JavaScript Patterns

JavaScript 模式

O'REILLY® | YAHOO! PRESS

中国电力出版社

Stoyan Stefanov 著

陈新 译

JavaScript模式

Stoyan Stefanov 著

陈新 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

JavaScript模式/（美）斯特凡洛夫（Stefanov, S.）著；陈新译. －北京：中国电力出版社，2012.4

书名原文：JavaScript Patterns

ISBN 978-7-5123-2923-2

I. J… II. ①斯… ②陈… III. ①Java语言－程序设计 IV. ①TP312

中国版本图书馆CIP数据核字（2012）第073567号

北京市版权局著作权合同登记

图字：01-2010-7922号

书　　名/ JavaScript模式

书　　号/ ISBN 978-7-5123-2923-2

责任编辑/ 刘炽

封面设计/ Karen Montgomery，张健

出版发行/ 中国电力出版社（http://www.cepp.sgcc.com.cn）

地　　址/ 北京市东城区北京站西街19号（邮政编码100005）

经　　销/ 全国新华书店

印　　刷/ 航远印刷有限公司印刷

开　　本/ 787毫米×980毫米　16开本　13.75印张　253千字

版　　次/ 2012年7月第一版　2012年7月第一次印刷

印　　数/ 0001－3000册

定　　价/ 38.00元（册）

目录

前言 1

第1章 简介 5

模式 5

JavaScript：基本概念 7

ECMAScript 5 9

JSLint 10

Console 10

第2章 基本技巧 12

编写可维护的代码 12

尽量少用全局变量 13

for循环 18

for-in循环 20

不要增加内置的原型 23

switch模式 23

避免使用隐式类型转换 24

使用parseInt()的数值约定 26

编码约定 27

命名约定 31

编写注释 33

编写API文档 34

编写可读性强的代码 38

同行互查 39
在正式发布时精简代码 39
运行JSLint 40
小结 41

第3章 字面量和构造函数 42

对象字面量 42
自定义构造函数 45
强制使用new的模式 47
数组字面量 50
JSON 52
正则表达式字面量 54
基本值类型包装器 55
错误对象 57
小结 58

第4章 函数 59

背景 59
回调模式 64
返回函数 69
自定义函数 70
即时函数 72
即时对象初始化 75
初始化时分支 77
函数属性——备忘模式 78
配置对象 80
Curry 81
小结 86

第5章 对象创建模式 89

命名空间模式 89
声明依赖关系 93

私有属性和方法 94
模块模式 99
沙箱模式 103
静态成员 108
对象常量 111
链模式 113
method()方法 115
小结 116

第6章 代码复用模式 117

传统与现代继承模式的比较 117
使用类式继承时的预期结果 118
类式继承模式#1——默认模式 119
类式继承模式#2——借用构造函数 122
类式继承模式#3——借用和设置原型 125
类式继承模式#4——共享原型 126
类式继承模式#5——临时构造函数 128
Klass 130
原型继承 133
通过复制属性实现继承 136
借用方法 139
小结 142

第7章 设计模式 143

单体模式 143
工厂模式 148
迭代器模式 151
装饰者模式 153
策略模式 158
外观模式 160
代理模式 162
中介者模式 170

观察者模式.....173
小结.....181

第8章 DOM和浏览器模式.....182

关注分离.....182
DOM脚本.....184
事件.....186
长期运行脚本.....190
远程脚本.....192
配置JavaScript.....197
载入策略.....199
小结.....208

前言

模式是针对普遍问题的解决方案。更进一步地说，模式是解决一类特定问题的模板。

模式会有助于将问题分解为一个个像积木一样的小模块，并集中精力处理从各种烦琐的细节中提炼出来的该问题所特有的部分。

模式通过提供一个通用的词典来帮助我们更好地进行交流，因此学习并识别模式是很重要的。

目标读者

这本书不是一本入门级的书，而是适用于希望将自身的JavaScript技巧提高到一个新层次的专业的开发人员和程序员。

本书并未讨论一些基本的原理（例如循环、条件语句和闭包等）。如果发现需要复习这些主题，请参考推荐读物列表。

同时，一些主题（例如创建和提升对象）可能在本书中看起来太基础了，但是我认为它们对于充分发挥该语言的能力有很大作用，因此从模式的视角来讨论了这些主题。

如果正在寻找更好的实践和更强大的模式来编写更优、更好的可维护性和更强健的JavaScript代码，那么这本书正是您所要的。

本书使用的排版约定

本书使用如下排版约定：

斜体（Italic）

用来表示新术语、URL、Email地址、文件名、文件扩展名等。

等宽字体（Constant width）

用来表示程序列表，同时在段落中引用的程序元素（例如变量、函数名、数据库、数据类型、环境变量、声明和关键字等）也用该格式表示。

等宽粗体（**Constant width bold**）

用于表示需要用户逐字符输入的命令或其他文本。

等宽*斜体（Constant width italic）*

用于表示应该以用户提供的值或根据上、下文决定的值加以替换的文本。

如何使用代码范例

本书可以帮助您完成工作。一般来说，可以在自己的程序和文档中使用本书的代码。除非将重新编译代码中重要的部分，您是不需要联系我们来获得授权的。举例来说，使用书中几段程序来编程是不需要获得我们的授权的，但是销售或发布O'Reilly出版书籍中配套光盘中代码是需要授权的。通过引用本书中范例来回答问题是不需要授权的，而将本书中重要部分的范例代码整合到您产品的文档中是需要授权的。

我们感谢您在使用我们代码的时候给出引用说明，但这不是硬性规定。一个引用说明通常包括了标题、作者、出版社和ISBN。举例来说，本书的引用说明“JavaSc Patterns, by Stoyan Stefanov（O'Reilly）. Copyright 2010 Yahoo!, Inc.,9780596806750.”

如果您不确定所使用的范例代码是否超出了上面给定的权限，可以随时通过电子邮件联系我们。我们的电子邮件地址是*permissions@oreilly.com*。

如何联系我们

请将关于本书的意见和问题发送给出版社：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

我们为本书提供了网页，该网页上面列出了勘误表、范例和任何其他附加的信息。您可以访问如下网页获得：

原文书：

http://www.oreilly.com/catalog/9780596806750

中文书：

http://www.oreilly.com.cn/index.php?func=book&isbn=978-7-5123-2923-2

要询问技术问题或对本书提出建议，请发送电子邮件至：

bookquestions@oreilly.com

要获得更多关于我们的书籍、会议、资源中心和O'Reilly网络的信息，请参见我们的网站：

http://www.oreilly.com

http://www.oreilly.com.cn

致谢

我一直都受惠于社区的一些难以置信的评论者，他们分享自己知识和能力，从而使得本书变得更好。他们的博客和推特具有让人敬畏的、敏锐的观察力，并包含了伟大的思想和模式。

- Dmitry Soshnikov (*http://dmitrysoshnikov.com*, *@DmitrySoshnikov*)
- Andrea Giammarchi (*http://webreflection.blogspot.com*, *@WebReflection*)
- Asen Bozhilov (*http://asenbozhilov.com*, *@abozhilov*)
- Juriy Zaytsev (*http://perfectionkills.com*, *@kangax*)
- Ryan Grove (*http://wonko.com*, *@yaypie*)
- Nicholas Zakas (*http://nczonline.net*, *@slicknet*)
- Remy Sharp (*http://remysharp.com*, *@rem*)
- Iliyan Peychev

感谢

本书中的一些模式是作者在实践和学习了一些流行的JavaScript库（例如jQuery和YUI）的基础上提出的，但大部分模式还是由JavaScript社区提出并描述的。因此，本书是众多开发者集体工作的成果。为了不打断关于本书历史和信誉的叙述，我们在本书的附属网站（*http://www.jspatterns.com/book/reading/*）上给出了参考文献和建议的附加读物列表。

如果我在参考文献中遗漏了某篇优秀的原始文献，请接受我的道歉并告知我，以便我可以将其增加到*http://www.jspatterns.com*的在线列表中。

推荐读物

本书不是一本入门读物，因此没有介绍一些基础的主题，例如循环和条件选择等。如果您需要获取关于本语言更多的细节，可以参考如下推荐读物：

- Object-Oriented JavaScript by yours truly (Packt Publishing)
- JavaScript: The Definitive Guide by David Flanagan (O'Reilly)
- JavaScript: The Good Parts by Douglas Crockford (O'Reilly)
- Pro JavaScript Design Patterns by Ross Hermes and Dustin Diaz (Apress)
- High Performance JavaScript by Nicholas Zakas (O'Reilly)
- Professional JavaScript for Web Developers by Nicholas Zakas (Wrox)

第1章

简介

JavaScript是一门基于Web的语言。它最初是作为在网页中操作一些特定类型元素（例如图像和表格字段）的方法，但现在该语言发展很快，远远超出了其预期。除了作为客户端浏览器所使用的脚本，现在JavaScript还可用于正在增加的更加多样化的平台编程。例如可以用于编写服务端代码（使用.NET或Node.js）、应用程序扩展（例如为Firefox和Photoshop编写扩展）、移动应用程序和命令行脚本。

JavaScript也是一门与众不同的语言。它没有类，并且函数是用于很多任务的顶层类对象。最初很多程序员认为该语言效率低下，但近年来该观念有所改变。有趣的是，例如Java和PHP这类语言开始增加闭包和匿名函数特性，而JavaScript程序员早就已经享用了这些特性。

JavaScript是一门动态性较强的语言，通过配置可以使其看上去和感觉起来像您过去习惯的其他语言一样。但接受JavaScript与其他语言的不同，并学习其特定的模式可能是更好的方法。

模式

广义上模式是指“重现事件或者对象的主题……它是一个可以用来产生其他事物的模板或者模型”（引自维基百科，*http://en.wikipedia.org/wiki/Pattern*）。

在软件开发过程中，模式是指一个通用问题的解决方案。一个模式不仅仅是一个可以用来复制粘贴的代码解决方案，更多地是提供了一个更好的实践经验、有用的抽象化表示和解决一类问题的模板。

学习和识别模式是非常重要的，理由如下：

- 通过学习模式，可以帮助我们使用经过实践证明有效的经验来编写代码，而无需做很多无用的工作。
- 模式提供了某种程度上的抽象。大脑在一定的时间内仅能记住一定数量的内容，因此当思考更复杂的问题时，使用模式可以让您集中精力去用已有的模式来解决该问题，而不需被一些低层次的细节所困扰。
- 模式可以改善开发者和开发团队之间的交流，通常开发者是采取远程交流而不是面对面进行交流。为一些编码技术或方法贴上标签并命名，可以很方便地确保双方能够了解对方确切想表达的意思。举例来说，如果需要表达“用大括号括上一个函数，并在刚刚定义的这个函数的结束位置放置另一个括号来调用该函数”这层意思，直接使用“立即函数”这样的称法可能更为简单明了。

本书主要讨论如下三种类型的模式：

- 设计模式。
- 编码模式。
- 反模式。

设计模式最初是在“Gang of Four”（该名称是根据本书四名作者的姓名而来的）这本书中定义的，该书早在1994年就出版了，当时的书名是“设计模式：可复用面向对象软件的基础”。这些设计模式的范例主要包括单件（singleton）、工厂方法（factory）、装饰（decorator）、观察者（observer）等。尽管设计模式是与语言无关的，但相对JavaScript的设计模式来说，过去设计模式主要是从强类型语言的视角进行研究的，例如C++语言和Java语言等。有时候将这些模式严格地应用于JavaScript这样弱类型动态语言是没有必要的。有时候这些模式是适用于处理强类型语言的一些本性和基于类的继承。在JavaScript语言中，可能有其他更好、更简单的选择。本书将在第7章讨论几种设计模式在JavaScript中实现的方法。

代码模式更为有趣，这是JavaScript特有的模式，它提供了关于该语言独特的很好的体验，例如various uses of functions。JavaScript的代码模式是本书的主题。

在本书中还会学到反模式。从名称上看，反模式有一些消极甚至是刺耳，但是事实并非如此。一个反模式并不是一个bug，或者是编码错误，它仅仅是常见的、引发的问题比解决的问题更多的一种方法。在代码中使用注释明确标记出反模式。

JavaScript：基本概念

首先快速地复习为后续章节提供背景支持的一些重要概念。

面向对象

JavaScript是一门面向对象的语言，很多程序员对此会很惊讶，因为他们之前都没看到JavaScript语言的这一特性。看到的任何一段JavaScript代码都很有可能是一个对象。只有五种基本类型不是对象：数值类型、字符串类型、布尔类型、空类型和未定义类型。其中前三个类型有对应的以基本类型封装形式体现的对象表示（将在接下来的一个章节进行讨论）。数值类型、字符串类型和布尔类型的值可以通过程序员或者位于幕后的JavaScript解析器来实现向对象的转换。

函数实际上也是对象，函数有属性和方法。

在任何一门语言中最简单的事情就是定义一个变量。在JavaScript中，一旦定义好了变量，同时也就已经正在处理对象了。首先，该变量会自动成为内置对象的一个属性，成为激活对象（如果该变量是一个全局变量，那么该变量会成为全局对象的一个属性）。第二，该变量实际上也是伪类，因为它拥有其自身的属性（称为attributes），该属性决定了该变量是否可以被修改、被删除和在一个for-in循环中进行枚举。这些属性在ECMAScript3中没有直接对外提供，但在第5版本的ECMAScript中，提供了一个特殊的描述符方法来操纵这些属性。

那么对象是什么东西呢？因为它们需要做很多事情，所以这些对象必须十分特殊。实际上对象是十分简单的。一个对象仅仅是一个容器，该容器包含了命名的属性、键－值对（大多数）的列表。这里面的属性可以是函数（函数对象），这种情形下我们称其为方法。

关于创建的对象的另外一件事情是可以在任意时间修改该对象（尽管ECMAScript5引入了API来防止突变）。可以对一个对象执行添加、删除和更新它的成员变量。如果关注隐私和访问，在模式方面也有对应的内容。

最后需要记住的是对象主要有两种类型：

原生的（Native）

在ECMAScript标准中有详细描述。

主机的（Host）

在主机环境中定义的（例如浏览器环境）。

原生的对象可以进一步分为内置对象（例如数组、日期对象等）和用户自定义对象（例如var o={};）等。

主机对象包含windows对象和所有的DOM对象。如果还不确定使用的是否是主机对象，可以尝试在不同的、无浏览器的环境下运行该代码，如果该代码能正确地运行，那么应该使用的是原生的对象。

没有类

可能在本书中将会多次看到这样的语句：在JavaScript中没有类。这对使用其他语言的老练的程序员可能是一个比较新颖的概念，因此需要多次重复来忘记类这个概念，并接受JavaScript只处理对象这样一个特点。

不使用类的做法可以使得编程更为简洁。在创建一个对象的时候，无需先拥有一个类。请考虑如下类Java的对象创建方式：

```
// 创建Java对象
HelloOO hello_oo=new HelloOO();
```

在需要创建一个简单的对象的时候，重复以上操作看起来是额外的开销。我们往往希望让对象更为简洁。

在JavaScript中可以在需要的时候创建一个空对象，然后开始为该对象添加感兴趣的成员变量。可以为该对象添加基本类型、函数和其他对象来作为该对象的属性。一个“空对象”实际上并不是完全空白的，它实际上是包含有一些内置的属性，但是没有其自身的属性。将在接下来的章节详细讨论该问题。

在“Gang of Four”这本书中的一条通用规则是：“尽量多使用对象的组合，而不是使用类的继承”。这句话的意思是通过已有的对象组合来获取新对象，是比通过很长的父－子继承链来创建新的对象更好的一种方法。在JavaScript中可以很简单地实践该建议，之所以这么简单是因为在JavaScript中没有类，因此使用对象的组合是唯一可以采取的方法。

原型（Prototypes）

JavaScript没有继承，尽管这是重用代码的一种方式（后面将通过一整章来介绍代码重用）。可以使用多种方法来实现继承，这里通常使用原型。原型是一个对象（别惊讶），并且创建的每一个都会自动获取一个Prototypes属性，该属性指向一个新的空对象。该对象几乎等同于采用对象字面量或Obect()创建的对象，区别在于它的constructor属性指向了所创建的函数，而不是指向内置的Object()函数。可以为该空对

象增加成员变量，以后其他对象也可以从该对象继承并像使用自己的属性一样使用该对象的属性。

本书将在后续章节详细讨论继承的细节，但现在请各位记住原型就是一个对象（不是一个类，也不是其他特殊的元素），每一个函数都有Prototype属性。

环境

JavaScript需要运行环境来执行。通常JavaScript是在浏览器中执行的，但是这不是唯一的运行环境。本书中介绍的模式大部分是和核心JavaScript（ECMAScript）相关的，因此它们是与环境无关的。除了以下两种情况以外：

- 第8章中特定针对浏览器的模式。
- 其他示范模式的实际应用程序范例。

环境会提供自身的主机对象，该对象在ECMAScript标准中没有定义，可能会带来没有特别提到的和不确定的行为。

ECMAScript 5

核心的JavaScript编程语言（不包含DOM、BOM和额外的主机对象）是基于ECMAScript标准（缩写是ES）。该标准的第3版是在1999年被官方所接受，并且是当前浏览器所使用的标准。该标准的第4版被放弃了，第5版在2009年12月得到通过，这相比第3版过去了10年。

第5版为ECMAScript增加了一些新的内置对象、方法和属性，但是最重要的是增加了所谓的strict模式，该模式实际上可以从JavaScript语言中移除某些特性，使得程序更为简洁和不容易出错。例如with语句的用法在这几年中有很多争议。现在在ES5的strict模式中将会引发错误，尽管在非strict模式中能正常运行。strict模式是通过一个普通的字符串来触发的，在该语言的较早的实现方式中将会简单地忽略该代码。这就意味着通过使用strict模式，可以实现向后兼容性，因为在之前不能理解该代码的浏览器中，它不会引起错误。

在一个作用域（可以是函数作用域、全局作用域或者在将字符串的起始位置传递给eval()）中，可以使用如下字符串：

```
Function my(){
    "use strict";
    // 函数的其余部分……
}
```

这就意味着函数中的代码是在ECMAScript语言的strict子集中运行。对于之前的浏览器，这仅仅是一个字符串，并没有分配给任何变量，因此不会被使用，进而不会导致错误。

在本语言的未来计划中将只允许使用strict模式。考虑到ES5是一个过渡版本，本语言现在鼓励开发者使用strict模式编写代码，但不强制要求。

本书将不探讨ES5增加的特有的模式，因为在本书编写的时候并没有任何浏览器实现了ES5。但本书中的范例通过以下方式推广使用新标准：

- 确保本书中提供的范例代码不会在strict模式中引起错误。
- 避免使用和指出类似`arguments.callee`之类的构造函数。
- 调用在ES5中有等价替代的ES3模式，例如`Object.create()`。

JSLint

JavaScript是一门解析型语言，没有静态编译时检查。因此有可能仅仅是因为几个拼写错误，配置一个有问题程序，而又没有意识到该问题的存在。在这种情况下JSLint就非常有用了。

JSLint（*http://jslint.com*）是Douglas Crockford编写的一个JavaScript代码质量检查工具，该工具可以检查代码，并对潜在的问题提出警告。强烈推荐使用JSLint运行您的代码。在刚开始使用的时候，该工具给出的警告可能会影响到心情，但是将很快从中发现错误，并适应这个专业JavaScript程序员都应具备的一个很重要的习惯。如果JSLint没有对代码给出任何警告，那么将会对代码更为自信，知道自己没有在匆忙中有所遗漏或语法错误。

从下一章开始将会多次提到JSLint。本书中所有示范代码都成功通过了JSLint检查（使用在本书编写期间的默认设置），除了偶尔有一些代码是用于展示反模式的（这些代码会清楚地标示出来）。

在默认设置中，JSLint希望用户最好在代码中使用strict模式。

Console

全书都使用了`Console`对象。该对象不是JavaScript语言的一部分，而是指当今大多数浏览器都提供的一个运行环境。在Firefox浏览器中，使用的是Firebug这个扩展工具。Firebug的控制台提供了一个图形化界面，可以方便用户快速地输入和测试小段的

JavaScript代码，并在当前加载的页面上显示结果（见图1-1）。强烈推荐您学习和探索Firebug这个工具。在WebKit浏览器（Safari和Chrome）中也有类似功能，是作为Web监视器的一部分提供的，而从IE8开始在开发工具中提供了类似功能。

本书中大部分代码使用了console对象来代替弹出alert()消息或刷新当前页面，因为这种方法输出将更为简便，而且不显得那么唐突。

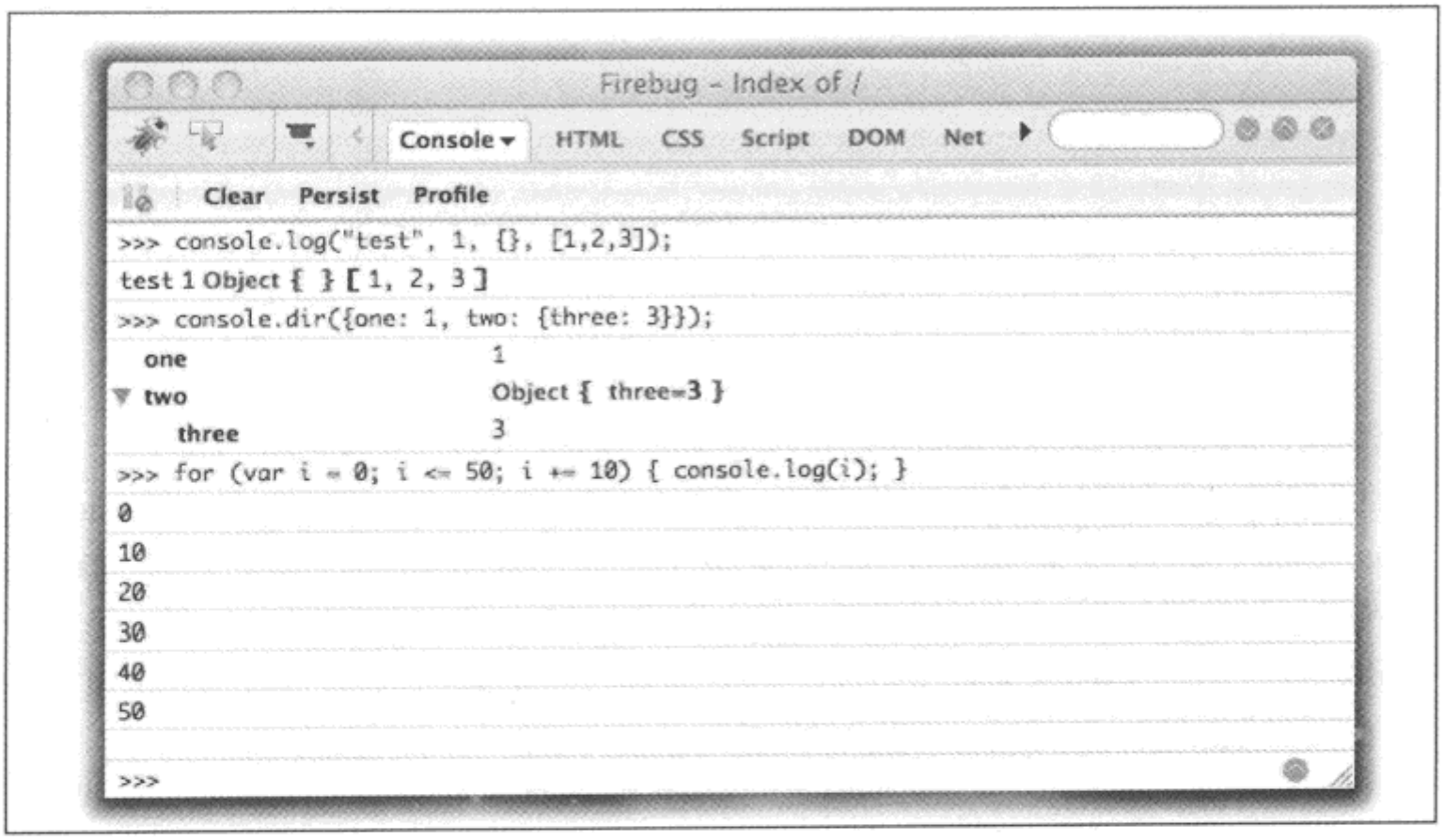

图1-1：使用Firebug控制台

通常使用log()方法来打印所有传递过来的参数，有时候使用dir()方法来枚举传递过来的对象，并打印出所有属性。下面是一个使用的范例：

```
Console.log("test" ,{},[1,2,3]);
Console.dir({one:1,two:{three:3}});
```

当在控制台输入时，无需输入console.log();只需要简单地省略即可。为了避免杂乱，有些范例代码也省略了console.log()。假定正在控制台测试如下代码：

```
Window.name==window['name'];//true
```

这和如下操作是一样的：

```
Console.log(window.name==window['name']);
```

将会在控制台输出true。

第2章

基本技巧

本章讨论一些帮助大家写出高质量的javascript代码的最核心的方法、模式和习惯，比如避免全局变量、使用单个变量var的声明、循环中重新缓存长度变量length（pre-caching length in loops）和符合代码约定等。本章也包含了一些与编码本身无关的习惯，但更多的内容在代码创建过程中，包括撰写API 文档、进行同行评审、运行JSLint等。这些习惯和优秀的方法能帮助大家写出更好、更易于理解、更易维护的代码。当几个月甚至几年之后重新阅读这些代码时，可以为这些代码感到自豪。

编写可维护的代码

软件bug的修改是需要成本的，并且这项成本总是在不断地增加，特别是对于已经广泛发布的产品代码而言，更是如此。最好的情况是当我们一发现bug，立刻就可以修改它，这种情况只发生在刚写完这些代码后不久。否则，一旦转移到新的任务上，忘记了这部分代码，就需要重新阅读这些代码：

- 花时间重新学习和理解相应的问题。
- 花时间理解当时用于解决相应问题的代码。

对于大型项目或大公司而言，还存在另一个问题，就是最终修改代码的人，往往并不是当初写代码的人，也不是发现bug的人。因此，减少理解自己以前写的代码的时间，或者减少理解团队中他人写的代码的时间，就变得非常关键。同时，这也影响到开发完成时间（商业收入）和开发者的情绪，毕竟开发新产品更能让人兴奋，而不是花费那么多时间在老项目维护上。

另外一个事实在于，软件开发人员通常读代码比写代码更耗时间。通常的情形是，当我

们专注于某个问题时，会坐下来花一下午的时间编写出大量的代码。这些代码可能当天就可运行，但要想成为一项成熟的应用项目，需要我们对代码进行重新检查、重新校正、重新调整。譬如：

- 发现bug。
- 项目加入新的特性。
- 项目需要在新的环境中运行（如市场上新出现的浏览器）。
- 代码变换意图。
- 代码在新的框架或体系下需要完全重写，甚至是应用一种新的语言。

因为这些改变，可能最初只是几小时工时写出来的代码，最终需要花费几周的工时来阅读。这就是为什么创建易维护的代码是一个项目成功与否的关键。

易维护的代码意味着代码具有如下特性：

- 阅读性好。
- 具有一致性。
- 预见性好。
- 看起来如同一个人编写。
- 有文档。

本章接下来将阐述在开始写Javascript脚本时要注意的关键。

尽量少用全局变量

Javascript使用函数管理作用域。变量在函数内声明，只在函数内有效，不能在外部使用。全局变量与之相反，在函数外部声明，在函数内无需声明即可很简单地使用。

每一个Javascript环境都有全局对象，可在函数外部使用this进行访问。创建的每一个全局变量都为全局对象所有。在浏览器中，为了方便，使用window表示全局对象本身。下面的代码片断表明在浏览器环境中如何创建和访问全局变量。

```
myglobal = "hello"; // 反模式
console.log(myglobal); // "你好"
console.log(window.myglobal); // "你好"
console.log(window["myglobal" ]; // "你好"
```

```
console.log(this.myglobal); // "你好"
```

全局变量的问题

全局变量的问题在于它们在整个Javascript应用或Web页面内共享。它们生存于同一个全局命名空间内，总有可能发生命名冲突。譬如当一个应用程序中两个独立的部分定义了同名的全局变量，但却有不同目的时。

网页经常会包含一些非页面开发人员编写的代码，譬如：

- 第三方Javascript库。
- 来自于广告合作伙伴的脚本。
- 来自于第三方用户的跟踪与分析脚本的代码。
- 各种小工具、徽章（badges）和按钮。

举个例子，某个第三方脚本定义了一个全局变量result，后来又在某个函数里定义了另一个全局变量，也叫result。这样造成的结果是后一个result变量覆盖了前一个，第三方脚本可能就停止了工作。

因此，与同一个页面上的其他脚本友好共存非常重要，要尽可能少地使用全局变量。在本书的后面章节，我们会研究最小化全局变量的数量的方法，如命名空间模式或自执行立即生效函数（the self-executing immediate functions）,但最重要的方法还是使用var声明变量。

Javascript总是在不知不觉中就出人意料地创建了全局变量，其原因在于Javascript的两个特性。第一个特性是javascript可直接使用变量，甚至无需声明。第二个特性是Javascript有个暗示全局变量（implied globals）的概念，即任何变量，如果未经声明，就为全局对象所有（也就像正确声明过的全局变量一样可以访问）。思考下面的例子：

```
function sum(x, y) {
    // 反模式: 暗示全局变量
    result = x + y;
    return result;
}
```

在这个例子中，result未经声明就使用了。代码虽然在一般情况下可以正常工作，但如果在调用该函数后，在全局命名空间使用了另外的result变量，问题就会出现。

首要的规则就是用var声明变量，正如下面改善后的sum()函数所示。

```
function sum(x, y) {
```

```
var result = x + y;
return result;
}
```

另一种创建隐式全局变量的反模式是带有var声明的链式赋值。在下面的代码片断中，a是局部变量，b是全局变量，这也许并不是你想要的。

```
//反模式，不要使用
function foo() {
var a = b = 0;

// ...
}
```

也许大家想知道，这一切是怎么发生的。这缘于从右至左的操作符优先级。首先，优先级较高的是表达式b=0，此时b未经声明。表达式的返回值为0，它被赋给var声明的局部变量a，如以下代码所示：

```
var a = (b = 0);
```

如果对链式赋值的所有变量都进行了声明，就不会创建出不期望的全局变量。例如：

```
function foo() {
    var a, b;
    // ...
    a = b = 0; // 均为局部变量
}
```

注意： 另一个避免全局变量的原因来源于代码移植。如果你希望你的代码运行在不同的环境（主机），使用全局变量就会非常危险。因为可能很偶然地，不存在于原环境（所以看起来是安全的），但存在于其他环境的主机变量就被覆盖了。

变量释放时的副作用

隐含全局变量与明确定义的全局变量有细微的不同，不同之处在于能否使用delete操作符撤销变量。

- 使用var创建的全局变量（这类变量在函数外部创建）不能删除。
- 不使用var创建的隐含全局变量（尽管它是在函数内部创建）可以删除。

这表明隐含全局变量严格来讲不是真正的变量，而是全局对象的属性。属性可以通过delete操作符删除，但变量不可以。

```
//定义三个全局变量
var global_var = 1;
```

```
global_novar = 2; // 反模式
(function () {
    global_fromfunc = 3; //反模式
}());

// 企图删除
delete global_var; // false
delete global_novar; // true
delete global_fromfunc; // true

// 测试删除情况
typeof global_var; // "number"类型
typeof global_novar; // "undefined"类型
typeof global_fromfunc; // "undefined"类型
```

在ES5 strict模式中，为没有声明的变量赋值会抛出错误（类似上述代码中的两种反模式）。

访问全局对象

在浏览器下，可通过window属性在代码的任意位置访问到全局对象（除非做了特别的处理而发生了意外，如声明了一个名为window的局部变量）。但在其他环境下，这个用起来方便的属性可能就不叫window，而是叫别的名称（甚至可能对于程序员是不可见的）。如果需要访问不带硬编码处理的标识window，可以按如下方式，从内嵌函数的作用域访问。

```
var global = (function () {
    return this;
}());
```

按这种方式通常能获得全局对象，因为this在函数内部作为一个函数调用（而不是通过构造器new创建）时，往往指向该全局对象。事实上在ECMAScript 5的严格模式下就不再这样用了。所以如果你的代码要运行在严格模式下，就要采用另外一种方式。比如，如果你正开发一个库，你可以将你的库里的代码打包在一个直接函数（将在第4章讨论）中，然后在全局作用域中，传递一个引用给this，把this看成传递到直接函数的一个参数。

单一var模式（Single var Pattern）

只使用一个var在函数顶部进行变量声明是一种非常有用的模式。它的好处在于：

- 提供一个单一的地址以查找到函数需要的所有局部变量。
- 防止出现变量在定义前就被使用的逻辑错误（参见“Hoisting:分散变量造成的A问题”）。

- 帮助牢记要声明变量，以尽可能少地使用全局变量。
- 更少的编码（无论是输入代码还是传输代码都更少了）。

单一var模式如下所示：

```
function func() {
    var a = 1,
        b = 2,
        sum = a + b,
        myobject = {},
        i,
        j;

    // 函数体…
}
```

使用一个var关键字声明由逗号分割的多个变量。在声明变量的同时初始化变量，为变量赋初值也是一种好的做法。这样可以防止逻辑错误（所有未初始化且未声明的变量，其值都为undefined），也可提高代码的可读性。当你在以后重新看这段代码时，你可以根据变量的初始值知道使用这些变量的意图。比如，它应该是一个对象还是一个整型？

在声明变量时也可能做些实质性的工作，比如上述代码中的sum = a + b。另一个例子是DOM（文档对象模型）的引用。如下面代码中所示，使用单一var声明将DOM引用赋给局部变量。

```
function updateElement() {
    var el = document.getElementById("result"),
        style = el.style;

    // 使用elt和style再做其他事…
}
```

提升：凌散变量的问题

Javascript 允许在函数的任意地方声明多个变量，无论在哪里声明，效果都等同于在函数顶部进行声明。这就是所谓的“提升”。当先使用变量再在函数后面声明变量时可能会导致逻辑错误。对Javascript而言，只要变量是在同一个范围（同一函数）里，就视为已经声明，哪怕是在变量声明前就使用。看看下面的例子。

```
//反模式
myname = "global"; // 全局变量
function func() {
    alert(myname); // "未定义"
    var myname = "local";
    alert(myname); // "局部变量"
}
func();
```

在这个例子中，可能会以为第一个alert()会提示为“全局变量”，第二个会提示“局部变量”，这是一个合乎情理的期望，因为在第一个alert中，myname没有声明，因此函数很可能“看到”全局变量myname。但事实上并不是这样，第一个alert会被指明为“未定义”，因为myname被看做声明为函数的本地变量（尽管是在后面声明）。所有的变量声明都提升到函数的最顶层。因此，为了避免这类混乱，最好在开始就声明要用的所有变量。

前面的代码片断运行结果和以下代码一样。

```
myname = "global"; // 全局变量
function func() {
    var myname; // 等同于 -> var myname = undefined;
    alert(myname); // "未定义"
    myname = "local";
    alert(myname); // "局部"
}
func();
```

注意： 为了完整起见，再谈谈实现级别上的事情，事实上它们更为复杂。代码处理上分两个阶段，第一，这是个阶段创建变量、函数声明及形式参数。这是解析和进入上、下文的阶段。第二个阶段是代码运行时执行过程，创建函数表达和不合格标识符（未定义变量）。但为了实际使用的目的，我们采纳了“提升”的概念，这个概念并没有在ECMAScript标准中定义，但却经常用来表述这种情形。

---

for循环

for循环经常用在遍历数组或类数组对象，如引数（arguments）和HTML容器（HTMLColltion）对象。通常for循环模式使用如下。

```
//次优循环
for (var i = 0; i < myarray.length; i++) {
// 对myarray[i]做操作
}
```

这种模式的问题在于每次循环迭代时都要访问数据的长度。这样会使代码变慢，特别是当myarray不是数据，而是HTML容器对象时。

HTML容器是DOM方法返回的对象，如：

- document.getElementsByName()
- document.getElementsByClassName()
- document.getElementsByTagName()

还有很多其他HTML容器，它们在DOM标准以前就引入了，并一直使用至今。包括（除此之外还有很多）：

`documents.images`
: 页面上所有的IMG元素。

`documents.links`
: 所有的A元素。

`documents.forms`
: 所有的Forms。

`document.forms[0].elements`
: 页面上第一个form内的所有字段。

容器的麻烦在于它们在document（HTML页面）下是活动的查询。也就是说，每次访问任何容器的长度时，也就是在查询活动的DOM，而通常DOM操作是非常耗时的。

这就是为什么好的`for`循环模式是将已经遍历过的数组（或容器）的长度缓存起来。如以下代码所示。

```
for (var i = 0, max = myarray.length; i < max; i++) {
    //对myarray[i]进行处理
}
```

这种方式下，对长度的值只提取一次，但应用到整个循环中。

在所有的浏览器中，通过将HTML容器上需要遍历的次数缓存起来都会大大提高速度。其中在Safari 3中速度会提高两倍，而在IE7中速度会提高170倍［如需更多细节，请参考《High Performance JavaScript by Nicholas Zakas》(O'Reilly)］。

注意，当要在循环中修改容器时（例如新增一个DOM元素），需要修改容器的长度。

下面是单变量模式，也可以将变量放到循环以外，如下所示：

```
function looper() {
    var i = 0,
        max,
        myarray = [];

    // ...

    for (i = 0, max = myarray.length; i < max; i++) {
    //处理 myarray[i]
    }
}
```

这种模式的好处在于一致性，因为它贯穿了单一变量的模式。缺陷在于创建代码时粘贴和复制整个循环比较麻烦。例如，如果要从一个函数复制循环至另一个函数，必须确保能将i和max携带至新函数中（如果这几个量在原函数中不再需要，则很可能会删掉它们了）。

对于循环的最后一个改进是，用i++替代以下两种表达式：

```
i = i + 1
i += 1
```

JSLint 推荐这样做，原因是++和－－提倡“excessive trickiness”,如果不同意这种说法，可以将JSLint操作的plusplus设成false（默认为真）。本书的后面，最后的模式使用了i += 1。

for模式中的两个变量引出了一些细微操作，原因是：

- 使用了最少的变量（而非最多）。
- 逐步减至0，这样通常更快，因为同0比较比同数组的长度比较，或同非0数组比较更有效率。

第一个修改后的模式是：

```
var i, myarray = [];

for (i = myarray.length; i--;) {
    // 处理 myarray[i]
}
```

第二个使用while循环：

```
var myarray = [],
    i = myarray.length;

while (i--) {
    //处理 myarray[i]
}
```

还有一些微操作将会在关键性的性能操作中提到。另外，JSLint一节中将会解释i--的使用。

for-in循环

for-in循环应该用来遍历非数组对象。使用for-in循环也被称为枚举（enumeration）。

从技术上来说，也可以使用for-in循环来遍历数组（因为在JavaScript中，数组也是对

象），但是不推荐用户这样使用，因为当该数组对象已经被自定义函数扩大后，这样做有可能会导致逻辑上的错误。因此推荐使用正常的for循环来处理数组，并使用for-in循环来处理对象。

当遍历对象属性来过滤遇到原型链的属性时，使用hasOwnProperty()方法是非常重要的。

请考虑如下例子：

```
// 对象
var man = {
    hands: 2,
    legs: 2,
    heads: 1
};

//代码的其他部分
//将一个方法添加到所有对象上
if (typeof  Object.prototype.clone === "undefined") {
    Object.prototype.clone = function () {};
}
```

在本例子中，使用文本定义了一个简单的名为man的对象。在man对象定义前面或者后面的其他位置，使用了一个名为clone()的有用的方法来增加Object的原型。该原型链是活动的，这也就意味着所有的对象都会自动获取针对新方法的访问。为了避免在枚举man的方法时枚举出clone()方法，需要调用hasOwnProperty()函数来过滤该原型属性。如果不使用过滤函数来进行过滤，将会显示出clone()，这在大多数情形下是不希望得到的结果。

```
// 1.
// for-in循环
for (var i in man) {
    if (man.hasOwnProperty(i)) { // filter
        console.log(i, ":", man[i]);
    }
}
/*
控制台中的结果
hands : 2
legs : 2
heads : 1
*/

// 2.
// 反模式:
// 不使用hasOwnProperty()进行检查后使用for-in 循环的结果
for (var i in man) {
    console.log(i, ":", man[i]);
}
```

```
/*
控制台中的结果
hands : 2
legs : 2
heads : 1
clone: function()
*/
```

另外一种使用hasOwnProperty()的模式是在Object.prototype中调用该函数，如下所示：

```
for (var i in man) {
    if (Object.prototype.hasOwnProperty.call(man, i)) { // 过滤
        console.log(i, ":", man[i]);
    }
}
```

在使用hasOwnProperty对man对象进行精练后，可以有效地避免命名冲突，也可以使用一个本地变量来缓存比较长的属性名，如下所示：

```
var i,
    hasOwn = Object.prototype.hasOwnProperty;
for (i in man) {
    if (hasOwn.call(man, i)) { // 过滤
        console.log(i, ":", man[i]);
    }
}
```

注意： 严格来讲，不使用hasOwnProperty()并没有错。依赖于具体任务和对代码的自信，您可以略过该方法并稍微加速循环执行的速度。但是当确认不了对象的内容（和原型链）时，最好还是加上hasOwnProperty()这样的安全检查。

有一种格式化的变种（该变种没有通过JSLint测试）略过了花括号，并将if语句放在同一行中。这样做的好处是该循环语句变得可读性更强（对于每个拥有属性X的元素，就直接对X进行操作）。这样做的结果是只需要更少的缩进就可以获取循环的目的：

```
// 警告：不能通过JSLint检查
var i,
    hasOwn = Object.prototype.hasOwnProperty;
for (i in man) if (hasOwn.call(man, i)) { // 过滤
    console.log(i, ":", man[i]);
}
```

不要增加内置的原型

增加构造函数的原型属性是一个增强功能性的强大的方法，但是有时候该方法会过于强大。

增加内置构造函数（例如Object(),Array()和Function()等）的原型是很有诱惑的，但是这可能会严重影响可维护性，因为这种做法将使代码变得更加不可预测。其他开发者在使用您的代码时可能期望内置的JavaScript方法的使用是一致的，而不期望有一些您自己添加的方法。

