模块渲染数据
    nav : [
        {
            text : '新闻头条',
            mode : 'news',
            url : 'http://www.example.com/01'
```

```

    },
    {
        text : '最新电影',
        mode : 'movie',
        url : 'http://www.example.com/02'
    },
    {
        text : '热门游戏',
        mode : 'game',
        url : 'http://www.example.com/03'
    },
    {
        text : '今日特价',
        mode : 'price',
        url : 'http://www.example.com/04'
    }
];
}
;

```

“万事俱备，只欠执行。那么，现在我们为 MVP 对象创建一个快捷执行方法 init。”

```

// MVP 入口
MVP.init = function(){
    this.presenter.init();
}

```

39.10 千呼万唤始出来的导航

“等到页面加载完毕后我们就可以渲染并创建我们的导航模块了。”

```

window.onload = function(){
    //执行管理器
    MVP.init();
}

```

“看，小白，这样我们就可以顺利完成我们的需求了，以后有什么变更只需要修改相应的管理器内容就可以了。”

39.11 模块开发中的应用

“视图层果然独立出来，不过以后添加其他模块，是不是还要找到管理器查找对应的模块代码修改呢？或者说，我在模块化开发中怎么更快地实现呢？”

“模块化开发中就不能按照上面的形式去实现了，在模块化开发中一个模块的实现是要依赖 MVP 对象实现的，因此我们要将 MVP 封装在模块内。”

```

F.module('lib/MVP', function(){
    // MVP 构造函数
    var MVP = function() {};
    // MVP 实现
})

```

```
// ....
return MVP;
});
```

39.12 MVP 构造函数

“有了 MVP 对象模块，我们就可以在其他模块中引用 MVP 模块了。不过目前为止我们还不能使用 MVP 为管理器添加其他控制器模块。所以我们完成 MVP 构造函数，实现通过 MVP（模块名称、模块管理器、服务器端获取的数据）的方式添加模块。”

```
// MVP 构造函数
var MVP = function(modName, pst, data) {
    // 在数据层中添加 modName 渲染数据模块
    MVP.model.setData(modName, data);
    // 在管理器层中添加 modName 管理器模块
    MVP.presenter.add(modName, pst);
};
```

39.13 增添管理器

“我们可以看出 MVP 构造函数做了两件事，首先为数据层添加模块数据，然后为管理器层添加管理器模块。我们已经在数据层 model 中实现了 setData 方法，所以只剩下管理器层中的 add 方法有待实现。”

```
// 管理器层
MVP.presenter = function() {
    // .....
    return {
        init : function(){},
        /**
         * 为管理器添加模块
         * @param modName 模块名称
         * @param pst     模块管理器
        */
        add : function(modName, pst){
            C[modName] = pst;
            return this;
        }
    };
}();
```

39.14 增加一个模块

“管理器的 add 方法允许我们以管理器名称+模块管理器的形式在管理器对象层中添加模块管理器，这样我们在外部就可以自由地添加模块了。比如我们创建一个简单的网址模块（如图 39-2 所示）。”

聚划算 1号店 九块邮 优购网 爱淘宝 1折网

▲图39-2 网址

“我们可以在外部模块中创建网址模块。”

```
/// 网址模块
F.module(['lib/MVP', 'lib/A'], function(MVP, $) {
    // 页面加载完成执行 参考附录中 A 框架
    $(function() {
        // 为 MVP 对象添加一个网址模块
        MVP(
            // 模块名称
            'sites',
            /**
             * 模块控制器
             * @param M          数据对象层引用
             * @param V          视图对象层引用
             * @param modName   模块名称
             */
            function(M, V, modName) {
                // 渲染模板<li><a href="#">{#text#}</a></li>
                var tpl = V('li>a[href="#">{@text}');
                $(
                    // 创建网址模块容器
                    .create('ul', {
                        'class' : 'store-nav',
                        'id' : modName
                    })
                    // 向网址模块容器中插入网址模块视图
                    .html(
                        // 创建网址模块视图
                        $.formatString(tpl, M.getData(modName))
                    )
                    // 插入页面中
                    .appendTo('#container');
                    // 其他交互与特效……
                },
                // 模块数据
                [
                    '聚划算',
                    '1号店',
                    '九块邮',
                    '优购网',
                    '爱淘宝',
                    '1折网'
                ]
            );
        });
    });
});
```

“模块创建完毕我们就可以执行所有模块控制器了。”