此外，给原型添加的属性在没有使用hasOwnProperty()时可能会在循环中出现，这会导致一些混乱。

因此，最好的方法就是不要给内置的原型增加属性。以下情形是例外，可以为内置的原型增加属性：

1. 当未来的ECMAScript版本或JavaScript的具体实现可能将该功能性作为一个统一的内置方法时。举例来说，可以添加ECMAScript 5中描述的方法，并等待浏览器来对其支持。在这种情形下，仅仅是提前定义了有用的方法。
2. 如果检查了自定义的属性或方法并未存在时。也许在其他地方已经实现了该方法，或者是某个您支持的浏览器中JavaScript引擎的一部分。
3. 您准确地用文档记录下来，并和团队交流清楚。

如果遇到以上情形，可以采用如下模式为原型增加自定义的方法：

```
if (typeof Object.prototype.myMethod !== "function") {
    Object.prototype.myMethod = function () {
        // implementation...
    };
}
```

switch模式

可以使用以下模式来提高switch语句的可读性和健壮性：

```
var inspect_me = 0,
    result = '';

switch (inspect_me) {
case 0:
    result = "zero";
    break;
case 1:
```

```
        result = "one";
        break;
    default:
        result = "unknown";
    }
```

switch语句通常采用如下格式：

- 使每个case和switch纵向排列整齐（大括号的缩进规则是个例外）。
- 在每个case语句中使用代码缩进。
- 在每个case语句结尾有一个明确的break语句。
- 避免使用fall-throughs（也就是有意不使用break语句，以使得程序会按顺序一直向下执行）。如果确实希望采用fall-throughs，那么请确信在代码中使用fall-throughs的确是最好的途径，因为在代码中这样做会让其他阅读您代码的人以为代码是有错误的。
- 用default语句来作为switch的结束：当以上所有情形都不匹配时，给出一个默认的结果。

避免使用隐式类型转换

JavaScript在使用比较语句时会执行隐式类型转换，这也是为什么执行false==0或"" == 0这类比较语句后会返回true。

为了避免隐式类型转换导致的混淆不清，请在使用比较语句的时候使用===和!===操作符来对数值和类型进行比较：

```
var zero = 0;
if (zero === false) {
    // 因为zero是0，而不是false，所以代码未执行 }
// 反模式
if (zero == false) {
    // 该代码块会被执行…
}
```

还有一种观念是认为使用===是多余的，仅仅使用==就足够了。例如，当使用typeof来获取已知类型时，会返回一个字符串，因此没有理由使用严格的等价比较。然而，Jslint要求严格的等价比较，这样的做法会使得代码看起来更为一致，并减少在阅读代码时的脑力开销（==的使用是故意的还是无意遗漏的）。

避免使用eval()

如果在代码中看到使用了eval()，请牢记一句俗语，“eval()是一个魔鬼”。该函数可以将任意字符串当做一个JavaScript代码来执行。当需要讨论的代码是预先就编写好了（不是在动态运行时决定），是没有理由需要使用eval()。而如果代码是在运行时动态生成的，则也有其他更好的方法来代替eval()实现其功能。举例来说，只需要简单地使用方括号将需要访问的动态属性括起来就行了：

```
// 反模式
var property = "name";
alert(eval("obj." + property));

// 推荐的方法
var property = "name";
alert(obj[property]);
```

使用eval()也包含一些安全隐患，因为这样做有可能执行被篡改过的代码（例如来自网络的代码）。这是在处理来自一个Ajax请求的JSON响应时常见的反模式。在那些情形下，最好是使用浏览器内置的方法来解析JSON请求，以确保安全性和有效性。对于原生不支持JSON.parse()的浏览器来说，可以使用来自JSON.org网站的类库。

还有一点比较重要的是要牢记通过setInterval()、setTimeout()和function()等构造函数来传递参数，在大部分情形下，会导致类似eval()的隐患，因此应该也尽量避免使用这些函数。在幕后，JavaScript仍然不得不评估和执行以程序代码方式传递过来的字符串：

```
// 反模式
setTimeout("myFunc()", 1000);
setTimeout("myFunc(1, 2, 3)", 1000);

// 推荐的模式
setTimeout(myFunc, 1000);
setTimeout(function () {
    myFunc(1, 2, 3);
}, 1000);
```

使用new Function()构造函数和eval()比较类似，因此该函数的使用也需要十分小心，该函数是一个功能强大的函数，但是通常容易被误用。如果一定需要使用eval()，那么可以考虑使用new Function()来替代eval()。这样做的一个潜在的好处是由于在new Function()中的代码将在局部函数空间中运行，因此代码中任何采用var定义的变量不会自动成为全局变量。另一个避免自动成为全局变量的方法是将eval()调用封装到一个即时函数中（更多关于即时函数的信息请参看第4章）。

请考虑接下来的代码，在这里只有un这个变量仍然是一个全局变量，会影响到命名空间：

```
console.log(typeof un);    // "未定义"
console.log(typeof deux);  // "未定义"
console.log(typeof trois); // "未定义"

var jsstring = "var un = 1; console.log(un);";
eval(jsstring); // logs "1"

jsstring = "var deux = 2; console.log(deux);";
new Function(jsstring)(); // logs "2"

jsstring = "var trois = 3; console.log(trois);";
(function () {
    eval(jsstring);
}()); // logs "3"

console.log(typeof un);    // "数值类型"
console.log(typeof deux);  // "未定义"
console.log(typeof trois); // "未定义"
```

另一个new Function()和eval()的区别在于eval()会影响到作用域链，而Function更多地类似于一个沙盒。无论在哪里执行Function，它都仅仅能看到全局作用域。因此对局部变量的影响比较小。在接下来的例子中，eval()可以访问和修改它外部作用域的变量，然而Function不行（请注意使用Function和使用new Function是一样的）。

```
(function () {
    var local = 1;
    eval("local = 3; console.log(local)"); // logs 3
    console.log(local); // logs 3
}());

(function () {
    var local = 1;
    Function("console.log(typeof local);")(); // logs 未定义
}());
```

使用parseInt()的数值约定

通过使用parseInt()，可以从一个字符串中获取数值。该函数的第二个参数是一个进制参数，通常可以忽略该参数，但是最好不要这样做，因为当解析的字符串是0开始就会出现错误：例如在处理日期时只有一部分日期会进入字段。在ECMAScript 3版本中，0开始的字符串会被当做一个八进制数，而在ECMAScript 5版本中发生了改变。为了避免不一致性和未预期的结果，请每次都具体指定进制参数：

```
var month = "06",
    year = "09";
month = parseInt(month, 10);
```

```
year = parseInt(year, 10);
```

在本例中，如果忽略了进制参数，使用类似parseInt(year)的方法，那么返回值将是0，因为“09”会当做一个八进制数进行处理[和使用parseInt(year,8)一样]，而09在八进制中不是一个合法的数值。

另外一个将字符串转换为数值的方法是：

```
+"08" // 结果是8
Number("08") // 8
```

这种方法通常会比parseINt()快很多，因为正如其名称一样，parseInt()是解析而不是简单地转换。但是如果希望输入"08 hello"会返回一个数值，那么除parseInt()之外，其他方法都会失败并返回NaN。

编码约定

确定并遵循编码约定是非常重要的，这可以使得代码更为一致、可预测、更容易阅读和理解。一个新加入团队的开发者通过了解编码约定，可以很快提高工作效率，理解团队其他人员编写的代码。

在会议和邮件列表中经常会爆发关于特定编码约定的激烈讨论（例如编码缩进是应该使用tab还是空格）。因此，如果在您的组织里建议采纳一个编码约定，请务必有心理准备面对抵触，以及听到不同的但同样有力的观点。记住，确定并一致遵循约定比这个具体约定是什么更为重要。

缩进

没有缩进的代码是很难阅读的。如果代码使用了不一致的缩进，也是很令人头痛的事情。因为这样的做法看起来遵循了约定，但是会导致很多令人困惑的疑问。标准化使用缩进是非常重要的。

一些程序员喜欢使用tab来进行缩进，因为任何人都可以调整编辑器来显示该tab，tab可以以自定义为若干个空格。一些程序员喜欢直接使用空格来缩进，通常是使用4个空格。只要团队中所有人都遵循相同的约定就可以了。例如在本书中使用4个空格来缩进，这也是JSLint的默认值。

那么需要对哪些内容执行缩进呢？规则很简单，只需要对大括号中所有的代码执行缩进。这主要包含函数体、循环体（do、while、for、for-in）、if语句、switch语句和对象字面量引用的属性。下面代码展示了一些缩进的例子：

```
function outer(a, b) {
    var c = 1,
        d = 2,
        inner;
    if (a > b) {
        inner = function () {
            return {
                r: c - d
            };
        };
    } else {
        inner = function () {
            return {
                r: c + d
            };
        };
    }
    return inner;
}
```

大括号

应该经常使用大括号，甚至在可选的情形下，都请使用大括号。技术上来说，在if语句和for语句中如果仅有一行语句，可以不需要大括号，但是为了一致性和更方便升级，最好还是使用大括号。

想像一下有一个for循环中仅有一条语句，可以忽略该大括号并不会有语法错误：

```
// 不好的做法
for (var i = 0; i < 10; i += 1)
    alert(i);
```

但是如果以后为该循环体增加了其他语句，会如何呢？

```
// 不好的做法
for (var i = 0; i < 10; i += 1)
    alert(i);
    alert(i + " is " + (i % 2 ? "odd" : "even"));
```

尽管采用了缩进，第二个alert语句还是在循环体外面。最好的办法就是一直使用大括号，甚至在循环体中仅有一条语句：

```
// 比较好的做法
for (var i = 0; i < 10; i += 1) {
    alert(i);
}
```

在if条件选择语句中也有类似情况：

```
// 不好的做法
```

```
if (true)
    alert(1);
else
    alert(2);
// 比较好的做法
if (true) {
    alert(1);
} else {
    alert(2);
}
```

开放的大括号位置

开发人员在讲开放的大括号放置于什么位置有不同的选择，是和语句放在同一行还是放在接下来的一行中呢？

```
if (true) {
    alert("It's TRUE!");
}
```

或者

```
if (true)
{
    alert("It's TRUE!");
}
```

在这个特定的范例中，采用哪种方式只是个人习惯的不同。但是有些情形下随着大括号的位置不同，程序的执行结果也会有所不同。这是由分号插入机制（semicolon insertion mechanism）导致的。JavaScript对代码不会很挑剔，当没有正确使用分号结束本语句时，它会自动补上。该行为在函数返回一个对象字面量并且开放的大括号位于接下来的一行时会导致问题：

```
// 警告：未预期的返回值
function func() {
    return
    {
        name: "Batman"
    };
}
```

如果期望该函数返回一个有name属性的对象时，您可能会很惊讶。因为由于隐式添加了分号，该函数返回了一个未定义的值。接下来的代码是一个等价的形式：

```
// 警告：未预期的返回值
function func() {
      return undefined;
      //并未访问到接下来的代码…
```

```
        {
            name: "Batman"
        };
}
```

总之，应该一直使用大括号并直到将开放的大括号放置在前面语句的同一行：

```
function func() {
    return {
        name: "Batman"
    };
}
```

注意： 关于分号的注释：和大括号一样，应该一直使用分号，甚至JavaScript解析器会隐式增加。这不仅是严格记录和代码编写方式，也会有助于避免之前范例中含糊不清的情况。

空格

使用空格也有助于改善代码的可读性和一致性。在撰写英文文章时在逗号和区间范围后面使用空格。在JavaScript采用同样的逻辑，可在列表表达式（等价于逗号）和语句结束（等价于完成一次“思考”）后面添加空格。

使用空格比较好的位置包含如下：

- 在分开for循环的各个部分的分号之后：例如，`for (var i = 0; i < 10; i += 1) {...}`
- 在for循环中初始化多个变量（i和最大值等）：`for (var i = 0, max = 10; i < max; i += 1) {...}`
- 在限定数组项的逗号后面：`var a = [1, 2, 3];`
- 对象属性的逗号之后和将属性名和属性值分开的冒号之后：`var o = {a: 1, b: 2};`
- 分隔开函数中各个参数的逗号之后：`myFunc(a, b, c)`
- 在函数声明的大括号之前： `function myFunc() {}`
- 在匿名函数表达式之后： `var myFunc = function () {};`

空格的另外一个很好的用途是用来分隔所有的操作符和操作，这也就是意味着在+, -, *, =, <, >, <=, >=, ===, !==, &&, ||, +=等之后使用空格：

```
// 大量空格，并且使用一致
// 使得代码可读性更好
// 允许在阅读的时候不用一口气读下去
var d = 0,
```

```
    a = b + 1;
if (a && b && c) {
    d = a % c;
    a += d;
}

// 反模式
// 缺少空格或者空格使用不一致
//使得代码比较混乱
var d= 0,
    a =b+1;
if (a&& b&&c) {
d=a %c;
    a+= d;
}
```

最后一点关于使用空格的情形是和大括号有关的，在如下情形使用空格是比较好的做法：

- 在函数中使用大括号开始符之前，例如if-else语句、循环语句和对象字面量等。
- 在大括号结束符和else或while之间。

大量使用空格的可能会导致文件长度变长，但是这并不是本章讨论的主题（这将在后面的章节中进行讨论）。

注意： 一个常用的增加可读性的做法是使用垂直的空格。可以使用空行来分隔代码的不同单元，就像在文章中使用空行来分隔不同想法一样。

命名约定

另一个提高代码可预测性和可维护性的方法是使用命名约定。这就意味着采用一致的方法来对变量和函数进行命名。

下面是关于命名约定的一些建议，您可以采纳这些约定，也可以按照自己的想法进行修改。再一次说明，拥有一个约定并一致地遵循该约定比该约定实际上是如何更为重要。

构造函数的首字母大写

JavaScript没有类，但是可以通过new调用构造函数：

```
var adam=new Person();
```

因为构造函数仍然仅仅是一个函数，它看起来是一个函数名，它和构造函数或者普通函数的行为差不多。

构造函数的首字母大写这样的命名方式就包含了以上含义。首字母小写的函数名和方法表明这些函数和方法不能使用new来调用：

```
function MyConstructor() {...}
function myFunction() {...}
```

在下一章中会介绍一些模式，使用这些模式可以让您强制构造函数体现出构造函数的特性，但是简单地遵循命名约定仍然是有用的，至少可以提高代码的可读性。

分隔单词

当变量名和函数名是由多个单词组成时，遵循如何将这些单词分隔开的约定是一个不错的主意。通常的约定是使用所谓的骆驼峰式命名法。在骆驼峰式命名法约定中，所有的单词除去每个单词的首字母以外，都用小写表示。

对于构造函数，可以使用大骆驼峰式命名法，例如`MyConstructor()`；而对于函数和方法名，可以使用小骆驼峰式命名法，例如`myFunction()`、`calculateArea()`、`getFirstName()`等。

那么对于不是函数的变量，该如何命名呢？开发者通常使用小骆驼峰式命名法来为变量命名。另外有一个不错的方法是所有的单词都使用小写，并用下划线分隔开各个单词，例如`first_name`、`favorite_band`和`sold_campany_name`等。这种表示方法可以方便地很明显地区分开函数和其他标志——基本常量和对象。

ECMAScript使用骆驼峰式命名法来为方法和属性进行命名，尽管多单词的属性名比较少见（例如正则表达式对象中的`lastIndex`属性和`ignoreCase`属性）。

其他命名模式

有时候开发者使用命名约定来弥补或者代替语言的特性。

举例来说，在JavaScript中无法定义常量（尽管有一些内置的数值，例如`Number.MAX_VALUE`），因此开发者采用将变量名全部大写的约定来表明该变量在程序生命周期中不可改变，例如：

```
// 精确的常量，请不要修改
var PI = 3.14,
    MAX_WIDTH = 800;
```

此外还有一种情形下，约定将所有的字符都大写：为全局变量命名时将所有的字符大写。使用全部大写字符命名的全局变量命名方式可以使得这些变量很容易地被识别出来。

还有一种使用约定来模仿功能性的做法是私有成员函数约定。尽管在JavaScript中可以实现真正的私有函数，有时候开发者发现仅仅使用一个下划线前缀来标识私有方法或者私有属性是更为简单的一种方法。请考虑接下来的范例：

```
var person = {
    getName: function () {
        return this._getFirst() + ' ' + this._getLast();
    },
    _getFirst: function () {
        // ...
    },
    _getLast: function () {
        // ...
    }
};
```

在这个例子中，getName()意味着这是API的一个公开的方法，而_getFirst()和_getLast()意味着这是一个私有函数。尽管它们都是普通的公开方法，但是使用下划线前缀的表示方法可以提醒使用person对象的用户，告诉他们这些方法在其他地方不能确保一定能够正常工作，不能直接调用。注意到JSLint会对下划线前缀给出警告，除非将nomen选项设置为false。

下面是一些使用下划线约定的变量：

- 使用下划线结尾来表明是私有变量，例如name_和getElements_()。
- 使用一个下划线前缀来标识受保护属性，使用两个下划线前缀来标识私有属性。
- 在Firefox中有一些属性，这些属性技术上不是JavaScript语言的一部分，它们采用两个下划线前缀和两个下划线后缀来命名，例如__proto__and__parent__。

编写注释

为代码编写注释是非常重要的，甚至在该代码除您之外其他人都不会接触到时都需要编写注释。通常人们在深入思考一个问题时，会非常清楚这段代码的工作原理。但是当过一周后再次回到该代码时，可能会花上很长时间来回想起那段代码到底是干什么的。

不需要注释一些比较明显的代码：例如每一个变量或每一行都注释。但通常有必要对所有的函数、函数参数、返回值和其他有趣或不同寻常的算法和技术都用文档记录下来。设想注释就是未来代码阅读者的一个提示，只需要阅读注释就能明白代码中有哪些函数和属性名。举例来说，当有一段5至6行的代码执行了一个具体的工作，如果有一行描述该代码功能并说明为什么该代码位于本位置的注释，那么阅读者就可以略过代码细节。

关于代码注释没有严格不变的规则和比例，有一些代码（例如正则表达式）可能实际上注释会比代码长度还长。

注意： 最重要的习惯，也是最难遵循的习惯就是不断更新注释，因为过期的注释可能会误导阅读者，这比没有注释还可怕。

在接下来的一节里，将看到注释会自动生成文档。

编写API文档

大多数开发者认为编写文档是一件令人厌烦的不值得做的任务。但事实并非如此，API文档可以从代码的注释中自动生成，通过这种方法实际上都无需编写就可以自动获得文档。大多数程序员认为这种想法是有极大吸引力的，因为从特定关键字和特定格式化的“命令”来自动生成可读的文档看起来很像一种编程操作。

传统的API文档最早是来自Java语言中，在Java SDK（软件开发工具集）中发布了一个名为javadoc的工具。很多其他语言也开始采纳了这个思想。在JavaScript中有两个优秀的工具，这两个工具都是免费并且开源的，分别是JSDoc Toolkit (*http://code.google.com/p/jsdoc-toolkit/*)和YUIDoc (*http://yuilibrary.com/projects/yuidoc*)。

生成API文档的步骤如下：

- 编写特殊格式的代码块。
- 运行工具来解析代码和注释。
- 发布工具解析的结果，大多数情况是采用HTML格式发布。

需要学习一些特殊的标签，如下所示：

```
/**
 * @tag value
 */
```

举例来说，如果有一个名为`reverse()`的函数，该函数可以将字符串翻转过来，该函数有一个字符串参数并返回另一个字符串。那么可以按照如下方式记录文档：

```
/**
 * 翻转一个字符串
 *
 * @param  {String} 输入需要翻转的字符串
 * @return {String} 翻转后的字符串
 */
var reverse = function (input) {
```

```
        // ...
        return output;
    };
```

这里可以看到@param是输入参数的标签，而@return是用来表示返回值的标签。文档工具会解析这些标签，并生成一系列格式化非常好的文档。

YUIDoc范例

YUIDoc过去是用来创建YUI（Yahoo! User Interface）类库文件的，现在已经广泛用于各种项目。需要遵循一些约定，以便更好地使用该工具。例如模块名和类的名称（尽管在JavaScript中没有类）。

现在请看一个使用YUIDoc生成文档的完整范例。

图2-1是一个最终生成的具备良好格式化的文档的预览。实际上还可以根据项目需要和个人感觉对模板进行自定义。

可以访问*http://jspatterns.com/book/2/*来获取一个更为生动的范例。

在本例中，整个应用程序由一个文件（app.js）组成，该文件中仅有一个模块(myapp)。在接下来的章节中将了解更多有关模块的信息，但是从现在开始，只需要将模块看做一个使得YUIDoc正常工作的标签即可。

app.js文件的开头通常如下所示：

```
/**
 * 我的JavaScript应用程序
 *
 * @module myapp
 */
```

然后使用一个命名空间来定义一个空对象：

```
var MYAPP = {};
```

然后定义一个包含两个方法（sum()和multi()）的math_stuff对象。

```
/**
 * 一个数字工具
 * @namespace MYAPP
 * @class math_stuff
```

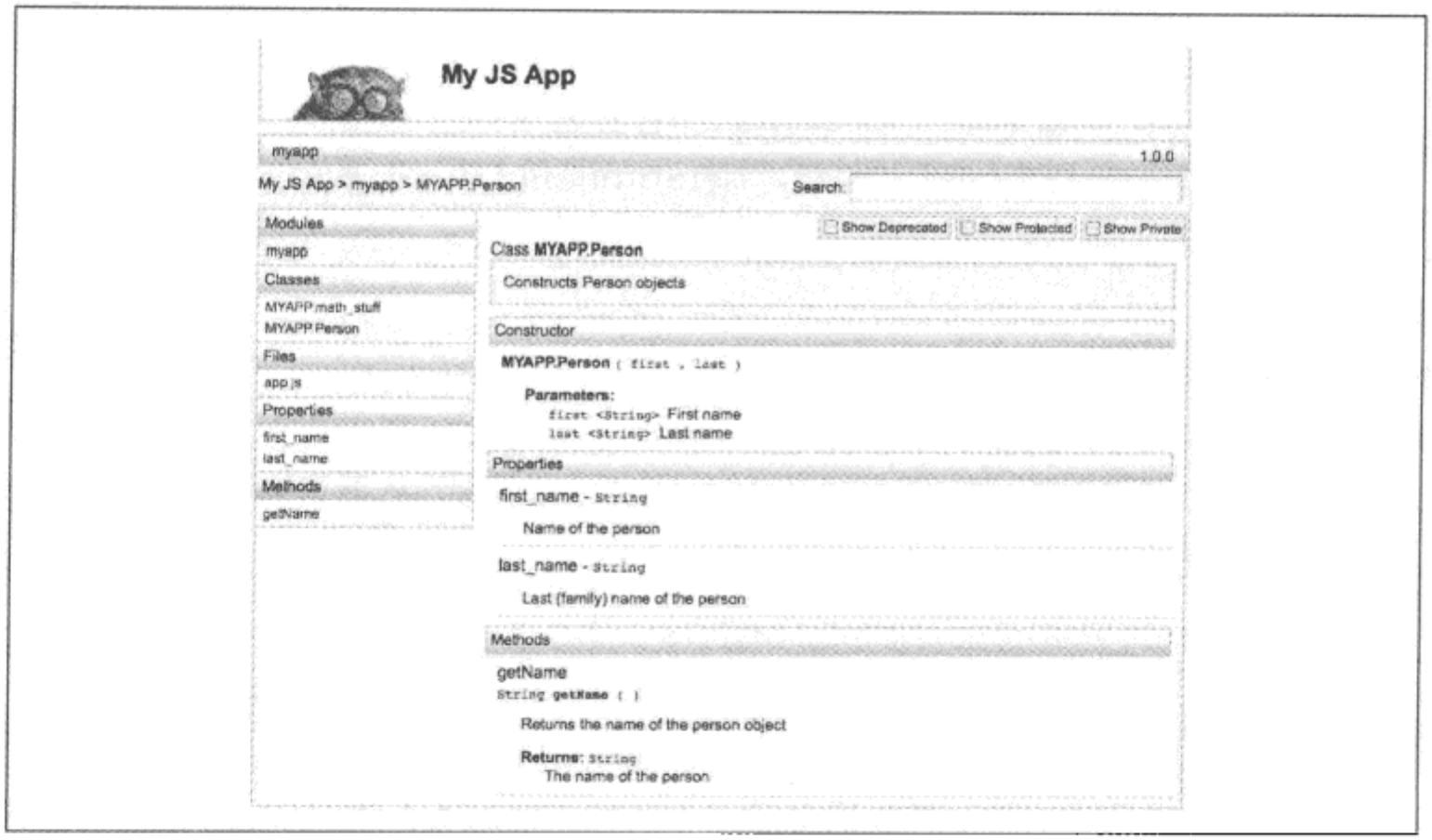

图2-1：YUIDoc生成的文档

```
 */
MYAPP.math_stuff = {
    /**
     * Sums two numbers
     *
     * @method sum
     * @param  {Number} 是第一个数
     * @param  {Number} 是第二个数
     * @return {Number} 两个输入的总和
     */
    sum: function (a, b) {
        return a + b;
    },
    /**
     * Multiplies two numbers
     *
     * @method multi
     * @param  {Number} 是第一个数
     * @param  {Number} 是第二个数
     * @return {Number} 两个输入相乘后结果
     */
    multi: function (a, b) {
        return a * b;
    }
};
```

以上就完成了第一个“类”。请注意这里加黑显示的标签：

@namespace

这用于命名包含以上对象的全局引用的名称。

@class

这里有些命名不当，它实际意思是指对象或者构造函数。

@method

定义对象中方法和方法名。

@param

列举函数使用的参数。其中将参数类型用大括号括起来，并在其后注释参数名及描述。

@return

类似于@param，这是用于描述返回值的，并且该方法没有名称。

对于第二个“类”，让我们使用构造函数来为原型增加一个方法，借此来感受一下文档系统如何处理采用不同方法创建的对象：

```
/**
 * Constructs Person objects
 * @class Person
 * @constructor
 * @namespace MYAPP
 * @param {String} first 是名字
 * @param {String} last 是姓氏
 */
MYAPP.Person = function (first, last) {
    /**
     * 人的姓名
     * @property first_name
     * @type String
     */
    this.first_name = first;
    /**
     * Last (family) name of the person
     * @property last_name
     * @type String
     */
    this.last_name  = last;
};
/**
 * Returns the name of the person object
 *
 * @method getName
 * @return {String} 人的姓名
 */
MYAPP.Person.prototype.getName = function () {
    return this.first_name + ' ' + this.last_name;
};
```

在图2-1中可以看到Person构造函数生成的文档是什么样子的。下面是关于使用粗体表示的说明：

- @constructor表明这个“类”实际上是一个构造函数。
- @property和@type描述了对象的属性。

YUIDoc系统是一个和语言无关的系统，它仅仅解析注释的部分，而不解析JavaScript代码。缺点在于不得不在注释中声明属性名、参数名和方法名，例如@property first_name。而优点在于一旦适应了使用该系统进行注释，就可以使用该系统为任何其他语言的代码编写注释。

编写可读性强的代码

为API编写注释不仅仅是一种提供参考文档的简便方法，而且还有其他用途——通过再次审视代码，提高代码质量。

几乎所有的作者和编辑都会告诉您编辑校对工作是十分重要的：这有可能是出版优秀书籍和文章中最重要的步骤。将最初的草稿内容写到纸上仅仅是第一步。草稿会传递一些信息给读者，但是这些信息可能不十分清晰、结构性不好，并且不好理解。

编写代码也是类似的。当您坐下来解决问题时写出的解决方案仅仅是一个初稿。该解决方案可以给出令人期待的输出，但是该方案是否是最佳方案呢？该代码是否可读、易于理解、维护和升级呢？当您再次审视代码时您将更加确定代码哪些部分可以改进——如何使得代码更容易继续更新、移除一些不足之处等。这就是编辑校对工作的重要性，它可以极大地帮助您创建高质量的代码。但是通常任务都是时间期限很紧张，没有更多的时间用于编辑校对（这的确是一个问题，实际情况可能是昨天就需要将工作提交）。这也就是为什么编写API文档是编辑校对的一个机会。

通常在编写文档注释时，会重新思考一些问题。有时候反省可以使得代码更清晰。举例来说，上面范例代码中方法的第三个参数使用比第二个参数频繁得多，并且第二个参数通常默认设置为true，那么就可以调整该方法的接口，交换第二个和第三个参数。

编写可读性强的代码意味着在编写代码，甚至仅仅是编写某个API时，心里都要想着该代码可能是要提供给其他人阅读的。这种思维方式将会有助于编辑校对和思考更好的解决问题的方法。

谈论到初稿时，还有一个“未雨绸缪”的观念。这看起来有一些极端，但是这是十分有意义的，特别是当手头上有一个关键任务项目时（人们的生活依赖于此）。这个观念是您想出来的第一个解决方案可能可以解决问题，但是这不过是一个草稿，是解决问题的

方法之一。第二个解决方法通常要更好，因为这时对问题的理解会更为深入。在第二个解决方案中，不允许直接从第一个方案中复制、粘贴出来，这样可以阻止为了简便而满足于不完美的解决方案。

同行互查

另外一种优化代码的方式是采取同行互查的方式。同行互查可以采用正式和标准化的途径，甚至是采用一些专用的工具。将同行互查作为开发过程中流水化的一部分是十分重要的。在如果没有足够时间来研究和采用审查工具时，也要坚持同行互查。这种情形下，可以简单地走到隔壁，请坐在那里的开发者来看一下您的代码。

和使用API文档或其他类型文档一样，同行互查的方式有助于编写更为清晰的代码。这里的原因在于其他人需要阅读和理解才能明白您的代码具体是用于做什么事情。

同行互查是一种非常好的实践方法，不仅仅是因为这样做可以得到更好的代码，而且通过这样做可以使得审查人和代码编写者可以交换和共享知识，进而学习到对方的经验和独有的方法。

如果您是一个单干者，并没有同行来审查代码，这也不应该成为障碍。可以将代码中的一部分开源，并将代码中有趣的片段放置在博客中，那么全世界的人都可以成为您的审查员。

另外一个比较好的实践方法是使用源码控制系统（例如CVS、Subversion和Git），这些系统会在有人更新代码后自动向项目团队发送邮件。大部分这些邮件是不会有人阅读的，但是有时候某个同事可能想从自己的工作中休息一下，并自发地看看您刚提交的代码。

在正式发布时精简代码

可以采用减少空白位置、注释和其他JavaScript代码中不重要的部分来减小需要从服务器端传输给浏览器的JavaScript文件，以便加速页面载入速度。通常该工作是采用工具来实现的（例如Yahoo！的YUICompress或Google的Closure Compiler）。在正式发布之前精简脚本是非常重要的，因为这样做的结果可以大大缩小JavaScript文件，通常可以减少一半左右。

下面是一个缩小JavaScript文件的一个范例（这是YUI2类库的事件工具之一）：

```
YAHOO.util.CustomEvent=function(D,C,B,A){this.type=D;this.scope=C||window;this.silent
=B;this.signature=A||YAHOO.util.CustomEvent.LIST;this.subscribers=[];if(!this.silent)
{}var E="_YUICEOnSubscribe";if(D!==E){this.subscribeEvent=new
```

```
YAHOO.util.CustomEvent(E,this,true);}...
```

除了去除空格、新的空行、注释以外，精简器还会将变量重新命名为一个较短的名称（当然是在保证安全的前提下），例如在处理过后的代码中用D、C、B、A来表示变量。精简器只会重命名局部变量，因为重命名全局变量可能会破坏代码。这也是为什么要尽量多使用局部变量的原因。如果在函数中多次使用了全局变量，例如DOM引用，最好是将其分配给一个局部变量。这将加速解析变量名的速度，以便在精简以后运行速度更快，下载速度更快。

备注一下，Google的Closure Compiler也会尝试重命名全局变量（在高级模式中），如果确定要利用该附加精简功能，请务必意识到这样做具有一定风险，需要特别引起注意。

精简发布版本代码是十分重要的，因为这将有助于提升页面性能，但是应该将这部分工作留给精简器来做。尝试手工编写精简的代码是错误的。在编写代码的时候仍然要使用有含义的变量名、一致性的空格和指示符、注释等。这样编写出来的代码可读性强，以便使得维护者（人）能够快速理解代码，并让精简器（机器）来精简文件大小。

运行JSLint

在前面的章节中已经介绍了JSLint，本章中也多次提到了该工具。现在应该已经确信使用JSLint运行代码是一个很好的编程模式。

那么JSLint会在代码中寻找什么呢？它会在代码中寻找本章中讨论的一些不好的模式（单一var模式、parseInt()的进制、一直使用大括号）和其他冲突：

- 无法执行的代码。
- 在定义之前使用变量。
- 不安全的UTF字符。
- 使用*void*、*with*或*eval*。
- 在正则表达式中不合适的转义字符。

JSLint是由JavaScript编写的（大部分代码可以通过JSLint的检查）。比较好的是JSLint既可以作为一个基于Web的工具来使用，也可以作为一个可下载的代码以用于具有JavaScript解析器的多个平台。可以下载下来并使用WSH（Windows Scripting Host）、JSC(Mac OSX的JavaScriptCore)或Rhino(Mozilla推出的JavaScript解析器)以在本地运行。

将JSLint下载下来并整合到文本编辑器中是一个很好的做法，这样就可以在每次保存

文件的时候都运行JSLint检查一下的习惯（为其建立一个键盘快捷方式也是很有帮助的）。

小结

在本章中，我们讨论了如何编写可维护的代码。该主题是十分重要的，不仅对于成功地完成软件项目十分有利，而且有利于开发者与开发团队的其他人员进行交流。下面回顾一下本章谈到的一些重要的优秀实践经验和模式：

- 减少使用全局变量，最理想的情况是一个应用程序仅有一个全局变量。
- 在每个函数中仅使用一个var变量声明，这有助于在一个地方查看所有变量，可以有效防范变量提升导致的错误。
- for循环、for-in循环、switch语句、“evel()是魔鬼”和不要扩充内置的原型。
- 遵循一些编码约定（一致性的空格、缩进、在可选的时候都坚持使用大括号和分号）和命名约定（针对构造函数、函数和变量的命名约定）。

本章也讨论了一些和代码无关，但一般和编程进程有关的的附加实践经验，编写注释、创建生成API文档、同行互查、不要尝试以损失可读性为代价的基础上编写精简代码、坚持使用JSLint检查代码等。

第3章

字面量和构造函数

在JavaScript中可以使用字面量表示（Literal notation）模式，这种方法更为准确，也更富有表现力，并且在对象定义中更不容易出错。本章主要讨论了通过字面量以构造对象的方法，比如对象、数组以及正则表达式等字面量构造方法，同时还讨论了与类似`Object()`和`Array()`等内置构造函数相比，为什么基于字面量表示法是更为可取。本章还引入了JSON格式，并用以演示数组和对象字面量如何用于定义数据传输格式。此外，本章还讨论了自定义构造函数，以及增强`new`操作符以确保构造函数行为符合预期的方法。

为了扩展本章的主要知识（即避免使用构造函数，相反更建议使用字面量的方法），为此对内置包装构造函数`Number()`、`String()`以及`Boolean()`进行了讨论，并且与原始数值、字符串以及布尔值进行了比较。最后，本章还给出了一个有关使用不同内置`Error()`构造函数的简短说明。

对象字面量

当考虑JavaScript中的对象时，比如简单考虑键－值对（key-value pair）哈希表（这与其他编程语言中称之为“关联数组”（associative arrays）的概念相类似）时，其中该值可以是原始类型的，或者是其他类型的对象，在这两种情况下，都将其称之为属性(property)。该值也可以是函数，在这种情况下，它们被称为方法（method）。

在JavaScript中所创建的自定义对象（也就是说，用户定义的原生对象）在任何时候都是可变的。许多内置原生对象的属性也是可变的。您可以从一个空对象开始，然后再根据需要向其添加函数。对于按需对象创建方式而言，对象字面量表示法是一种非常理想的选择方法。

考虑以下例子：

```
// 开始时定义一个空对象
var dog = {};

// 向dog对象添加一个属性
dog.name = "Benji";

// 现在，向dog对象添加一个方法
dog.getName = function () {
    return dog.name;
};
```

在前面的例子中，dog对象开始时是处于干净状态，即一个空对象。然后可以向该对象添加一个属性和一个方法。在程序生命周期内的任何时候，都可以执行以下操作。

- 改变属性和方法的值，如下所示：

```
dog.getName = function () {
    // 重新定义该返回方法
    // 返回一个硬编码的值
    return "Fido";
};
```

- 完全删除属性/方法：

```
delete dog.name;
```

- 添加更多的属性和方法：

```
dog.say = function () {
    return "Woof!";
};
dog.fleas = true;
```

当然，并不要求必须从空对象开始。对象字面量模式可以使您在创建对象时向其添加函数。如以下例子所示：

```
var dog = {
    name: "Benji",
    getName: function () {
        return this.name;
    }
};
```

注意： 在本书中，可以在多处看到“空白对象(blank object)”和“空对象(empty object)”。重要的是要注意，这只是为了表述的简洁性，而实际在JavaScript中没有任何这样的事物是空对象。即使是最简单的{}对象也具有从Object.prototype继承的属性和方法。通过“空”这个词语，就会很容易理解该对象除了继承的属性以外，并没有自身的属性。

对象字面量语法

如果不习惯使用对象字面量表示法，刚开始看起来可能有点奇怪，但是一旦使用得越多，就会越喜欢这种方法。从本质上说，语法规则如下所示：

- 将对象包装在大括号中（左大括号“{”和右大括号“}”）。
- 对象中以逗号分隔属性和方法。在名称－值(name-value) 最后的尾随逗号是允许的，但是在IE浏览器中会产生错误，因此尽量不要使用它。
- 用冒号来分割属性名称和属性的值。
- 当给变量赋值时，请不要忘记右大括号“}”后的分号。

来自构造函数的对象

JavaScript中并没有类的概念，但是它却支持极大的灵活性，因为不必预先知道对象的一切细节，也不需要类“蓝图(blueprint)”。但是，JavaScript中也有构造函数，其使用的语法与Java或其他基于类的语言中创建对象的语法相似。

可以使用自己的构造函数，或使用一些类似Object()、Date()、String()的内置构造函数以创建对象。

在下面的例子中展示了以两种等价的方法来创建两个相同的对象：

```
// 第一种方法——使用字面量
var car = {goes: "far"};

// 另一种方法——使用内置构造函数
// 警告：这是一个反模式
var car = new Object();
car.goes = "far";
```

从上面这个例子可以看出，字面量表示法的显著优点在于它仅需要输入更短的字符。优先选择字面量模式以创建对象的另一个原因在于它强调了该对象仅是一个可变哈希映射，而不是从对象中提取的属性或方法。

与使用object构造函数相对，使用字面量的另一个原因在于它并没有作用域解析。因为可能以同样的名字创建了一个局部构造函数，解释器需要从调用Object()的位置开始一直向上查询作用域链，直到发现全局object构造函数。

对象构造函数捕捉

当您能够使用对象字面量时，就没有理由使用new Object()构造函数，但是可能会继承

其他人编写的遗留代码，因此应该意识到该构造函数的“特征”（或者说是另一个不使用构造函数的原因）。这里涉及的“特征”在于Object()构造函数仅接受一个参数，并且还依赖传递的值，该Object()可能会委派另一个内置构造函数来创建对象，并且返回了一个并非期望的不同对象。

下面的一些例子将数字、字符串、以及布尔值到传递到new Object()构造函数中，其结果是获得了以不同构造函数所创建的对象：

```
// 警告：前面的反模式

// 一个空对象
var o = new Object();
console.log(o.constructor === Object); // true

// 一个数值对象
var o = new Object(1);
console.log(o.constructor === Number); // true
console.log(o.toFixed(2)); // "1.00"
// 一个字符串对象
var o = new Object("I am a string");
console.log(o.constructor === String); // true

// 一般的对象并没有substring()方法，
// 但是字符串对象都有该方法
console.log(typeof o.substring); // "function"
// 一个布尔对象
var o = new Object(true);
console.log(o.constructor === Boolean); // true
```

当传递给Object()构造函数的值是动态的，并且直到运行时才能确定其类型时，Object()构造函数的这种行为可能会导致意料不到的结果。因此，总的来说，不要使用new Object()构造函数，相反应该使用更为简单、可靠的对象字面量模式。

自定义构造函数

除了对象字面量模式和内置的构造函数以外，可以使用自己的构造函数来创建对象，如下面的例子所示：

```
var adam = new Person("Adam");
adam.say(); // 输出结果为"I am Adam"
```

这种新模式看起来非常像使用一个名为Person的类创建一个Java对象。虽然语法上看起来是相似的，但是实际上在JavaScript中并没有类，并且Person也只是一个函数而已。

下面是Person构造函数的定义：

```
var Person = function (name) {
    this.name = name;
    this.say = function () {
        return "I am " + this.name;
    };
};
```

当以new操作符调用构造函数时，函数内部将会发生以下情况：

- 创建一个空对象并且this变量引用了该对象，同时还继承了该函数的原型。
- 属性和方法被加入到this引用的对象中。
- 新创建的对象由this所引用，并且最后隐式地返回this（如果没有显式地返回其它对象）。

以上情况看起来就像是在后台发生了如下事情：

```
var Person = function (name) {

    // 使用对象字面量模式创建一个新对象
    // var this = {};

    // 向this添加属性和方法
    this.name = name;
    this.say = function () {
        return "I am " + this.name;
    };

    // return this;
};
```

在本例中，为了简单起见，将say()方法添加到this中。其造成的结果是在任何时候调用new Person()时都会在内存中创建一个新的函数。这种方法的效率显然非常低下，因为多个实例之间的say()方法实际上并没有改变。更好的选择应该是将方法添加到Person类的原型中。

```
Person.prototype.say = function () {
    return "I am " + this.name;
};
```

我们将在后续的章节中讨论更多有关原型（prototype）和继承方面的知识，但是在这里，应该记住可重用的成员，比如可重用方法，都应该放置到对象的原型中。

在本书后面的部分中还将更加详细地阐述一件事情，但是为了完整性起见，在这里仍然值得提及该问题。这也就是我们所说的，构造函数内部发生的行为在某种意义上与后台发生的相类似。

```
// var this = {};
```

以上语句并不是真相的全部。因为“空”对象实际上并不空，它已经从Person的原型中继承了许多成员。因此，它更像是下面的语句：

```
// var this = Object.create(Person.prototype);
```

在本书的后面，将讨论Object.create()所表示的意思。

构造函数的返回值

当用new操作符创建对象时，构造函数总是返回一个对象；默认情况下返回的是this所引用的对象。如果在构造函数中并不向this添加任何属性，将返回“空”对象(这里的“空”，指的是除了从构造函数的原型中所继承的成员以外)。

构造函数将隐式返回this，甚至于在函数中没有显式地加入return语句。但是，可以根据需要返回任意其他对象。在下面的例子中，that引用了新创建的对象，并且返回了that所引用的对象。

```
var Objectmaker = function () {

    // 下面的'name'属性将被忽略
    // 这是因为构造函数决定改为返回另一个对象
    this.name = "This is it";
    // 创建并返回一个新对象
    var that = {};
    that.name = "And that's that";
    return that;
};
// 测试
var o = new Objectmaker();
console.log(o.name); // 输出为"And that's that"
```

正如从上面所看到的，可以在构造函数中自由地返回任意对象，只要它是一个对象。试图返回并非对象的值，这虽然并不会造成错误，但是函数却会简单的忽略该值，相反，构造函数将会返回this所引用的对象。

强制使用new的模式

如前所述，构造函数仍然只是函数，只不过它却以new的方式调用。如果在调用构造函数时忘记指定new操作符会发生什么？这并不会导致语法或运行时错误，但可能导致逻辑错误和意外的行为发生。发生这类问题是因为您忘记了使用new操作符，从而导致构造函数中的this指向了全局对象（在浏览器中，this将会指向window）。

当构造函数与this.member这种表示法相似，并且调用构造函数时没有使用new，那么实际上在全局对象中创建了一个新的名为member的属性，并且该属性可以通过window.member或member直接访问。这种行为是非常不希望看到的，因为众所周知，程序员应该总是争取保持全局命名空间的整洁。

```
// 构造函数
function Waffle() {
    this.tastes = "yummy";
}

// 定义一个新对象
var good_morning = new Waffle();
console.log(typeof good_morning); // "object"
console.log(good_morning.tastes); // "yummy"

// 反模式:
// 忘记使用`new`操作者
var good_morning = Waffle();
console.log(typeof good_morning); // 将输出"undefined"
console.log(window.tastes); // 将输出"yummy"
```

上面的这种意外行为在ECMAScript 5中得到了解决，并且在严格模式中，this不会指向全局对象。如果不能使用ES5标准，那么应该采取一些措施，以确保构造函数运行行为总是正常的，即使是在调用构造函数时没有使用new操作符。

命名约定

最简单的替代方法是使用命名约定，如同前面章节中所讨论的那样，使构造函数名称中的首字母变成大写的（MyConstructor），并且使“普通”函数和方法的名称中的首字母变成小写（myFunction）。

使用that

遵循命名约定一定程度上有助于避免上面忘记使用new所带来的问题，但命名约定也只是一种建议，并不能强制保证正确的行为。下面的模式可以帮助您确保构造函数的行为总是表现出构造函数应有的行为。下面的模式中，并不是将所有的成员添加到this中，可以将这些成员添加到that中，并且最后返回that。

```
function Waffle() {
    var that = {};
    that.tastes = "yummy";
    return that;
}
```

对于简单的对象，甚至不需要类似that这样的局部变量，可以仅仅从字面量中返回一个对象，如下所示：

```
function Waffle() {
    return {
        tastes: "yummy"
    };
}
```

使用上面任何一种Waffle()的实现方式都总是会返回一个对象，而无论它是如何被调用的：

```
var first = new Waffle(),
    second = Waffle();
console.log(first.tastes); // 输出"yummy"
console.log(second.tastes); // 输出"yummy"
```

这种模式的问题在于它会丢失到原型的链接，因此任何您添加到Waffle()原型的成员，对于对象来说都是不可用的。

注意： 需要注意的是，变量名that仅是一个命名公约，它并不是语言的一个部分。可以使用任意名称，可以是一些常见的变量名，包括self和me。

自调用构造函数

为了解决前面模式的缺点，并使得原型（prototype）属性可在实例对象中使用，那么可以考虑下面的方法。具体来说，可以在构造函数中检查this是否为构造函数的一个实例，如果为否，构造函数可以再次调用自身，并且在这次调用中正确地使用new操作符：

```
function Waffle() {

    if (!(this instanceof Waffle)) {
        return new Waffle();
    }

    this.tastes = "yummy";

}
Waffle.prototype.wantAnother = true;

// 测试调用
var first = new Waffle(),
    second = Waffle();

console.log(first.tastes); // 输出"yummy"
console.log(second.tastes); // 输出"yummy"
```

```
console.log(first.wantAnother); // 输出true
console.log(second.wantAnother); // 输出true
```

另一种用于检测实例对象的通用方法是将其与arguments.callee进行比较，而不是在代码中硬编码构造函数名称。如下所示：

```
if (!(this instanceof arguments.callee)) {
    return new arguments.callee();
}
```

使用这种模式是基于这样一个事实：即在每个函数内部，当该函数被调用时，将会创建一个名为arguments的对象，其中包含了传递给该函数的所有参数。同时，arguments对象中有一个名为callee的属性，该属性会指向被调用的函数。需要注意的是，在ES5的严格模式中并不支持arguments.callee属性，因此，最好限制在将来才使用该属性，并且在现有代码中删除所找到的任何实例。

数组字面量

JavaScript中的数组与语言中的绝大多数事物比较相似，即都是对象。可以创建通过内置的构造函数Array()来创建，但同时它们也可以有字面量表示，这与对象字面量类似，不过相比而言，数组字面量表示法更为简单，且更可取。

在下面的例子中，可以使用相同的元素，并以两种不同的方法创建两个数组，即使用Array()构造函数和使用字面量模式。

```
// 具有三个元素的数组
// 警告：反模式
var a = new Array("itsy", "bitsy", "spider");

// 完全相同的数组
var a = ["itsy", "bitsy", "spider"];

console.log(typeof a); // 输出"object"，这是由于数组本身也是对象类型
console.log(a.constructor === Array); // 输出true
```

数组字面量语法

数组字面量表示法（Array literal notation）并没有太多的内容：它只是一个逗号分隔的元素列表，并且整个列表包装在方括号中。可以给数组元素指定任意类型的值，包括对象或者其他数组。

数组字面量语法是非常简单、明确，并且优美的。毕竟，一个数组仅是一个零值索引列表。为此，也没有必要通过引入构造函数以及使用new操作符使得事情变得复杂起来。

数组构造函数的特殊性

避开new Array()的另一个理由是用于避免构造函数中可能产生的陷阱。

当向Array()构造函数传递单个数字时，它并不会成为第一个数组元素的值。相反，它却设定了数组的长度。这意味着newArray(3)这个语句创建了一个长度为3的数组，但是该数组中并没有实际的元素。如果试图访问该数组中的任何元素，由于该数组中不存在任何元素，将会获得undefined值。下面的代码示例向您展示了当使用字面量以及具有单个值的构造函数时所发生的不同行为。

```
// 具有一个元素的数组
var a = [3];
console.log(a.length); // 1
console.log(a[0]); // 3

// 具有三个元素的数组
var a = new Array(3);
console.log(a.length); // 3
console.log(typeof a[0]); // 输出"undefined"
```

上面的例子中，虽然这种行为可能看起来并非预期的结果，但是与您向new Array()传递一个整数相比，如果向该构造函数传递一个浮点数，则情况变得更加糟糕，如下所示。

```
// 使用数组字面量
var a = [3.14];
console.log(a[0]); // 3.14

var a = new Array(3.14); // 输出RangeError: invalid array length（范围错误，不合法的
                         数组长度）
console.log(typeof a); // 输出"undefined"
```

为了避免您在运行时创建动态数组可能产生的潜在错误，坚持使用数组字面量表示法，程序将会更加安全。

注意： 虽然有一些使用Array()构造函数的灵巧方法，比如重复字符串。下面的代码片段返回了一个具有255个空白字符的字符串（为什么不是256，这个问题留给有好奇心的读者去思考）：

```
var white = new Array(256).join(' ');
```

检查数组性质

以数组作为操作数并使用typeof操作符，其结果将会返回“object”。

虽然这种行为是有意义的（数组也是对象），但对于排除错误却没有什么帮助。通常，需要知道某个值是否是一个数组。有时候，可以检查代码是否存在length属性或者一些数组方法，比如slice()方法，以此来确定该值是否具有“数组性质”(array-ness)。但是

这些检查机制并不健壮，因为没有任何理由确定一个非数组对象就不能具有同样名称的属性和方法。另外一些人使用instanceof Array进行检查，但是这种检查机制在某些IE浏览器版本中的不同框架中运行并不正确。

ECMAScript 5定义了一个新方法Array.isArray()，该函数在参数为数组时返回true，比如：

```
Array.isArray([]); // true

// 试图以一个类似数组的对象欺骗检查
Array.isArray({
    length: 1,
    "0": 1,
    slice: function () {}
}); // false
```

如果在您的环境中无法使用这种新方法，可以通过调用Object.prototype.toString()方法对其进行检查。如果在数组上、下文环境中调用了toString的call()方法，它应该返回字符串“[object Array]”。如果该上、下文是一个对象，则它应该返回字符串“[object Object]”。因此，可以采用如下形式进行检测：

```
if (typeof Array.isArray === "undefined") {
    Array.isArray = function (arg) {
        return Object.prototype.toString.call(arg) === "[object Array]";
    };
}
```

JSON

已经熟悉了前面所讨论的数组和对象字面量，现在让我们了解一下JSON，它代表了JavaScript对象表示（JavaScript Object Notation）以及数据传输格式。它是一种轻量级数据交换格式，且可以很方便地用于多种语言，尤其是在JavaScript中。

实际上，对于JSON而言并没有任何新的知识，它只是一个数组和对象字面量表示方法的组合。下面是一个JSON字符串的例子：

```
{"name": "value", "some": [1, 2, 3]}
```

JSON和文字对象之间唯一的语法差异在于，在JSON中，属性名称需要包装在引号中才能成为合法的JSON。而在对象字面量中，仅当属性名称不是有效的标识符时才会需要引号，比如，字符之间有空格{"first name": "Dave"}。

此外，在JSON字符串中，不能使用函数或正则表达式字面量。

使用JSON

正如在前面章节中所提到的，并不推荐盲目使用eval()对任意JSON字符串进行求值，其原因在于安全性的影响。如果使用JSON.parse()方法解析字符串，其安全性会更好，因为该方法自从ES5以来一直都是语言的一个部分，并且现代浏览器中的JavaScript引擎天然地都支持该方法。对于早期的JavaScript引擎，可以使用JSON.org库（*http://www.json.org/json2.js*）以访问JSON对象及其方法。

```
// 一个输入JSON字符串
var jstr = '{"mykey": "my value"}';

// 反模式
var data = eval('(' + jstr + ')');

// 优先使用的方法
var data = JSON.parse(jstr);

console.log(data.mykey); // "my value"
```

如果已经使用了一个JavaScript库，它可能恰好带有一个解析JSON的程序，因此可能就并不需要额外的JSON.org库。比如，为了使用YUI3，可以采用如下方法：

```
// 一个输入JSON字符串
var jstr = '{"mykey": "my value"}';
// 解析该字符串，并使用一个YUI实例
// 将它转换为一个对象
YUI().use('json-parse', function (Y) {
    var data = Y.JSON.parse(jstr);
    console.log(data.mykey); // "my value"
});
```

在jQuery中存在一个parseJSON()方法：

```
// 一个输入JSON字符串
var jstr = '{"mykey": "my value"}';

var data = jQuery.parseJSON(jstr);
console.log(data.mykey); // "my value"
```

与JSON.parse()相对的方法是JSON.stringify()。它可以采用任意的对象或数组(或基本数据类型)，并且将其序列化为一个JSON字符串。

```
var dog = {
    name: "Fido",
    dob:  new Date(),
    legs: [1, 2, 3, 4]
};

var jsonstr = JSON.stringify(dog);
```

```
// jsonstr is now:
// {"name":"Fido","dob":"2010-04-11T22:36:22.436Z","legs":[1,2,3,4]}
```

正则表达式字面量

JavaScript中的正则表达式也是对象，您可以用两种方法创建正则表达式：

- 使用new RegExp()构造函数。
- 使用正则表达式字面量。

下面的示例代码演示了两种可用于创建正则表达式以匹配反斜杠的方法：

```
// 正则表达式字面量
var re = /\\/gm;

// 构造函数
var re = new RegExp("\\\\", "gm");
```

正如从上面代码中所看到的，正则表达式字面量表示法显得更加简短，并且不要求按照类似类（class-like）的构造函数方式思考。因此，建议优先原则字面量模式。

此外，当使用RegExp()构造函数时，不仅需要转义引号，并且通常还需要双反斜杠，如前面的代码片段所示，在这里需要四个反斜杠才能匹配单个反斜杠。这使得正则表达式模式显得更长，并且更加难以阅读和修改。刚开始使用正则表达式时比较困难，而任何能够使其简化的方法都是受欢迎的，因此最好坚持使用该字面量表示法。

正则表达式字面量语法

正则表达式字面量表示法使用了斜杠（分隔号“/”）来包装用于匹配的正则表达式模式。在第二个斜杠之后，可以将该模式修改为不加引号的字母形式：

- g——全局匹配。
- m——多行。
- i——大小写敏感的匹配。

模式修改器可以以任何顺序或者组合方式出现：

```
var re = /pattern/gmi;
```

当调用类似String.prototype.replace()的方法以接受正则表达式对象作为参数时，使用正则表达式字面量有助于编写出更简洁的代码。

```
var no_letters = "abc123XYZ".replace(/[a-z]/gi, "");
console.log(no_letters); // 输出123
```

使用new RegExp()的原因之一在于，某些场景中无法事先确定模式，而只能在运行时以字符串方式创建。

正则表达式字面量和构造函数之间的其中一个区别在于，字面量在解析时只有一次创建了一个对象。也就是说，如果您在一个循环中创建了相同的正则表达式，那么后面返回的对象与前面创建的对象相同，并且其所有的属性将被设置为第一次的值。考虑下面的例子，该例子演示了如何两次返回同一个对象。

```
function getRE() {
    var re = /[a-z]/;
    re.foo = "bar";
    return re;
}

var reg = getRE(),
    re2 = getRE();

console.log(reg === re2); // 输出true
reg.foo = "baz";
console.log(re2.foo); // 输出"baz"
```

注意： 这种行为已经在ES5中已经得到了改变，并且字面量会创建新的对象。在许多浏览器环境中，这种行为也得到了更正，因此，这种模式并非不可依赖。

最后需要说明的是，调用RegExp()时不使用new的行为与使用new的行为是相同的。

基本值类型包装器

JavaScript有五个基本的值类型：数字、字符串、布尔、null和undefined。除了null和undefined以外，其他三个具有所谓的基本包装对象（primitive wrapper object）。可以使用内置构造函数Number()、String()和Boolean()创建包装对象。

为了说明基本（primitive）数字和数字对象（object）之间的差异，请考虑以下例子：

```
// 一个基本数值
var n = 100;
console.log(typeof n); // 输出"number"

// 一个数值Number对象
var nobj = new Number(100);
console.log(typeof nobj); // 输出"object"
```

包装对象包含了一些有用的属性和方法。例如，数字对象具有形如toFixed()和toExponential()的方法。字符串对象具有substring()、charAt()、toLowerCase()等方法（其中还包括其他的）以及length属性。与使用基本类型相比较而言，这些包装对象的方法使用起来非常方便，这也是用这种方法创建对象的一个很好的理由。但是这些方法在基本值类型上也能够起作用。只要您调用这些方法，基本值类型就可以在后台被临时转换为一个对象，并且表现得犹如一个对象。

```
// 用来作为对象的基本字符串
var s = "hello";
console.log(s.toUpperCase()); // "HELLO"

// 值本身可以作为一个对象
"monkey".slice(3, 6);  // "key"

// 与数值的方法相同
(22 / 7).toPrecision(3); // "3.14"
```

由于基本值类型也可以充当对象，只要需要它们这样做，不过通常并没有理由去使用更为冗长的包装构造函数。很显然，比如当可以简单地使用“hi”时，就不必编写“new String("hi");”这样冗长的语句。

```
// 避免使用以下结构：
var s = new String("my string");
var n = new Number(101);
var b = new Boolean(true);

// 更好且更加简单的语句
var s = "my string";
var n = 101;
var b = true;
```

通常使用包装对象的原因之一是您有扩充值以及持久保存状态的需要。这是由于基本值类型并不是对象，它们不可能扩充属性。

```
// 基本字符串
var greet = "Hello there";

// 为了使用split()方法，基本数据类型被转换成对象
greet.split(' ')[0]; // "Hello"

// 试图增加一个原始数据类型并不是导致错误
greet.smile = true;

// 但是它并不是实际运行
typeof greet.smile; // "undefined"
```

在前面的代码片段中，greet只能临时转换成对象，以使得该属性/方法可访问，且运行时不会产生错误。另一方面，如果使用new String()来定义一个对象greet，那么对其扩

充的smile属性将会按照预期运行。扩充一个字符串、数字或布尔值的情况比较少见，除非这种行为就是您所需要的，否则可能并不需要包装构造函数。

当使用时没有带new操作符时，包装构造函数将传递给它们的参数转换成一个基本类型值，如下所示：

```
typeof Number(1); // 输出"number"
typeof Number("1"); // 输出"number"
typeof Number(new Number()); // 输出"number"
typeof String(1); // 输出"string"
typeof Boolean(1); // 输出"boolean"
```

错误对象

JavaScript中有一些内置错误构造函数（error constructor），比如Error()、SyntaxError()、TypeError()以及其他，这些处错误构造函数都带有throw语句。通过这些错误构造函数创建的错误对象具有下列属性：

name

用于创建对象的构造函数的名称属性。它可以是一般的“Error”或者更为专门的构造函数，比如“RangeError”。

message

当创建对象时传递给构造函数的字符串。

错误对象也还有一些其他的属性，比如发生错误的行号和文件名，但这些额外属性都是浏览器扩展属性，在多个浏览器实现中实现并不一致，因而并不可靠。

另一方面，throw适用于任何对象，并不一定是由某个错误构造函数所创建的对象，因此可以选择抛出自己的对象。这种错误对象也可以有属性“name”、“message”，以及任意希望传递给catch语句来处理的其他类型的信息。当涉及自定义错误对象时，可以发挥创造性，并且用这些错误对象来将应用程序的状态恢复到正常状态。

```
try {
    // 发生意外的事情，抛出一个错误
    throw {
        name: "MyErrorType", // 自定义错误类型
        message: "oops",
        extra: "This was rather embarrassing",
        remedy: genericErrorHandler // 指定应该处理该错误的函数
    };
} catch (e) {
    // 通知用户
    alert(e.message); // 输出"oops"
```

```
    // 优美的处理错误
    e.remedy(); // 调用函数genericErrorHandler()
}
```

错误构造函数以函数的形式调用（不带new）时，其表现行为与构造函数（带new）相同，并且返回同一个错误对象。

小结

在本章中，已经学习到不同的字面量模式，与使用构造函数相比较而言，这些都是更为简单的替代方法。本章主要讨论了下列主题：

- 对象字面量表示法，这是一种优美的对象创建方式，它以包装在大括号中的逗号分隔的键－值（key-value）对的方式创建对象。
- 构造函数，主要包括内置构造函数(几乎总是有一个更好且更短的字面量表示法)和自定义构造函数。
- 一种可以确保自定义构造函数总是表现得像以new调用的方法。
- 数组字面量表示法，方括号内以逗号分隔的值列表。
- JSON，一种包含对象和数组字面量的数据格式。
- 正则表达式字面量。
- 其他内置构造函数以避免使用String()、Number()、Boolean()等，此外还讨论了多种不同的Error()构造函数。

在一般情况下，除了Date()构造函数以外，很少需要使用其他内置构造函数。下面的表格总结了这些构造函数及其相应的、选择更优的字面量模式。

Built-in constructors (avoid)

```
var o = new Object();
var a = new Array();
var re = new RegExp(
    "[a-z]",
    "g"
);
var s = new String();
var n = new Number();
var b = new Boolean();
throw new Error("uh-oh");
```

Literals and primitives (prefer)

```
var o = {};
var a = [];
var re = /[a-z]/g;


var s = ""
var n = 0;
var b = false;
   throw {
        name: "Error"
        message: "uh-oh"
   }
...or
   throw Error("uh-oh");
```

第4章

函数

对于JavaScript程序员来说，掌握函数是最基本的技能，因为JavaScript语言中在许多情况下都要用到函数。它们需要执行各种各样的任务，而这些任务在其他语言中却可能有相应的特殊语法来完成。

在本章中，将学习到以多种不同的方式来定义JavaScript中的函数，您会学习到函数表达式和函数声明，并且还会看到局部作用域(local scope)和变量声明提升（variable hoisting）的工作原理。然后，还将会学习到大量对API（为函数提供更好的界面）、代码初始化（以更少的全局变量），以及程序性能（也就是说，降低运行开销）有帮助的模式。

让我们开始深入探讨函数，首先回顾并阐明一些重要的基础知识。

背景

JavaScript中的函数有两个主要特点使其显得比较特殊。第一个特点在于函数是第一类对象（first-class object），第二个特点在于它们可以提供作用域。

函数就是对象，其表现如下：

- 函数可以在运行时动态创建，还可以在程序执行过程中创建。
- 函数可以分配给变量，可以将它们的引用复制到其他变量，可以被扩展，此外，除少数特殊情况外，函数还可以被删除。
- 可以作为参数传递给其他函数，并且还可以由其他函数返回。
- 函数可以有自己的属性和方法。

因此，对于函数A来说，它可能是一个对象，并且具有自己的属性和方法，而且其中的方法之一可能恰好又是另一个函数B。此外，函数B可以接受函数C作为参数，并且在执行时可以返回另外的函数D。初看起来，有许多函数需要记录。但是适应各种函数应用后，将开始欣赏函数所提供的能力、灵活性、以及表现力。一般来说，当考虑比较JavaScript中的函数、对象时，其唯一的特性在于该对象（即函数）是可调用的，这意味着它是可执行的。

事实上，当看到new Function()构造函数运行时，函数就是对象的意义就变得非常明确了：

```
// 反模式
// 仅用于演示目的
var add = new Function('a, b', 'return a + b');
add(1, 2); // returns 3
```

在以上这段代码中，毫无疑问，add()是一个对象，毕竟它是由一个构造函数所创建。然而，使用Function()构造函数并不是一个好主意［如同使用eval()一样都不好］，这是由于代码是以字符串方式传递并重新计算。同样，这也不便于编写(以及阅读)，这是因为必须用引号隔开代码，并且如果出于可读性的目的而希望在函数中正确地缩进代码，那么还需要格外注意。

第二个重要的特征在于函数提供了作用域。在JavaScript中有没有花括号{}语法以定义局部作用域。也就是说，块并不创建作用域。JavaScript中仅存在函数作用域。在函数内部以var关键词定义的任何变量都是局部变量，对于函数外部是可不见的。考虑到花括号{}并不提供局部作用域，因此如果在if条件语句或在for以及while循环中，使用var关键词定义一个变量，这并不意味着该变量对于if或for来说是局部变量。它仅对于包装函数来说是局部变量，并且如果没有包装函数，它将成为一个全局变量。正如第2章所讨论的，最小化全局变量的数目是一个好习惯，因此，在涉及使变量作用域受控制的情况下，函数就是必不可少的工具。

消除术语的歧义

让我们花费一点时间讨论用于定义函数的相关代码的术语，因为在谈论到模式时，使用准确、约定的名称与代码是同等重要的。

考虑下面的代码片断：

```
// 命名函数表达式
var add = function add(a, b) {
    return a + b;
};
```

前面的代码显示了一个函数，它使用了命名函数表达式（named function expression）。

如果跳过函数表达式中的名称（例子中的第二个add），将会得到一个未命名（unnamed）函数表达式，也简称为函数表达式，或者最常见的是将其称之为匿名函数（anonymous function）。如下所示：

```
// 函数表达式，又名匿名函数
var add = function (a, b) {
    return a + b;
};
```

因此广义上称为函数表达式，并且命名函数表达式是一个函数表达式的一种特殊情况，通常发生在定义可选的命名时。

当省略了第二个add并且以一个未命名函数表达式作为结束，这并不会影响该函数的定义以及后续的调用。唯一的区别在于该函数对象的name属性将会成为一个空字符串。name属性是Javascript语言的一个扩展(它并不是ECMA标准的一部分)，但是在许多环境中得到了广泛的应用。如果您保留了第二个add，那么add.name属性将会包含字符“add”。当使用调试器时，比如Firebug，或者当从自身递归调用同一个函数时，name属性是非常有用的；否则，可以跳过该属性。

最后，获得了函数声明（function declaration）。这些声明看起来与其他语言中所使用的函数极为相似：

```
function foo() {
    // 此处为函数主体
}
```

就语法而言，命名函数表达式与函数的声明看起来很相似，尤其是如果不将函数表达式的结果分配给变量（正如将在本章后面的回调模式中所看到的）的时候。有时候，没有其他方法可以区分出函数声明和命令函数表达式的差异，除非查看函数出现的上、下文语境，正如将在下一节中所看到的。

在尾随的分号中，这两者之间在语法上存在差异。函数声明中并不需要分号结尾，但在函数表达式中需要分号，并且应该总是使用分号，即使编辑器中分号自动插入机制可能帮您完成了此工作。

注意： 函数字面量（function literal）这个术语也经常被使用，它可能表示一个函数表达式或命名函数表达式。由于这种模糊性含义，如果不使用该术语可能会更好。

声明Vs 表达式：名称和变量声明提升(Hoisting)

因此，应该使用哪种方法，函数声明或者函数表达式？在不能使用声明的情况下，下面将为您解决这种困境。其示例包括将函数对象作为参数传递，或者在对象字面量（object literal）中定义方法：

```
// 这是一个函数表达式
// 它作为参数传递给函数 `callMe`
callMe(function () {
    // 这里是命名函数表达式
    // 也被称为匿名函数
});

// 这是命令函数表达式
callMe(function me() {
    // 这里是命名函数表达式，
    // 并且其名称为"me"
});

// 另一个函数表达式
var myobject = {
    say: function () {
        // 这里是函数表达式
    }
};
```

函数声明只能出现在“程序代码”中，这表示它仅能在其他函数体内部或全局空间中。它们的定义不能分配给变量或属性，也不能以参数形式出现在函数调用中。下面是一个允许使用函数声明的例子，其中所有函数foo()、bar()和local()都是以函数声明模式进行定义：

```
// 全局作用域
function foo() {}

function local() {
    // 局部作用域
    function bar() {}
    return bar;
}
```

函数的命名属性

当选择函数定义模式的时候，另一个需要考虑的事情是有关只读name属性的可用性。同样，这个属性并不是标准，但在许多环境中都可以使用它。在函数声明和命名函数表达式中，已经定义了name属性。在匿名函数表达式中，它依赖于其实现方式。其name属性可能是未定义的（比如在IE浏览器环境中），也可能是以空字符串（比如在Firefox、WebKit浏览器环境中）来定义name属性。

```
function foo() {} // 声明
var bar = function () {}; // 表达式
var baz = function baz() {}; // 命名表达式

foo.name; // 输出"foo"
bar.name; // 输出""
baz.name; // 输出"baz"
```

在Firebug或其他调试器中调试代码时，name属性是非常有用的。当调试器需要展示函数中的错误时，它可以检测name属性是否存在，并将其作为一个指示符。name属性也可用于在自身内部递归调用同一个函数。如果对这两种情况都没有兴趣，那么未命名函数表达式将显得更加容易、更简洁。

不利用使用函数声明的情况下，并且偏好使用函数表达式。原因在于表达式突出了函数与所有其他对象一样都是对象这个事实，并且不需要特殊语言结构。

注意： 使用命名函数表达式并且将其分配给一个具有不同名称的变量，在技术上是可行的，如下例所示：

```
var foo = function bar() {};
```

然而在一些浏览器（比如IE浏览器）中，这种用法的行为没有被正确地实现，所以并不推荐使用这种模式。

函数的提升

从前面的讨论中，可能会得出函数声明的行为几乎等同于命名（named）函数表达式的行为。然而这并不完全正确，其区别在于提升（hoisting）行为。

注意： 变量声明提升（hoisting）这个术语在ECMAScript中并没有定义，但它却是一种用于描述行为的常见的好方法。

正如您所知，对于所有的变量，无论在函数体的何处进行声明，都会在后台被提升到函数顶部。而这对于函数同样适用，其原因在于函数只是分配给变量的对象。唯一“明白”的地方在于当使用函数声明时，函数定义也被提升，而不仅仅是函数声明被提升。请注意考虑下列代码片断：

```
// 反模式
// 仅用于演示

// 全局函数
function foo() {
    alert('global foo');
}
```

```
function bar() {
    alert('global bar');
}

function hoistMe() {

    console.log(typeof foo); // 输出"function"
    console.log(typeof bar); // 输出"undefined"

    foo(); // 输出"local foo"
    bar(); // 输出TypeError: bar is not a function

    // 函数声明
    // 变量'foo'以及其实现者被提升
    function foo() {
        alert('local foo');
    }

    // 函数表达式
    // 仅变量'bar'被提升
    // 函数实现并未被提升
    var bar = function () {
        alert('local bar');
    };

}
hoistMe();
```

这个例子中可以看到，如同正常的变量一样，仅存在于hoistMe()函数中的foo和bar移动到了顶部，从而覆盖了全局foo和bar函数。两者之间的区别在于局部foo()的定义被提升到顶部且能正常运行，即使在后面才定义它。bar()的定义并没有提升，仅有它的声明被提升。这也就是为什么代码执行到达bar()的定义时，其显示结果是undefined且并没有作为函数来使调用［然而，在作用域链中仍然防止全局bar()被“看到”］。

现在，所需要的有关函数的背景和术语已没有什么问题，让我们来看看一些Javascript所提供的与函数相关的良好模式，首先从回调（callback）模式开始讨论。再次强调，重要的是要记住Javascript中函数的两个特征：

- 它们都是对象。
- 它们提供局部作用域。

回调模式

函数都是对象，这表示它们可以作为参数传递给其他函数。当您将函数introduceBugs()作为参数传递给函数writeCode()时，那么在某一时刻writeCode()可能会执行（或者调用）introduceBugs()。在这种情况下，introduceBugs()就被称之为回调函数（*callback function*），也可简称为回调（*callback*）。

```
function writeCode(callback) {
    // 执行一些事务...
    callback();
    // ...
}

function introduceBugs() {
    // ... 引入漏洞
}

writeCode(introduceBugs);
```

请注意introduceBugs()作为参数传递给writeCode()时是不带括号的。括号表示要执行函数，而在这种情况下，我们仅需要传递该函数的应用，而让writeCode()在适当的时候来执行它（也就是说，返回以后调用）。

回调示例

让我们举个例子，开始时并不使用回调函数，然后在后面对其重构。假设有一个通用的函数执行一些复杂的处理工作，并且其返回结果为一个大块数据。可以调用这些通用函数，比如findNodes()，它的任务是抓取页面的DOM树，并返回一个您所关注的页面元素数组。

```
var findNodes = function () {
    var i = 100000, // 大而繁重的循环
        nodes = [], // 存储该结果
        found; // 找到了下一个节点
    while (i) {
        i -= 1;
        // 这里是复杂的逻辑...
        nodes.push(found);
    }
    return nodes;
};
```

保持函数的通用性并且使其仅返回一个DOM节点数组，而不对实际元素做任何处理，这是一个非常好的思想。修改节点的语义逻辑可以在不同的函数中实现，比如在一个名为hide()的函数，该函数顾名思义，其功能用于隐藏页面中的节点。

```
var hide = function (nodes) {
    var i = 0, max = nodes.length;
    for (; i < max; i += 1) {
        nodes[i].style.display = "none";
    }
};

// 执行该函数
hide(findNodes());
```

以上这个实现是低效的，因为hide()必须再次循环遍历由findNodes()所返回的数组节点。如果能避免这种循环，并且只要在findNodes()中选择便可隐藏节点，那么这将是更高效的实现方式。但是如果在findNodes()中实现隐藏逻辑，由于检索和修改逻辑耦合（coupling），那么它不再是一个通用函数。对这种问题的解决方法是采用回调模式，可以将节点隐藏逻辑以回调函数方式传递给findNodes()并委托其执行：

```
// 重构findNodes()以接受一个回调函数
var findNodes = function (callback) {
    var i = 100000,
        nodes = [],
        found;

    // 检查回调函数是否为可调用的
    if (typeof callback !== "function") {
        callback = false;
    }

    while (i) {
        i -= 1;

        // 这里是复杂的逻辑……

        // 现在运行回调函数
        if (callback) {
            callback(found);
        }
        nodes.push(found);
    }
    return nodes;
};
```

这是一种很直接的实现方法。findNodes()执行的唯一额外任务是，检查是否提供了可选回调函数，并且如果存在的话，那么就执行该函数。其中，回调函数是可选的，因此重构后的findNodes()仍然可以像以前一样使用，而不会破坏依赖旧API的原始代码。

现在，hide()的实现要简单得多，因为它并不需要循环遍历所有节点：

```
// 回调函数
var hide = function (node) {
    node.style.display = "none";
};

// 找到指定节点，并在后续执行中将其隐藏起来
findNodes(hide);
```

如前面的代码所示，回调函数可以是一个已有的函数，或者也可以是一个匿名函数，可以在调用主函数时创建它。例如，下面的代码展示了如何使用同样的findNodes()函数以显示节点：

```
// 传递一个匿名回调函数
findNodes(function (node) {
    node.style.display = "block";
});
```

回调与作用域

在前面的例子中，回调执行的语句部分如下所示：

```
callback(parameters);
```

虽然在许多情况下这种方法都是简单而且有效的，但经常存在一些场景，其回调并不是一次性的匿名函数或全局函数，而是对象的方法。如果该回调方法使用this来引用它所属的对象，这可能会导致意想不到的行为发生。

假设回调函数是paint()，它是一个名为myapp的对象的方法：

```
var myapp = {};
myapp.color = "green";
myapp.paint = function (node) {
    node.style.color = this.color;
};
```

函数findNodes()执行以下语句：

```
var findNodes = function (callback) {
    // ...
    if (typeof callback === "function") {
        callback(found);
    }
    // ...
};
```

如果调用findNodes(myapp.paint)，它并不会按照预期那样运行，这是由于this.color没有被定义。由于findNodes()是一个全局函数，因此，对象this引用了全局对象。类似的，如果findNodes()是一个名为dom的对象的方法(dom.findNodes())，那么回调内部的this将指向dom，而不是预期的myapp。

对这个问题的解决方案是传递回调函数，并且另外还传递该回调函数所属的对象：

```
findNodes(myapp.paint, myapp);
```

然后，还需要修改findNodes()以绑定所传递进入的对象：

```
var findNodes = function (callback, callback_obj) {
    //...
```

```
        if (typeof callback === "function") {
            callback.call(callback_obj, found);
        }
        // ...
    };
```

在后面的章节中，将会有更多有关绑定与使用call()和apply()的相关主题。

传递一个对象和一个方法以用做回调函数的另一种选择是，将其中的方法作为字符串来传递。因此无需重复两次输入该对象的名称，即：

```
findNodes(myapp.paint, myapp);
```

可以变为

```
findNodes("paint", myapp);
```

然后，findNodes()将会按照以下代码行执行任务：

```
var findNodes = function (callback, callback_obj) {

    if (typeof callback === "string") {
        callback = callback_obj[callback];
    }

    //...
    if (typeof callback === "function") {
        callback.call(callback_obj, found);
    }
    // ...
};
```

异步事件监听器

回调模式有许多常见用途，比如，当附加一个事件监听器到页面上的一个元素时，实际上提供了一个回调函数指针，该函数将会在事件发生时被调用。下面是一个简单的例子，展示了当监听到文档单击事件时如何传递回调函数console.log()。

```
document.addEventListener("click", console.log, false);
```

大多数的客户端浏览器编程都是事件驱动的。当页面加载完毕时，将会触发load事件。然后，用户与页面交互时，将会造成触发各种事件，比如click、keypress、mouseover、mousemove等。JavaScript特别适合于事件驱动编程，因为回调模式支持您的程序以异步方式运行，也就是说，可以乱序方式运行。

“不要给我们打电话，我们会给你打电话。这是好莱坞的一句名言，指的是当许多候选人试镜以竞争获得电影中的同一个角色时常见的托辞。实际上，影片工作人员也不可能

在所有时间内都回答所有候选人的电话。在异步事件驱动的JavaScript中，也有相似的现象发生。只不过不是给您电话号码，您提供的是将在合适时间被调用的回调函数。可能会提供超过所需数量的回调函数，因为某些事件可能从来都不会发生。比如，如果用户从不单击“Buy now!”(立即购买)，那么验证信用卡号格式的函数将从不会被调用。

超时

使用回调模式的另一个例子是，当使用浏览器的window对象所提供的超时方法：setTimeout()和setInterval()。这些方法也会接受并执行回调函数：

```
var thePlotThickens = function () {
    console.log('500ms later...');
};
setTimeout(thePlotThickens, 500);
```

请再次注意函数thePlotThickens是如何以变量方式传递的，传递该函数时并没有带括号，因为并不想立即执行该函数，而只是想指向该函数以便setTimeout()在以后使用。传递的字符串"thePlotThickens()"也并不是函数指针，而只是一个类似eval()重新运算的常见反模式。

库中的回调模式

回调模式是一种简单而又强大的模式，当设计一个库时它可以派上用场。进入软件库的代码应该尽可能地是通用和可复用的代码，而回调可以帮助实现这种通用化。不需要预测和实现能想到的每一项功能，因为这样会迅速使库膨胀，而绝大多数用户永远不会需要其中大量的功能。相反，可以专注于核心功能并提供“挂钩”形式的回调函数，这将使您很容易地构建、扩展，以及自定义库方法。

返回函数

函数也是对象，因此它们可以用做为返回值。这表示一个函数并不需要以某种数据值或数据数组作为其执行结果返回。函数可以返回另一个更专门的函数，也可以按需创建另一个函数，这取决于其输入。

下面是一个简单的例子：一个函数执行了一些工作，可能包含一些一次性的初始化，然后根据它的返回值继续运行。返回值恰好是另一个函数，而它也是可执行的：

```
var setup = function () {
    alert(1);
    return function () {
        alert(2);
```

```
    };
};

// 使用setup函数
var my = setup(); // alerts 1
my(); // alerts 2
```

由于setup()包装了返回函数，它创建了一个闭包，可以使用这个闭包存储一些私有数据，而这些数据仅可被该返回函数访问，但外部代码却无法访问。下面是一个计数器的例子，每次当您调用该函数时，它将会产生一个递增的值：

```
var setup = function () {
    var count = 0;
    return function () {
        return (count += 1);
    };
};

// 用法
var next = setup();
next(); // 返回1
next(); // 2
next(); // 3
```

自定义函数

函数可以动态定义，也可以分配给变量。如果创建了一个新函数并且将其分配给保存了另外函数的同一个变量，那么就以一个新函数覆盖了旧函数。在某种程度上，回收了旧函数指针以指向一个新函数。而这一切发生在旧函数体的内部。在这种情况下，该函数以一个新的实现覆盖并重新定义了自身。这可能听起来比实际上更复杂，下面让我们看一个简单的例子：

```
var scareMe = function () {
    alert("Boo!");
    scareMe = function () {
        alert("Double boo!");
    };
};

// 使用自定义函数
scareMe(); // 输出Boo!
scareMe(); // 输出Double boo!
```

当您的函数有一些初始化准备工作要做，并且仅需要执行一次，那么这种模式就非常有用。因为并没有任何理由去执行本可以避免的重复工作，即该函数的一些部分可能并不再需要。在这种情况下，自定义函数（self-defining function）可以更新自身的实现。

使用此模式可以显著地帮助您提升应用程序的性能，这是由于重新定义的函数仅执行了更少的工作。

注意： 这种模式的另一个名称是“惰性函数定义”（lazy function definition），因为该函数直到第一次使用时才被正确地定义，并且其具有后向惰性，执行了更少的工作。

该模式的其中一个缺点在于，当它重定义自身时已经添加到原始函数的任何属性都会丢失。此外，如果该函数使用了不同的名称，比如分配给不同的变量或者以对象的方法来使用，那么重定义部分将永远不会发生，并且将会执行原始函数体。

下面让我们来看一个例子，该例中`scareMe()`函数将以第一类对象（first-class object）使用的方式来使用：

1. 添加一个新的属性。
2. 函数对象被分配给一个新的变量。
3. 该函数也以一个方法的形式使用。

考虑下面的代码片段：

```
// 1. 添加一个新的属性
scareMe.property = "properly";

// 2. 赋值给另一个不同名称的变量
var prank = scareMe;

// 3. 作为一个方法使用
var spooky = {
    boo: scareMe
};

// calling with a new name
prank(); // 输出"Boo!"
prank(); // 输出"Boo!"
console.log(prank.property); // 输出"properly"

// 作为一个方法来调用
spooky.boo(); // 输出"Boo!"
spooky.boo(); // 输出"Boo!"
console.log(spooky.boo.property);  // 输出"properly"

// 使用自定义函数
scareMe(); // 输出Double boo!
scareMe(); // 输出Double boo!
console.log(scareMe.property); // 输出undefined
```

正如您所看到的，当将该函数分配给一个新的变量时，如预期的那样，函数的自定义(self-definition)并没有发生。每次当调用`prank()`时，它都通知“Boo!”消息，同时它还

覆盖了全局scareMe()函数，但是prank()自身保持了可见旧函数，其中还包括属性。当该函数以spooky对象的boo()方法使用时，也发生了同样的情况。所有这些调用不断地重写全局scareMe()指针，以至于当它最终被调用时，它才第一次具有更新函数主体并通知“Double boo”消息的权利。此外，它也不能访问scareMe.property属性。

即时函数

即时函数模式（Immediate Function pattern）是一种可以支持在定义函数后立即执行该函数的语法。下面是一个即时函数的示例：

```
(function () {
    alert('watch out!');
}());
```