```
$(function() {
    MVP.init();
})
```

“当然对于模块间的通信我们还可以用观察者模式来实现，使我们的系统更加完善。”

下章剧透

本章我们学习了 MVP 模式，通过 MVP 模式我们可以解决视图层与数据层之间的耦合，但是每次创建页面时，视图层都要被管理器直接调用，也就是说创建什么样的视图由管理器说了算。然而如何才能让视图层更独立，自己当家做主，创建什么样的视图“我说了算”呢，下一章我们一起探索吧。

忆之获

MVP 与 MVC 相比最重要的特征就是 MVP 中将视图层与数据层完全解耦，使得对视图层的修改不会影响到数据层，数据层内的数据改动又不会影响到视图层。因此，我们在管理器中对数据或者视图灵活地调用就可使数据层内的数据与视图层内的视图得到更高效的复用。因此，MVP 模式也可以实现一个管理器，可以调用多个数据，或者创建多种视图，而且是不受限制的。因而管理器有更高的操作权限，因此对于业务逻辑与需求的实现只需专注于管理器的开发即可，当然管理器内过多的逻辑也使得其开发与维护成本提高。

我问你答

运用 MVP 模式，为页面创建新闻模块，并添加新闻管理层（控制新闻显示样式以及数量）交互。

第 40 章 视图的逆袭——MVVM 模式

MVVM 模式，模型（Model）-视图（View）-视图模型（ViewModel）：为视图层（View）量身定做一套视图模型（ViewModel），并在视图模型（ViewModel）中创建属性和方法，为视图层（View）绑定数据（Model）并实现交互。

上一章节里我们学习了 MVP 模式，不过在 MVP 中，实现何种需求（创建哪种页面）的主动权在管理器中，因此必须通过创建管理器实现需求。然而某些情况下，一些开发者对于复杂的 JavaScript 了解得不是很深入，操作管理器成本很大，他们能否直接通过 html 创建视图实现页面的需求呢？这种异想天开的想法可以实现吗？

40.1 视图层的思考

“小铭，每次实现页面某个组件需求时，总是要操作管理器或者控制器，然而有些时候，页面 UI 功能类似，为此创建的管理器或者控制器代码或多或少有些重复。MVP 模式给了我一些启发，既然我们可以将视图独立出来，那么我们可不可以通过创建视图反过来控制管理器实现组件需求呢。”小白问。

小铭听见感到很惊讶，“这么长时间你总算学会以创造性的思维思考问题了，现在你可以自己独挡一面了，以后再有什么项目让你开发我也放心了。话说回来，刚才你说的想法很不错，不过我再给你补充一下，创建视图即是创建页面内的视图，因此本质上就是在页面中书写 HTML 代码，因此如果将视图作用提升，通过在页面中直接书写 HTML 代码创建视图组件，让控制器或者管理器去监听这些视图组件，并处理这些组件完成预期功能，这种实现方式可以让那些只懂得 HTML 代码的开发者，轻松完成一些功能需求。”

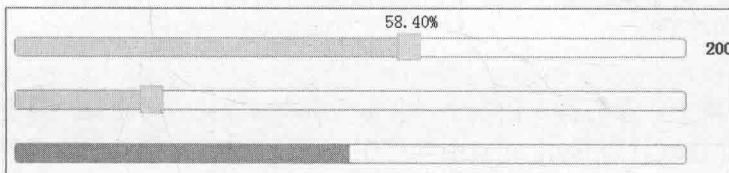
“听你这么一说豁然开朗，要是这样做那么我们是不是要对控制器做更复杂的封装呢？”

“没错，其实我们讨论的这个问题早已有实现，被江湖中人称之为 MVVM 模式，而我们所说的控制器或者管理器就是 MVVM 模式中的 VM 层，即视图模型层。而前面的 M 和 V 分别指的是数据模型层与视图层。因此在 MVVM 模式中 VM 是模式的核心。”

“原来我的想法早已有人实现了，这样实现起来我心里更有底了。”

40.2 滚动条与进度条

“既然你有这样的想法，那么我们写一个实例吧，比如我们要创建滚动条与进度条（如图 40-1 所示）。”



▲图 40-1 滚动条与进度条

“我们的预期目标是直接通过如下方式创建滚动条和进度条。”

```
<div class="first" data-bind="type : 'slider', data : demo1"></div>
<div class="second" data-bind="type : 'slider', data : demo2">test</div>
<div class="third" data-bind="type : 'progressbar', data : demo3"></div>
```

“太神奇了，不用写 JavaScript 代码，直接书写 HTML 代码就可以创建 3 个 UI 组件了？你是如何做到的？”小白对小铭的设想感到有些惊讶。

40.3 组件的探讨

“是不是感到很惊讶。”小铭笑了笑继续说，“总结上面代码特点，首先上面的 HTML 代码即是 MVVM 中的视图（V）层，3 个组件可分为两类，第一类是前两个，它们是滑动条组件，第二类是最后一个，它是进度条组件。我们可以通过 HTML 元素中自定义属性 data-bind 内的 type 值就可看出。而 data-bind 属性中的内容的格式与我们的对象很像，只是缺少一对大括号，后面的 data 值代表的是组件所需的数据模型，当然这些数据要在数据模型层中给出。”

“听你这么一说，对你的意图有些了解了，你是想通过 data-bind 自定义属性值为元素绑定 JavaScript 行为吧。”

“这句话说到点子上了。我之前说过，对于视图（V）层内的元素是要被视图模型（VM）层监听的，因此我们在视图模型（VM）层中实现对这些元素的监听，并为他们绑定行为。因此我们接下来要做的就是创建 VM 对象，不过在此之前我们要创建一个 VM 环境。”

40.4 视图模型层

```
// ~屏蔽压缩报错
~(function() {
    // 在闭包中获取全局变量
    var window = this || (0, eval)('this');
```

```

// 获取页面字体大小，作为创建页面 UI 尺寸参照物
var FONTSIZE = function() {
    // 获取页面 body 元素字体大小并转化成整数。
    return parseInt(document.body.currentStyle ? document.body.currentStyle['fontSize'] : getComputedStyle(document.body, false)['fontSize']);
}();
// 视图模型对象
var VM = function(){}();
// 将视图模型对象绑定在 Window 上，供外部获取
window.VM = VM;
})();

```

“在 VM 的环境中，我们首先获取全局变量 Window（当然对于非浏览器环境下全局变量并非 Window），并获取页面 body 元素中字号作为创建页面 UI 组件的参考尺寸。最后将模型视图（VM）层对象绑定在 Window 上供外部操作。接下来我们要做的事情就是处理视图模型对象 VM。在该对象中我们保存实例化组件方法，由于创建每类组件方法不同，因此我们将创建 UI 组件方法作为策略对象的方法保存，这样日后更容易拓展其他 UI 组件。”

```

var VM = function(){
    // 组件创建策略对象
    var Method = {
        // 进度条组件创建方法
        progressbar : function(){}
        // 滑动条组件创建方法
        slider : function(){}
    }
}();

```

“由于在我们的示例中只有两个 UI 组件，因此我们在策略对象中只创建两个组件策略方法，progressbar 可以创建进度条，slider 可以创建滑动条。下面我们依次实现这两个策略方法。”

“这应该是 MV 对象的核心吧，日后我们如果增加其他组件只需要在策略对象中添加创建组件的策略方法就可以了吧。”小白问。

40.5 创建进度条

“没错，当然你也可以为 Method 对象添加拓展方法，这样我们即可在外部拓展该策略对象以创建出更复杂多样的 UI 组件了。话又说回来，我们现在要把主要精力放在 progressbar 方法上，我们观察进度条组件，其结构很简单，在进度条组件内部只有一个完成进度容器，因此创建进度条组件很简单。”

```

/**
 * 进度条组件创建方法
 * dom 进度条容器
 * data 进度条数据模型
 */
progressbar : function(dom, data) {
    // 进度条进度完成容器
}

```

```

var progress = document.createElement('div'),
    // 数据模型数据, 结构: {position : 50}
    param = data.data;
// 设置进度完成容器尺寸
progress.style.width = (param.position || 100) + '%';
// 为进度条组件添加 UI 样式
dom.className += ' ui-progressbar';
// 进度完成容器元素插入进度条容器中
dom.appendChild(progress);
}

```

40.6 创建滑动条

“有了创建 progressbar 组件的经验我们再创建 slider 组件便轻松许多。我们观察滑动条，发现它们都有一个深灰色的滑动拨片、滑动容器以及滑动容器内部的深灰色的进度容器，当然对于最复杂的滑动条来说，它还要拥有容量提示信息与滑动拨片提示信息，因此我们按照示例中的滑动条的结构将滑动条创建出来。”