这种模式本质上只是一个函数表达式(无论是命名或匿名的)，该函数会在创建后立刻执行。在ECMAScript标准中并没有定义术语“即时函数(immediate function)”，但是这种模式非常简洁，下面将介绍并讨论该模式。

该模式由以下几部分组成：

- 可以使用函数表达式定义一个函数（函数声明则无法达到这个效果）。
- 在末尾添加一组括号，这将导致该函数立即执行。
- 将整个函数包装在括号中（只有不将该函数分配给变量才需要这样做）。

下面的替代语法也是很常见的(请注意右括号的位置)，但JSLint偏好使用第一种语法：

```
(function () {
    alert('watch out!');
})();
```

这种模式是有非常有用的，因为它为初始化代码提供了一个作用域沙箱(sandbox)。考虑以下常见的场景：当页面加载时，代码必须执行一些设置任务，比如附加事件处理程序、创建对象等诸如此类的任务。所有这些工作仅需要执行一次，因此没有理由去创建一个可复用的命名函数。但是代码也还需要一些临时变量，而在初始化阶段完成后就不再需要这些变量。然而，以全局变量形式创建所有那些变量是一个差劲的方法。这就是为什么需要一个即时函数的原因，用以将所有代码包装到它的局部作用域中，且不会将任何变量泄露到全局作用域中：

```
(function () {

    var days = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'],
```

```
        today = new Date(),
        msg = 'Today is ' + days[today.getDay()] + ', ' + today.getDate();

    alert(msg);

}()); // 输出"Today is Fri, 13"
```

如果上面这些代码没有包装到即时函数中，那么days、today和msg等变量将会成为全局变量，并遗留在初始化代码中。

即时函数的参数

也可以将参数到传递即时函数中，如下面的例子所演示的那样：

```
// 打印字符串：
// I met Joe Black on Fri Aug 13 2010 23:26:59 GMT-0800 (PST)

(function (who, when) {

    console.log("I met " + who + " on " + when);

}("Joe Black", new Date()));
```

一般情况下，全局对象是以参数方式传递给即时函数的，以便于在不使用window:指定全局作用域限定的情况下可以在函数内部访问该对象，这样将使得代码在浏览器环境之外时具有更好的互操作性：

```
(function (global) {

    // 通过`global`访问全局变量

}(this));
```

请注意，一般来说，不应该传递过多的参数到即时函数中，因为这样将迅速成为一种阅读负担，使在理解代码运行流程时需要不断地滚动到该函数的顶部和底部。

即时函数的返回值

正如任何其他函数一样，即时函数可以返回值，并且这些返回值也可以分配给变量：

```
var result = (function () {
    return 2 + 2;
}());
```

另一种方式也可达到同样的效果，即忽略包装函数的括号，因为将即时函数的返回值分配给一个变量时并不需要这些括号。忽略第一组括号时，代码示例如下所示：

```
var result = function () {
```

```
    return 2 + 2;
}();
```

这个语法虽然比较简单，但它可能看起来有点令人误解。在没有注意到该函数尾部的()时，一些阅读代码的人可能会认为result变量指向一个函数。实际上，result指向由即时函数返回的值，在这种情况下，指向了数字4。

另一种语法也可以得到同样的结果：

```
var result = (function () {
    return 2 + 2;
})();
```

前面的例子中，运行即时函数后返回的结果是一个整型值。但是实际上即时函数不仅可以返回原始值，还可以返回任意类型的值，包括另一个函数。因此，可以使用即时函数的作用域以存储一些私有数据，而这特定于返回的内部函数。

在接下来的例子中，即时函数返回的值是一个函数，它将分配给变量getResult，并且将简单地返回res值，该值被预计算并存储在即时函数的闭包中：

```
var getResult = (function () {
    var res = 2 + 2;
    return function () {
        return res;
    };
}());
```

当定义对象属性时也可以使用即时函数。想象一下，如果需要定义一个在对象生存期内永远都不会改变的属性，但是在定义它之前需要执行一些工作以找出正确的值。此时，可以使用一个即时函数包装这些工作，并且即时函数的返回值将会成为属性值。下面的代码展示了一个使用示例：

```
var o = {
    message: (function () {
        var who = "me",
            what = "call";
        return what + " " + who;
    }()),
    getMsg: function () {
        return this.message;
    }
};
// 用法
o.getMsg(); // 输出"call me"
o.message;  // 输出"call me"
```

在上面这个例子中，是一个字符串属性，而不是一个函数，但它需要一个函数，该函数在脚本加载时执行，并且帮助定义该o.message属性。

优点和用法

即时函数模式得到了广泛的使用。它可以帮助包装许多想要执行的工作，且不会在后台留下任何全局变量。定义的所有这些变量将会是用于自调用函数（self-invoking function）的局部变量，并且不用担心全局空间被临时变量所污染。

注意： 即时函数的其他名称包括“自调用”(self-invoking)以及“自执行”（self-executing）函数，因为该函数在定义后立即执行。

这种模式也经常用于书签工具中，因为书签工具可以在任何网页上运行，并且保持全局命名空间的整洁（并且书签代码是不显眼的）也是至关重要的。

该模式还支持将个别功能包装在自包含模块（self-contained module）中。想象一下，网页是静态的，并且在没有JavaScript的情况下运行良好。然后在逐步增强的精神下，以某种方式添加了一段代码以增强该面的功能。可以将这些代码（也可以将其称之为“模块”或者“功能”）包装到一个即时函数中，并且确保该页面在存在或不存在该代码的两种情况下都能够良好地运行。然后，可以添加更多的增强功能，同时还可以将其删除、分割对其进行测试、允许用户禁用等。

可以使用下面的模板来定义一个功能，让我们将其称之为module1：

```
// 文件module1.js中定义的模块module1
(function () {

  // 模块1的所有代码...
}());
```

遵循这个模板，可以编码其他模块。然后，当将该代码发布到在线站点时，可以决定哪些功能准备应用于黄金时间，并且使用构建脚本将对应文件合并。

即时对象初始化

保护全局作用域不受污染的另一种方法，类似于前面介绍的即时函数模式，也就是下面介绍的即时对象初始化（immediate object initialization）模式。这种模式使用带有init()方法的对象，该方法在创建对象后将会立即执行。init()函数需要负责所有的初始化任务。

下面是即时对象模式的一个示例：

```
({
    // 在这里可以定义设定值
    // 又名配置常数
    maxwidth: 600,
    maxheight: 400,

    // 还可以定义一些实用的方法
    gimmeMax: function () {
        return this.maxwidth + "x" + this.maxheight;
    },

    // 初始化
    init: function () {
        console.log(this.gimmeMax());
        // 更多初始化任务...
    }
}).init();
```

就语法而言，对待这种模式就像在使用对象字面量创建一个普通的对象。也可以将字面量包装到括号中（分组操作符），它指示JavaScript引擎将大括号作为对象字面量，而不是作为一个代码块（也不是if或者for循环）。在结束该括号之后，可以立即调用init()方法。

也可以将对象和init()调用同时包装到组括号中，而不是仅包装对象。换言之，这两种方法都可以运行：

```
({...}).init();
({...}.init());
```

这种模式的优点与即时函数模式的优点是相同的：可以在执行一次性的初始化任务时保护全局命名空间。与仅仅将一堆代码包装到匿名函数的方法相比，这种模式看起来涉及更多的语法特征，但是如果初始化任务更加复杂(它们也往往的确比较复杂)，它会使整个初始化过程显得更有结构化。比如，私有帮助函数是非常清晰可辨别的，因为它们是临时对象的属性，而在即时函数模式中，它们就很可能只是分散在各处的函数而已。

这种模式的缺点在于，绝大多数JavaScript 缩小器(minifier)工具并不会像仅包装在函数中的代码那样有效的缩减这种模式。私有属性和方法并不会被重新命名为更短的名称，因为从一个缩小器工具的角度来看，在缩减名称长度时并不容易保证安全性。在编写代码的时候，处于“高级”模式状态的Google闭包编译器(Closure Compiler)是唯一一个将即时对象的属性重命名为更短名称的缩小器工具，它将前面的例子变成如下所示的代码：

```
({d:600,c:400,a:function(){return this.d+"x"+this.c},b:function(){console.log(this.
a())}}).b();
```

注意： 这种模式主要适用于一次性的任务，而且在init()完毕后也没有对该对象的访问。如果想在init()完毕后保存对该对象的一个引用，可以通过在init()尾部添加“return this;”语句实现该功能。

初始化时分支

初始化时分支［Init-time branching，也称为加载时分支（load-time branching）］是一种优化模式。当知道某个条件在整个程序生命周期内都不会发生改变的时候，仅对该条件测试一次是很有意义的。浏览器嗅探(或功能检测) 就是一个典型的例子。

例如，在发现XMLHttpRequest可作为原生对象支持后，在程序执行过程中，底层的浏览器并没有机会改变，并且出乎意料您又需要处理ActiveX对象。由于环境并不会改变，代码就没有理由在每次需要另一个XHR对象时继续保持嗅探(并且将得到同样的结论)。

查明DOM元素的计算样式或附加的事件处理程序是另外一个可以受益于初始化时分支模式的场景。绝大多数程序开发员都已经编写过这样的代码，至少有一次在他们的客户端编程生命期内，即可用于附加或删除事件监听器的工具，如下面的例子所示：

```
// 之前
var utils = {
    addListener: function (el, type, fn) {
        if (typeof window.addEventListener === 'function') {
            el.addEventListener(type, fn, false);
        } else if (typeof document.attachEvent === 'function') { // IE
            el.attachEvent('on' + type, fn);
        } else { // 更早版本的浏览器
            el['on' + type] = fn;
        }
    },
    removeListener: function (el, type, fn) {
        // 几乎一样...
    }
};
```

此段代码的问题在于效率比较低下。 每次在调用utils.addListener()或utils.removeListener()时，都将会重复地执行相同的检查。

当使用初始化时分支的时候，可以在脚本初始化加载时一次性探测出浏览器特征。此时，可以在整个页面生命期内重定义函数运行方式。下面是一个可以处理这个任务的例子：

```
// 之后

// 接口
var utils = {
    addListener: null,
    removeListener: null
};
// 实现
if (typeof window.addEventListener === 'function') {
    utils.addListener = function (el, type, fn) {
        el.addEventListener(type, fn, false);
    };
    utils.removeListener = function (el, type, fn) {
        el.removeEventListener(type, fn, false);
    };
} else if (typeof document.attachEvent === 'function') { // 判断为IE浏览器
    utils.addListener = function (el, type, fn) {
        el.attachEvent('on' + type, fn);
    };
    utils.removeListener = function (el, type, fn) {
        el.detachEvent('on' + type, fn);
    };
} else { // 更早版本的浏览器
    utils.addListener = function (el, type, fn) {
        el['on' + type] = fn;
    };
    utils.removeListener = function (el, type, fn) {
        el['on' + type] = null;
    };
}
```

现在是时候对浏览器嗅探的使用提出警示了。当使用这个模式时，请不要过度假设浏览器特征。比如，如果已经探测浏览器并不支持window.addEventListener，请不要假设所处理的浏览器就是IE浏览器，且它也不支持原生XMLHttpRequest，虽然在浏览器历史中的某一时刻是真实存在的。可能在某些情况下，可以安全地假设那些特征是共存的，比如.addEventListener和.removeEventListener就是共存的，但是通常情况下，浏览器特征都是独立更新的。最好的策略就是分开嗅探浏览器特征，然后使用加载时分支仅执行一次嗅探。

函数属性——备忘模式

函数是对象，因此它们具有属性。事实上，它们确实还有属性和方法。例如，对于每一个函数，无论使用什么样的语法来创建它，它都会自动获得一个length属性，其中包含了该函数期望的参数数量。

```
function func(a, b, c) {}
console.log(func.length); // 3
```

可以在任何时候将自定义属性添加到你的函数中。自定义属性的其中一个用例是缓存函数结果(即返回值)，因此，在下一次调用该函数时就不用重做潜在的繁重计算。缓存函数结果也被称为备忘（memoization）。

在下面的例子中，函数myfunc创建了一个属性cache，该属性可以通过myFunc.cache像通常那样进行访问。cache属性是一个对象(即哈希对象)，其中使用传递给函数的参数param作为键，而计算结果作为值。计算结果可以是需要的任意复杂数据结构：

```
var myFunc = function (param) {
    if (!myFunc.cache[param]) {
        var result = {};
        // ... 开销很大的操作 ...
        myFunc.cache[param] = result;
    }
    return myFunc.cache[param];
};

// 缓存存储
myFunc.cache = {};
```

上面的代码假定该函数只需要一个参数param，并且它是一个基本数据类型(比如字符串类型)。如果有更多以及更复杂的参数，对此的通用解决方案是将它们序列化。例如，可以将参数对象序列化为一个JSON字符串，并使用该字符串作为cache对象的键：

```
var myFunc = function () {

    var cachekey = JSON.stringify(Array.prototype.slice.call(arguments)),
        result;

    if (!myFunc.cache[cachekey]) {
        result = {};
        // ... 开销很大的操作 ...
        myFunc.cache[cachekey] = result;
    }
    return myFunc.cache[cachekey];
};

// 缓存存储
myFunc.cache = {};
```

请注意在序列化过程中，对象的“标识”将会丢失。如果有两个不同的对象并且恰好都具有相同的属性，这两个对象将会共享同一个缓存条目。

编写前面的函数的另一种方法是使用arguments.callee来引用该函数，而不是硬编码函数名称。虽然在目前这是可能的，要认识到在ECMAScript 5的严格模式中并不支持arguments.callee。

```
var myFunc = function (param) {

    var f = arguments.callee,
        result;

    if (!f.cache[param]) {
        result = {};
        // ... 开销很大的操作 ...
        f.cache[param] = result;
    }
    return f.cache[param];
};

// 缓存存储
myFunc.cache = {};
```

配置对象

配置对象模式（configuration object pattern）是一种提供更整洁的API的方法，尤其是在建立一个库或任何将被其他程序使用的代码的情况。

随着软件开发和维护的推进，软件需求也发生着变化，这是开发过程中的事实。这种情况经常发生在仅以头脑记住的一些需求就开始工作时，不过后来将会添加更多的功能。

想象一下，正在编写一个名为addPerson()的函数，该函数接受人员的名和姓参数，并将该人员添加到列表中：

```
function addPerson(first, last) {...}
```

后来，了解到实际上还需要存储人员的出生日期，以及可选的性别和住址等信息。因此，可以修改该函数并添加新的参数(注意将可选参数放置在列表的末尾)：

```
function addPerson(first, last, dob, gender, address) {...}
```

在这一点上，该函数的参数列表就变得有一点长。然后，知道需要添加一个用户名，并且这是绝对必要的而非可选的。现在函数的调用者将必须传递参数，即使是可选参数也要传递，同时还要注意不要混淆了参数的顺序。

```
addPerson("Bruce", "Wayne", new Date(), null, null, "batman");
```

使用时需要传递大量的参数并不是很方便。一个更好的办法是仅使用一个参数对象来替代所有参数，让我们将该参数对象称为conf，也就是“配置”的意思：

```
addPerson(conf);
```

然后，该函数的使用者可以这样做：

```
var conf = {
    username: "batman",
    first: "Bruce",
    last: "Wayne"
};
addPerson(conf);
```

配置对象的优点在于：

- 不需要记住众多的参数以及其顺序。
- 可以安全忽略可选参数。
- 更加易于阅读和维护。
- 更加易于添加和删除参数。

而配置对象的不利之处在于：

- 需要记住参数名称。
- 属性名称无法被压缩。

当函数创建DOM元素时，这种模式可能是非常有用的，例如，可以用在设置元素的CSS样式中，因为元素和样式可能具有大量的可选特征和属性。

Curry

本章剩下的部分主要讨论有关Curry化和部分函数应用的主题。但是在深入讨论该主题之前，让我们首先看看函数应用准确的含义。

函数应用

在一些纯粹的函数式编程语言中，函数并不描述为被调用（即called或invoked），而是描述为应用（applied）。在JavaScript中，我们可以做同样的事情，使用方法`Function.prototype.apply()`来应用函数，这是由于JavaScript中的函数实际上是对象，并且它们还具有如下方法。

下面是一个函数应用的示例：

```
// 定义函数
var sayHi = function (who) {
    return "Hello" + (who ? ", " + who : "") + "!";
};

// 调用函数
```

```
sayHi(); // 输出"Hello"
sayHi('world'); // 输出"Hello, world!"

// 应用函数
sayHi.apply(null, ["hello"]); // 输出"Hello, hello!"
```

正如从上面的例子中所看到的，调用（invoking）函数和应用（applying）函数可以得到完全相同的结果。apply()带有两个参数：第一个参数为将要绑定到该函数内部this的一个对象，而第二个参数是一个数组或多个参数变量，这些参数将变成可用于该函数内部的类似数组的arguments对象。如果第一个参数为null（空），那么this将指向全局对象，此时得到的结果就恰好如同当调用一个非指定对象时的方法。

当函数是一个对象的方法时，此时不能传递null引用（如同前面的例子一样）。在这种情况下，这里的对象将成为apply()的第一个参数：

```
var alien = {
    sayHi: function (who) {
        return "Hello" + (who ? ", " + who : "") + "!";
    }
};

alien.sayHi('world'); // 输出"Hello, world!"
sayHi.apply(alien, ["humans"]); // 输出"Hello, humans!"
```

在上面的代码片段中，sayHi()内部的this指向了alien对象。而在上一个例子中，this指向了全局对象。

正如上面的两个例子所展示的那样，这些都表明我们考虑的调用函数并不只是“句法糖（syntactic sugar）”，而是等价于函数应用。

请注意，除了apply()以外， Function.prototype对象还有一个call()方法，但是这仍然只是建立在apply()之上的“语法糖（syntax sugar）”而已。有时候最好使用该“语法糖”：即当函数仅带有一个参数时，可以根据实际情况避免创建只有一个元素的数组的工作。

```
// 第二种更有效率，节省了一个数组
sayHi.apply(alien, ["humans"]); // 输出"Hello, humans!"
sayHi.call(alien, "humans"); // 输出"Hello, humans!"
```

部分应用

现在，我们已经知道调用函数实际上就是将一个参数集合应用到一个函数中，那么有没有可能只传递部分参数，而不是所有的参数？而这种情况就与手动处理一个数学函数所常采用的方法是相似的。

假定有一个函数add()用以将两个数字加在一起：x和y。下面的代码段展示了在给定x值为5，且y值为4的情况下的解决方案。

```
// 出于演示的目的
// 并不是合法的JavaScript

function add(x, y) {
    return x + y;
}

// 有以下函数
add(5, 4);

// 第1步，替换一个参数
function add(5, y) {
    return 5 + y;
}

// 第2步，替换其他参数
function add(5, 4) {
    return 5 + 4;
}
```

在上面的代码片段中，步骤1和步骤2并不是合法的JavaScript，但这演示了如何手工解决部分函数应用的问题。可以获取第一个参数的值，并且在整个函数中用已知的值5替代未知的x，然后重复同样的步骤直至用完了所有的参数。

对这个例子中的步骤1可以称为部分应用（partial application），即我们仅应用了第一个参数。当执行部分应用时，并不会获得结果（解答），相反会获得另一个函数。

下面的代码片段演示了假想的partialApply()方法的使用示例：

```
var add = function (x, y) {
    return x + y;
};

// 完全应用
add.apply(null, [5, 4]); // 9

// 部分应用
var newadd = add.partialApply(null, [5]);
// 应用一个参数到新函数中
newadd.apply(null, [4]); // 9
```

如上面的代码所示，部分应用向我们提供了另一个新函数，随后再以其他参数调用该函数。这种运行方式实际上与add(5)(4)有一些类似，这是由于add(5)返回了一个可在后来用(4)来调用的函数。此外，我们所熟悉的add(5, 4)调用方式可能并不像是“句法糖(syntactic sugar)”，相反，使用add(5)(4)才像是“句法糖”。

现在，返回到现实，JavaScript中并没有partialApply()方法和函数，默认情况下也并不会表现出与上面类似的行为。但是可以构造出这些函数，因为JavaScript的动态性足够支持这种行为。

使函数理解并处理部分应用的过程就称为Curry过程（Currying）。

Curry化

这里的Curry与辛辣的印度菜是没有任何关系的，它来源于数学家Haskell Curry的名字（Haskell编程语言也是以他的名字命名）。Curry化是一个转换过程，即我们执行函数转换的过程。Curry化的另一个替代名称为schönfinkelisation，它是以另一个数学家Moses Schönfinkel命名的，他是这种转换的最早发明人。

那么应该如何schönfinkel化（或schönfinkelize以及curry）一个函数？其他的函数式语言可能已经将这种Curry化转换构建到语言本身中，并且所有的函数已经默认转换过。在JavaScript中，可以将add()函数修改成一个用于处理部分应用的Curry化函数。

下面，让我们举一个例子：

```
// curry化的add()函数
// 接受部分参数列表
function add(x, y) {
    var oldx = x, oldy = y;
    if (typeof oldy === "undefined") { // 部分
        return function (newy) {
            return oldx + newy;
        };
    }
    // 完全应用
    return x + y;
}

// 测试
typeof add(5); // 输出"function"
add(3)(4); // 7
// 创建并存储一个新函数
var add2000 = add(2000);
add2000(10); // 输出2010
```

在上面的代码段中，当第一次调用add()时，它为返回的内部函数创建了一个闭包。该闭包将原始的x和y值存储到私有变量oldx和oldy中。第一个私有变量oldx将在内部函数执行的时候使用。如果没有部分应用，并且同时传递x和y值，该函数则继续执行，并简单地将其相加。这种add()实现与实际需求相比显得比较冗长，在这里只是出于演示的目的才这样实现。下一个代码段中将显示一个更为精简的实现版本，其中并没有oldx和oldy，

仅是因为原始x隐式地存储在闭包中，并且还将y作为局部变量复用，而不是像前面的例子那样创建一个新的变量newy：

```
// curry化的add()函数
// 接受部分参数列表
function add(x, y) {
    if (typeof y === "undefined") { // 部分
        return function (y) {
            return x + y;
        };
    }
    // 完全应用
    return x + y;
}
```

在这些例子中，函数add()本身负责处理部分应用。但是能够以更通用的方式执行相同的任务吗？也就是说，是否可以将任意的函数转换成一个新的可以接受部分参数的函数？接下来的代码片段展示了一个通用函数的例子，暂且称为schonfinkelize()，该函数执行了函数转换。使用名称schonfinkelize()，一部分原因在于这个单词的发音比较困难，而另一部分原因在于听起来像一个动词(使用“curry”可能会引起歧义)，此外也需要一个动词来表示这是一个函数的转换。

下面是一个通用curry化函数的示例：

```
function schonfinkelize(fn) {
    var slice = Array.prototype.slice,
        stored_args = slice.call(arguments, 1);
    return function () {
        var new_args = slice.call(arguments),
            args = stored_args.concat(new_args);
        return fn.apply(null, args);
    };
}
```

schonfinkelize()函数可能不应该有那么复杂，只有由于JavaScript中arguments并不是一个真实的数组。从Array.prototype中借用slice()方法可以帮助我们将arguments变成一个数组，并且使用该数组工作更加方便。当schonfinkelize()第一次调用时，它存储了一个指向slice()方法的私有引用（名为slice），并且还存储了调用该方法后的参数（存入stored_args中），该方法仅剥离了第一个参数，这是因为第一个参数是将被curry化的函数。然后，schonfinkelize()返回了一个新函数。当这个新函数被调用时，它访问了已经私有存储的参数stored_args以及slice引用。这个新函数必须将原有的部分应用参数（stored_args）合并到新参数（new_args），然后再将它们应用到原始函数fn中（也仅在闭包中私有可用）。

现在，已经准备好可将任意函数curry的通用方法，让我们尝试执行以下一些测试：

```
// 普通函数
function add(x, y) {
    return x + y;
}

// 将一个函数curry化以获得一个新的函数
var newadd = schonfinkelize(add, 5);
newadd(4); // 输出9

// 另一种选择——直接调用新函数
schonfinkelize(add, 6)(7); // 输出13
```

转换函数schonfinkelize()并不局限于单个参数或者单步Curry化。下面是一些更多的用法示例：

```
// 普通函数
function add(a, b, c, d, e) {
    return a + b + c + d + e;
}

// 可运行于任意数量的参数
schonfinkelize(add, 1, 2, 3)(5, 5); // 16

// 两步curry化
var addOne = schonfinkelize(add, 1);
addOne(10, 10, 10, 10); // 输出41
var addSix = schonfinkelize(addOne, 2, 3);
addSix(5, 5); // 输出16
```

何时使用Curry化

当发现正在调用同一个函数，并且传递的参数绝大多数都是相同的，那么该函数可能是用于Curry化的一个很好的候选参数。可以通过将一个函数集合部分应用（partially applyi）到函数中，从而动态创建一个新函数。这个新函数将会保存重复的参数(因此，不必每次都传递这些参数)，并且还会使用预填充原始函数所期望的完整参数列表。

小结

在Javascript中，有关函数的知识以及函数的正确用法是至关重要的。本章讨论了有关函数的背景和术语。学习了JavaScript中函数的两个重要特征，即：

1. 函数是第一类对象（first-class object），可以作为带有属性和方法的值以及参数进行传递。

2. 函数提供了局部作用域，而其他大括号并不能提供这种局部作用域。此外还需要记住的是，声明的局部变量可被提升到局部作用域的顶部。

创建函数的语法包括：

1. 命名函数表达式。
2. 函数表达式(与上面的相同，但缺少一个名字)，通常也称为匿名函数。
3. 函数声明，与其他语言中的函数的语法类似。

在涵盖了函数的背景和语法之后，将学习到一些有用的模式，可以将它们分为以下几类：

1. API模式，它们可以帮助您为函数提供更好且更整洁的接口。这些模式包括以下几个：

   回调模式

   将函数作为参数进行传递。

   配置对象

   有助于保持受到控制的函数的参数数量。

   返回函数

   当一个函数的返回值时另一个函数时。

   Curry化

   当新函数是基于现有函数，并加上部分参数列表创建时。

2. 初始化模式，它们可以帮助您在不污染全局命名空间的情况下，使用临时变量以一种更加整洁、结构化的方式执行初始化以及设置任务（当涉及Web网页和应用程序时是非常普遍的）。这些模式包括：

   即时函数

   只要定义之后就立即执行。

   即时对象初始化

   匿名对象组织了初始化任务，提供了可被立即调用的方法。

   初始化时分支

   帮助分支代码在初始化代码执行过程中仅检测一次，这与以后在程序生命期内多次检测相反。

3. 性能模式，可以帮助加速代码运行，这些模式包括：

备忘模式

使用函数属性以便使得计算过的值无须再次计算。

自定义模式

以新的主体重写本身，以使得在第二次或以后调用时仅需执行更少的工作。

第5章

对象创建模式

在JavaScript中创建对象是很容易的，可以使用对象字面量（object literal）或者使用构造函数。在本章中，我们越过那些方法以寻求一些额外的对象创建模式。

JavaScript是一种简洁明了的语言，其中并没有在其他语言中经常使用的一些特殊语法特征，比如命名空间（namespace）、模块（module）、包（package）、私有属性（private property），以及静态成员等语法。本章带您通过一些常见的模式以实现、替换那些语法特征，或者仅以不同于那些语法特征的方式来思考问题。

本章中我们将看到命名空间（namespace）、依赖声明（dependency declaration）、模块模式（module pattern）、沙箱模式（sandbox pattern）。它们都可以帮助您组织应用程序代码的结构，并且降低隐含的全局变量带来的后果。其他讨论的主题包括私有和特权成员、对象常量、链和一个启发类的方式以定义构造函数。

命名空间模式

命名空间（namespace）有助于减少程序中所需要的全局变量的数量，并且同时还有助于避免命名冲突或过长的名字前缀。

JavaScript语言的语法中并没有内置命名空间，但是这种特征是非常容易实现的。可以为应用程序或库创建一个（理想上最好只有一个）全局对象，然后可以将所有功能添加到该全局对象中，从而在具有大量函数、对象和其他变量的情况下并不会污染全局范围。

请考虑下面的例子：

```
// 之前：5个全局变量
```

```
// 警告：反模块

// 构造函数
function Parent() {}
function Child() {}

// 一个变量
var some_var = 1;

// 一些对象
var module1 = {};
module1.data = {a: 1, b: 2};
var module2 = {};
```

可以通过为应用程序创建一个全局对象这种方式来重构上面这种类型的代码，比如创建全局对象MYAPP，然后改变所有函数和变量以使其成为您的全局对象的属性。

```
// 之后：1个全局变量

// 全局变量
var MYAPP = {};

// 构造函数
MYAPP.Parent = function () {};
MYAPP.Child = function () {};

// 一个变量
MYAPP.some_var = 1;

// 一个对象容器
MYAPP.modules = {};

// 嵌套对象
MYAPP.modules.module1 = {};
MYAPP.modules.module1.data = {a: 1, b: 2};
MYAPP.modules.module2 = {};
```

对于全局命名空间对象的名称，可以任意选择，比如以应用程序或库的名称、域名或公司名来命令。通常程序员都会根据公约以全部大写的方式来命令全局变量，因此对于代码阅读者来说这些全局变量是非常引人注目的（但是请记住，全部大写也常用于常量的命名）。

这种模式是一种组织代码的命名空间的好方法，不仅可以避免您代码中的命名冲突，并且还可以避免在同一个页面中您的代码和第三方代码之间的命名冲突，比如与JavaScript库和窗口（widget）的冲突。强烈推荐使用这种模式，并且可完美地应用于许多任务中，但是它也确实有一些缺点：

- 需要输入更多的字符，每个变量和函数前都要附加前缀，总体上增加了需要下载的代码量。

- 仅有一个全局实例意味着任何部分的代码都可以修改该全局实例，并且其余的功能够获得更新后的状态。
- 长嵌套名字意味着更长（更慢）的属性解析查询时间。

在本章后面所讨论的沙箱模式（sandbox pattern）可以解决以上这些缺点。

通用命名空间函数

由于程序复杂性的增加、代码的某些部分被分割成不同的文件，以及使用条件包含语句等多个因素，仅假设您的代码是第一个定义某个命名空间或它内部的一个属性，这种做法现在已经变得不再安全。添加到命名空间的一些属性可能已经存在，这导致可能会覆盖它们。因此，在添加一个属性或者创建一个命名空间之前，最好是首先检查它是否已经存在，如下例子所示：

```
// 不安全的代码
var MYAPP = {};
// 更好的代码风格
if (typeof MYAPP === "undefined") {
    var MYAPP = {};
}
// 或者用更短的语句
var MYAPP = MYAPP || {};
```

可以看到这些附加的检查是如何迅速导致大量的重复代码。例如，如果想定义MYAPP.modules.module2，必须构造三次检查，每次检查都要针对定义的一个对象或者属性。这也就是为什么需要一个可以很方便地处理命名空间细节的可重用函数的原因。让我们称该函数为namespace()并以如下方式使用：

```
// 使用命名空间函数
MYAPP.namespace('MYAPP.modules.module2');

// 相当于以下代码
// var MYAPP = {
//     modules: {
//         module2: {}
//     }
// };
```

接下来是一个命名空间函数的实现示例。这个实现是非破坏性的，也就是说，如果已经存在一个命名空间，便不会再重新创建它：

```
var MYAPP = MYAPP || {};

MYAPP.namespace = function (ns_string) {
    var parts = ns_string.split('.'),
```

```
            parent = MYAPP,
            i;

        // 剥离最前面的冗余全局变量
        if (parts[0] === "MYAPP") {
            parts = parts.slice(1);
        }

        for (i = 0; i < parts.length; i += 1) {
            // 如果它不存在，就创建一个属性
            if (typeof parent[parts[i]] === "undefined") {
                parent[parts[i]] = {};
            }
            parent = parent[parts[i]];
        }
        return parent;
    };
```

本实现使所有下列这些用法都能正常运行：

```
// 将返回值赋给一个局部变量
var module2 = MYAPP.namespace('MYAPP.modules.module2');
module2 === MYAPP.modules.module2; // true

// 忽略最前面的`MYAPP`
MYAPP.namespace('modules.module51');

// 长命名空间
MYAPP.namespace('once.upon.a.time.there.was.this.long.nested.property');
```

图5-1显示了在Firebug中观察前面例子中的命名空间的样式：

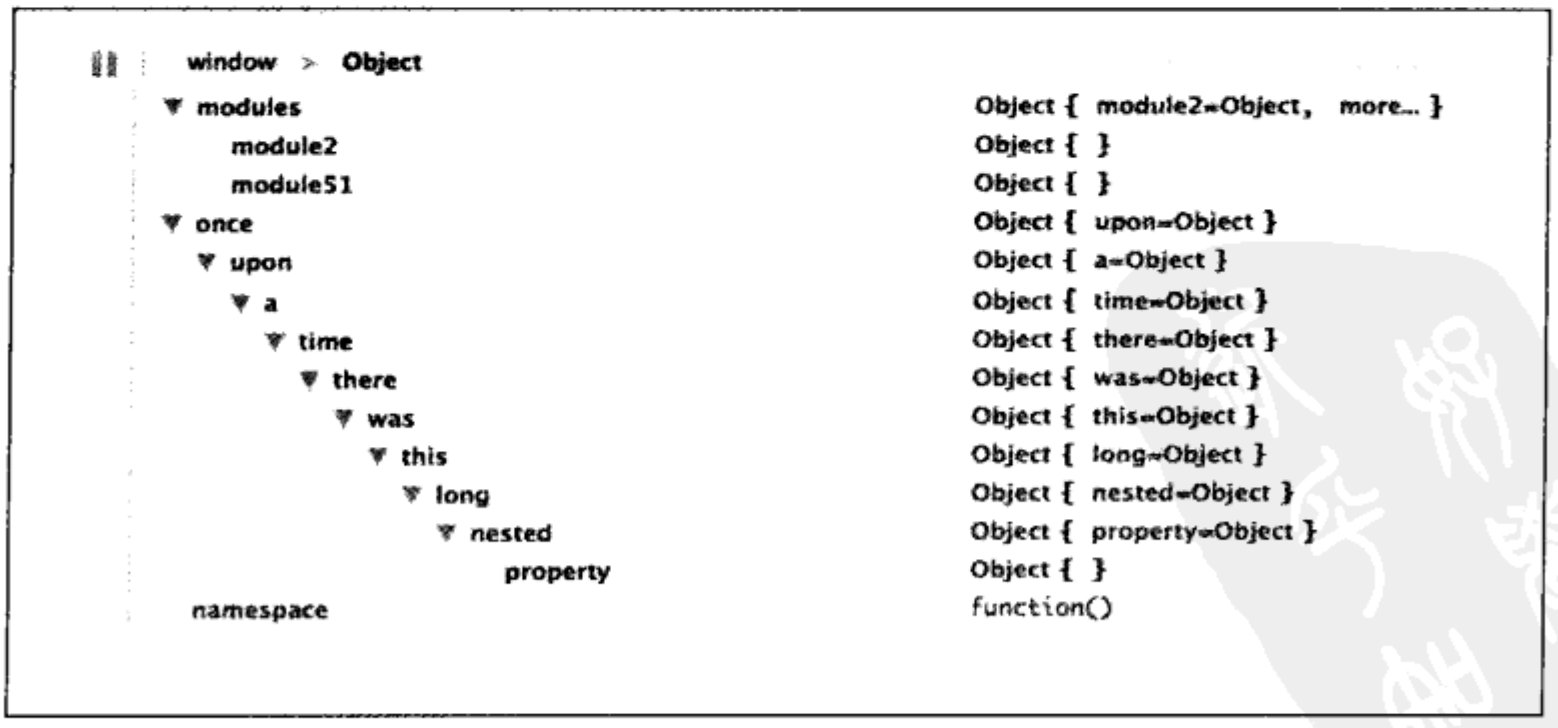

图5-1：在Firebug中观察MYAPP命名空间

声明依赖关系

JavaScript库通常是模块化且依据命名空间组织的，这使您能够仅包含所需的模块。例如，在YUI2库中有一个充当命名空间的全局变量YAHOO，而模块是该全局变量的属性，比如YAHOO.util.Dom（DOM模块）和YAHOO.util.Event（事件模块）。

在您的函数或模块顶部声明代码所依赖的模块是一个非常好的主意。该声明仅涉及创建一个局部变量并使其指向所需的模块。

```
var myFunction = function () {
    // 依赖
    var event = YAHOO.util.Event,
        dom = YAHOO.util.Dom;

    // 使用事件和DOM变量
    // 下面的函数……
};
```

这是一个极其简单的模式，但同时它却有许多优点：

- 显式的依赖声明向您代码的用户表明了他们确定需要的特定脚本文件已经包含在该页面中。
- 在函数顶部的前期声明可以使您很容易地发现并解析依赖。
- 解析局部变量（比如DOM）的速度总是要比解析全局变量（比如YAHOO）要快，甚至比使用全局变量的嵌套属性（比如YAHOO.util.Dom）还要快，这导致了更好的性能。当使用这种依赖声明模式时，全局符号解析仅会在函数中执行一次。在此之后将会使用局部变量，这种解析速度也快得多。
- 类似于YUICompressor和Google闭包编译器的这些高级小工具可以重命名局部变量（因此，event有可能变成一个字符，比如A），这导致了更小的代码量，但是这些工具从不会对全局变量进行重命名，因为这样做是不安全的。

下面的代码片段演示了在缩小后的代码添加依赖声明模式的效果。虽然test2()遵循了该模式，它看起来有点更为复杂，这是因为它需要更多的代码行以及额外的变量，但实际上它在缩小代码后导致了更少的代码量，这意味着用户仅需下载更少的代码。

```
function test1() {
    alert(MYAPP.modules.m1);
    alert(MYAPP.modules.m2);
    alert(MYAPP.modules.m51);
}

/*
缩减的 test1 主体:
```

```
alert(MYAPP.modules.m1);alert(MYAPP.modules.m2);alert(MYAPP.modules.m51)
*/

function test2() {
    var modules = MYAPP.modules;
    alert(modules.m1);
    alert(modules.m2);
    alert(modules.m51);
}

/*
缩减的 test2 主体:
var a=MYAPP.modules;alert(a.m1);alert(a.m2);alert(a.m51)
*/
```

私有属性和方法

JavaScript并没有特殊的语法来表示私有、保护、或公共属性和方法，在这一点上与Java或其他语言是不同的。JavaScript中所有对象的成员是公共的：

```
var myobj = {
    myprop: 1,
    getProp: function () {
        return this.myprop;
    }
};
console.log(myobj.myprop); // `myprop` 是公有可访问的
console.log(myobj.getProp()); // getProp() 也是公有可访问的
```

当使用构造函数创建对象时也同样如此，即所有的成员仍然都是公共的：

```
function Gadget() {
    this.name = 'iPod';
    this.stretch = function () {
        return 'iPad';
    };
}
var toy = new Gadget();
console.log(toy.name); // `name` 是公有的
console.log(toy.stretch()); // stretch() 是公有的
```

私有成员

虽然JavaScript语言中并没有用于私有成员的特殊语法，但是可以使用闭包（closure）来实现这种功能。构造函数创建了一个闭包，而在闭包范围内部的任意变量都不会暴露给构造函数以外的代码。然而，这些私有变量仍然可以用于公共方法中：即定义在构造函数中，且作为返回对象的一个部分暴露给外部的方法。让我们看一个例子，其中name是一个私有成员，在构造函数外部是不可访问的：

```
function Gadget() {
    //私有成员
    var name = 'iPod';
    // 公有函数
    this.getName = function () {
        return name;
    };
}
var toy = new Gadget();

// `name`是undefined的，它是私有的
console.log(toy.name); // 输出undefined
// 公有方法访问 `name`
console.log(toy.getName()); // 输出"iPod"
```

正如所看到的，很容易在JavaScript实现私有性。需要做的只是在函数中将需要保持为私有属性的数据包装起来，并确保它对函数来说是局部变量，这意味着外部函数不能访问它。

特权方法

特权方法（Privileged Method）的概念并不涉及任何特殊语法，它只是指那些可以访问私有成员的公共方法（因此它拥有更多的特权）的一个名称而已。

在前面的例子中，getName()就是一个特权的方法，因为它具有访问私有属性name的“特殊”权限。

私有性失效

当关注私有性的时候会出现一些边缘情况：

- Firefox的一些早期版本可以将第二个参数传递给eval()，而那是一个上、下文（context）对象，可以使您潜入函数的私有作用域。Mozilla Rhino中的__parent__属性也与此相似，它也能够使您访问函数的私有作用域。这些边缘情况并不适用于如今广泛使用的浏览器。
- 当直接从一个特权方法中返回私有一个变量，且该变量恰好是一个对象或者数组，那么外面的代码仍然可以访问该私有变量，这是因为它通过引用传递的。

让我们来更加仔细的看看第二种情况。 以下Gadget的实现看起来就像是无意造成失效的：

```
function Gadget() {
    // 私有成员
    var specs = {
```

```
            screen_width:  320,
            screen_height: 480,
            color: "white"
        };

        // 公有函数
        this.getSpecs = function () {
            return specs;
        };
    }
```

这里的问题是在于getSpecs()返回了一个引用的specs对象。这使得Gadget的用户可以修改表面上看来是隐藏和私有的specs对象：

```
var toy = new Gadget(),
    specs = toy.getSpecs();
specs.color = "black";
specs.price = "free";

console.dir(toy.getSpecs());
```

打印到Firebug控制台中的结果如图5-2所示。

color	"black"
price	"free"
screen_height	480
screen_width	320

图5-2：在Firebug中观察MYAPP命名空间

对这种意外行为的解决方法是保持细心，即不要传递需要保持私有性的对象和数组的引用。解决这个问题的一种方法是，使getSpecs()返回一个新对象，而该对象仅包含客户关注的原对象中的数据。这也是众所周知的最低授权原则（Principle of Least Authority，POLA），其中规定了应该永远不要给予超过需要的特权。在这种情况下，如果Gadget的消费者仅关注该gadget组件是否与一个特定方框的尺寸相符合，那么它需要的仅是尺寸规格。因此，并不需要分发所有的数据，可以创建getDimensions()使其仅返回一个包含宽度和高度的新对象。此时，可能根本不需要实现getSpecs()。