```

/*
 * 滑动条组件创建方法
 * dom 滑动条容器
 * data 滑动条数据模型
 */
slider : function(dom, data){
    // 滑动条拨片
    var bar = document.createElement('span'),
        //滑动条进度容器
        progress = document.createElement('div'),
        // 滑动条容量提示信息
        totleText = null,
        // 滑动条拨片提示信息
        progressText = null,
        // 数据模型数据, 结构: {position : 60, totle : 200}
        param = data.data,
        // 容器元素宽度
        width = dom.clientWidth,
        // 容器元素横坐标值
        left = dom.offsetLeft,
        // 拨片位置 (以模型数据中 position 数据计算)
        realWidth = (param.position || 100) * width / 100;
    // 清空滑动条容器, 为创建滑动条做准备
    dom.innerHTML = '';
    // 如果模型数据中提供容器总量信息 (param.totle), 则创建滚动条提示文案
    if(param.totle){
        // 容器总量提示文案
        text = document.createElement('b');
        // 拨片位置提示文案
        progressText = document.createElement('em');
        // 设置容器总量提示文案
        text.innerHTML = param.totle;
        // 将容器总量提示文案元素添加到滑动条组件中
        dom.appendChild(text);
    }
}

```

```

        // 将拨片位置提示文案元素添加到滑动条组件中
        dom.appendChild(progressText);
    }
    // 设置滑动条
    setStyle(realWidth);
    // 为滑动条组件添加 UI 样式
    dom.className += ' ui-slider';
    // 将进度容器添加到滑动条组件中
    dom.appendChild(progress);
    // 将拨片添加到滑动条组件中
    dom.appendChild(bar);
    // 设置滑动条
    function setStyle(w) {
        // 设置进度容器宽度
        progress.style.width = w + 'px';
        // 设置拨片横坐标
        bar.style.left = w - FONTSIZE / 2 + 'px';
        // 如果有拨片提示文案
        if(progressText) {
            // 设置拨片提示文案横坐标
            progressText.style.left = w - FONTSIZE / 2 * 2.4 + 'px';
            // 设置拨片提示文案内容
            progressText.innerHTML = parseFloat(w / width * 100).toFixed(2) + '%';
        }
    }
}
}

```

40.7 让滑动条动起来

“滚动条创建出来后我们还需简单为滚动条添加交互事件，这样滚动条才能被用户使用，比如当用户拖动滚动条拨片时，滚动条提示文案与拨片位置以及进度容器宽度都要随之改变，当鼠标松开时滑动交互停止。”

```

slider : function(dom, data) {
    // 创建组件逻辑
    // 按下鼠标拨片
    bar.onmousedown = function() {
        // 移动拨片（鼠标光标在页面中滑动，事件绑定给 document 是为了优化交互体验，使鼠标光标可以在页面中自由滑动）
        document.onmousemove = function(event) {
            // 获取事件源
            var e = event || window.event;
            // 鼠标光标相对于滑动条容器位置原点移动的横坐标
            var w = e.clientX - left;
            // 设置滑动条
            setStyle(w < width ? (w > 0 ? w : 0) : width);
        }
        // 阻止页面滑动选取事件
        document.onselectstart = function() {
            return false;
        }
    }
}

```

```
// 停止滑动交互（鼠标按键松开）
document.onmouseup = function() {
    // 取消文档鼠标光标移动事件
    document.onmousemove = null;
    // 取消文档滑动选取事件
    document.onselectstart = null;
}
}
```

40.8 为组件点睛

“创建组件的方法是 VM 对象中偏重于视图方面的逻辑，当然完整的 VM 对象还需要对模型数据的处理，因此我们要有一个获取数据的方法 getBindData，通过元素中给定的数据映射标识获取对应数据来渲染我们的视图。”

```
var VM = function() {
    var Method = {
        // .....
    }
    /**
     * 获取视图层组件渲染数据的映射信息
     * dom      组件元素
     */
    function getBindData(dom) {
        // 获取组件自定义属性 data-bind 值
        var data = dom.getAttribute('data-bind');
        // 将自定义属性 data-bind 值转化为对象
        return !!data && (new Function("return (" + data + ")"))();
    }
}();
```

40.9 寻找我的组件

“有了创建视图组件的方法以及获取组件映射数据的方法，对我们实现 VM 对象功能来说如虎添翼，所以，我们在 VM 中只需获取页面中那些需要渲染的组件即可。”