当需要传递所有的数据时，另外一种解决方法使用一个通用性的对象克隆（object-cloning）函数以创建specs对象的副本。下一章提供了两个这样的函数，其中一个名为extend()，它可以针对给定对象创建一个浅复制（shallow copy）副本（仅复制顶级单数）。而另一个函数名为extendDeep()，它可以通过递归复制所有的属性以及其嵌套属性而创建深度复制（deep copy）副本。

对象字面量以及私有性

到目前为止，我们仅看到了使用构造函数获得私有性的例子。但是当使用对象字面量（object literal）来创建对象会是什么情况呢？它是否还有可能拥有私有成员？

正如在前面所看到的，需要的只是一个能够包装私有数据的函数。因此，在使用对象字面量的情况下，可以使用一个额外的匿名即时函数（anonymous immediate function）创建闭包来实现私有性。如下面的例子所示：

```
var myobj; // 这将会是对象
(function () {
    // 私有成员
    var name = "my, oh my";

    // 实现公有部分
    // 注意，没有`var`修饰符
    myobj = {
        // 特权方法
        getName: function () {
            return name;
        }
    };
}());

myobj.getName(); // 输出"my, oh my"
```

下面的例子与上面的具有同样的思想，但是在实现上略有不同：

```
var myobj = (function () {
    // 私有成员
    var name = "my, oh my";

    // 实现公有部分
    return {
        getName: function () {
            return name;
        }
    };
}());

myobj.getName(); //输出"my, oh my"
```

这个例子也是称之为“模块模式”（module pattern）的基础框架，我们将在后面对此进行研究。

原型和私有性

当将私有成员与构造函数一起使用时，其中的一个缺点在于每次调用构造函数以创建对象时，这些私有成员都会被重新创建。

构造函数中添加到this中的任何成员都实际上面临以上问题。为了避免复制工作以及节省内存，可以将常用属性和方法添加到构造函数的prototype属性中。这样，通过同一个构造函数创建的多个实例可以共享常见的部分数据。此外，还可以在多个实例中共享隐藏的私有成员。为了实现这一点，可以使用以下两个模式的组合：即构造函数中的私有属性以及对象字面量中的私有属性。由于prototype属性仅是一个对象，因此可以使用对象字面量创建该对象。

下面的例子展示了如何实现以上目标：

```
function Gadget() {
    // 私有成员
    var name = 'iPod';
    // 公有函数
    this.getName = function () {
        return name;
    };
}

Gadget.prototype = (function () {
    // 私有成员
    var browser = "Mobile Webkit";
    // 公有原型成员
    return {
        getBrowser: function () {
            return browser;
        }
    };
}());

var toy = new Gadget();
console.log(toy.getName()); // 特权 "own" 方法
console.log(toy.getBrowser()); // 特权原型方法
```

将私有方法揭示为公共方法

揭示模式（revelation pattern）可用于将私有方法暴露成为公共方法。当为了对象的运转而将所有功能放置在一个对象中以及想尽可能地保护该对象是至关重要的时候，这种揭示模式就显得非常有用。不过，同时可能也想为其中的一些功能提供公共可访问的接口，因为那可能也是有用的。当将这些私有方法暴露为公共方法时，也使它们变得更为脆弱。因为使用公共API的一些用户可能会修改原对象，甚至是无意的修改。在ECMAScript 5中，可以选择将一个对象冻结，但是在前一版本的语言中是不具备该功能的。现在开始讨论揭示模式［该术语是由Christian Heilmann所创造的，其原句是“揭示模块模式”（revealing module pattern）］。

让我们举个例子，它建立在其中一种私有模式之上，即对象字面量中的私有成员。

```
var myarray;

(function () {

    var astr = "[object Array]",
        toString = Object.prototype.toString;

    function isArray(a) {
        return toString.call(a) === astr;
    }

    function indexOf(haystack, needle) {
        var i = 0,
            max = haystack.length;
        for (; i < max; i += 1) {
            if (haystack[i] === needle) {
                return i;
            }
        }
        return -1;
    }

    myarray = {
        isArray: isArray,
        indexOf: indexOf,
        inArray: indexOf
    };

}());
```

在以上例子中，有两个私有变量以及两个私有函数，isArray()和indexOf()。在匿名函数（immediate function）的最后，对象myarray中填充了认为适于公共访问的功能。在这种情况下，同一个私有函数indexOf()可以暴露为ECMAScript 5风格的indexOf以及PHP范式的inArray。下面测试新的myarray对象：

```
myarray.isArray([1,2]); // true
myarray.isArray({0: 1}); // false
myarray.indexOf(["a", "b", "z"], "z"); // 2
myarray.inArray(["a", "b", "z"], "z"); // 2
```

现在，如果发生了意外的事情，例如公共indexOf()方法发生意外，但私有indexOf()方法仍然是安全的，因此inArray()将继续正常运行：

```
myarray.indexOf = null;
myarray.inArray(["a", "b", "z"], "z"); // 2
```

模块模式

目前模块模式（Module Pattern）得到了广泛使用，因为它提供了结构化的思想并且有助于组织日益增长的代码。与其他语言不同的是，JavaScript并没有包（package）的特殊

语法，但是模块模式提供了一种创建自包含非耦合（self-contained de-coupled）代码片段的有利工具，可以将它视为黑盒功能，并且可以根据您所编写软件的需求（千变万化的需求）添加、替换或删除这些模块。

模块模式是本书中迄今为止介绍过的多种模式的组合，也就是以下模式的组合：

- 命名空间。
- 即时函数。
- 私有和特权成员。
- 声明依赖。

该模式的第一步是建立一个命名空间。让我们使用本章前面介绍的namespace()函数，并且启动可以提供有用数组方法的工具模块。

```
MYAPP.namespace('MYAPP.utilities.array');
```

下一步是定义该模块。对于需要保持私有性的情况，本模式则使用了一个可以提供私有作用域的即时函数。该即时函数返回了一个对象，即具有公共接口的实际模块，可以通过这些接口来使用这些模块。

```
MYAPP.utilities.array = (function () {
    return {
        // todo...
    };
}());
```

接下来，让我们向该公共接口添加一些方法：

```
MYAPP.utilities.array = (function () {
    return {
        inArray: function (needle, haystack) {
            // ...
        },
        isArray: function (a) {
            // ...
        }
    };
}());
```

通过使用由即时函数提供的私有作用域，可以根据需要声明一些私有属性和方法。在即时函数的顶部，正好也就是声明模块可能有任何依赖的位置。在变量声明之后，可以任意地放置有助于建立该模块的任何一次性的初始化代码。最终结果是一个由即时函数返回的对象，其中该对象包含了您模块的公共API。

```
MYAPP.namespace('MYAPP.utilities.array');
```

```
MYAPP.utilities.array = (function () {

        // 依赖
    var uobj  = MYAPP.utilities.object,
        ulang = MYAPP.utilities.lang,

        // 私有属性
        array_string = "[object Array]",
        ops = Object.prototype.toString;

        // 私有方法
        // ...

        // var变量定义结束

    // 可选的一次性初始化过程
    // ...

    // 公有 API
    return {

        inArray: function (needle, haystack) {
            for (var i = 0, max = haystack.length; i < max; i += 1) {
                if (haystack[i] === needle) {

                    return true;
                }
            }
        },
        isArray: function (a) {
            return ops.call(a) === array_string;
        }
        // ... 更多方法和属性
    };
}());
```

模块模式得到了广泛使用，并且强烈建议使用这种方式组织您的代码，尤其是当代码日益增长的时候。

揭示模块模式

我们在本章中已经讨论了揭示模式（revelation pattern），同时还考虑了私有模式（privacy pattern）。模块模式也可以组织成与之相似的方式，其中所有的方法都需要保持私有性，并且只能暴露那些最后决定设立API的那些方法。

根据以上思想，其代码如下所示：

```
MYAPP.utilities.array = (function () {

        // 私有属性
    var array_string = "[object Array]",
```

```
        ops = Object.prototype.toString,

        // 私有方法
        inArray = function (haystack, needle) {
            for (var i = 0, max = haystack.length; i < max; i += 1) {
                if (haystack[i] === needle) {
                    return i;
                }
            }
            return -1;
        },
        isArray = function (a) {
            return ops.call(a) === array_string;
        };
        //  var变量定义结束

    // 揭示公有API
    return {
        isArray: isArray,
        indexOf: inArray
    };
}());
```

创建构造函数的模块

前面的例子中创建了一个对象MYAPP.utilities.array，但有时候使用构造函数创建对象更为方便。当然，可以仍然使用模块模式来执行创建对象的操作。它们之间唯一的区别在于包装了模块的即时函数最终将会返回一个函数，而不是返回一个对象。

考虑以下使用模块模式的例子，在该例子中创建了一个构造函数MYAPP.utilities.Array：

```
MYAPP.namespace('MYAPP.utilities.Array');

MYAPP.utilities.Array = (function () {

        // 依赖
    var uobj  = MYAPP.utilities.object,
        ulang = MYAPP.utilities.lang,

        // 私有属性和方法
        Constr;

        // var 变量定义结束

    // 可选的一次性初始化过程
    // ...

    // 公有API——构造函数
    Constr = function (o) {
        this.elements = this.toArray(o);
    };
    // 公有 API——原型
```

```
    Constr.prototype = {
        constructor: MYAPP.utilities.Array,
        version: "2.0",
        toArray: function (obj) {
            for (var i = 0, a = [], len = obj.length; i < len; i += 1) {
                a[i] = obj[i];
            }
            return a;
        }
    };

    // 返回要分配给新命名空间的构造函数
    return Constr;
}());
```

使用这个新构造函数的方法如下所示：

```
var arr = new MYAPP.utilities.Array(obj);
```

将全局变量导入到模块中

在常见的变化模式中，可以将参数传递到包装了模块的即时函数中。可以传递任何值，但是通常这些都是对全局变量、甚至是全局对象本身的引用。导入全局变量有助于加速即时函数中的全局符号解析的速度，因为这些导入的变量成为了该函数的局部变量。

```
MYAPP.utilities.module = (function (app, global) {

    // 引用全局对象
    // 以及现在被转换成局部变量的
    // 全局应用程序命名空间对象

}(MYAPP, this));
```

沙箱模式

沙箱模式（sandbox pattern）解决了命名空间模式的如下几个缺点：

- 对单个全局变量的依赖变成了对应用程序的全局变量依赖。在命名空间模式中，是没有办法使同一个应用程序或库的两个版本运行在同一个页面中，因为这两者都需要同一个全局符号名，比如全局变量MYAPP。
- 对这种以点分割的名字来说，需要输入更长的字符，并且在运行时需要解析更长的时间，比如MYAPP.utilities.array。

顾名思义，沙箱模式提供了一个可用于模块运行的环境，且不会对其他模块和个人沙箱造成任何影响。

在YUI的第3版中大量使用了该模式，例如，但请记住，下面的讨论是一个简单的参考实现，并没有试图介绍YUI3的沙箱实现是如何工作的。

全局构造函数

在命名空间模式中，有一个全局对象；在沙箱模式中，则是一个全局构造函数，让我们称之为Sandbox()。可以使用该构造函数创建对象并且还可以传递回调函数，它变成了代码的隔离沙箱运行环境。

使用沙箱的方法如下所示：

```
new Sandbox(function (box) {
    // 你的代码写在这里...
});
```

对象box与命名空间例子中的MYAPP是相似的，它有您所需要的所有库函数，能够使代码正常运行。

让我们向该模式添加两个新的特征：

- 通过一些神奇特征(第3章中的强制new模式)，可以假设在创建对象时不需要new操作符。
- Sandbox()构造函数可以接受一个额外的配置参数（或多个参数），其中该参数指定了对象实例所需要的模块名。我们希望代码是模块化的，因此绝大部分Sandbox()提供的功能将将被限制在模块中。

有了以上这两个额外特征，让我们查看实例化对象的代码看起来是什么样子的。

可以忽略new操作符，并且使用一些如下所示的虚构 “ajax” 和 “event” 模块来创建对象：

```
Sandbox(['ajax', 'event'], function (box) {
    // console.log(box);
});
```

下面的例子与前一个相类似，但下面例子中的模块名是以单个参数的形式传递的：

```
Sandbox('ajax', 'dom', function (box) {
    // console.log(box);
});
```

根据上面例子的启示，认为使用通配符 “*” 参数表示 “使用所有可用模块” 的表示方

法如何？此外，为了方便起见，让我们假设当没有传递任何模块时，沙箱也会将其认定为“*”。因此，根据以上惯例，我们可以使用如下两种方式来使用所有可用模块：

```
Sandbox('*', function (box) {
    // console.log(box);
});

Sandbox(function (box) {
    // console.log(box);
});
```

此外，该模式的另外一个使用样例则演示了如何多次实例化沙箱对象的方法。甚至还可以将一个模块嵌入到另外一个模块中，并且这两者之间不会相互干扰。

```
Sandbox('dom', 'event', function (box) {

    // 使用 DOM和事件来运行

    Sandbox('ajax', function (box) {
       // 另一个沙箱化（sandboxed）的"box"对象
       // 这里的 "box" 对象与函数外部的
       // "box" 并不相同

       //...

       // 用 Ajax来处理
    });

    // 这里没有 Ajax 模块

});
```

从以上这些例子中可以看到，当使用本沙箱模式时，可以通过将代码包装到回调函数中从而保护全局命名空间。

如果需要，也可以利用“函数就是对象”这个事实，然后将数据存储为该Sandbox()构造函数的静态属性。

最后，可以根据所需要的模块类型创建不同的实例，并且这些实例互相独立运行。

现在让我们来看看应该如何着手实现Sandbox()构造函数以及其模块，从而支持所有这些功能。

增加模块

在实现实际的构造函数之前，让我们看看如何才能够增加模块的功能。

Sandbox()构造函数也是一个对象，因此可以向它添加一个名为modules的静态属性。该

属性是包含键－值对（key-value pair）的另一个对象，其中这些键是模块名字，而值则是实现每个模块的对应函数。

```
Sandbox.modules = {};

Sandbox.modules.dom = function (box) {
    box.getElement = function () {};
    box.getStyle = function () {};
    box.foo = "bar";
};

Sandbox.modules.event = function (box) {
    // 如果需要，就访问Sandbox原型，如下语句:
    // box.constructor.prototype.m = "mmm";
    box.attachEvent = function () {};
    box.dettachEvent = function () {};
};

Sandbox.modules.ajax = function (box) {
    box.makeRequest = function () {};
    box.getResponse = function () {};
};
```

在上面这个例子中，我们增加了模块DOM、event，以及ajax，这些都是在库或复杂Web应用中常见的功能片段。

实现每个模块的函数可以接受当前实例box作为参数，并且可以向该实例中添加额外的属性和方法。

实现构造函数

最后，让我们来实现该Sandbox()构造函数（当然，可能希望重命名这种类型的构造函数，以便使得这些名字对于库或者应用程序来说是具有字面意义的）：

```
function Sandbox() {
        // 将参数转换成一个数组
    var args = Array.prototype.slice.call(arguments),
        // 最后一个参数是回调函数
        callback = args.pop(),
        // 模块可以作为一个数组传递，或作为单独的参数传递
        modules = (args[0] && typeof args[0] === "string") ? args : args[0],
        i;

    // 确保该函数
    // 作为构造函数被调用
    if (!(this instanceof Sandbox)) {
        return new Sandbox(modules, callback);
    }

    // 需要向 `this` 添加的属性：
    this.a = 1;
```

```
        this.b = 2;

        // 现在向该核心 `this` 对象添加模块
        // 不指定模块名称或指定"*" 都表示 "使用所有模块"
        if (!modules || modules === '*') {
            modules = [];
            for (i in Sandbox.modules) {
                if (Sandbox.modules.hasOwnProperty(i)) {
                    modules.push(i);
                }
            }
        }

        // 初始化所需的模块
        for (i = 0; i < modules.length; i += 1) {
            Sandbox.modules[modules[i]](this);
        }

        // call the callback
        callback(this);
    }

    // 需要的任何原型属性
    Sandbox.prototype = {
        name: "My Application",
        version: "1.0",
        getName: function () {
            return this.name;
        }
    };
```

在以上代码实现中，其关键部分为：

- 存在一个类型检查语句，检查this是否为Sandbox的实例。如果为否（这表示在没有使用new操作符的情况下调用了Sandbox()），那么我们会再次以构造函数的形式调用该函数。
- 可以在该构造函数中将一些属性添加到this中。此外，还可以将一些属性添加到构造函数的原型中。
- 所需的模块可以模块名称数组的形式传递，或以单个参数的形式传递，还可通过*通配符（或省略）的形式传递，这表示我们应该载入所有可用的模块。请注意，在这个示例实现中，我们并不关心从其他文件中加载所需的功能，但这绝对也是一个可选的实现功能。比如，YUI3库中就支持这种功能。可以仅加载最基本的模块（也称之为“种子”），并且根据与命名公约对应的模块名称，从外部文件中加载任何所需的模块。
- 当我们知道所需的模块时，便可以据此进行初始化，这表示可以调用实现每个模块的函数。

- 该构造函数的最后一个参数是一个回调函数。该回调函数将会在使用新创建的实例时最后被调用。这个回调函数实际上是用户的沙箱，它可以获得一个填充了所需功能的box对象。

静态成员

静态属性和方法也就是那些从一个实例到另外一个实例都不会发生改变的属性和方法。在基于类的语言中，静态成员是通过使用特殊语法而创建的，并且在使用过程中如同类本身的成员一样。例如，对于MathUtils类的静态方法max()，其调用形式为MathUtils.max(3, 5)，这就是一个共有静态成员的例子，而该静态方法max()可以在没有创建任何该类的实例时使用。此外也存在私有静态成员，对于该类的用户而言是不可见的，但是仍然可以在类的多个实例间共享。下面让我们看看如何在JavaScript中实现私有和公有静态成员。

公有静态成员

在JavaScript中并没有特殊语法来表示静态成员。但是可以通过使用构造函数并且向其添加属性这种方式，从而获得与“类式”（classy，即具有class性质）语言相同的语法。这种方式可以良好运行，这是因为构造函数与所有其他函数一样都是对象，并且它们都可以拥有属性。在前面章节中讨论的备忘模式（memoization pattern）也采用了同样的思想，即向函数中添加属性。

下面的例子定义了一个具有静态方法isShiny()的构造函数Gadget，以及一个普通的实例方法setPrice()。其中，isShiny()是一个静态方法，因为该方法并不需要特定的gadget对象就能够运行［就如同不需要特别的gadget以找出所有处于shiny（发光）状态的gadget］。另一方面，setPrice()方法则需要一个对象才能运行，因为gadget可被设定为不同的定价：

```
// 构造函数
var Gadget = function () {};

// 静态方法
Gadget.isShiny = function () {
    return "you bet";
};

// 向该原型添加一个普通方法
Gadget.prototype.setPrice = function (price) {
    this.price = price;
};
```

现在，让我们调用这些方法以进行测试。构造函数中的静态isShiny()方法可被直接调用，然而普通的方法则需要一个实例：

```
// 调用静态方法
Gadget.isShiny(); // 输出"you bet"

// 创建一个实例并调用其方法
var iphone = new Gadget();
iphone.setPrice(500);
```

试图以静态方式调用一个实例方法是无法正常运行的。同样，如果使用实例iphone对象调用静态方法也无法正常运行：

```
typeof Gadget.setPrice; // 输出"undefined"
typeof iphone.isShiny;  // 输出"undefined"
```

有时候也可以很方便地使静态方法与实例一起工作。这种功能比较容易实现，只需要向原型中添加一个新的方法即可，其中该新方法作为一个指向原始静态方法的外观（Façade）：

```
Gadget.prototype.isShiny = Gadget.isShiny;
iphone.isShiny(); // 输出"you bet"
```

在这种情况下，如果在静态方法内部使用this要特别注意。当执行Gadget.isShiny()时，那么isShiny()内部的this将会指向Gadget构造函数。如果执行iphone.isShiny()，那么this将会指向iphone。

最后一个例子向您展示了如何以静态或非静态方式调用同一个方法，而在这两种场景下依赖于调用模式的不同，其表现行为略有不同。下面的instanceof函数有助于确定方法是如何被调用的。

```
// 构造函数
var Gadget = function (price) {
    this.price = price;
};

// 静态方法
Gadget.isShiny = function () {

    // 这种方法总是可以运行
    var msg = "you bet";

    if (this instanceof Gadget) {
        // this only works if called non-statically
        msg += ", it costs $" + this.price + '!';
    }

    return msg;
```

```
};

// 向该原型添加一个普通方法
Gadget.prototype.isShiny = function () {
    return Gadget.isShiny.call(this);
};
```

测试静态方法调用：

```
Gadget.isShiny(); // 输出"you bet"
```

测试实例，非静态调用：

```
var a = new Gadget('499.99');
a.isShiny(); // 输出"you bet, it costs $499.99!"
```

私有静态成员

到目前为止，本章所讨论的是公有静态方法，现在让我们来看看如何实现私有静态成员。就私有静态成员而言，指的是成员具有如下属性：

- 以同一个构造函数创建的所有对象共享该成员。
- 构造函数外部不可访问该成员。

下面让我们看一个例子，其中counter是构造函数Gadget中的一个私有静态属性。在本章中已经存在有关私有属性的讨论，因此这一部分仍然是相同的。需要一个函数作为闭包并且包装私有成员。然后，让我们使同一个包装函数立即执行并返回一个新函数。返回的函数值分配给变量Gadget，并且成为了新的构造函数：

```
var Gadget = (function () {

    // 静态变量/属性
    var counter = 0;

    // 返回
    // 该构造函数新的实现
    return function () {
        console.log(counter += 1);
    };

}()); // 立即执行
```

新的Gadget构造函数只是简单的递增和记录其私有counter成员。使用以下几个实例进行测试，可以看到counter的确是在多个实例之间共享：

```
var g1 = new Gadget(); // logs 1
var g2 = new Gadget(); // logs 2
```

```
var g3 = new Gadget(); // logs 3
```

由于我们对每个对象都以1为单位递增counter，这个静态属性实际上成为了对象ID标识符，它唯一标识了以Gadget构造函数创建的每个对象。这种唯一标识符可能是很有用的，因此为什么不通过特权方法（privileged method）将其公开？下面是基于前面示例基础上建立的一个例子，其主要增加了一个特权方法getLastId()以访问静态私有属性：

```
// 构造函数
var Gadget = (function () {

    // 静态变量/属性
    var counter = 0,
        NewGadget;

    // 这将成为
    // 新的构造函数的实现
    NewGadget = function () {
        counter += 1;
    };

    // 特权方法
    NewGadget.prototype.getLastId = function () {
        return counter;
    };

    // 覆盖该构造函数
    return NewGadget;

}()); // 立即执行
```

以下是测试新的实现：

```
var iphone = new Gadget();
iphone.getLastId(); // 1
var ipod = new Gadget();
ipod.getLastId();   // 2
var ipad = new Gadget();
ipad.getLastId();   // 3
```

静态属性（共有和私有）使用会带来很多便利。它们可以包含非实例相关的方法和数据，并且不会为每个实例重新创建静态属性。在第7章中，当涉及单体（singleton）模式时，可以看到一个使用静态属性以实现类似类（class-like）的单体构造函数的例子。

对象常量

JavaScript中没有常量的概念，虽然许多现代的编程环境可能为您提供了用以创建常量const语句。

作为一种变通方案，JavaScript中常见的一种方法是使用命名约定，使那些不应该被修改的变量全部用大写字母以突出显示。实际上这个命名约定已经用于内置JavaScript对象中了。

```
Math.PI; // 3.141592653589793
Math.SQRT2; // 1.4142135623730951
Number.MAX_VALUE; // 1.7976931348623157e+308
```

对于您自己的常量，也可以采用相同的命名约定，并且将它们以静态属性的方式添加到构造函数中。

```
// 构造函数
var Widget = function () {
    // 实现...
};

// 常数
Widget.MAX_HEIGHT = 320;
Widget.MAX_WIDTH = 480;
```

同样的命名约定还可应用于以字面量创建的对象中，这些常量可以是以大写字母命名的正常属性。

如果你真的想有一个不可变的值，可以创建一个私有属性并提供一个取值（getter）方法，但并不提供设值函数（setter）。不过在许多情况下，当可以采用简单的命名公约取值时，这种不提供设值函数的方法可能显得矫枉过正。

下面是一个通用的constant（常量）对象实现方法示例，它提供了下列方法：

set(*name, value*)
: 定义一个新的常量。

isDefined(*name*)
: 检测指定常量是否存在。

get(*name*)
: 读取指定常量的值。

在这个实现中，只有原始值（primitive value）允许设为常量。此外，一些额外的注意事项是要确保声明的常量名与内置属性名不会冲突，比如toString或hasOwnProperty等，可以通过使用hasOwnProperty()检查名称，并且在所有的常量名前面添加随机生成的前缀，从而确保名称之间相互适应。

```
var constant = (function () {
    var constants = {},
```

```
    ownProp = Object.prototype.hasOwnProperty,
    allowed = {
        string: 1,
        number: 1,
        boolean: 1
    },
    prefix = (Math.random() + "_").slice(2);
return {
    set: function (name, value) {
        if (this.isDefined(name)) {
            return false;
        }
        if (!ownProp.call(allowed, typeof value)) {
            return false;
        }
        constants[prefix + name] = value;
        return true;
    },
    isDefined: function (name) {
        return ownProp.call(constants, prefix + name);
    },
    get: function (name) {
        if (this.isDefined(name)) {
            return constants[prefix + name];
        }
        return null;
    }
};
```

测试以上实现代码：

```
// 检查是否已经定义
constant.isDefined("maxwidth"); // false

// 定义
constant.set("maxwidth", 480); // true

// 再次检查
constant.isDefined("maxwidth"); // true

// 试图重新定义
constant.set("maxwidth", 320); // false

// 该值是否仍保持不变
constant.get("maxwidth"); // 480
```

链模式

链模式（Chaining Pattern）可以使您能够一个接一个的调用对象的方法，而无需将前一个操作返回的值赋给变量，并且无需将您的调用分割成多行：

```
myobj.method1("hello").method2().method3("world").method4();
```

当创建的方法其返回的是无任何意义的值时，可以使它们返回this，即正在使用的对象的实例。这将使对象的用户调用前面链接的下一个方法：

```
var obj = {
    value: 1,
    increment: function () {
        this.value += 1;
        return this;
    },
    add: function (v) {
        this.value += v;
        return this;
    },
    shout: function () {
        alert(this.value);
    }
};

// 链方法调用
obj.increment().add(3).shout(); // 5

// 与逐个调用的方式相反
obj.increment();
obj.add(3);
obj.shout(); // 5
```

链模式的优点和缺点

使用链模式的一个优点在于可以节省一些输入的字符，并且还可以创建更简洁的代码，使其读起来就像是一个句子。

另一个优点在于它可以帮助您考虑分割函数，以创建更加简短、具有特定功能的函数，而不是创建尝试实现太多功能的函数。从长远看来，这提高了代码的可维护性。

链模式的一个缺点在于以这种方式编写的代码更加难以调试。或许知道在某个特定的代码行中发生错误，但是在此行中实际执行了太多的步骤。当链中多个方法的其中一个静默失效时，无法知道是哪一个方法发生失效。《Clean Code》这本书的作者Robert Martin甚至将其称为“火车失事（train wreck）”模式。

在任何情况下，识别出这种模式都很有好处。当编写的方法并没有明显和有意义的返回值时，可以总是返回this。该模式得到了广泛的应用，比如在jQuery库中就使用了该模式。此外，如果查看DOM的API，那么还可以注意到它的结构也倾向于链模式，如下所示：

```
document.getElementsByTagName('head')[0].appendChild(newnode);
```

method()方法

JavaScript可能会使那些以类的方式思考的程序员感到困惑。这也就是为什么一些开发人员倾向于选择使JavaScript更加类似类（class-like）。其中一个这样的尝试是由Douglas Crockford引入method()方法的思想。现在回想起来，他承认使JavaScript类似类（class-like）的思想是并不值得推荐的方案，但是它仍然是一种令人关注的模式，有可能在一些应用程序中遇到这种模式。

使用构造函数看起来就像是在使用Java中的类。它们还能够支持您向构造函数主体中的this添加实例属性。然而这种向this添加方法的机制其效率低下，原因在于它们最终都会与每个实例一起被重新创建，并且消耗更多的内存空间。这也就是为什么可复用方法应该添加到构造函数的prototype属性中的原因。许多程序员可能对prototype属性看起来都比较陌生，因此可以将其隐藏在方法之后。

注意： 向编程语言中添加便利的功能通常也被称之为语法糖（syntactic sugar）或简单的称之为糖（sugar）。在这种情况下，可以将method()方法称之为“糖方法（sugar method）”。

使用糖method()定义类的方法，其形式如下所示：

```
var Person = function (name) {
    this.name = name;
}.
    method('getName', function () {
        return this.name;
    }).
    method('setName', function (name) {
        this.name = name;
        return this;
    });
```

请注意构造函数是如何链接到method()的调用，其依次链接到下一个method()的调用，后面以此类推。这个例子遵循了前面介绍的链模式，它何以帮助您以单个声明语句定义整个“类”。

method()方法有两个参数：

- 新方法的名称。
- 方法的实现。

从代码中可以看到，这个新方法然后可添加到Person “类”中。实现仅是另一个函数，并且实现函数内的this指向了由Person所创建的对象，而这正是您所期望的。

下面向您展示了如何使用Person()创建并使用一个新的对象：

```
var a = new Person('Adam');
a.getName(); // 输出'Adam'
a.setName('Eve').getName(); 输出// 'Eve'
```

再次强调，请注意链模式已经生效，这是由于setName()返回了this，从而使得以上代码可以正常运行。

最后，下面展示了method()方法是如何实现的：

```
if (typeof Function.prototype.method !== "function") {
    Function.prototype.method = function (name, implementation) {
        this.prototype[name] = implementation;
        return this;
    };
}
```

在method()的实现中，首先我们应该认真的检查该方法是否已经实现过。如果没有，那么继续添加函数，并将其作为implementation参数传递给构造函数的原型。在这种情况下，this指的是构造函数，其原型得到了增强。

小结

在本章中，您学习到多种不同的模式以创建对象，而这些模式超越了基本的使用对象字面量以及构造函数创建的方法。

您学习了命名空间（namespacing）模式以保持全局代码干净，并且可以帮助您组织代码结构。还学习了有关简单、却非常有用的声明依赖（declaring dependency）。然后对一些关于私有模式（privacy pattern）的主题进行了详细探讨，其主要包括私有成员、特权方法、边缘情况下的私有性、具有私有成员的对象字面量的使用以及将私有方法揭示为公共方法等内容。所有用于构建块的这些模式都是非常流行且强大的模块模式(module pattern)。

然后，学习了沙箱模式（sandbox pattern），它可作为长命名空间的替代方法，也可以帮助您为代码和模块创建独立的环境。

最后总结以上讨论，我们深入查看了对象常量、静态方法(包括共有和私有两种)、链，以及一个新奇的method()方法。

第6章

代码复用模式

代码复用是一个非常重要而有趣的主题，简而言之，这是由于人们很自然的争取编写尽可能少的代码，并且尽可能多的复用自己或者其他人已经编写过的现有代码。尤其是那些具有质量优秀、通过测试、可维护、可扩展性、文档化的可复用代码。

在谈及代码复用的时候，首先想到的是代码的继承性（inheritance），而本章中大部分也专门致力于代码复用这个主题。在这里可以看到多种方法都可以实现“基于类特性的(classical)”和“非基于类特性的（nonclassical）”继承特性。但重要的是要记住其最终目标，我们要复用代码。继承性就是程序员用以实现代码复用这个目标的一种方法(或手段)，而且它也并不是唯一的方法。在本章中，可以看到如何利用其他对象组合成所需的对象，也可以看到如何使用mix-in 技术[注1]，还可以看到如何在技术上没有永久继承的情况下仅借用和复用所需的功能。

当开始接触代码复用任务时，请记住GoF（Gang of Four，指《Design Patterns》的四位作者）等人在其著作中提出的有关创建对象的建议原则：“优先使用对象组合，而不是类继承。”

传统与现代继承模式的比较

经常会在有关JavaScript继承模式的主题讨论中听到术语“传统继承（classical inheritance）”，为此，让我们首先阐明“传统（classical）”所代表的意义。该术语从词义上来说，并不是与用于古董、历史沉积、或者广为接受并认定为正确的处理事务的方法这些词义相同。实际上，该术语只是单词“Class（类）”的一种表现形式。

注1：　混入或者渗元技术，这是一种在不改变原对象的情况下对其进行扩展的技术。

许多编程语言都具有类的概念，并以此作为对象的蓝图（blueprints）。在那些编程语言中，每个对象都是一个类的特定实例（比如，在Java语言环境中），并且在不存在某个类的时候并不能创建该类的对象。在JavaScript中，由于并没有类的概念，因此实例的概念也就没有多大意义。JavaScript中的对象是简单的键－值（key-value）对，可以动态的创建和修改这些对象。

但JavaScript具有构造函数，并且new操作符的语法与那些使用类的编程语言在语法上有许多相似之处。

在Java中可以采用下列方式创建对象：

```
Person adam = new Person();
```

而在JavaScript中则可以采用下列方式创建对象：

```
var adam = new Person();
```

除了与Java中强类型限制的情况不同之外，在JavaScript中也必须声明adam是Person类型，其语法与Java看起来是一样的。JavaScript的构造函数在调用时Person看起来似乎是一个类，但重要的是要记住Person仍然只是一个函数。这种语法上的相似性导致了许多程序员按照类的方式考虑JavaScript，并产生了一些假定在类的基础上的开发思路和继承模式。我们将这种实现方式称之为“类式（classical）”继承模式。在这里让我们将“现代（modern）”模式表述为：其他任何不需要以类的方式考虑的模式。

当在项目开发中涉及采用继承模式的时候，有相当多的选择。建议总是争取采用一种现代的继承模式，除非团队在不涉及到类的时候对此真的感到不适应。

本章首先讨论了类式继承模式，然后再转而讨论现代继承模式。

使用类式继承时的预期结果

实现类式继承（classical inheritance）的目标是通过构造函数Child()获取来自于另外一个构造函数Parent()的属性，从而创建对象。

注意： 虽然这里讨论的是类式继承模式，但是请让我们尽量避免使用“class（类）”这个单词。将其表述为“构造函数（constructor function或者constructor）”时虽然字数更长一些，但其表述更为精确且不会产生歧义。一般情况下，在开发团队交流时请努力消除单词“class”的使用，因为当涉及JavaScript时，“class”这个单词对于不同的人可能意味着不同的含义。

下面是定义两个构造函数Parent()和Child()的一个例子：

```
// 父构造函数
function Parent(name) {
    this.name = name || 'Adam';
}

// 向该原型添加功能
Parent.prototype.say = function () {
    return this.name;
};

// 空白的子构造函数
function Child(name) {}

// 继承的魔力在这里发生
inherit(Child, Parent);
```

在上述代码中，存在父［(parent()］与子［(child()］两个构造函数，say()方法被添加到父构造函数的原型（prototype）中，并且一个名为inherit()的函数调用负责处理它们之间的继承关系。其中，inherit()函数并非由编程语言提供的，为此，程序员必须自己来实现该函数。下面，让我们看看实现该函数的几种常见方法。

类式继承模式#1——默认模式

最常用的一种默认方法是使用Parent()构造函数创建一个对象，并将该对象赋值给Child()的原型。下面是可复用继承函数inherit()的第一种实现方法：

```
function inherit(C, P) {
    C.prototype = new P();
}
```

重要的是需要记住，原型属性应该指向一个对象，而不是一个函数，因此它必须指向一个由父构造函数所创建的实例（一个对象），而不是指向构造函数本身。也就是说，要注意使用new操作符来创建新对象，因为需要new才能使这种模式运作。

在以后的应用程序中，当使用new Child()语句创建一个对象时，它会通过原型从Parent()实例中获取它的功能，如下面的例子所示：

```
var kid = new Child();
kid.say(); // 输出"Adam"
```

追溯原型链

使用这种默认的继承模式时，同时继承了自身的属性（即加入到this的实例相关属性，比如name），以及原型属性和方法［比如say()］。

让我们回顾一下在这种继承模式下原型链的工作原理。出于讨论的目的，让我们将对象视做存在于内存中某处的块，该内存块可以包含数据以及指向其他块的引用。当使用new Parent()语句创建一个对象时，会创建一个这样的块，即如图6-1所示标记的块#2。在#2块中保存了name属性的数据。如果您尝试访问say()方法，虽然[例如，使用(new Parent).say()语句]块#2中并不包含say()方法，但是通过使用指向构造函数Parent()的prototype（原型）属性的隐式链接__proto__，便可以访问对象#1（Parent.prototype），而对象#1又确实知道关于say()的地址。所有这一切都在后台发生，并不用为这种复杂的原型链而烦恼，但重要的是需要理解它的工作原理以及所需要访问或可能修改的数据位于何处。请注意，这里仅使用__proto__来解释原型链，即使在一些环境中（比如Firefox浏览器）提供了该属性，在程序开发语言中也并不能使用该属性。

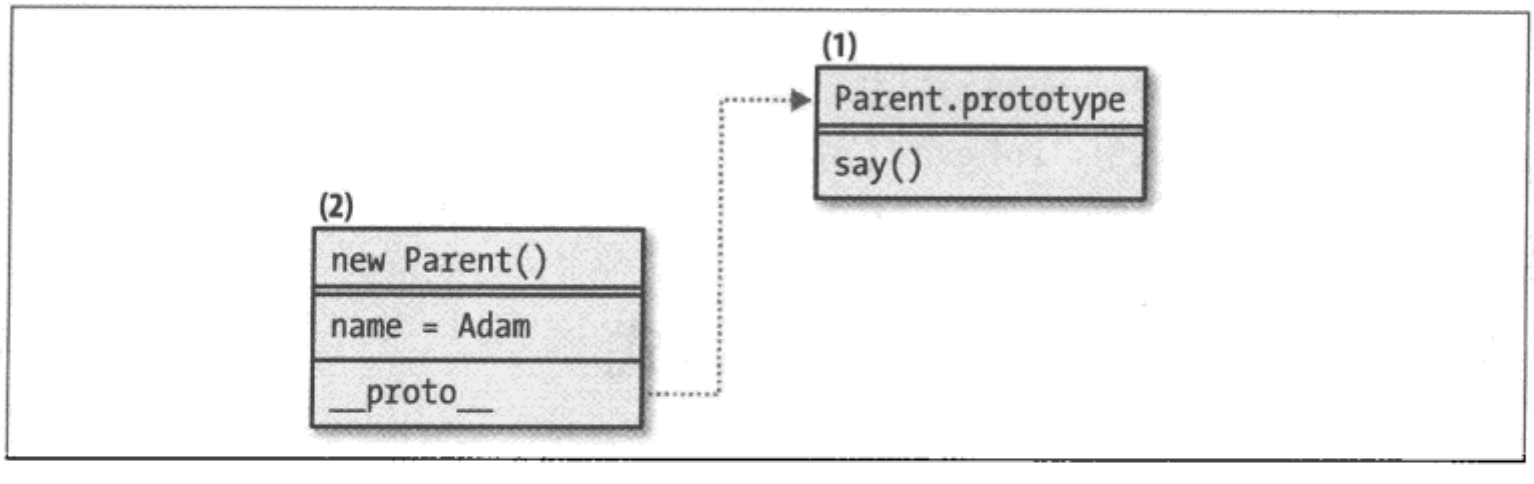

图6-1：Parent()构造函数的原型链

现在，让我们查看在使用inherit()函数后，当使用var kid = new Child()创建新对象时会发生什么情况，如图6-2中的图表所示。

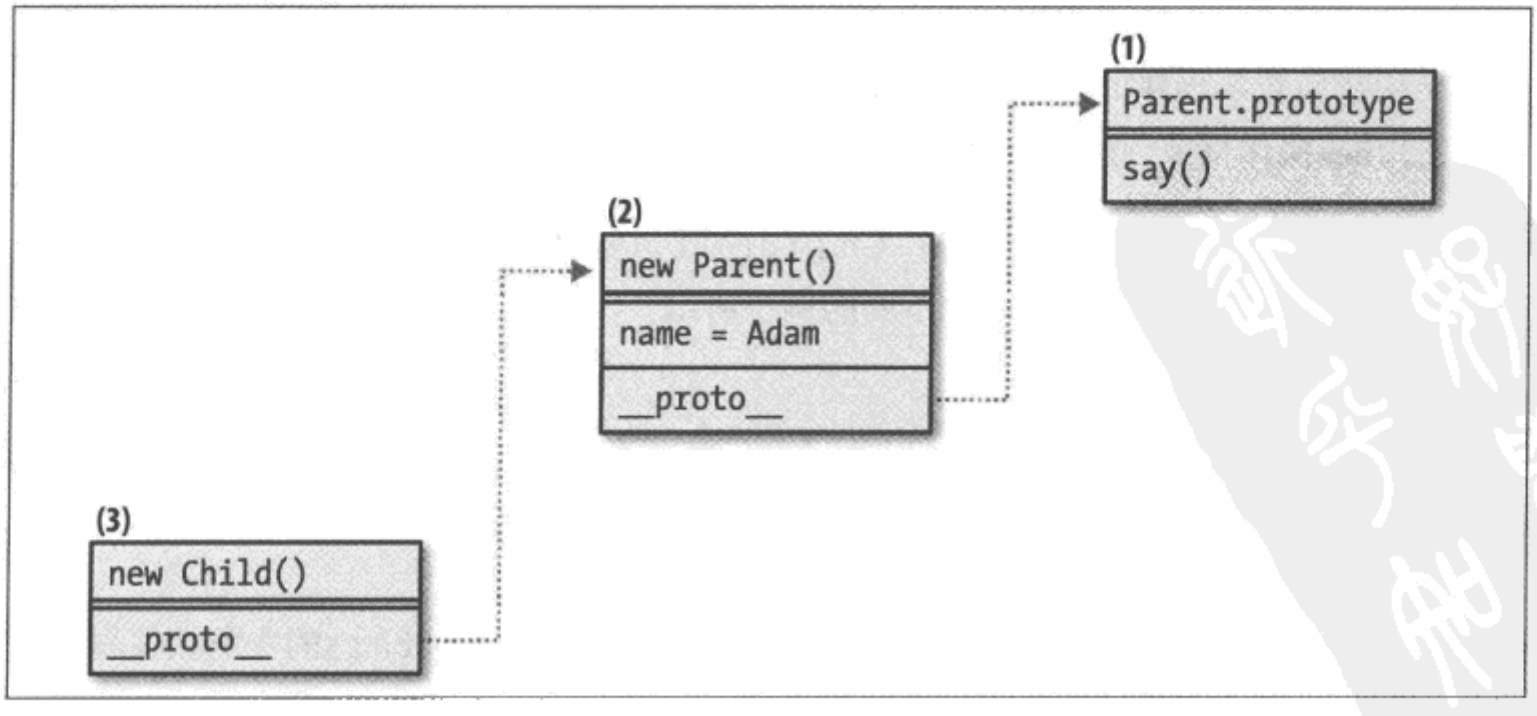

图6-2：调用继承函数后的原型链

从图6-2中可以看到，Child()构造函数是空的，并且没有任何属性添加到Child.prototype中。因此，使用new Child()语句所创建的对象除了包含隐式链接__proto__以外，它几乎是空的。在这种情况下，__proto__指向了在inherit()函数中使用new Parent()语句所创建的对象。

现在，当执行kid.say()时会发生什么情况？对象#3中并没有这样的say()方法，因此它将通过原型链查询到对象#2。然而，对象#2中也没有该方法，因此它又顺着原型链查询到对象#1，而对象#1正好具有say()方法。然而，在say()中引用了this.name，该引用仍然还需要解析。因此，查询再次启动。在这种情况下，this指向对象#3，对象#3中并没有name属性。为此，将查询对象#2，而对象#2中确实有name属性，其值为“Adam”。

最后，让我们更进一步查看原型链的概念，比如说，我们有以下这样的代码：

```
var kid = new Child();
kid.name = 输出"Patrick";
kid.say(); // 输出"Patrick"
```

图6-3显示了在上述这种情况下原型链的工作过程。

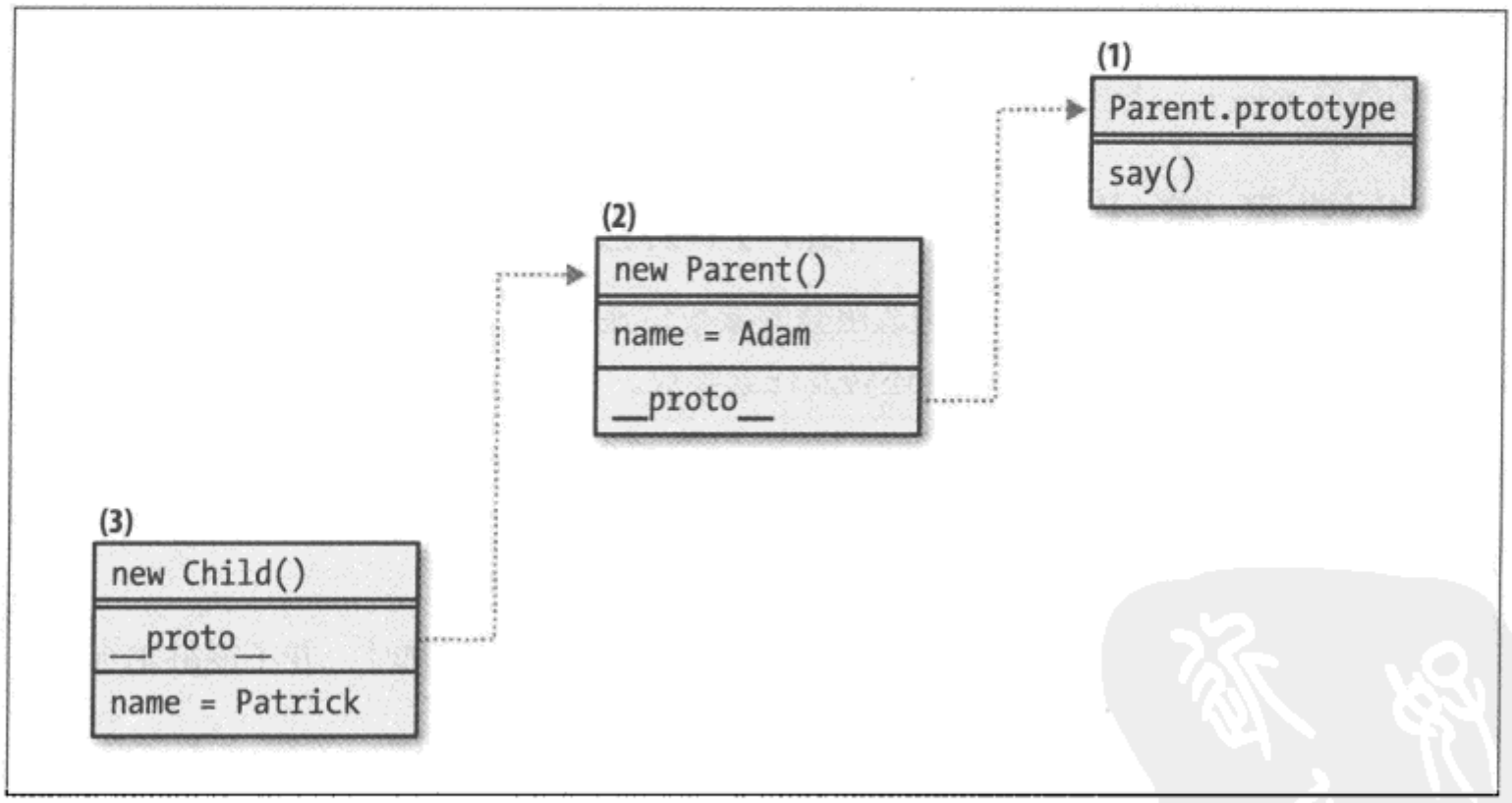

图6-3：在调用继承函数并在child对象中添加一个属性后的原型链

设置kid.name的语句并不会修改对象#2的name属性，但是它却直接在kid对象#3上创建了一个自身的属性。当执行kid.say()时，将依次在对象#3、对象#2中查询say()方法，并且最终在对象#1中查找到该方法，这与前面所述的过程相似。但是，如果这次是查找this.name（这是与kid.name相同的），那么其过程是很快的，这是由于该属性立刻就能够在对象#3中找到，而无需通过原型链。

如果使用delete kid.name语句删除新属性，那么对象#2的name属性将“表现出来（shine through）”，并在连续的查找过程中找到其name属性。

使用模式#1时的缺点

本模式的其中一个缺点在于同时继承了两个对象的属性，即添加到this的属性以及原型属性。在绝大多数的时候，并不需要这些自身的属性，因为它们很可能是指向一个特定的实例，而不是复用。

注意：对于构造函数的一般经验法则是：应该将可复用的成员添加到原型中。

另一个关于使用通用inherit()函数的问题在于它并不支持将参数传递到子构造函数中，而子构造函数然后又将参数传递到父构造函数中，考虑以下这个例子：

```
var s = new Child('Seth');
s.say(); // 输出"Adam"
```

以上输出结果可能并不是您所期望的[注2]。虽然子构造函数可以将参数传递到父构造函数中，但是那样的话，在每次需要一个新的子对象时都必须重新执行这种继承机制，而且该机制其效率低下，其原因在于最终会反复地重新创建父对象。

类式继承模式#2——借用构造函数

本模式解决了从子构造函数到父构造函数的参数传递问题。本模式借用了父构造函数，它传递子对象以绑定到this，并且还转发任意参数。

```
function Child(a, c, b, d) {
    Parent.apply(this, arguments);
}
```

在这种方式中，只能继承在父构造函数中添加到this的属性。同时，并不能继承那些已添加到原型中的成员。

使用该借用构造函数模式时，子对象获得了继承成员的副本，这与类式继承模式#1中仅获取引用的方式是不同的。下面的例子演示了其差异：

```
// 父构造函数
function Article() {
    this.tags = ['js', 'css'];
}
var article = new Article();
```

注2：s初始化时使用的是'Seth'，但是调用say()时却输出了"Adam"。

```
// blog 文章对象继承了article对象
// via the classical pattern #1
function BlogPost() {}
BlogPost.prototype = article;
var blog = new BlogPost();
// 注意以上代码，你并不需要`new Article()`
// 这是因为你已经有一个可用的实例

// static page （静态页面）继承了 article
// 通过借用构造函数模式
function StaticPage() {
    Article.call(this);
}
var page = new StaticPage();

alert(article.hasOwnProperty('tags')); // true
alert(blog.hasOwnProperty('tags')); // false
alert(page.hasOwnProperty('tags')); // true
```

在以上代码片段中，有两种方式都继承了父构造函数Article()。默认模式导致了blog对象通过原型以获得对tags属性的访问，因此blog对象中没有将article作为自身的属性，因此当调用hasOwnProperty()时会返回false。相反，page对象本身则具有一个tags属性，这是由于它在使用借用构造函数的时候，新对象会获得父对象中tags成员的副本(不是引用)。

请注意修改继承的tags属性时表现出来的差异：

```
blog.tags.push('html');
page.tags.push('php');
alert(article.tags.join(', ')); // 输出"js, css, html"
```

在上面这个例子中，子对象blog修改了其 tags属性，而这种方式同时也会修改父对象article，这是由于本质上blog.tags和article.tags都指向了同一个数组。但是，修改page.tags时却不会影响其父对象article，这是由于在继承过程中page.tags是独立创建的一个副本。

原型链

当使用本模式以及熟悉的Parent()和Child()构造函数时，让我们来看原型链(prototype chain)的工作流程。其中，Child()需要根据这个新模式的需求略加修改：

```
// 父构造函数
function Parent(name) {
    this.name = name || 'Adam';
}

// 向该原型添加功能
Parent.prototype.say = function () {
```

```
    return this.name;
};

// 子构造函数
function Child(name) {
    Parent.apply(this, arguments);
}

var kid = new Child("Patrick");
kid.name; // 输出"Patrick"
typeof kid.say; // 输出"undefined"
```

如果仔细查看图6-4，将会注意到在new Child对象和Parent对象之间不再有链接。出现这种现象的原因在于本模式中根本就没有使用Child.prototype，并且它只是指向一个空对象。使用本模式时，kid获得了自身的属性name，但是却从未继承过say()方法，如果试图调用该方法将会导致错误。继承是一个一次性的操作，它仅会复制父对象的属性并将其作为子对象自身的属性，仅此而已。因此，也就不会保留__proto__链接。

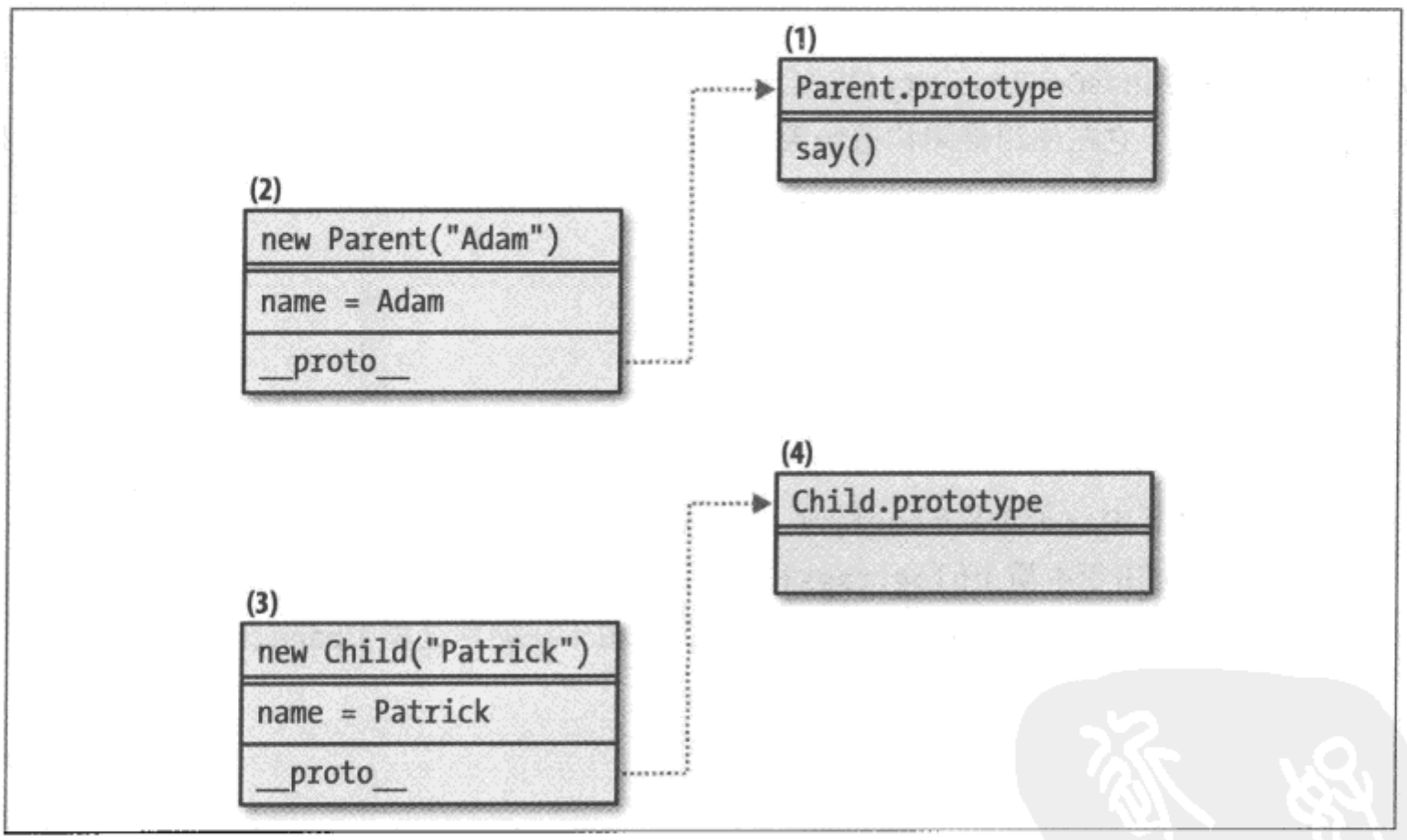

图6-4：使用借用构造函数模式时断开的原型链

通过借用构造函数实现多重继承

当使用借用构造函数模式时，可以通过借用多个构造函数从而简单的实现多重继承。

```
function Cat() {
    this.legs = 4;
    this.say = function () {
```

```
        return "meaowww";
    }
}
function Bird() {
    this.wings = 2;
    this.fly = true;
}
function CatWings() {
    Cat.apply(this);
    Bird.apply(this);
}
var jane = new CatWings();
console.dir(jane);
```

上述代码运行的结果如图6-5所示[注3]。在解析任意的副本属性时，将会通过最后一个获胜的方式来解析该属性。

图6-5：在Firebug中检查CatWing对象

借用构造函数模式的优缺点

借用构造函数模式的缺点是很明显的，如前面所述，其问题在于无法从原型中继承任何东西，并且原型也仅是添加可重用方法以及属性的位置，它并不会为每个实例重新创建原型。

本模式的一个优点在于可以获得父对象自身成员的真实副本，并且也不会存在子对象意外覆盖父对象属性的风险。

因此，在前面的情况中，如何才能使子对象也能够继承原型属性？以及如何使kid能够访问say()方法？下面的模式将解决这个问题。

类式继承模式#3——借用和设置原型

类式继承模式#3主要思想是结合前两种模式，即先借用构造函数，然后还设置子构造函数的原型使其指向一个构造函数创建的新实例。代码如下所示：

```
function Child(a, c, b, d) {
    Parent.apply(this, arguments);
}
```

注3：　Firebug是Firefox下的插件，可用于调试JavaScript。

```
Child.prototype = new Parent();
```

这样做的优点在于，以上代码运行后的结果对象能够获得父对象本身的成员副本以及指向父对象中可复用功能(以原型成员方式实现的那些功能)的引用。同时，子对象也能够将任意参数传递到父构造函数中。这种行为可能是最接近您希望在Java中实现的方式。可以继承父对象中的一切东西，同时这种方法也能够安全的修改自身属性，且不会带来修改其父对象的风险。

这种模式的一个缺点是，父构造函数被调用了两次，因此这导致了其效率低下的问题。最后，自身的属性（比如本例中的name属性）会被继承两次。

让我们查看下列代码并进行一些测试：

```
// 父构造函数
function Parent(name) {
    this.name = name || 'Adam';
}

// adding functionality to the prototype
Parent.prototype.say = function () {
    return this.name;
};

// 子构造函数
function Child(name) {
    Parent.apply(this, arguments);
}
Child.prototype = new Parent();

var kid = new Child("Patrick");
kid.name; // 输出"Patrick"
kid.say(); // 输出"Patrick"
delete kid.name;
kid.say(); // 输出"Adam"
```

在上面的代码中，不同于先前的模式，现在say()方法已被正确地继承。还可以注意到name属性却被继承了两次，在我们删除了kid本身的name属性的副本以后，随后看到的输出是原型链表现出来（shine through）所引出的name属性。

图6-6显示了对象之间的链接关系。这些关系非常类似于图6-3中所示的原型链，但这里我们所采用的继承方式是不同的。

类式继承模式#4——共享原型

不同于前面的那种需要两次调用父构造函数的模式(类式继承模式#3)，接下来介绍的模式根本就不涉及调用任何父构造函数。

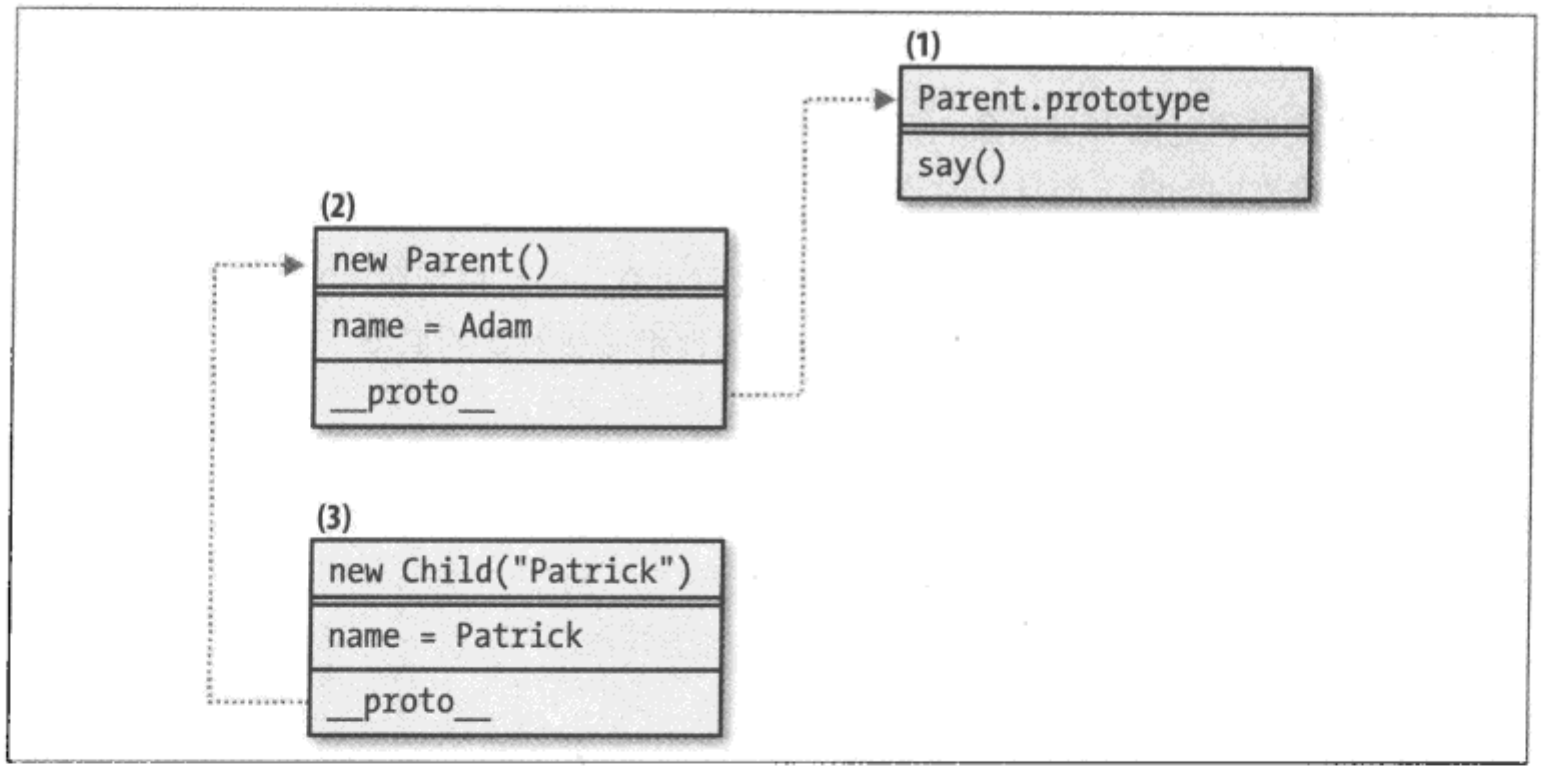

图6-6：除继承的自身成员之外还保持了原型链

本模式的经验法则在于：可复用成员应该转移到原型中而不是放置在this中。因此，出于继承的目的，任何值得继承的东西都应该放置在原型中实现。所以，可以仅将子对象的原型与父对象的原型设置为相同的即可，代码如下所示：

```
function inherit(C, P) {
    C.prototype = P.prototype;
}
```

这种模式能够向您提供简短而迅速的原型链查询，这是由于所有的对象实际上共享了同一个原型。但是，这同时也是一个缺点，因为如果在继承链下方的某处存在一个子对象或者孙子对象修改了原型，它将会影响到所有的父对象和祖先对象。

如图6-7所示，下面的子对象和父对象共享了同一个原型，并且可以同等的访问say()方法。然而，需要注意到子对象并没有继承name属性。

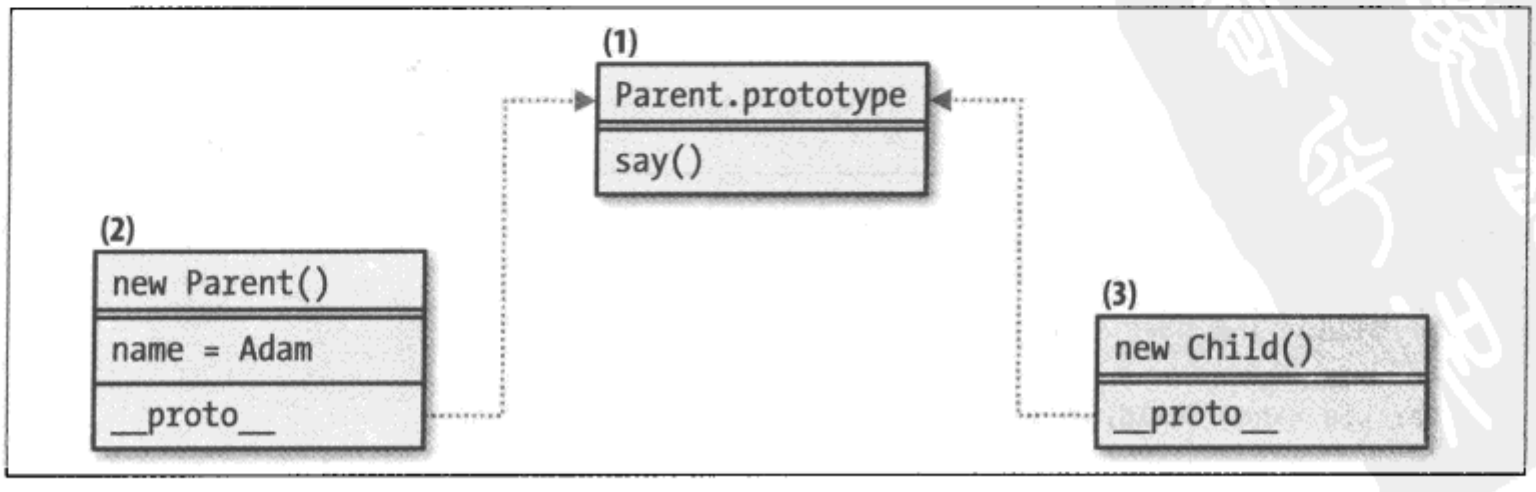

图6-7：共享同一个原型时对象之间的关系

类式继承模式#5——临时构造函数

类式继承模式#5通过断开父对象与子对象的原型之间的直接链接关系，从而解决共享同一个原型所带来的问题，而且同时还能够继续受益于原型链带来的好处。

下面的代码是本模式的一种实现方式，在该代码中有一个空白函数F()，该函数充当了子对象和父对象之间的代理。F()的prototype属性指向父对象的原型。子对象的原型则是一个空白函数实例。

```
function inherit(C, P) {
    var F = function () {};
    F.prototype = P.prototype;
    C.prototype = new F();
}
```

这种模式在行为上与默认模式（类式继承模式#1）略有不同，这是由于这里的子对象仅继承了原型的属性（见图6-8）。这种情况通常来说是很好的，实际上也是更加可取的，因为原型也正是放置可复用功能的位置。在这种模式中，父构造函数添加到this中的任何成员都不会被继承。

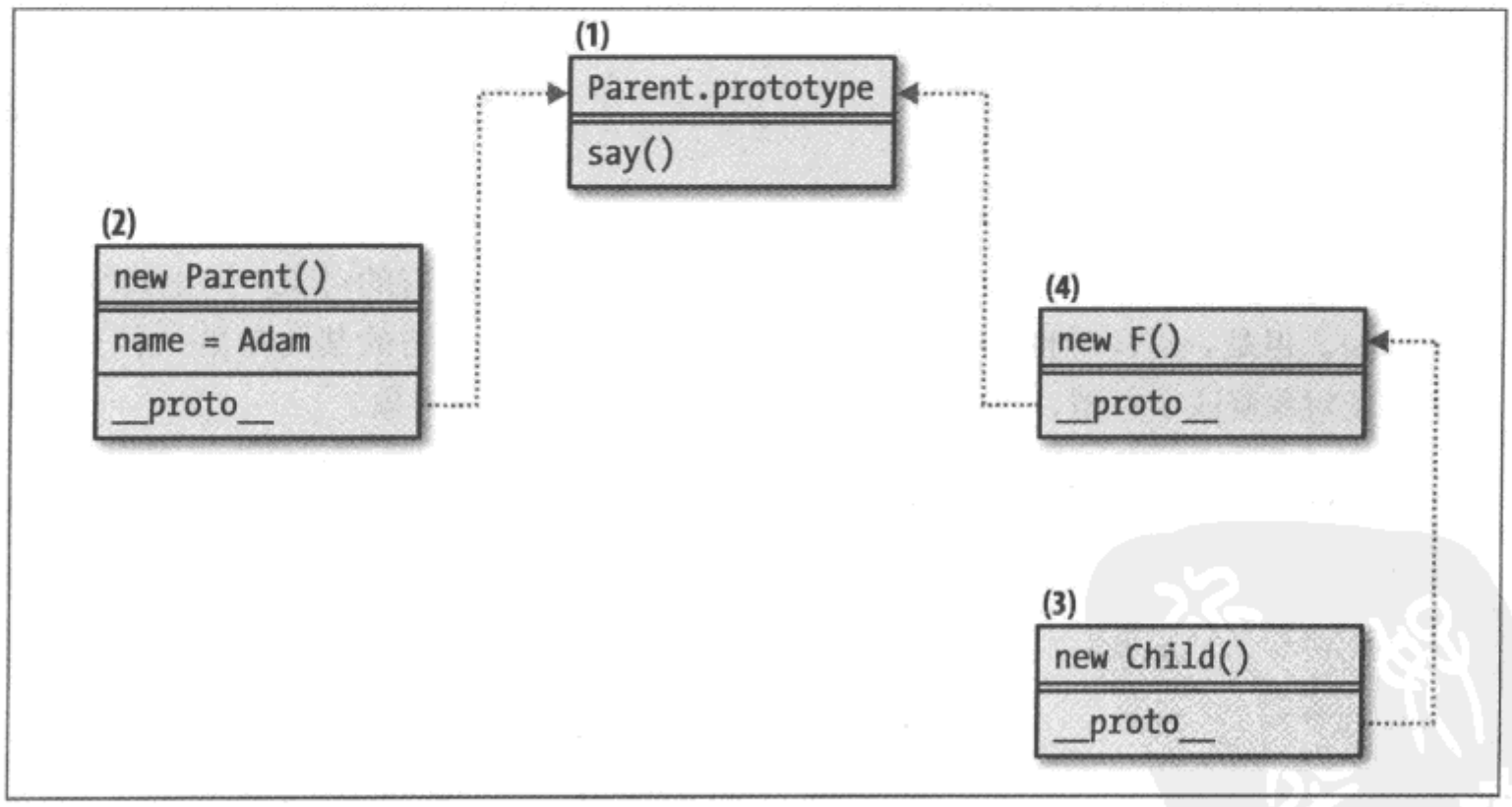

图6-8：通过使用临时(代理)构造函数F()的类式继承

让我们创建一个新的子对象，并审查其行为：

```
var kid = new Child();
```

如果访问kid.name，其结果将是undefined类型。在这种情况下，name是父对象所拥有的

一个属性，然而在继承的时候我们实际上从未调用过new Parent()，因此也从未创建过该属性。当您访问kid.say()时，在对象#3中该方法并不可用，因此需要开始查询原型链。然而对象#4中也没有该方法，但是对象#1中确实存在该方法并且位于内存中的同一位置，因此所有继承了Parent()的不同构造函数，以及所有由其子构造函数所创建的对象都可重用该say()方法。

存储超类

在上面模式的基础上，还可以添加一个指向原始父对象的引用。这就像在其他编程语言中访问超类一样，这可以偶尔派上用场。

该属性被称之为uber，这仅是由于“super”是保留的关键词，并且“superclass”可能导致粗心的程序员不加思考便顺势根据该关键词认为JavaScript中具有类（class）。下面是该类式继承模式的一个改进实现：

```
function inherit(C, P) {
    var F = function () {};
    F.prototype = P.prototype;
    C.prototype = new F();
    C.uber = P.prototype;
}
```

重置构造函数指针

最后，针对对这个近乎完美的类式继承函数，还需要做的一件事情就是重置该构造函数的指针，以免在将来的某个时候还需要该构造函数。

如果不重置该构造函数的指针，那么所有子对象将会报告Parent()是它们的构造函数，这是没有任何用处的。因此，使用前面的inherit()实现代码，可以观察到此行为：

```
// 父，子，继承
function Parent() {}
function Child() {}
inherit(Child, Parent);

// 投石问路
var kid = new Child();
kid.constructor.name; // "Parent"
kid.constructor === Parent; // true
```

虽然我们很少用到constructor属性，但是这种功能却可以很方便的用于运行时对象的内省。可以重置constructor属性使其指向期望的构造函数且不会影响其功能，这是由于该属性主要是用于提供对象的信息。

这个类式继承模式最后的圣杯（Holy Grail）版本看起来如下所示：

```
function inherit(C, P) {
    var F = function () {};
    F.prototype = P.prototype;
    C.prototype = new F();
    C.uber = P.prototype;
    C.prototype.constructor = C;
}
```

如果认为这种模式是适用于项目中的最佳方法，需要说明的是，在开源YUI库（可能还有其他库）中也存在一个与本函数相似的函数，并且它还在没有类的情况下实现了类式继承。

注意： 这种模式也被称之为使用代理函数（proxy function）或代理构造函数（proxy constructor）的模式，而不是使用临时构造函数（temporary constructor）的模式，这是因为临时构造函数实际上是一个用于到获得父对象的原型的代理。

对于该圣杯模式的一个常见优化是避免在每次需要继承时都创建临时(代理)构造函数。仅创建一次临时构造函数，并且修改它的原型，这已经是非常充分的。在具体实现方式上，可以使用即时函数（immediate function）并且在闭包中存储代理函数。

```
var inherit = (function () {
    var F = function () {};
    return function (C, P) {
        F.prototype = P.prototype;
        C.prototype = new F();
        C.uber = P.prototype;
        C.prototype.constructor = C;
    }
}());
```

Klass

许多JavaScript库都模拟了类的概念，并引进了一些语法糖（sugar syntax，即增加了更便利的语法）。这些库中类的实现方式各有不同，但是往往都有一些共性，其中都包括了以下内容：

- 都有一套有关如何命名类方法的公约，这也被认为是类的构造函数，比如initialize、_init以及一些其他类似的构造函数名，并且在创建对象时这些方法将会被自动调用。
- 存在从其他类所继承的类。

- 在子类中可以访问父类或超类。

注意：让我们从这里开始改变思维习惯，由于在本章的这一部分中讨论的主题是有关模拟类的概念，因此仅在这里自由地使用“class”这个词语。

在没有深入研究其细节的情况下，让我们看一个在JavaScript中模拟类的实现示例。首先，从客户的角度来看应该如何使用该解决方案？

```
var Man = klass(null, {
    __construct: function (what) {
        console.log("Man's constructor");
        this.name = what;
    },
    getName: function () {
        return this.name;
    }
});
```

上面代码中的语法糖以一个名为klass()的函数形式出现。在其他一些实现中，可能会看到它以Klass()构造函数或以增强的Object.prototype出现。但在本例子中，让我们将其保持为一个简单的函数。

该函数有两个参数：第一个参数为将被继承的父类；第二个参数为对象字面量（object literal）所提供的新类的实现。由于受到PHP的影响，让我们建立一个公约，即类的构造函数必须是名为__construct的方法。在前面的代码片段中，创建了一个名为Man的新类，该类并没有继承任何其他类（这意味着在后台继承了Object类）。Man类中有一个在__construct中所创建的属性name，以及一个方法getName()。该类同时也是一个构造函数，因此下面的代码仍然能够正常运行（看起来就像一个类的实例化）。

```
var first = new Man('Adam'); // 记录了 "Man's constructor"
first.getName(); // "Adam"
```

现在，让我们扩充该类并创建一个SuperMan类：

```
var SuperMan = klass(Man, {
    __construct: function (what) {
        console.log("SuperMan's constructor");
    },
    getName: function () {
        var name = SuperMan.uber.getName.call(this);
        return "I am " + name;
    }
});
```

在上面的代码中，传递给klass()的第一个参数是将被继承的父类Man。同时请注意到在getName()中，其父类的函数getName()由于通过使用SuperMan的uber（super）静态属性而首先被调用。为证实该行为，我们进行如下测试：

```
var clark = new SuperMan('Clark Kent');
clark.getName(); // 结果为"I am Clark Kent"
```

记录到控制台的第一行输出为“Man's constructor”，然后输出“Superman's constructor”。实际上，在大多数基于类的语言中，每次在调用子类的构造函数时，父类的构造函数也将会被自动调用。因此，在Javascript中为何不模拟成与那些语言是一样的呢？

测试instanceof操作符将返回如下期望的结果：

```
clark instanceof Man; // true
clark instanceof SuperMan; // true
```

最后，让我们查看klass()函数是如何实现的：

```
var klass = function (Parent, props) {

    var Child, F, i;

    // 1.
    // 新构造函数
    Child = function () {
        if (Child.uber && Child.uber.hasOwnProperty("__construct")) {
            Child.uber.__construct.apply(this, arguments);
        }
        if (Child.prototype.hasOwnProperty("__construct")) {
            Child.prototype.__construct.apply(this, arguments);
        }
    };

    // 2.
    // 继承
    Parent = Parent || Object;
    F = function () {};
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.uber = Parent.prototype;
    Child.prototype.constructor = Child;

    // 3.
    // 添加实现方法
    for (i in props) {
        if (props.hasOwnProperty(i)) {
            Child.prototype[i] = props[i];
        }
    }
```

```
        // 返回该 "class"
        return Child;
    };
```

在klass()的实现中有三个令人关注且独特的部分：

1. 创建了Child()构造函数。该函数将是最后返回的，并且该函数也用作类。在这个函数中，如果存在__construct方法，那么将会调用该方法。另外，在此之前，通过使用静态uber属性，其父类的__construct方法也会被自动调用（同样，如果存在该方法的话）。可能在有些情况下，当没有定义uber属性时，比如直接从Object类中继承时，这与从Man类的定义中继承是相似的情况。
2. 第二部分在一定程度上处理继承关系。它只是采用了本章前面章节中所讨论的类式继承的圣杯版本模式。从代码上看，只有一个新的语句（Parent = Parent || Object），即如果没有传递需要被继承的类，那么就将Parent设置为Object。
3. 最后一节是遍历所有的实现方法（比如，本例中的__construct和getName），这些是该类的实际定义，并且也是将它们添加到Child的原型中的部分代码。

那么，什么时候应该使用这种模式？实际上答案是如果您避免使用它会更好，原因在于它导致了整个类的概念混淆，而在Javascript语言中严格来说不存在类的概念。这种模式增加了新的语法并且需要学习和记忆新规则。也就是说，如果您或者团队使用类时感觉很轻松，并且同时对于原型感到不适应，那么就值得探索使用这种模式。这种模式允许您完全忘记原型，并且其优点还在于您可以调整语法和公约以使其与您喜爱的语言风格相类似。

原型继承

下面我们开始讨论一种称之为原型继承（prototypal inheritance）的“现代”无类继承模式。在本模式中并不涉及类，这里的对象都是继承自其他对象。以这种方式考虑：有一个想要复用的对象，并且想创建的第二个对象需要从第一个对象中获取其功能。

下面的代码展示该如何开始着手实现这种模式：

```
// 要继承的对象
var parent = {
    name: "Papa"
};

// 新对象
var child = object(parent);

// 测试
alert(child.name); // "Papa"
```

在前面的代码片段中，存在一个以对象字面量（object literal）创建的名为parent的现有对象，并且要创建另外一个与parent具有相同属性和方法的名为child的对象。child对象是由一个名为object()的函数所创建。JavaScript中并不存在该函数(不要与构造函数object()弄混淆)，为此，让我们看看应该如何定义该函数。

与类式继承模式的圣杯版本相似，首先，可以使用空的临时构造函数F()。然后，将F()的原型属性设置为父对象。最后，返回一个临时构造函数的新实例：

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```

图6-9显示了在使用原型继承模式时的原型链。图中的child最初是一个空对象，它没有自身的属性，但是同时它又通过受益于__proto__链接而具有其父对象的全部功能。

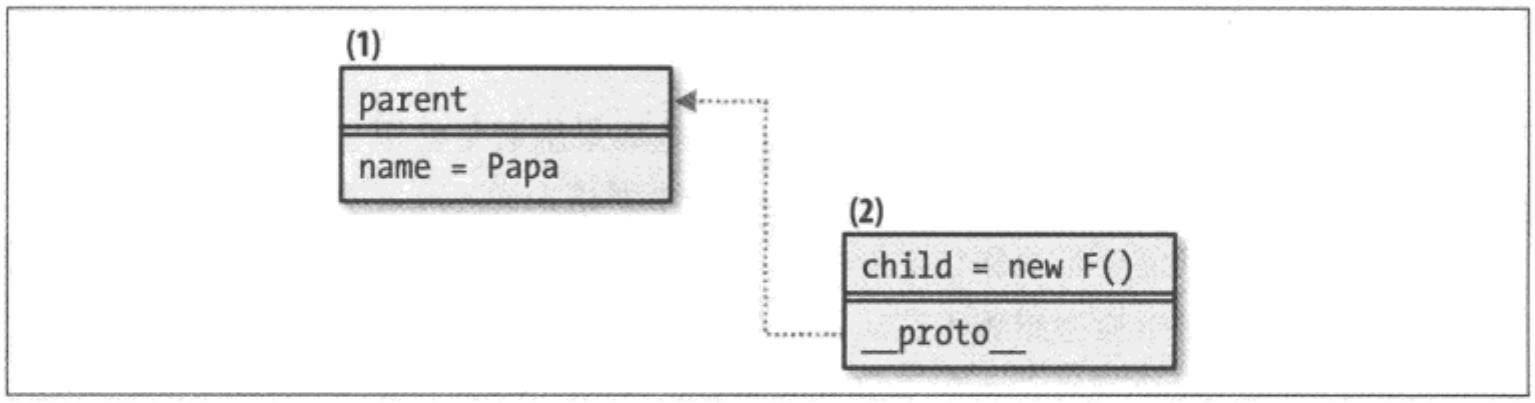

图6-9：原型继承模式

讨论

在原型继承模式中，并不需要使用字面量符号（literal notation）来创建父对象（尽管这可能是一种比较常见的方式）。如下代码所示，可以使用构造函数创建父对象，请注意，如果这样做的话，“自身”属性和构造函数的原型的属性都将被继承：

```
// 父构造函数
function Person() {
    // an "own" property
    this.name = "Adam";
}
// 添加到原型的属性
Person.prototype.getName = function () {
    return this.name;
};

// 创建一个新的Person类对象
var papa = new Person();
```

```
// 继承
var kid = object(papa);

// 测试自身的属性
// 和继承的原型属性
kid.getName(); // "Adam"
```

在本模式的另外一个变化中，可以选择仅继承现有构造函数的原型对象。请记住，对象继承自对象，而不论父对象是如何创建的。下面使用了前面的例子演示该变化，仅需稍加修改代码即可：

```
// 父构造的函数
function Person() {
    // an "own" property
    this.name = "Adam";
}
//添加到原型的属性
Person.prototype.getName = function () {
    return this.name;
};

// 继承
var kid = object(Person.prototype);

typeof kid.getName; // 结果为"function"类型因为它在原型中
typeof kid.name; // 结果为"undefined"类型,因为只有该原型是继承的
```

增加到ECMAScript 5中

在ECMAScript 5中，原型继承模式已经正式成为该语言的一部分。这种模式是通过方法Object.create()来实现的。也就是说，不需要推出与object()相类似的函数，它已经内嵌在该语言中：

```
var child = Object.create(parent);
```

Object.create()接受一个额外的参数，即一个对象。这个额外对象的属性将会被添加到新对象中，以此作为新对象自身的属性，然后Object.create()返回该新对象。这提供了很大的方便，使您可以仅采用一个方法调用即可实现继承并在此基础上构建子对象。例如：

```
var child = Object.create(parent, {
    age: { value: 2 } // ECMA5 描述符号
});
child.hasOwnProperty("age"); // 结果为true
```

可能还会发现一些JavaScript库中已经实现了原型继承模式。例如，在YUI3中是Y.Object()方法：

```
YUI().use('*', function (Y) {
    var child = Y.Object(parent);
});
```

通过复制属性实现继承

让我们看另一种继承模式，即通过复制属性（copying properties）实现继承。在这种模式中，对象将从另外一个对象中获取功能，其方法是仅需将其复制即可。下面是一个示例函数extend()实现复制继承的例子：

```
function extend(parent, child) {
    var i;
    child = child || {};
    for (i in parent) {
        if (parent.hasOwnProperty(i)) {
            child[i] = parent[i];
        }
    }
    return child;
}
```

上面的代码是一个简单的实现，它仅遍历父对象的成员并将其复制出来。在本示例实现中，child对象是可选的。如果不传递需要扩展的已有对象，那么它就会创建并返回一个全新的对象。

```
var dad = {name: "Adam"};
var kid = extend(dad);
kid.name; // 结果为"Adam"
```

上面给出的是一种所谓浅复制（shallow copy）的对象。另一方面，深度复制（deep copy）意味着属性检查，如果即将复制的属性是一个对象或者一个数组，这样的话，它将会递归遍历该属性并且还会将该属性中的元素复制出来。在使用浅复制(由于JavaScript中的对象是通过引用而传递的)的时候，如果改变了子对象的属性，并且该属性恰好是一个对象，那么这种操作表示也正在修改父对象。其实，这也是更可取的方法，但是当处理其他对象和数组时，这种浅复制也可能导致意外发生。考虑下列情况：

```
var dad = {
    counts: [1, 2, 3],
    reads: {paper: true}
};
var kid = extend(dad);
kid.counts.push(4);
dad.counts.toString(); // "1,2,3,4"
dad.reads === kid.reads; // 结果为true
```

现在，让我们修改extend()函数以实现深度复制。所有需要做的事情就是检查某个属性的类型是否为对象，如果是这样的话，需要递归复制出该对象的属性。另外，还需要检查该对象是否为一个真实对象或者一个数组，我们可以使用第3章中讨论的方法检查其数组性质（array-ness）。因此，深度复制版本的extend()函数看起来如下所示：

```
function extendDeep(parent, child) {
    var i,
        toStr = Object.prototype.toString,
        astr = "[object Array]";

    child = child || {};

    for (i in parent) {
        if (parent.hasOwnProperty(i)) {
            if (typeof parent[i] === "object") {
                child[i] = (toStr.call(parent[i]) === astr) ? [] : {};
                extendDeep(parent[i], child[i]);
            } else {
                child[i] = parent[i];
            }
        }
    }
    return child;
}
```

现在开始测试这种新的实现方式，由于它能够为我们创建对象的真实副本，因此子对象的修改并不会影响其父对象。

```
var dad = {
    counts: [1, 2, 3],
    reads: {paper: true}
};
var kid = extendDeep(dad);

kid.counts.push(4);
kid.counts.toString(); // 结果为"1,2,3,4"
dad.counts.toString(); // 结果为"1,2,3"


dad.reads === kid.reads; //结果为 false
kid.reads.paper = false;


kid.reads.web = true;
dad.reads.paper; // 结果为true
```

这种属性复制模式比较简单且得到了广泛运用，例如，Firebug（使用JavaScript编写的Firefox扩展插件）中具有一个名为extend()的方法，该方法就可以实现浅复制，而jQuery库中的extend()则可创建深度复制的副本。YUI3提供了一个名为Y.clone()的方

法，该方法也可创建深度复制的副本，并且还通过将函数绑定到子对象以复制出函数（更多有关绑定的信息请参见本章后面的部分）。

值得注意的是，在本模式中根本没有涉及任何原型，本模式仅与对象以及它们自身的属性相关。

混入

可以针对这种通过属性复制实现继承的思想做进一步的扩展，现在让我们考虑一种“mix-in”（混入）模式。mix-in模式并不是复制一个完整的对象，而是从多个对象中复制出任意的成员并将这些成员组合成一个新的对象。

mix-in实现比较简单，只需遍历每个参数，并且复制出传递给该函数的每个对象中的每个属性。

```
function mix() {
    var arg, prop, child = {};
    for (arg = 0; arg < arguments.length; arg += 1) {
        for (prop in arguments[arg]) {
            if (arguments[arg].hasOwnProperty(prop)) {
                child[prop] = arguments[arg][prop];
            }
        }
        return child;
}
```

现在，您有一个通用的mix-in函数，可以向它传递任意数量的对象，其结果将获得一个具有所有源对象属性的新对象。下面是一个使用示例：

```
var cake = mix(
   {eggs: 2, large: true},
   {butter: 1, salted: true},
   {flour: "3 cups"},
   {sugar: "sure!"}
);
```

在图6-10中，显示了在Firebug控制台中通过执行console.dir(cake)命令以展示新混入对象cake的属性。

注意： 如果已经学习过那些正式包含mix-in概念的编程语言，并且习惯于mix-in的概念，那么可能希望修改一个或多个父对象时可以影响其子对象，但是在本节给定的实现中并不是这样的。在这里我们仅简单循环、复制自身的属性，以及断开与父对象之间的链接。

---

butter	1
eggs	2
flour	"3 cups"
large	true
salted	true
sugar	"sure!"