```
var VM = function() {
    var Method = {
        // .....
    }
    function getBindData() {}

    // 组件实例化方法
    return function() {
        // 获取页面中所有元素
        var doms = document.body.getElementsByTagName('*'),
            // 元素自定义数据句柄
            ctx = null;
        // ui 处理是会向页面中插入元素，此时 doms.length 会改变，此时动态获取 dom.length
        for(var i = 0; i < doms.length; i++) {
```

```

        // 获取元素自定义数据
        ctx = getBindData(dom[i]);
        // 如果元素是 UI 组件，则根据自定义属性中组件类型，渲染该组件
        ctx.type && Method[ctx.type] && Method[ctx.type](doms[i], ctx);
    }
}

}();

```

40.10 展现组件

“此时我们的 VM 部分基本完成，最后我们要为 M 数据模型层添加一些数据。”

```

// 数据模型层中获取到的组件渲染数据
// 带有提示文案的滑动条
var demo1 = {
    position : 60,
    totle : 200
},
// 简易版滑动条
demo2 = {
    position : 20
},
// 进度条
demo3 = {position : 50};

```

“我们渲染组件很容易。”

```

window.onload = function(){
    // 渲染组件
    VM();
}

```

“当然，为了完善组件，我们还可以为页面添加滚动条事件与窗口尺寸重置事件来优化组件交互。”

下章剧透

本章通过学习 MVVM 模式，我们学会了一两种灵活创建组件的方式。这是本书的最后一章。学到这里相信大家已有所成。那么接下来的模式……将由你来总结。

忆之获

MVVM 模式与前面的 MVC 模式、MVP 模式类似，或者说 MVVM 模式是由这两种模式演变而来的，因此它也主要用来分离视图（View）和数据模型（Model）。不同的是 MVVM 模式使视图层更灵活，可以独立于数据模型层、视图模型层而独立修改，自由创建。当然这也使

得数据模型层可以独立变化，甚至一个视图模型层可以对应多个视图层或者数据模型层。MVVM 模式是对视图模型层的高度抽象，因此当多个视图层对应同一个视图模型层时，也使得视图模型层内的代码逻辑变得高度复用。而这种开发模式最重要的一个特征恐怕就是视图层的独立开发，这样可以使那些不懂 JavaScript 的人，只要了解 HTML 内容并按照视图层规范格式创建视图即可完成一个复杂的页面开发，而让那些开发人员可专注于开发视图模型层里面的业务逻辑了。当然测试问题在 MVVM 中也变得很容易，只需要针对视图模型层撰写测试代码即可。

我问你答

通过运用 MVVM 模式，创建带有下拉框功能的输入框组件。

附录 A

```
/***
 * A Library v1.0.0
 *
 * Author      Zhangrongming
 * Date: 2014-11-30
 */
~(function(window){
    /**
     * @name 框架单体对象 A
     * @param selector 选择器或页面加载回调函数
     * @param context 查找元素上下文
     */
    var A = function(selector, context){
        // 如果 selector 为方法则为窗口添加页面加载完成事件监听
        if(typeof selector == 'function'){
            A(window).on('load', selector);
        }else{
            // 创建 A 对象
            return new A.fn.init(selector, context);
        }
    }
    // 原型方法
    A.fn = A.prototype = {
        // 强化构造函数
        constructor : A,
        // 构造函数
        init : function(selector, context){
            // modify 选择器为元素
            if(typeof selector === 'object'){
                this[0] = selector;
                this.length = 1;
                return this;
            };
            // 设置获取到的元素长度属性
            this.length = 0,
            // 纠正上下文
            context = document.getElementById(context) || document;
            // 如果是 id 选择器
            if(~selector.indexOf('#')){
                this[0] = document.getElementById(selector.slice(1));
                this.length = 1;
            }
        }
    }
})
```