图6-10：在Firebug中查看cake对象

借用方法

有时候，可能恰好仅需要现有对象其中的一个或两个方法。在想要重用这些方法的同时，但是又确实不希望与源对象形成父-子继承关系。也就是说，只想使用所需要的方法，而不希望继承所有那些永远都不会用到的其他方法。在这种情况下，可以通过使用借用方法（borrowing method）模式来实现，而这是受益于call()和apply()函数方法。您已经在本书中见到过这种模式，比如，甚至于在本章中extendDeep()函数的实现中都见到过。

如您所知，JavaScript中的函数也是对象，并且它们自身也附带着一些有趣的方法，比如apply()和call()方法。这两者之间的唯一区别在于其中一个可以接受传递给将被调用方法的参数数组，而另一个仅逐个的接受参数。可以使用这些方法以借用现有对象的功能。

```
// call() 例子
notmyobj.doStuff.call(myobj, param1, p2, p3);
// apply() 例子
notmyobj.doStuff.apply(myobj, [param1, p2, p3]);
```

在以上代码中，存在一个名为MyObj的对象，并且还知道其他名为notmyobj的对象中有一个名为doStuff()的有用方法。您无需经历继承所带来的麻烦，也无需继承myobj对象永远都不会用到的一些方法，可以仅临时性的借用方法doStuff()即可。

可以传递对象、任意参数以及借用方法，并将它们绑定到您的对象中以作为this本身的成员。从根本上说，您的对象将在一小段时间内伪装成其他对象，从而借用其所需的方法。这就像得到了继承的好处，但是却无需支付遗产税（这里的“税”以您不需要的额外属性和方法的形式出现）。

例子：借用数组的方法

本模式的一个常见实现方法是借用数组方法。

数组具有一些有用的方法，而形如arguments的类似数组的对象并不具有这些方法。因此，arguments可以借用数组的方法，比如slice()方法。如下面的例子所示：

```
function f() {
    var args = [].slice.call(arguments, 1, 3);
    return args;
}

// 例子
f(1, 2, 3, 4, 5, 6); // 返回 [2,3]
```

在这个例子中，创建一个空数组的原因只是为了使用数组的方法。此外，能够实现同样功能但是语句稍微长一点的方式是直接从Array的原型中借用方法，即使用Array.prototype.slice.call()方法。这种方式需要输入更长一点的字符，但是却可以节省创建一个空数组的工作。

借用和绑定

考虑到借用方法不是通过调用call()/apply()就是通过简单的赋值，在借用方法的内部，this所指向的对象是基于调用表达式而确定的。但是有时候，最好能够“锁定”this的值，或者将其绑定到特定对象并预先确定该对象。

让我们看下面这个例子，其中存在一个名为one的对象，且具有say()方法：

```
var one = {
    name: "object",
    say: function (greet) {
        return greet + ", " + this.name;
    }
};
// 测试
one.say('hi'); // 结果为"hi, object"
```

现在，另一个对象two中并没有say()方法，但是可以从对象one中借用该方法，如下代码所示：

```
var two = {
    name: "another object"
};
one.say.apply(two, ['hello']); // 结果为"hello, another object"
```

在上述例子中，借用的say()方法内部的this指向了two对象，因而this.name的值为“another object”。但是在什么样的场景中，应该将函数指针赋值给一个全局变量，或者将该函数作为回调函数来传递？在客户端编程中有许多事件和回调函数，因此确实发生了很多这样混淆的事情。

```
// 给变量赋值
// `this` 将指向全局变量
var say = one.say;
say('hoho'); // 结果为"hoho, undefined"

// 作为回调函数传递
var yetanother = {
    name: "Yet another object",
    method: function (callback) {
        return callback('Hola');
    }
};
yetanother.method(one.say); // 结果为"Holla, undefined"
```

在以上两种情况下，say()方法内部的this指向了全局对象，并且整个代码段都无法按照预期正常运行。为了修复（也就是说，绑定）对象与方法之间的关系，我们可以使用如下的一个简单函数：

```
function bind(o, m) {
    return function () {
        return m.apply(o, [].slice.call(arguments));
    };
}
```

这个bind()函数接受了一个对象o和一个方法m，并将两者绑定起来，然后返回另一个函数。其中，返回的函数可以通过闭包来访问o和m。因此，即使在bind()返回后，内部函数（inner function）仍然可以访问o和m，并且总是指向原始对象和方法。下面，让我们使用bind()创建一个新的函数：

```
var twosay = bind(two, one.say);
twosay('yo'); // 结果为"yo, another object"
```

正如您上面所看到的，即使twosay()以全局函数方式而创建的，但是say()方法内部的this并没有指向全局对象，实际上它指向了传递给bind()的对象two。无论您如何调用twosay()，该方法永远是绑定到对象two上。

奢侈的拥有绑定所需要付出的代价就是额外闭包的开销。

Function.prototype.bind()

ECMAScript 5中将bind()方法添加到Function.prototype，使得bind()就像apply()和call()一样简单易用。因此，可以执行如下表达式：

```
var newFunc = obj.someFunc.bind(myobj, 1, 2, 3);
```

上述表达式的含义是将someFunc()和myobj绑定在一起，并且预填充someFunc()期望的前三个参数。这也是第4章中所讨论的部分函数应用的一个例子。

当程序在ES5之前的环境中运行时，让我们看看应该如何实现Function.prototype.bind()：

```
if (typeof Function.prototype.bind === "undefined") {
   Function.prototype.bind = function (thisArg) {
       var fn = this,
           slice = Array.prototype.slice,
           args = slice.call(arguments, 1);

       return function () {
           return fn.apply(thisArg, args.concat(slice.call(arguments)));
       };
   };
}
```

这个实现可能看起来有点熟悉，它使用了部分应用并拼接了参数列表，即那些传递给bind()的参数(除了第一个以外)，以及那些传递给由bind()所返回的新函数的参数，其中该新函数将在以后被调用。下面是一个使用示例：

```
var twosay2 = one.say.bind(two);
twosay2('Bonjour'); // 结果为"Bonjour, another object"
```

在前面的例子中，除了提供了将被绑定的对象以外，并没有向bind()传递任何参数。在下面的例子，让我们传递一个参数以实现部分应用：

```
var twosay3 = one.say.bind(two, 'Enchanté');
twosay3(); // 结果为"Enchanté, another object"
```

小结

当在JavaScript中涉及继承时，有许多可供选择的方法。这些方法对于学习和理解多种不同的模式大有裨益，因为它们有助于提高您对语言的掌握程度。在本章中，您了解了几种类式继承模式以及几种现代继承模式，从而可以解决继承相关的问题。

然而，在开发过程中经常面临的继承可能并不是一个问题。其中一部分的原因在于，事实上使用的Javascript库可能以这样或那样的方式解决了该问题，而另一方面的原因在于很少需要在JavaScript中建立长而且复杂的继承链。在静态强类型语言中，继承可能是唯一复用代码的方法。在JavaScript中，经常有更简洁且优美的方法，其中包括借用方法、绑定、复制属性以及从多个对象中混入（mixing-in）属性等多种方法。

请记住，代码重用才是最终目的，而继承只是实现这一目标的方法之一。

第7章

设计模式

GoF合作出版的《设计模式》这本书提供了许多有关与面向对象软件设计中常见问题的解决方案。这些模式已经出现了相当长的一段时间，已经被证明在许多情况下都非常有用。这也是为什么需要自己熟悉并谈论这些模式的原因。

虽然这些设计模式是与语言和实现方式无关的，并且人们已经对此研究了多年，但都主要是从强类型的静态类语言的角度开展研究，比如C++ 和 Java语言。

JavaScript是一种弱类型、动态的、基于原型的语言，这种语言特性使得它非常容易、甚至是普通的方式实现其中的一些模式。

让我们先从第一个例子开始，即单体（singleton）模式，理解在与基于类的静态语言相比时，JavaScript中存在哪些区别。

单体模式

单体（Singleton）模式的思想在于保证一个特定类仅有一个实例。这意味着当您第二次使用同一个类创建新对象的时候，应该得到与第一次所创建对象完全相同对象。

但是如何将这种模式应用到JavaScript？在JavaScript中有没有类，只有对象。当您创建一个新对象时，实际上没有其他对象与其类似，因此新对象已经是单体了。使用对象字面量（object literal）创建一个简单的对象也是一个单体的例子：

```
var obj = {
    myprop:'my value'
};
```

在JavaScript中，对象之间永远不会完全相等，除非它们是同一个对象，因此即使创建一个具有完全相同成员的同类对象，它也不会与第一个对象完全相同：

```
var obj2 = {
    myprop: 'my value'
};
obj === obj2; // false
obj == obj2;  // false
```

因此，可以认为每次在使用对象字面量创建对象的时候，实际上就正在创建一个单体，并且并不涉及任何特殊语法。

注意： 请注意到有时当人们在JavaScript上下文中谈论“单体”时，他们的意思是指第5章中所讨论的模块模式。

使用new操作符

JavaScript中并没有类，因此对单体咬文嚼字的定义严格说来并没有意义。但是JavaScript中具有new语法可使用构造函数来创建对象，而且有时可能需要使用这种语法的单体实现。这种思想在于当使用同一个构造函数以new操作符来创建多个对象时，应该仅获得指向完全相同的对象的新指针。

注意： 对于在一些基于类的语言（即静态、强类型语言）中其函数不是“第一类型对象”的那些语言来说，下面讨论的主题并不是那么有用，而是更多的作为一种理论上的模仿变通方案的运用。

下面的代码片段显示了其预期行为（假定不认可多元宇宙的观点，并且仅接受外在世界只有一个宇宙的观点）：

```
var uni  = new Universe();
var uni2 = new Universe();
uni === uni2; // 结果为true
```

在上面这个例子中，uni对象仅在第一次调用构造函数时被创建。在第二次（以及第三次、第四次等）创建时将会返回同一个uni对象。这就是为什么uni=== uni2，因为它们本质上是指向同一个对象的两个引用。那么如何在JavaScript中实现这种模式呢？

需要Universe构造函数缓存该对象实例this，以便当第二次调用该构造函数时能够创建并返回同一个对象。有多种选择可以实现这一目标：

- 可以使用全局变量来存储该实例。但是并不推荐使用这种方法，因为在一般原则

下，全局变量是有缺点的。此外，任何人都能够覆盖该全局变量，即使是意外事件。因此，让我们不要再进一步讨论这种方法。

- 可以在构造函数的静态属性中缓存该实例。JavaScript中的函数也是对象，因此它们也可以有属性。您可以使用类似Universe.instance的属性并将实例缓存在该属性中。这是一种很好的实现方法，这种简洁的解决方案唯一的缺点在于instance属性是公开可访问的属性，在外部代码中可能会修改该属性，以至于让您丢失了该实例。
- 可以将该实例包装在闭包中。这样可以保证该实例的私有性并且保证该实例不会被构造函数之外的代码所修改，其代价是带来了额外的闭包开销。

下面，让我们来看看第二种和第三种方法的实现示例。

静态属性中的实例

下面的代码是一个在Universe构造函数的静态属性中缓存单个实例的例子：

```
function Universe() {

    // 我们有一个现有的实例吗？
    if (typeof Universe.instance === "object") {
        return Universe.instance;
    }

    // 正常进行
    this.start_time = 0;
    this.bang = "Big";

    // 缓存
    Universe.instance = this;

    // 隐式返回
    // return this;
}

// 测试
var uni  = new Universe();
var uni2 = new Universe();
uni === uni2; // 结果为true
```

正如您所看到的，这是一个非常直接的解决方法，其唯一的缺点在于其instance属性是公开的。虽然其他代码不太可能会无意中修改该属性，但是仍然存在这种可能性。

闭包中的实例

另外一种实现类似于类的单体方法是采用闭包来保护该单个实例。可以通过使用在第5章中所讨论的私有静态成员模式实现这种单体模式。这里的秘诀就是重写构造函数：

```
function Universe() {
    // 缓存实例
    var instance = this;

    // 正常进行
    this.start_time = 0;
    this.bang = "Big";

    // 重写该构造函数
    Universe = function () {
        return instance;
    };
}

// 测试
var uni  = new Universe();
var uni2 = new Universe();
uni === uni2; // 结果为true
```

在上述代码运行时，当第一次调用原始构造函数时，它像往常一样返回this。然后，在第二次、第三次调用时，依此类推，此时将执行重写构造函数。该重写构造函数通过闭包访问了私有instance变量，并且仅简单的返回了该instance。

这个实现实际上是来自于第4章的自定义函数（self-defining function）模式的另一个例子。而这种方法的缺点我们已经在第4章中讨论过，主要在于重写构造函数［在本例中，也就是构造函数Universe()］会丢失所有在初始定义和重定义时刻之间添加到它里面的属性。在这里的特定情况下，任何添加到Universe()的原型中的对象都不会存在指向由原始实现所创建实例的活动链接。

通过下面的一些测试，可以看到这个问题：

```
// 向原型添加属性
Universe.prototype.nothing = true;

var uni  = new Universe();

// 在创建初始化对象之后，
// 再次向该原型添加属性
Universe.prototype.everything = true;

var uni2 = new Universe();
```

开始测试：

```
// 仅有最初的原型
// 链接到对象上

uni.nothing; // 结果为true
uni2.nothing; // 结果为true
```

```
uni.everything; // 结果为undefined
uni2.everything; // 结果为undefined

// 结果看上去是正确的:
uni.constructor.name; // 结果为"Universe"

// 但这是很奇怪的:
uni.constructor === Universe; // 结果为false
```

之所以uni.constructor不再与Universe()构造函数相同，是因为uni.constructor仍然指向了原始的构造函数，而不是重新定义的那个构造函数。

从需求上来说，如果需要使原型和构造函数指针按照预期的那样运行，那么可以通过做一些调整来实现这个目标：

```
function Universe() {

    // 缓存实例
    var instance;

    // 重写构造函数
    Universe = function Universe() {
        return instance;
    };

    // 保留原型属性
    Universe.prototype = this;

    // 实例
    instance = new Universe();

    // 重置构造函数指针
    instance.constructor = Universe;

    // 所有功能
    instance.start_time = 0;
    instance.bang = "Big";

    return instance;
}
```

现在，所有的测试用例应该按预期运行：

```
// 更新原型并创建实例
Universe.prototype.nothing = true; // 结果为true
var uni = new Universe();
Universe.prototype.everything = true; // 结果为true
var uni2 = new Universe();

// 它们是相同的实例
uni === uni2; // 结果为true

// 无论这些原型属性是何时定义的，
```

```
// 所有原型属性都起作用
uni.nothing && uni.everything && uni2.nothing && uni2.everything; // 结果为true
// 正常属性起作用
uni.bang; // 结果为"Big"
// 该构造函数指向正确
uni.constructor === Universe; // 结果为true
```

另一种解决方案也是将构造函数和实例包装在即时函数中。在第一次调用构造函数时，它会创建一个对象，并且使得私有instance指向该对象。从第二次调用之后，该构造函数仅返回该私有变量。通过这个新的实现方式，前面所有代码片段的测试也都会按照预期运行。

```
var Universe;

(function () {

    var instance;

    Universe = function Universe() {

        if (instance) {
            return instance;
        }

        instance = this;

        // 所有功能
        this.start_time = 0;
        this.bang = "Big";

    };

}());
```

工厂模式

设计工厂模式的目的是为了创建对象。它通常在类或者类的静态方法中实现，具有下列目标：

- 当创建相似对象时执行重复操作。
- 在编译时不知道具体类型（类）的情况下，为工厂客户提供一种创建对象的接口。

其中，在静态类语言中第二点显得更为重要，因为静态语言创建类的实例是非平凡的（non-trivial），即事先(在编译时)并不知道实例所属的类。而在JavaScript中，这部分目标实现起来相当容易。

通过工厂方法（或类）创建的对象在设计上都继承了相同的父对象这个思想，它们都是实现专门功能的特定子类。有时候公共父类是一个包含了工厂方法的同一个类。

让我们看一个实现示例：

- 公共父构造函数CarMaker。
- 一个名为factory()的CarMaker的静态方法，该方法创建car对象。
- 从CarMaker继承的专门构造函数CarMaker.Compact、CarMaker.SUV和CarMaker.Convertible。所有这些构造函数都被定义为父类的静态属性，以便保证全局命名空间免受污染，因此我们也知道了当需要这些构造函数的时候可以在哪里找到它们。

让我们先来看看如何使用这个已经完成的实现：

```
var corolla  = CarMaker.factory('Compact');
var solstice = CarMaker.factory('Convertible');
var cherokee = CarMaker.factory('SUV');
corolla.drive();  // 结果为"Vroom, I have 4 doors"
solstice.drive(); //结果为"Vroom, I have 2 doors"
cherokee.drive(); //结果为"Vroom, I have 17 doors"
```

其中，这一部分：

```
var corolla = CarMaker.factory('Compact');
```

可能是工厂模式中最易辨别的部分。现在看到的工厂方法接受在运行时以字符串形式指定的类型，然后创建并返回所请求类型的对象。代码中看不到任何具有new或对象字面量的构造函数，其中仅有一个函数根据字符串所指定类型来创建对象。

下面是工厂模式的实现示例，这将会使得前面的代码片段正常运行：

```
// 父构造函数
function CarMaker() {}

// a method of the parent
CarMaker.prototype.drive = function () {
    return "Vroom, I have " + this.doors + " doors";
};

// 静态工厂方法
CarMaker.factory = function (type) {
    var constr = type,
        newcar;

    // 如果构造函数不存在，则发生错误
    if (typeof CarMaker[constr] !== "function") {
        throw {
            name: "Error",
            message: constr + " doesn't exist"
        };
    }
```

```
        // 在这里，构造函数是已知存在的
        // 我们使得原型继承父类，但仅继承一次
        if (typeof CarMaker[constr].prototype.drive !== "function") {
            CarMaker[constr].prototype = new CarMaker();
        }
        // 创建一个新的实例
        newcar = new CarMaker[constr]();
        // 可选择性的调用一些方法然后返回...
        return newcar;
    };

    // 定义特定的汽车制造商
    CarMaker.Compact = function () {
        this.doors = 4;
    };
    CarMaker.Convertible = function () {
        this.doors = 2;
    };
    CarMaker.SUV = function () {
        this.doors = 24;
    };
```

实现该工厂模式时并没有特别的困难。所有需要做的就是寻找能够创建所需类型对象的构造函数。在这种情况下，简洁的命名习惯可用于将对象类型映射到创建该对象的构造函数中。继承部分仅是可以放进工厂方法的一个公用重复代码片段的范例，而不是对每种类型的构造函数的重复。

内置对象工厂

而对于“自然工厂”的例子，可以考虑内置的全局Object()构造函数。它也表现出工厂的行为，因为它根据输入类型而创建不同的对象。如果传递一个原始数字，那么它能够在后台以Number()构造函数创建一个对象。对于字符串和布尔值也同样成立。对于任何其他值，甚至包括无输入的值，它都会创建一个常规的对象。

下面是该行为的一些例子和测试。请注意，无论使用new操作符与否，都可以调用Object()：

```
    var o = new Object(),
        n = new Object(1),
        s = Object('1'),
        b = Object(true);

    // test
    o.constructor === Object;  //结果为true
    n.constructor === Number;  //结果为true
    s.constructor === String;  //结果为true
    b.constructor === Boolean; //结果为true
```

事实上，Object()也是一个实际用途不大的工厂，值得将它作为例子而提及的原因在于它是我们身边常见的工厂模式。

迭代器模式

在迭代器模式中，通常有一个包含某种数据集合的对象。该数据可能存储在一个复杂数据结构内部，而要提供一种简单的方法能够访问数据结构中每个元素。对象的消费者并不需要知道如何组织数据，所有需要做的就是取出单个数据进行工作。

在迭代器模式中，对象需要提供一个next()方法。依次调用next()必须返回下一个连续的元素。当然，在特定数据结构中，“下一个”所代表的意义是由您来决定的。

假定对象名为agg，可以在类似下面这样的一个循环中通过简单调用next()即可访问每个数据元素：

```
var element;
while (element = agg.next()) {
    // 处理该元素...
    console.log(element);
}
```

在迭代器模式中，聚合对象通常还提供了一个较为方便的hasNext()方法，因此，该对象的用户可以使用该方法来确定是否已经到达了数据的末尾。此外，还有另一种顺序访问所有元素的方法，这次是使用hasNext()，其用法如下所示：

```
while (agg.hasNext()) {
    // 处理下一个元素...
    console.log(agg.next());
}
```

现在已经有了用例，让我们来看看如何实现这样的聚合对象。

当实现迭代器模式时，私下的存储数据和指向下一个可用元素的指针（索引）是很有意义的。为了演示一个实现示例，让我们假定数据只是普通数组，而“特殊”的检索下一个连续元素的逻辑为返回每隔一个的数组元素。

```
var agg = (function () {

    var index = 0,
        data = [1, 2, 3, 4, 5],
        length = data.length;

    return {

        next: function () {
            var element;
```

```
            if (!this.hasNext()) {
                return null;
            }

            element = data[index];
            index = index + 2;
            return element;
        },

        hasNext: function () {
            return index < length;
        }

    };
}());
```

为了提供更简单的访问方式以及多次迭代数据的能力，您的对象可以提供额外的便利方法：

rewind()
: 重置指针到初始位置。

current()
: 返回当前元素，因为不可能在不前进指针的情况下使用next()执行该操作。

实现这些方法不存在任何困难：

```
var agg = (function () {

    // [snip...]

    return {

        // [snip...]

        rewind: function () {
            index = 0;
        },
        current: function () {
            return data[index];
        }

    };
}());
```

现在测试该迭代器：

```
// 这个循环记录1，然后3，接着5
while (agg.hasNext()) {
    console.log(agg.next());
}

// 回退
```

```
agg.rewind();
console.log(agg.current()); // 1
```

输出结果将记录在控制台中：即依次输出1，3，5（从循环中），并且最后输出1(在回绕以后)。

装饰者模式

在装饰者模式中，可以在运行时动态添加附加功能到对象中。当处理静态类时，这可能是一个挑战。在JavaScript中，由于对象是可变的，因此，添加功能到对象中的过程本身并不是问题。

装饰者模式的一个比较方便的特征在于其预期行为的可定制和可配置特性。可以从仅具有一些基本功能的普通对象开始，然后从可用装饰资源池中选择需要用于增强普通对象的那些功能，并且按照顺序进行装饰，尤其是当装饰顺序很重要的时候。

用法

让我们仔细看一下该模式使用示例。假设正在开发一个销售商品的Web应用，每一笔新销售都是一个新的sale对象。该sale对象“知道”有关项目的价格，并且可以通过调用sale.getPrice()方法返回其价格。根据不同的情况，可以用额外的功能装饰这个对象。试想这样一个场景，销售客户在加拿大的魁北克省。在这种情况下，买方需要支付联邦税和魁北克省税。遵循这种装饰者模式，您会说需要使用联邦税装饰者和魁北克税装饰者来“装饰”这个对象。然后，可以用价格格式化功能装饰该对象。这种应用场景看起来如下所示：

```
var sale = new Sale(100);          // 该价格为 100 美元
sale = sale.decorate('fedtax');    // 增加联邦税
sale = sale.decorate('quebec');    // 增加省级税
sale = sale.decorate('money');     // 格式化为美元货币形式
sale.getPrice();                   // "$112.88"
```

在另一种情况下，买方可能在一个没有省税的省份，并且您可能也想使用加元的形式对其价格进行格式化，因此，您可以按照下列方式这样做：

```
var sale = new Sale(100);          // 该价格为100美元
sale = sale.decorate('fedtax');    // 增加联邦税
sale = sale.decorate('cdn');       // 格式化为CDN形式
sale.getPrice();                   // "CDN$ 105.00"
```

正如您所看到的，这是一种非常灵活的方法，可用于增加功能以及调整运行时对象。让我们来看看如何处理该模式的实现。

实现

实现装饰者模式的其中一种方法是使得每个装饰者成为一个对象，并且该对象包含了应该被重载的方法。每个装饰者实际上继承了目前已经被前一个装饰者进行增强后的对象。每个装饰方法在uber（继承的对象）上调用了同样的方法并获取其值，此外它还继续执行了一些操作。

最终的效果是，当您在第一个用法示例中执行sale.getPrice()时，调用了money装饰者的方法（见图7-1）。但由于每个装饰方法首先调用父对象的方法，money的getPrice()将会首先调用quebec的getPrice()，这又需要依次调用fedtax的getPrice()等。该调用链一路攀升到由Sale()构造函数所实现的原始未经装饰的getPrice()。

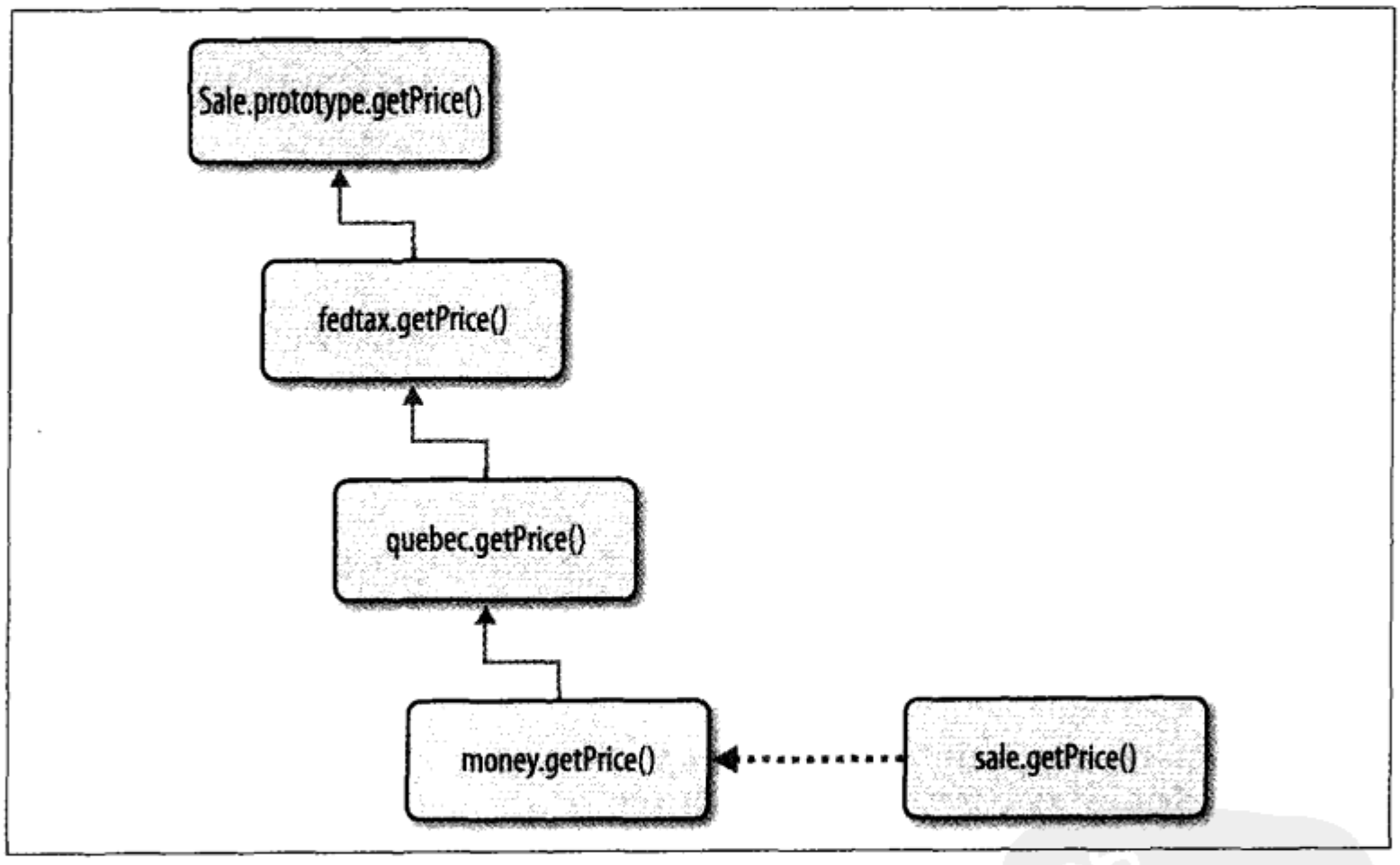

图7-1：装饰者模式的实现

该实现是从一个构造函数和一个原型方法开始的：

```
function Sale(price) {
    this.price = price || 100;
}
Sale.prototype.getPrice = function () {
    return this.price;
};
```

装饰者对象都将以构造函数的属性这种方式来实现：

```
Sale.decorators = {};
```

让我们看一个装饰者示例。它是一个实现了自定义getPrice()方法的对象。请注意，该方法首先会从父对象的方法中获取值，然后再修改该值：

```
Sale.decorators.fedtax = {
    getPrice: function () {
        var price = this.uber.getPrice();
        price += price * 5 / 100;
        return price;
    }
};
```

同样地，我们可以实现其他装饰者，其数量由需求来定。它们可以是核心Sale()功能的扩展，也可以实现成插件。它们甚至可以“驻留”在其他文件中，并且可被第三方开发人员所开发和共享。

```
Sale.decorators.quebec = {
    getPrice: function () {
        var price = this.uber.getPrice();
        price += price * 7.5 / 100;
        return price;
    }
};

Sale.decorators.money = {
    getPrice: function () {
        return "$" + this.uber.getPrice().toFixed(2);
    }
};

Sale.decorators.cdn = {
    getPrice: function () {
        return "CDN$ " + this.uber.getPrice().toFixed(2);
    }
};
```

最后，让我们看看称之为decorate()的“神奇”方法，它可以将所有的块拼接在一起。请记住，调用该方法的方式如下所示：

```
sale = sale.decorate('fedtax')
```

“fedtax”字符串将对应于Sale.decorators.fedtax中实现的对象。新装饰的对象newobj将继承目前我们所拥有的对象(无论是原始对象，还是已经添加了最后的装饰者的对象)，这也就是对象this。为了完成继承部分的代码，我们使用了前一章中的临时构造函数模式。首先，我们也设置了newobj的uber属性，以便于子对象可以访问父对象。然后，我们从装饰者中将所有的额外属性复制到新装饰的对象newobj中。最后，返回newobj，而在我们具体的用法例子中，它实际上成为了更新的sale对象。

```
Sale.prototype.decorate = function (decorator) {
    var F = function () {},
        overrides = this.constructor.decorators[decorator],
        i, newobj;
    F.prototype = this;
    newobj = new F();
    newobj.uber = F.prototype;
    for (i in overrides) {
        if (overrides.hasOwnProperty(i)) {
            newobj[i] = overrides[i];
        }
    }
    return newobj;
};
```

使用列表实现

现在让我们探讨一个稍微不同的实现方法，它利用了JavaScript语言的动态性质，并且根本不需要使用继承。此外，并不是使每个装饰方法调用链中前面的方法，我们可以简单将前面方法的结果作为参数传递到下一个方法。

这种实现方法还可以很容易的支持反装饰（undecorating）或撤销装饰，这意味着可以简单地从装饰者列表中删除一个项目。

下面的用法示例将略微简单一点，这是由于我们没有将从decorate()返回的值赋给对象。在这个实现中，decorate()并没有对该对象执行任何操作，它只是将返回的值追加到列表中：

```
var sale = new Sale(100); // 该价格为 100 美元
sale.decorate('fedtax');  // 增加联邦税
sale.decorate('quebec');  // 增加省级税
sale.decorate('money');   // 格式化为美元货币形式
sale.getPrice();          // "$112.88"
```

现在，Sale()构造函数中有一个装饰者列表并以此作为自身的属性：

```
function Sale(price) {
    this.price = (price > 0) || 100;
    this.decorators_list = [];
}
```

可用装饰者将再次以Sale.decorators属性的方式实现。请注意，现在getPrice()方法变得更为简单了，这是因为它们并没有调用父对象的getPrice()以获得中间结果，而这个结果将作为参数传递给它们：

```
Sale.decorators = {};
Sale.decorators.fedtax = {
```

```
    getPrice: function (price) {
        return price + price * 5 / 100;
    }
};
Sale.decorators.quebec = {
    getPrice: function (price) {
        return price + price * 7.5 / 100;
    }
};
Sale.decorators.money = {
    getPrice: function (price) {
        return "$" + price.toFixed(2);
    }
};
```

在下面的代码中，有趣的部分发生在父对象的decorate()和getPrice()方法中。在以前的实现中，decorate()具有一定的复杂性，而getPrice()却是相当简单。然而，在本实现中却采用了恰好相反的方式：decorate()仅用于追加列表，而getPrice()却完成所有工作。这些工作包括遍历当前添加的装饰者以及调用每个装饰者的getPrice()方法、传递从前一个方法中获得的结果。

```
Sale.prototype.decorate = function (decorator) {
    this.decorators_list.push(decorator);
};

Sale.prototype.getPrice = function () {
    var price = this.price,
        i,
        max = this.decorators_list.length,
        name;
    for (i = 0; i < max; i += 1) {
        name = this.decorators_list[i];
        price = Sale.decorators[name].getPrice(price);
    }
    return price;
};
```

装饰者模式的第二种实现方法更为简单，并且也不涉及继承。此外，装饰方法也是非常简单的。所有这些工作都是通过“同意”被装饰的那个方法来完成。在这个实现示例中，getPrice()是唯一允许装饰的方法。如果想拥有更多可以被装饰的方法，那么每个额外的装饰方法都需要重复遍历装饰者列表这一部分的代码。然而，这很容易抽象成一个辅助方法，通过它来接受方法并使其成为“可装饰”的方法。在这样的实现中，sale中的decorators_list属性变成了一个对象，且该对象中的每个属性都是以装饰对象数组中的方法和值命名。

策略模式

策略模式支持您在运行时选择算法。代码的客户端可以使用同一个接口来工作，但是它却根据客户正在试图执行任务的上下文，从多个算法中选择用于处理特定任务的算法。

使用策略模式的其中一个例子是解决表单验证的问题。可以创建一个具有validate()方法的验证器（validator）对象。无论表单的具体类型是什么，该方法都将会被调用，并且总是返回相同的结果，一个未经验证的数据列表以及任意的错误消息。

但是根据具体的表单形式以及待验证的数据，验证器的客户端可能选择不同类型的检查方法。验证器将选择最佳的策略（strategy）以处理任务，并且将具体的数据验证委托给适当的算法。

数据验证示例

假设有以下数据块，它可能来自于网页上的一个表单，而您需要验证它是否有效：

```
var data = {
    first_name: "Super",
    last_name: "Man",
    age: "unknown",
    username: "o_O"
};
```

在这个具体的例子中，为了使验证器知道什么是最好的策略，首先需要配置该验证器，并且设置认为是有效的且可接受的规则。

比如说在表单验证中，您对姓氏不做要求且接受任意字符作为名字，但是要求年龄必须为数字，并且用户名中仅出现字母和数字且无特殊符号。该配置如下所示：

```
validator.config = {
    first_name: 'isNonEmpty',
    age: 'isNumber',
    username: 'isAlphaNum'
};
```

现在，validator对象已经配置完毕并可用于数据处理，您可以调用validator对象的validate()方法并将任意验证错误信息打印到控制台中：

```
validator.validate(data);
if (validator.hasErrors()) {
    console.log(validator.messages.join("\n"));
}
```

上述语句将会打印出下列错误消息：

```
Invalid value for *age*, the value can only be a valid number, e.g. 1, 3.14 or 2010
Invalid value for *username*, the value can only contain characters and numbers, no
special
symbols
```

现在，让我们看看验证程序是如何实现该validator的。用于检查的可用算法也是对象，它们具有一个预定义的接口，提供了一个validate()方法和一个可用于提示错误消息的一行帮助信息。

```
// 非空值的检查
validator.types.isNonEmpty = {
    validate: function (value) {
        return value !== "";
    },
    instructions: "the value cannot be empty"
};
};

// 检查值是否是一个数字
validator.types.isNumber = {
    validate: function (value) {
        return !isNaN(value);
    },
    instructions: "the value can only be a valid number, e.g. 1, 3.14 or 2010"
};

// 检查该值是否只包含字母和数字
validator.types.isAlphaNum = {
    validate: function (value) {
        return !/[^a-z0-9]/i.test(value);
    },
    instructions: "the value can only contain characters and numbers, no special symbols"
};
```

最后，核心的validator对象如下所示：

```
var validator = {

    // 所有可用的检查
    types: {},

    // 在当前验证会话中的
    // 错误消息
    messages: [],

    // 当前验证配置
    //名称: 验证类型
    config: {},

    // 接口方法
    // `data` 为键－值对
```

```
validate: function (data) {

    var i, msg, type, checker, result_ok;

    // 重置所有消息
    this.messages = [];

    for (i in data) {

        if (data.hasOwnProperty(i)) {

            type = this.config[i];
            checker = this.types[type];

            if (!type) {
                continue; //不需要验证
            }
            if (!checker) { // uh-oh
                throw {
                    name: "ValidationError",
                    message: "No handler to validate type " + type
                };
            }

            result_ok = checker.validate(data[i]);
            if (!result_ok) {
                msg = "Invalid value for *" + i + "*, " + checker.instructions;
                this.messages.push(msg);
            }
        }
        return this.hasErrors();
    },

    // 帮助程序
    hasErrors: function () {
        return this.messages.length !== 0;
    }
};
```