```

// 如果是类选择器
} else if(~selector.indexOf('.')){
    var doms = [],
        className = selector.slice(1);
    // 支持通过类获取元素的方法
    if(context.getElementsByClassName){
        doms = context.getElementsByClassName(className);
    } else{
        doms = context.getElementsByTagName('*');
    }
    // 设置获取到的元素
    for(var i = 0, len = doms.length; i < len; i++){
        if(doms[i].className
&& !~doms[i].className.indexOf(className)){
            this[this.length] = doms[i];
            // 纠正长度
            this.length++;
        }
    }
}
// 否则为元素名选择器
} else{
    var doms = context.getElementsByTagName(selector),
        i = 0,
        len = doms.length;
    for(; i < len; i++){
        this[i] = doms[i];
    }
    this.length = len;
}
// 设置当前对象的选择上下文
this.context = context;
// 设置当前对象的选择器
this.selector = selector;
return this;
},
// 元素长度
length : 0,
// 增强数组
push: [].push,
splice: [].splice
}
// 设置构造函数原型
A.fn.init.prototype = A.fn;
/***
 * @name 对象拓展
 * @param[0] 目标对象
 * @param[1,...] 拓展对象
 ***/
A.extend = A.fn.extend = function(){
    var i = 1,
        len = arguments.length,
        target = arguments[0],
        j;
    // 如果一个参数，则为当前对象拓展方法
    if(i == len){
        target = this;
        i--;
    }
}

```

```

}

// 遍历拓展对象
for(; i < len; i++) {
    // 遍历拓展对象中方法与属性
    for(j in arguments[i]){
        // 浅复制
        target[j] = arguments[i][j];
    }
}
// 返回目标对象
return target;
};

// 单体对象 A 方法拓展
A.extend({
    /**
     * @name 将横线式命名字符串转化为驼峰式
     * eg : 'test-demo' -> 'testDemo'
     */
    camelCase : function(str){
        return str.replace(/-(\w)/g, function(match, letter){
            return letter.toUpperCase();
        });
    },
    /**
     * @name 去除字符串两端空白符
     * eg : ' t es t ' -> 't es t'
     */
    trim : function(str){
        return str.replace(/^\s+|\s$/g, '')
    }
    /**
     * @name 创建一个元素并包装成 A 对象
     * @param type 元素类型
     * @param value 元素属性对象
     */
    ,create : function(type, value){
        var dom = document.createElement(type);
        return A(dom).attr(value);
    }
    /**
     * @name 格式化模板
     * @param str 模板字符串
     * @param data 渲染数据
     * eg: '<div>{#value#}</div>' + {value:'test'} -> '<div>test</div>'
     */
    ,formatString : function(str, data){
        var html = '';
        //如果渲染数据是数组，则遍历数组并渲染
        if(data instanceof Array){
            for(var i = 0, len = data.length; i < len; i++){
                html += arguments.callee(str, data[i]);
            }
        }
        return html;
    }else{
        // 搜索{#key#}格式字符串，并在 data 中查找对应的 key 属性替换
        return str.replace(/\{\#(\w+)\#\}/g, function(match, key){
            return typeof data === 'string' ? data : (typeof data[key] ===
    
```

```

'undefined' ? '' : data[key]));
    }
}

});

// 事件绑定方法
var _on = (function(){
    // 如果标准浏览器
    if(document.addEventListener){
        return function(dom, type, fn, data){
            dom.addEventListener(type, function(e){
                fn.call(dom, e, data);
            }, false);
        }
    }
    // 如果 IE 浏览器
    else if(document.attachEvent){
        return function(dom, type, fn, data){
            dom.attachEvent('on' + type, function(e){
                fn.call(dom, e, data);
            });
        }
    }
    // 如果是老版本浏览器
    else{
        return function(dom, type, fn, data){
            dom['on' + type] = function(e){
                fn.call(dom, e, data);
            };
        }
    }
})();
A.fn.extend({
    // 添加事件
    on : function(type, fn, data){
        var i = this.length;
        for(; --i >= 0;){
            // 通过闭包实现对 i 变量保存
            _on(this[i], type, fn, data);
        }
        return this;
    },
    // 设置或者获取元素样式
    css : function(){
        var arg = arguments,
            len = arg.length;
        // 如果无获取到的元素则返回
        if(this.length < 1){
            return this;
        }
        // 如果是一个参数
        if(len === 1){
            // 如果参数是字符串则返回获取到的第一个元素的样式
            if(typeof arg[0] === 'string'){
                // ie 浏览器
                if(this[0].currentStyle){
                    return this[0].currentStyle[name];
                }else{
                    return getComputedStyle(this[0], false)[name];
                }
            }
        }
    }
});

```

```

        }
        // 如果参数为对象则为获取到的所有元素设置样式
    }else if(typeof arg[0] === 'object'){
        for(var i in arg[0]){
            for(var j = this.length - 1; j >= 0; j--){
                this[j].style[A.camelCase(i)] = arg[0][i];
            }
        }
    }
    // 如果两个参数
}else if(len === 2){
    // 为获取到的所有元素设置样式
    for(var j = this.length - 1; j >= 0; j--){
        this[j].style[A.camelCase(arg[0])] = arg[1];
    }
}
return this;
}
// 设置或者获取元素属性
,attr : function(){
    var arg = arguments,
        len = arg.length;
    // 如果无获取到的元素则返回
    if(this.length < 1){
        return this;
    }
    // 如果是一个参数
    if(len === 1){
        // 如果参数是字符串则返回获取到的第一个元素的属性值
        if(typeof arg[0] === 'string'){
            return this[0].getAttribute(arg[0]);
        }
        // 如果参数为对象则为获取到的所有元素设置属性
    }else if(typeof arg[0] === 'object'){
        for(var i in arg[0]){
            for(var j = this.length - 1; j >= 0; j--){
                this[j].setAttribute(i, arg[0][i]);
            }
        }
    }
    // 如果是两个参数
}else if(len === 2){
    // 为获取到的所有元素设置属性
    for(var j = this.length - 1; j >= 0; j--){
        this[j].setAttribute(arg[0], arg[1]);
    }
}
return this;
}
// 获取或者设置元素内容
,html : function(){
    var arg = arguments,
        len = arg.length;
    // 如果无获取到的元素则返回
    if(this.length < 1){
        return this;
    }
    // 如果无参数则返回获取到的第一个元素内容
}

```

```

if(len === 0){
    return this[0].innerHTML;
// 如果是一个参数，则设置获取到的所有元素内容
} else if(len === 1){
    for(var i = this.length - 1; i >= 0; i--){
        this[i].innerHTML = arg[0];
    }
// 如果两个参数，且第二个参数值为 true，则为获取到的所有元素追加内容
} else if(len === 2 && arg[1]){
    for(var i = this.length - 1; i >= 0; i--){
        this[i].innerHTML += arg[0];
    }
}
return this;
}
/**
 * @name 判断类存在
 * @param val 类名
 */
,hasClass : function(val){
    // 如果无获取到的元素则返回
    if(!this[0]){
        return;
    }
    // 类名去除首尾空白符
    var value = A.trim(val);
    // 如果获取到的第一个元素类名包含 val 则返回 true，否则返回 false
    return this[0].className&&this[0].className.indexOf(value)>=0?true:false;
}
/**
 * @name 添加类
 * @param val 类名
 */
,addClass : function(val){
    var value = A.trim(val),
        str = '';
    // 遍历所有获取到的元素
    for(var i = 0, len = this.length; i < len; i++){
        str = this[i].className;
        // 如果元素类名包含添加类则为元素添加类
        if(!~str.indexOf(value)){
            this[i].className += ' ' + value;
        }
    }
    return this;
}
/**
 * @name 移除类
 * @param val 类名
 */
,removeClass : function(val){
    var value = A.trim(val),
        classNameArr,           // 将元素类名转化为数组
        result;                 // 元素类名最终结果
    // 遍历所有获取到的元素
    for(var i = 0, len = this.length; i < len; i++){
        // 如果类名包含删除类

```

```

if(this[i].className && ~this[i].className.indexOf(value)) {
    // 通过空格符将元素类名切割成数组
    classNameArr = this[i].className.split(' ');
    result = '';
    // 遍历类名
    for(var j = classNameArr.length - 1; j >= 0; j--) {
        // 去除类名首尾空白符
        classNameArr[j] = A.trim(classNameArr[j]);
        // 如果类名存在并且类名不等于移除类，则保留该类
        result+=classNameArr[j]&&classNameArr[j]!=value?''+className
    }
    Arr[j]='';
}
// 重置元素类名
this[i].className = result;
}
return this;
}
/**
 * @name 插入元素
 * @param parent 父元素
 */
appendTo : function(parent) {
    var doms = A(parent);
    // 如果获取到父元素
    if(doms.length) {
        // 遍历父元素
        for(var j = this.length - 1; j >= 0; j--) {
            // 简化元素克隆(cloneNode)操作，只向第一个父元素中插入子元素
            doms[0].appendChild(this[j]);
        }
    }
}
});

// 运动框架单体对象
var Tween = {
    // 计时器句柄
    timer : 0,
    // 运动成员队列
    queen : [],
    // 运动间隔
    interval : 16,
    // 缓冲函数
    easing : {
        // 默认运动缓存算法 匀速运动
        def : function (time, startValue, changeValue, duration) {
            return changeValue * time / duration + startValue
        },
        // 缓慢结束
        easeOutQuart: function (time, startValue, changeValue, duration) {
            return -changeValue*((time=time/duration-1)*time*time*time-1)+startValue;
        }
    }
};

/**
 * @name 添加运动成员

```

```

* @param instance 运动成员
*/
add : function(instance){
    // 添加成员
    this.queen.push(instance);
    // 运行框架
    this.run();
},
/**
 * @name 停止框架运行
 */
clear : function(){
    clearInterval(this.timer);
    this.timer = 0;
},
/**
 * @name 运行框架
 */
run : function(){
    // 如果在运行则返回
    if(this.timer)
        return;
    // 重置计时器
    this.clear();
    // 运行框架
    this.timer = setInterval(this.loop, this.interval);
},
/**
 * @name 运动框架循环方法
 */
loop : function(){
    // 如果运动队列中没有成员
    if(Tween.queen.length === 0){
        // 停止框架运行
        Tween.clear();
        // 返回
        return;
    }
    // 获取当前时间
    var now = +new Date();
    // 遍历运动成员
    for(var i = Tween.queen.length - 1; i >= 0; i--){
        // 获取当前成员
        var instance = Tween.queen[i];
        // 当前成员已运动的时间
        instance.passed = now - instance.start;
        // 如果当前成员已运动的时间小于当前成员运动时间
        if(instance.passed < instance.duration){
            // 执行当前成员主函数
            Tween.workFn(instance);
        }else{
            // 结束当前成员运行
            Tween.endFn(instance);
        }
    }
},
/**

```

```

* @name 运行方法
* @param instance 运动成员
*/
workFn : function(instance){
    // 获取当前成员在当前时刻下的运动进程
    instance.Tween = this.easing[instance.type](instance.passed, instance.
    from,
    instance.to - instance.from, instance.duration);
    // 执行主函数
    this.exec(instance);
},
/***
 * @name 结束方法
* @param instance 运动成员
*/
endFn : function(instance){
    instance.passed = instance.duration;
    instance.Tween = instance.to;
    this.exec(instance);
    this.distory(instance);
},
/***
 * @name 执行主函数
* @param instance 运动成员
*/
exec : function(instance){
    try{
        // 执行当前成员主函数
        instance.main(instance.dom)
    }catch(e){}
},
/***
 * @name 注销运动成员
* @param instance 运动成员
*/
distory : function(instance){
    // 结束当前成员
    instance.end();
    // 在运动成员队列中删除该成员
    this.Tween.splice(this.Tween.indexOf(instance), 1);
    // 删除成员中的每一个属性,
    for(var i in instance){
        delete instance[i];
    }
}
/***
 * @name 获取当前成员在运动成员中的位置
* @param instance 运动成员
*/
Tween.Tween.indexOf = function(){
    var that = this;
    // 如果有该方法则返回,如果没有则创建一个方法
    return Tween.Tween.indexOf || function(instance){
        // 遍历每个成员
        for(var i = 0, len = that.length; i < len; i++){
            // 如果该成员是需求成员则返回该成员在队列中的位置
            if(that[i] === instance){

```

```

        return i;
    }
}
// 否则返回-1，表示不存在
return -1;
}
}();
// A.fn 对象拓展方法
A.fn.extend({
    /**
     * @name 动画模块
     * @param obj 动画成员对象
     */
    animate : function(obj){
        // 适配运动对象
        var obj = A.extend({
            duration : 400,           // 默认运行时间
            type : 'def',             // 默认动画缓存函数
            from: 0,                  // 开始点
            to : 1,                   // 结束点
            start : +new Date(),      // 开始时间
            dom : this,                // 当前元素
            main : function(){},       // 运行主函数
            end : function(){},        // 结束函数
        }, obj);
        // 向运动框架中载入运动对象成员
        Tween.add(obj);
    }
});
/**
 * @name 避免框架别名冲突（主要用于页面中引入多核框架）
 * @param library 其他框架
 */
A.noConflict = function(library){
    // 如果传递其他框架
    if(library){
        // 为 library 绑定$别名
        window.$ = library;
    }else{
        // 否则删除$别名
        window.$ = null;
        delete window.$;
    }
    // 返回 A 对象
    return A;
}
// 为全局对象绑定 A 框架，并绑定别名$
window.$ = window.A = A;
})(window);

```