正如您所看到的，validator对象是通用的，可以像这样将其保存下来以用于验证用例。增强validator对象的方法是添加更多的类型检查。如果在多个页面中使用它，很快就会有一个优良的特定检查集合。以后，针对每个新的用例，所需要做的就是配置该验证器并运行validate()方法。

外观模式

外观（facade）模式是一种简单的模式，它为对象提供了一个可供选择的接口。这是一种非常好的设计实践，可保持方法的简洁性并且不会使它们处理过多的工作。如果原来有许多接受多个参数的uber方法，相比而言，按照本实现方法，最终将会创建更多数量

的方法。有时候，两个或更多的方法可能普遍的被一起调用。在这样的情况下，创建另一个方法以包装重复的方法调用是非常有意义的。

例如，当处理浏览器事件时，您有以下方法：

`stopPropagation()`
: 中止事件以避免其冒泡上升到父节点。

`preventDefault()`
: 阻止浏览器执行默认动作（例如，阻止下面的链接或提交表单）。

以上是两个单独的方法且各自具有不同的目标，它们之间应保持互相独立，但在同一时间，它们经常被一起调用。为此，并不需要在程序中到处复制这两个方法调用，可以创建一个外观方法从而同时调用这两个方法：

```
var myevent = {
    // ...
    stop: function (e) {
        e.preventDefault();
        e.stopPropagation();
    }
    // ...
};
```

外观模式也非常适合于浏览器脚本处理，据此可将浏览器之间的差异隐藏在外观之后。继续返回到前面的例子，可以添加代码来处理在IE的事件API中的差异。

```
var myevent = {
    // ...
    stop: function (e) {
        // 其他
        if (typeof e.preventDefault === "function") {
            e.preventDefault();
        }
        if (typeof e.stopPropagation === "function") {
            e.stopPropagation();
        }
        // IE 浏览器
        if (typeof e.returnValue === "boolean") {
            e.returnValue = false;
        }
        if (typeof e.cancelBubble === "boolean") {
            e.cancelBubble = true;
        }
    }
    // ...
};
```

外观模式对于重新设计和重构的工作也很有帮助。当需要替换一个具有不同实现的对象

时，不得不花费一段时间对它重新进行修改(这是一个复杂的对象)，而且同时还要编写使用该对象的新代码。通过使用外观模式，可以首先考虑新对象的API，然后继续在原有对象的前面创建一个外观。这样，当您着手完全取代原有对象的时候，仅需修改更少的客户端代码，这是由于任何最新的客户端代码都已经使用了这个新API。

代理模式

在代理设计模式中，一个对象充当另一个对象的接口。它与外观模式的区别之处在于，在外观模式中您所拥有的是合并了多个方法调用的便利方法。代理则介于对象的客户端和对象本身之间，并且对该对象的访问进行保护。

这种模式可能看起来像是额外的开销，但是出于性能因素的考虑它却非常有用。代理充当了某个对象（也称为“本体对象”）的守护对象，并且试图使本体对象做尽可能少的工作。

使用这种模式的其中一个例子是我们可以称为延迟初始化（lazy initialization）的方法。试想一下，假设初始化本体对象开销非常大，而恰好又在客户端初始化该本体对象以后，应用程序实际上却从来没有使用过它。在这种情况下，代理可以通过替换本体对象的接口来解决这个问题。代理接收初始化请求，但是直到该本体对象明确的将被实际使用之前，代理从不会将该请求传递给本体对象。

图7-2举例说明了这种情况，即首先由客户端发出一个初始化请求，然后代理以一切正常作为响应，但实际上却并没有将该消息传递到本体对象，直到客户端明显需要本体对象完成一些工作的时候。只有那个时候，代理才将两个消息一起传递。

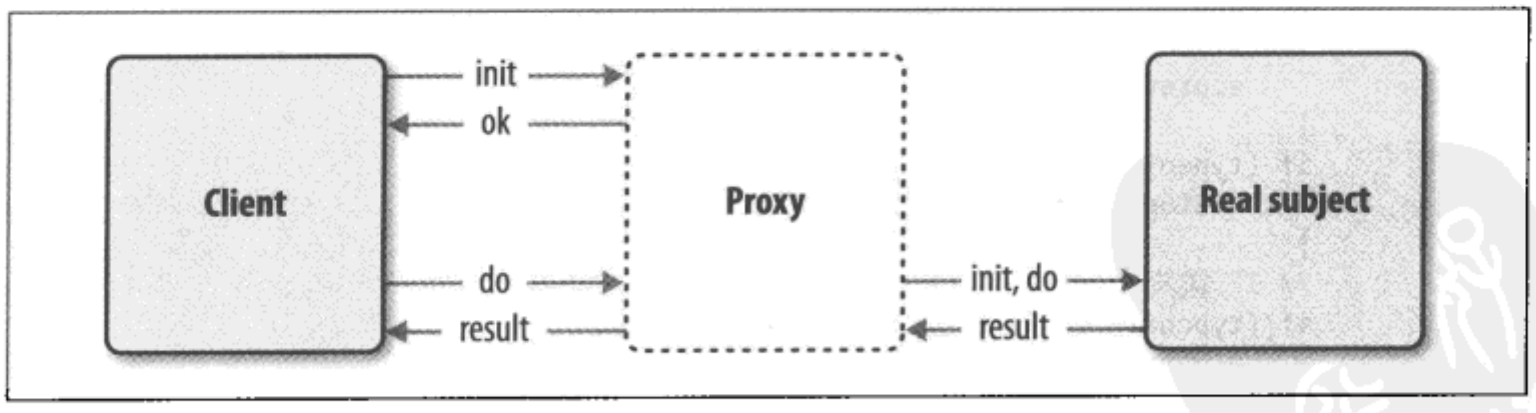

图7-2：当通过代理时客户端可本体对象之间的关系

范例

当本体对象执行一些开销很大的操作时，代理模式就显得非常有用。在Web应用中，可

以执行的开销最大的操作就是网络请求，因此，尽可能合并更多的HTTP请求就显得非常重要。让我们举一个例子，该例子执行了上述操作并演示代理模式所起的作用。

视频展开

假定我们有一个可以播放选定艺术家视频的小应用程序（见图7-3）。实际上，您可以将该在线示例用做娱乐，并且还可以在网址*http://www.jspatterns.com/book/7/proxy.html*查看其代码。

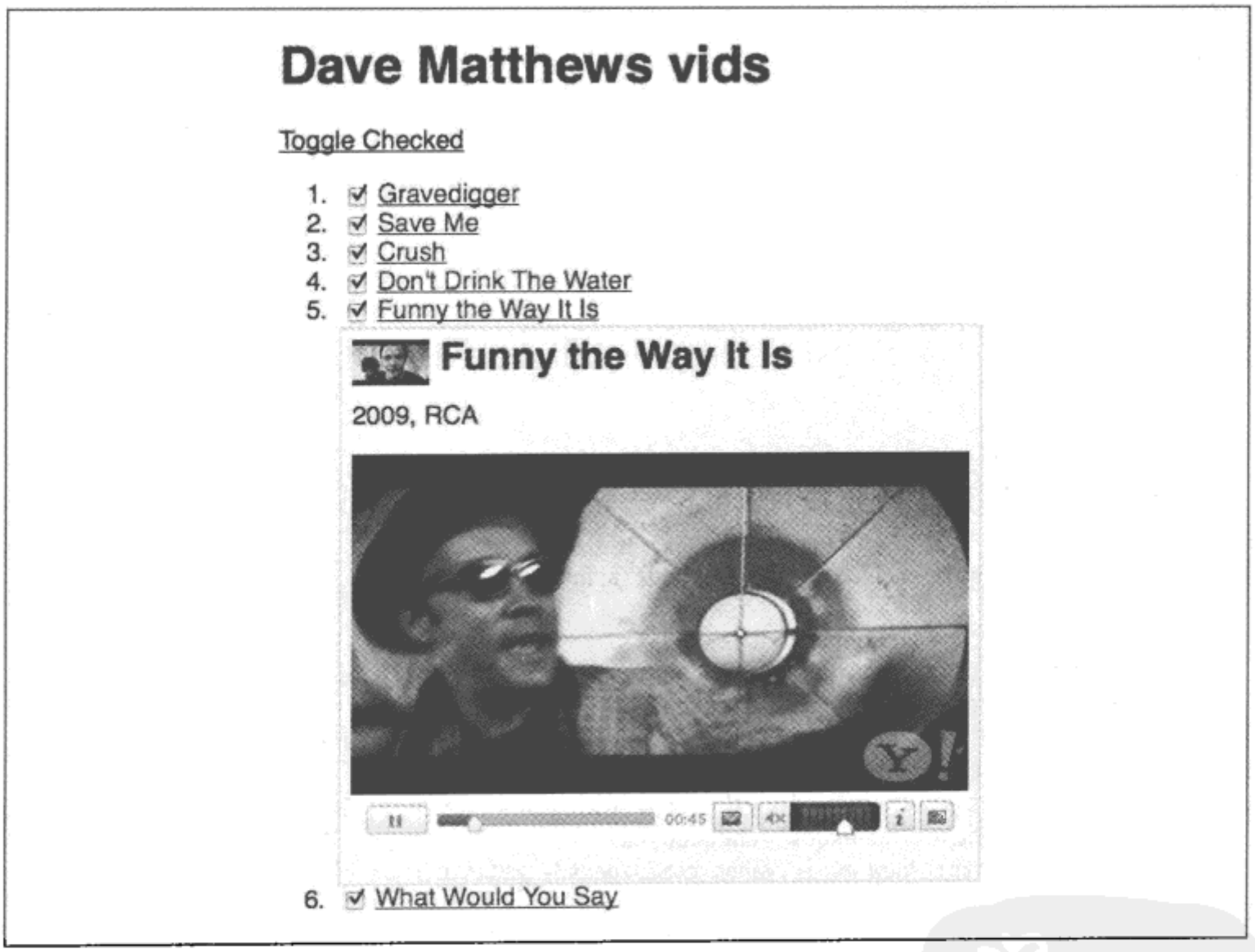

图7-3：视频展开操作

在该页面中有一个视频标题的清单。当用户点击一个视频标题时，该标题下面的区域将展开以显示有关该视频的更多信息，并且还能启用视频播放功能。详细的视频信息和网址并不是该页面的一部分，这需要通过建立Web服务调用以进行检索才能获得这些信息。Web服务可以接受以多个视频ID作为参数的查询，因此我们可以通过构造更少的HTTP请求数量并且每次检索多个视频的数据，从而加速该应用程序。

我们的应用程序支持同时展开多个（或全部）视频信息，因此这是合并Web服务请求的一个完美机会。

没有代理的情况

在该应用程序中，主要的“参与者”有两个对象：

Videos
: 负责展开/折叠［方法videos.getInfo()］信息区域（info areas）以及播放视频（方法videos.getPlayer）。

http
: 通过调用方法http.makeRequest()来负责与服务器的通信。

当没有代理的时候，videos.getInfo()将针对每个视频调用一次http.makeRequest()。当我们添加一个代理时，它将成为一个新的称之为proxy的参与者，它位于对象videos和对象http之间，并且将调用委托给makeRequest()，此外还在可能的情况下合并这些调用。

下面，让我们首先来看看在没有代理的情况下的代码，然后再添加代理以提高应用程序的响应能力。

HTML

下面的HTML代码只是一个链接列表：

```
<p><span id="toggle-all">Toggle Checked</span></p>
<ol id="vids">
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--2158073">Gravedigger</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--4472739">Save Me</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--45286339">Crush</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--2144530">Don't Drink The Water</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--217241800">Funny the Way It Is</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--2144532">What Would You Say</a></li>
</ol>
```

事件处理程序

现在让我们来查看该事件处理程序。首先定义下面便利的简写函数$：

```
var $ = function (id) {
    return document.getElementById(id);
};
```

下面代码中使用了事件委托（event delegation，有关此模式信息请参见第8章）模式，让我们使用单个函数来处理在id为“vids”（即id="vids"）的有序列表中出现的所有点击。

```
$('vids').onclick = function (e) {
    var src, id;

    e = e || window.event;
    src = e.target || e.srcElement;

    if (src.nodeName !== "A") {
        return;
    }

    if (typeof e.preventDefault === "function") {
        e.preventDefault();
    }
    e.returnValue = false;

    id = src.href.split('--')[1];

    if (src.className === "play") {
        src.parentNode.innerHTML = videos.getPlayer(id);
        return;
    }

    src.parentNode.id = "v" + id;
    videos.getInfo(id);
};
```

在以上包罗万象的点击处理程序（click handler）中，我们对其中的两次点击非常感兴趣：一个是展开/折叠信息部分［调用getInfo()］；另一个是播放视频(当目标的类名为“play”时)，这意味着信息区域已被展开，然后我们便可以调用getPlayer()。其中，这些影片的ID提取自链接hrefs。

另外一个点击处理程序对点击做出反应时将切换所有的信息片段（info section），它本质上只是再次调用了getInfo()，不过是在一个循环中调用getInfo()：

```
$('toggle-all').onclick = function (e) {

    var hrefs,
        i,
        max,
        id;

    hrefs = $('vids').getElementsByTagName('a');
    for (i = 0, max = hrefs.length; i < max; i += 1) {
        // 跳过播放链接
        if (hrefs[i].className === "play") {
            continue;
        }
        // 跳过选中节目
```

```
            if (!hrefs[i].parentNode.firstChild.checked) {
                continue;
            }

            id = hrefs[i].href.split('--')[1];
            hrefs[i].parentNode.id = "v" + id;
            videos.getInfo(id);
        }
    };
```

videos对象

该videos对象有三个方法：

getPlayer()

返回HTML请求以播放Flash视频（与本讨论无关）。

updateList()

该回调函数接收所有来自Web服务的数据，并且生成HTML代码以用于扩展信息片段中。在这个方法中根本没有什么特别有趣的事情发生。

getInfo()

该方法用于切换信息片段的可见性，并且还在http对象的调用中将updateList()作为回调函数传递出去。

下面是该对象的一个代码片段：

```
var videos = {

    getPlayer: function (id) {...},
    updateList: function (data) {...},

    getInfo: function (id) {

        var info = $('info' + id);

        if (!info) {
            http.makeRequest([id], "videos.updateList");
            return;
        }

        if (info.style.display === "none") {
            info.style.display = '';
        } else {
            info.style.display = 'none';
        }

    }
};
```

http对象

http对象只有一个方法，该方法产生JSONP格式的请求以发送到Yahoo!的YQL Web服务：

```
var http = {
    makeRequest: function (ids, callback) {
        var url = 'http://query.yahooapis.com/v1/public/yql?q=',
            sql = 'select * from music.video.id where ids IN ("%ID%")',
            format = "format=json",
            handler = "callback=" + callback,
            script = document.createElement('script');

        sql = sql.replace('%ID%', ids.join('","'));
        sql = encodeURIComponent(sql);

        url += sql + '&' + format + '&' + handler;
        script.src = url;

        document.body.appendChild(script);
    }
};
```

注意： YQL（雅虎查询语言）是一种元（meta）Web服务，它提供了一种通过使用类似SQL的语法以获取大量其他Web服务的能力，且无需研究每个服务的API。

当所有六个视频同时切换时，六个独立的请求将被发送到Web服务， YQL查询的语法如下所示：

```
select * from music.video.id where ids IN ("2158073")
```

进入代理模式

前面介绍的代码运行良好，但是我们可以更进一步。现在让proxy对象进入本场景并且接管HTTP和videos之间的通信。它试图使用一个简单的逻辑将多个请求合并起来：即一个50ms的视频缓冲区。videos对象并不直接调用HTTP服务而是调用proxy。然后，该proxy在转发该请求之前一直等待。如果来自于videos的其他调用进入了50ms的等待期，这些请求将会被合并为单个请求。50ms的延迟对于用户而言是相当不易察觉的，但是却有助于合并请求，此外，当点击“切换”（toggle）并同时展开超过一个的视频时，该延迟还能加速用户体验。此外，它还显著降低了服务器的负载，这是由于该Web服务器仅需要处理数量更少的请求。

将两个视频请求合并以后的YQL查询将如下所示：

```
select * from music.video.id where ids IN ("2158073", "123456")
```

在修改版本的代码中，其唯一的变化在于videos.getInfo()现在调用的是proxy.makeRequest()，而不是http.makeRequest()，如下所示：

```
proxy.makeRequest(id, videos.updateList, videos);
```

该proxy建立了一个队列以收集过去50ms接收到的视频ID，然后排空该队列，同时还调用http并向它提供自己的回调函数，这是由于videos.updateList()回调函数仅能处理单个数据记录。

下面是该proxy的代码：

```
var proxy = {
    ids: [],
    delay: 50,
    timeout: null,
    callback: null,
    context: null,
    makeRequest: function (id, callback, context) {

        // 加入到队列中
        this.ids.push(id);

        this.callback = callback;
        this.context  = context;

        // 设置超时时间
        if (!this.timeout) {
            this.timeout = setTimeout(function () {
                proxy.flush();
            }, this.delay);
        }
    },
    flush: function () {

        http.makeRequest(this.ids, "proxy.handler");

        // 清除超时设置和队列
        this.timeout = null;
        this.ids = [];

    },
    handler: function (data) {
        var i, max;

        // 单个视频
        if (parseInt(data.query.count, 10) === 1) {
            proxy.callback.call(proxy.context, data.query.results.Video);
            return;
        }

        // 多个视频
        for (i = 0, max = data.query.results.Video.length; i < max; i += 1) {
            proxy.callback.call(proxy.context, data.query.results.Video[i]);
        }
```

```
        }
    };
```

通过引入该代理，仅需对原始代码进行简单的修改就能够提供将多个Web服务请求合并成单个请求的能力。

图7-4和图7-5分别举例说明了生成三轮往返消息到服务器（无代理时）与使用代理时仅有一轮往返消息相比较的情景。

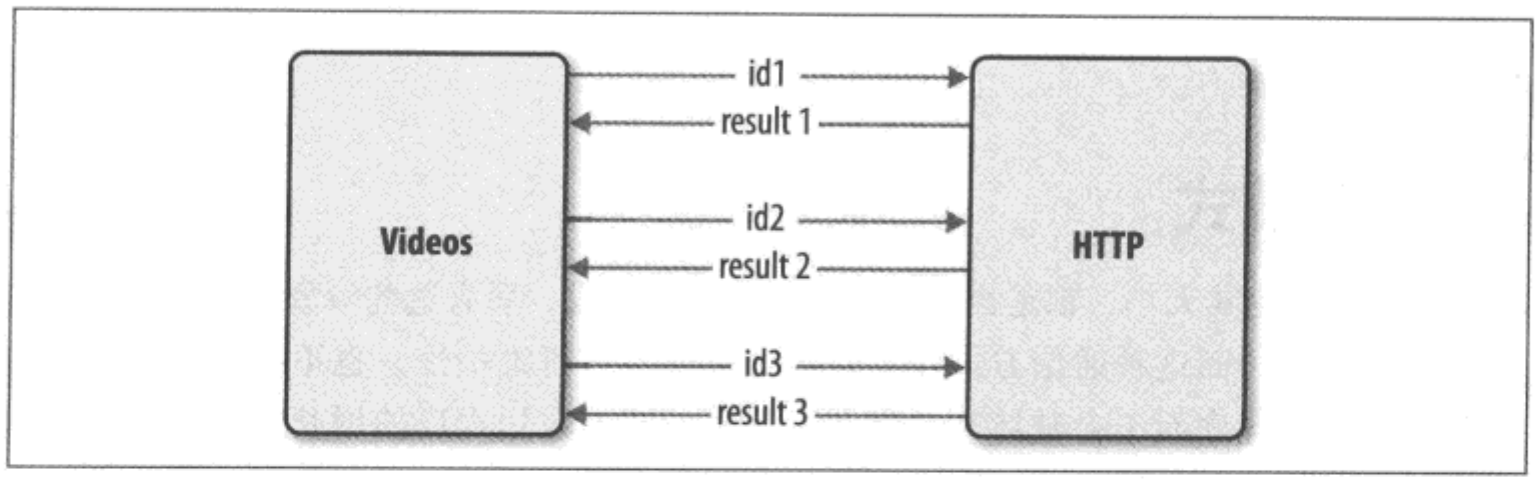

图7-4：到服务器的三轮往返消息

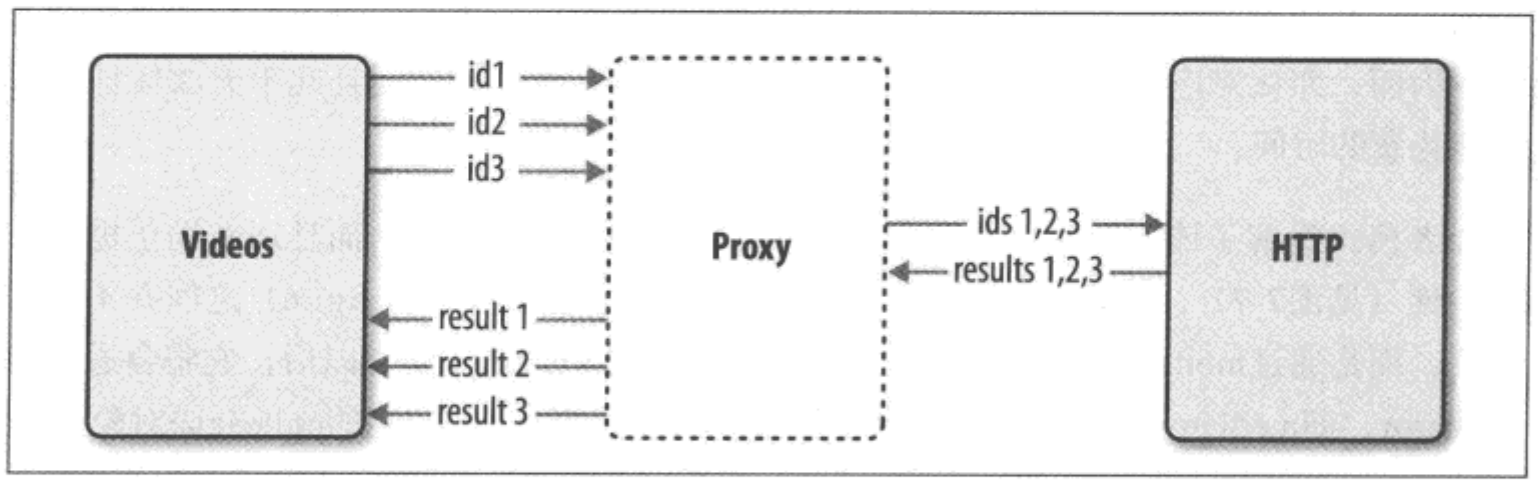

图7-5：使用代理以合并和减少到服务器的往返次数

缓存代理

在本例子中，客户端对象（videos）足够聪明到不会再次请求同一个视频的信息。但是实际情况并不总是如此。代理可以通过将以前的请求结果缓存到新的cache属性中（见图7-6），从而更进一步的保护对本体对象http的访问。那么，如果videos对象恰好再一次请求有关同一个视频ID的信息，proxy可以从缓存中取出该信息，从而节省了该网络往返消息。

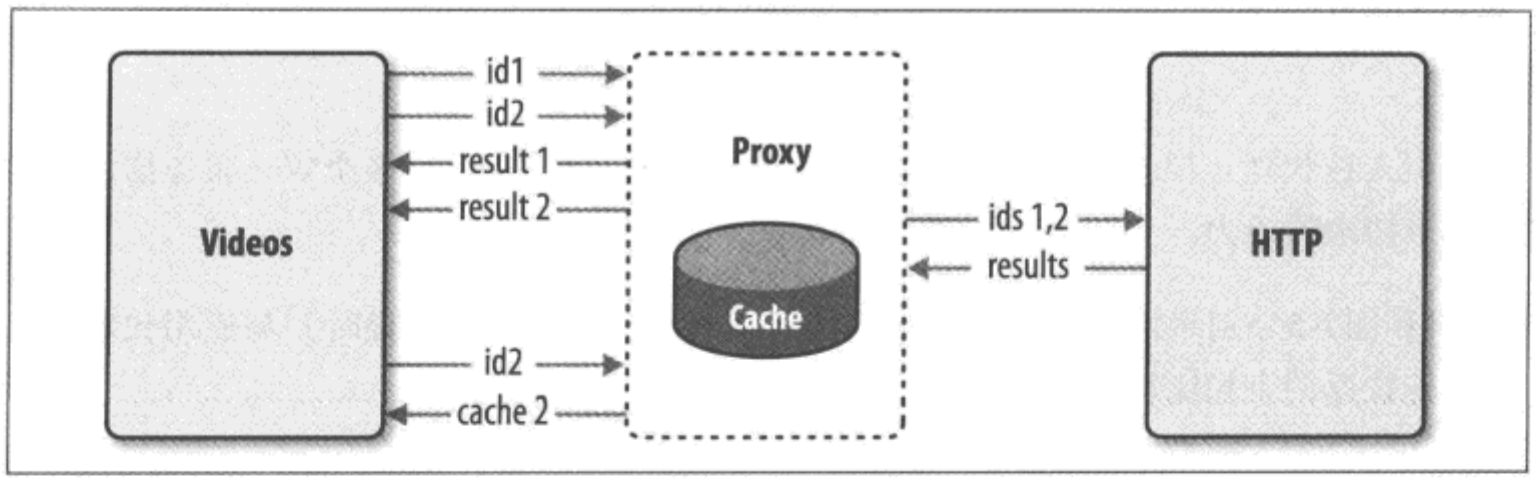

图7-6：缓存代理

中介者模式

应用程序，无论其大小，都是由一些单个的对象所组成。所有这些对象需要一种方式来实现相互通信，而这种通信方式在一定程度上不降低可维护性，也不损害那种安全的改变部分应用程序而不会破坏其余部分的能力。随着应用程序的增长，将添加越来越多的对象。然后在代码重构期间，对象将被删除或重新整理。当对象互相知道太多信息并且直接通信(调用对方的方法并改变属性)时，这将会导致产生不良的紧耦合（tight coupling）问题。当对象间紧密耦合时，很难在改变单个对象的同时不影响其他多个对象。因而，即使对应用程序进行最简单的修改也变得不再容易，而且几乎无法估计修改可能花费的时间。

中介者模式缓解了该问题并促进形成松耦合（loose coupling），而且还有助于提高可维护性（见图7-7）。在这种模式中，独立的对象（图7-7中的colleague）之间并不直接通信，而是通过mediator对象。当其中一个colleague对象改变状态以后，它将会通知该mediator，而mediator将会把该变化传达到任意其他应该知道此变化的colleague对象。

中介者示例

下面让我们探讨使用中介模式的例子。该应用程序是一个游戏程序，其中两名玩家分别给予半分钟的时间以竞争决胜出谁会比另一个按更多次数的按钮。在比赛中玩家1按2，而玩家2按0(这样他们会更加舒服一点，而不会为了争夺键盘而争吵)。计分板依据当前得分进行更新。

本例中参与的对象如下所示：

- 玩家1。
- 玩家2。

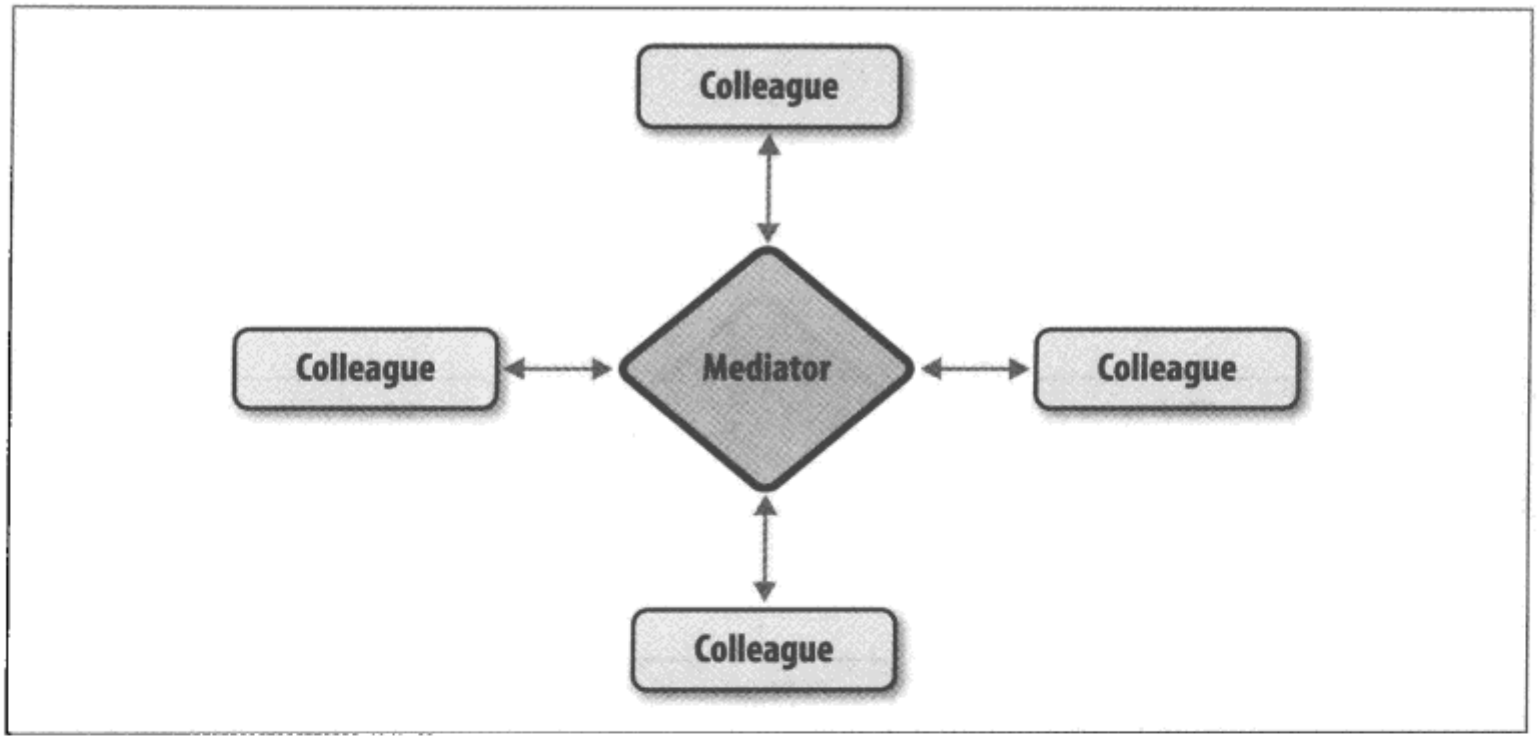

图7-7：缓存代理

- 计分板(Scoreboard)。
- 中介者(Mediator)。

中介者知道所有其他对象的信息。它与输入设备（键盘）进行通信并处理键盘按键事件，并且还要决定是哪个玩家前进了一个回合，随后还将该消息通知给玩家（见图7-8）。玩家玩游戏的同时（即仅用一分来更新其自己的分数），还要通知中介者他所做的事情。中介者将更新后的分数传达给计分板，计分板随后更新显示的分值。

除了中介者以外，没有对象知道任何其它对象。这种模式使得更新游戏变得非常简便，比如，通过该中介者可以很容易添加一个新的玩家或者另一个显示剩余时间的显示窗口。

可以在网址*http://jspatterns.com/book/7/mediator.html*看到该游戏的在线版本，并且还可以查看其源代码。

player对象是由Player()构造函数所创建的，具有points和name属性。原型中的play()方法每次以1递增分数，然后通知中介者。

```
function Player(name) {
    this.points = 0;
    this.name = name;
}
Player.prototype.play = function () {
    this.points += 1;
    mediator.played();
};
```

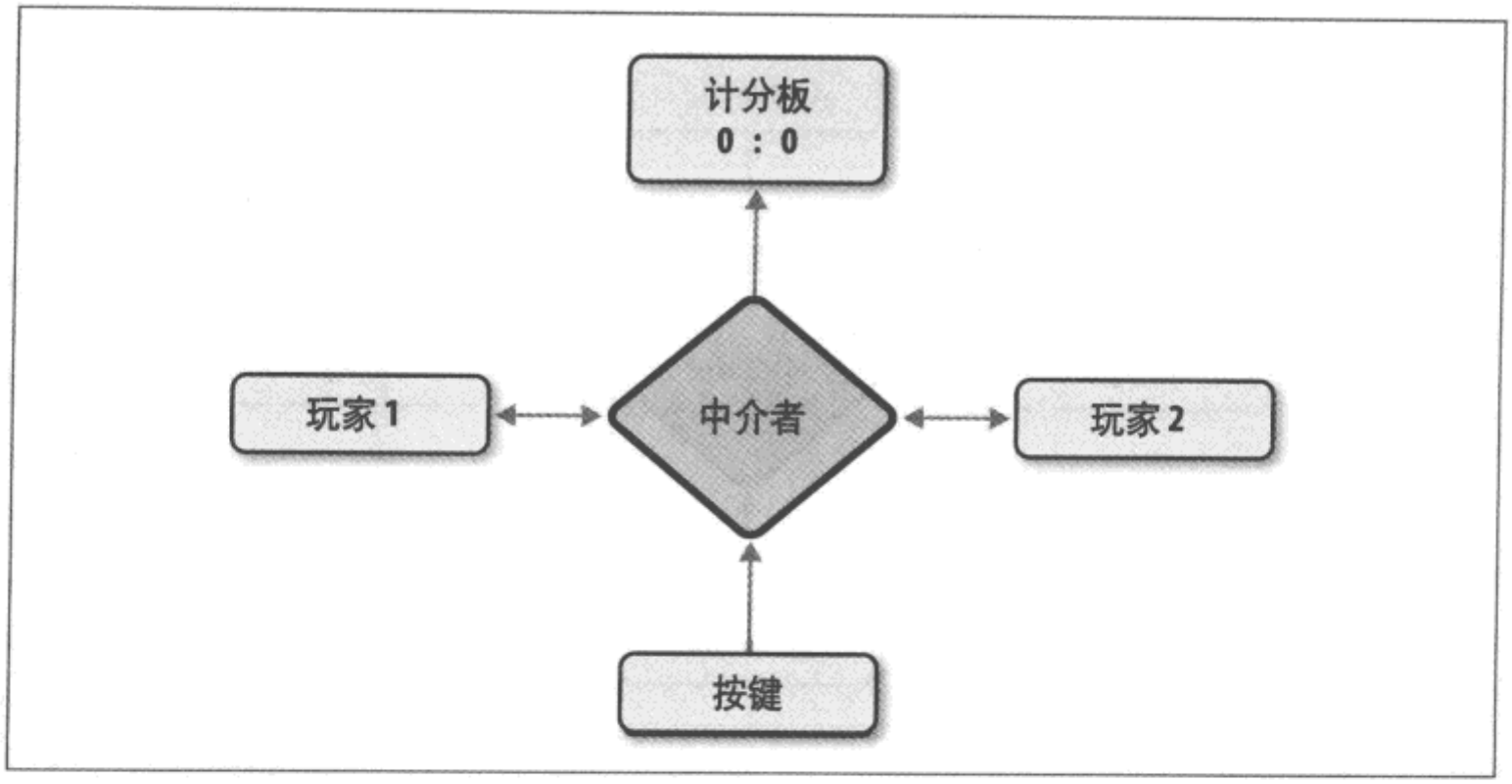

图7-8：按键游戏中的参与对象

scoreboard对象中有一个update()方法，在轮到每个玩家游戏结束之后mediator对象将调用该方法。scoreboard并不知道任何玩家的接口并且也没有保存分值，它仅根据mediator给定的值显示当前分数：

```
var scoreboard = {
    // 待更新的HTML 元素
    element: document.getElementById('results'),

    // 更新得分显示
    update: function (score) {

        var i, msg = '';
        for (i in score) {
            if (score.hasOwnProperty(i)) {
                msg += '<p><strong>' + i + '<\/strong>: ';
                msg += score[i];
                msg += '<\/p>';
            }
        }
        this.element.innerHTML = msg;
    }
};
```

现在，让我们来查看一下mediator对象。它首先初始化游戏，在它的setup()方法中创建player对象，然后将这些player对象记录到自己的players属性中。其中，played()方法将在每轮游戏后由player所调用。该方法更新score哈希表并将其发送到scoreboard中以用于显示分值。最后一个方法为keypress()，它用于处理键盘事件，确定那个玩家前进了一个回合并通知该玩家：

```
var mediator = {

    // 所有的玩家（player对象）
    players: {},

    // 初始化
    setup: function () {
        var players = this.players;
        players.home = new Player('Home');
        players.guest = new Player('Guest');

    },

    // 如果有人玩，则更新得分值
    played: function () {
        var players = this.players,
            score = {
                Home:  players.home.points,
                Guest: players.guest.points
            };

        scoreboard.update(score);
    },

    // 处理用户交互
    keypress: function (e) {
        e = e || window.event; // IE浏览器
        if (e.which === 49) { // 按键 "1"
            mediator.players.home.play();
            return;
        }
        if (e.which === 48) { // 按键 "0"
            mediator.players.guest.play();
            return;
        }
    }
};
```

而最后的事情就是要建立以及拆除该游戏：

```
// 运行!
mediator.setup();
window.onkeypress = mediator.keypress;

// 游戏在30秒内结束
setTimeout(function () {
    window.onkeypress = null;
    alert('Game over!');
}, 30000);
```

观察者模式

观察者（observer）模式广泛应用于客户端JavaScript编程中。所有的浏览器事件（鼠

标悬停，按键等事件）是该模式的例子。它的另一个名字也称为自定义事件（custom events），与那些由浏览器触发的事件相比，自定义事件表示是由您编程实现的事件。此外，该模式的另外一个别名是订阅/发布（subscriber/publisher）模式。

设计这种模式背后的主要动机是促进形成松散耦合。在这种模式中，并不是一个对象调用另一个对象的方法，而是一个对象订阅另一个对象的特定活动并在状态改变后获得通知。订阅者也称之为观察者，而被观察的对象称为发布者或者主题。当发生了一个重要的事件时，发布者将会通知(调用)所有订阅者并且可能经常以事件对象的形式传递消息。

示例#1：杂志订阅

为了理解如何实现这种模式，让我们看一个具体的例子。假设有一个发布者paper，它每天出版报纸以及月刊杂志。订阅者joe将被通知任何时候所发生的新闻。

该paper对象需要有一个subscribers属性，该属性是一个存储所有订阅者的数组。订阅行为只是将其加入到这个数组中。当一个事件发生时，paper将会循环遍历订阅者列表并通知他们。通知意味着调用订阅者对象的某个方法。因此，当用户订阅信息的时候，该订阅者需要向paper的subscribe()提供它的其中一个方法。

paper也提供了unsubscribe()方法，该方法表示从订阅者数组（即subscribers属性）中删除订阅者。Paper最后一个重要的方法是publish()，它会调用这些订阅者的方法。总而言之，发布者对象paper需要具有以下这些成员：

`subscribers`
: 一个数组。

`subscribe()`
: 将订阅者添加到subscribers数组。

`unsubscribe()`
: 从订阅者数组subscribers中删除订阅者。

`publish()`
: 循环遍历subscribers中的每个元素，并且调用他们注册时所提供的方法。

所有这三种方法都需要一个type参数，因为发布者可能触发多个事件（比如同时发布一本杂志和一份报纸）而用户可能仅选择订阅其中一种，而不是另外一种。

由于这些成员对于任何发布者对象都是通用的，将它们作为独立对象的一个部分来实现

是很有意义的。那样我们可以将其复制到任何对象中，并且将任意给定的对象变成为一个发布者。

下面是该通用发布者功能的一个实现示例，它定义了前面列举出的所有需要的成员，还加上了一个帮助方法visitSubscribers()：

```
var publisher = {
    subscribers: {
        any: [] // 事件类型: 订阅者 (subscribers)
    },
    subscribe: function (fn, type) {
        type = type || 'any';
        if (typeof this.subscribers[type] === "undefined") {
            this.subscribers[type] = [];
        }
        this.subscribers[type].push(fn);
    },
    unsubscribe: function (fn, type) {
        this.visitSubscribers('unsubscribe', fn, type);
    },
    publish: function (publication, type) {
        this.visitSubscribers('publish', publication, type);
    },
    visitSubscribers: function (action, arg, type) {
        var pubtype = type || 'any',
            subscribers = this.subscribers[pubtype],
            i,
            max = subscribers.length;

        for (i = 0; i < max; i += 1) {
            if (action === 'publish') {
                subscribers[i](arg);
            } else {
                if (subscribers[i] === arg) {
                    subscribers.splice(i, 1);
                }
            }
        }
    }
};
```

而这里有一个函数makePublisher()，它接受一个对象作为参数，通过把上述通用发布者的方法复制到该对象中，从而将其转换为一个发布者：

```
function makePublisher(o) {
    var i;
    for (i in publisher) {
        if (publisher.hasOwnProperty(i) && typeof publisher[i] === "function") {
            o[i] = publisher[i];
        }
    }
```

```
        o.subscribers = {any: []};
    }
```

现在，让我们来实现paper对象。它所能做的就是发布日报和月刊：

```
    var paper = {
        daily: function () {
            this.publish("big news today");
        },
        monthly: function () {
            this.publish("interesting analysis", "monthly");
        }
    };
```

将paper构造成一个发行者：

```
    makePublisher(paper);
```

由于已经有了一个发布者，让我们来看看订阅者对象joe，该对象有两个方法：

```
    var joe = {
        drinkCoffee: function (paper) {
            console.log('Just read ' + paper);
        },
        sundayPreNap: function (monthly) {
            console.log('About to fall asleep reading this ' + monthly);
        }
    };
```

现在，paper注册joe（也就是说，joe向paper订阅）：

```
    paper.subscribe(joe.drinkCoffee);
    paper.subscribe(joe.sundayPreNap, 'monthly');
```

正如您所看到的，joe为默认“任意”事件提供了一个可被调用的方法，而另一个可被调用的方法则用于当“monthly”类型的事件发生时的情况。现在，让我们触发一些事件：

```
    paper.daily();
    paper.daily();
    paper.daily();
    paper.monthly();
```

所有这些出版物产生的事件将会调用joe的适当方法，控制台中输出的结果如下所示：

```
    Just read big news today
    Just read big news today
    Just read big news today
    About to fall asleep reading this interesting analysis
```

该代码好的部分在于，paper对象中并没有硬编码joe，而joe中也并没有硬编码paper。此外，本代码中还没有那些知道所有一切的中介者对象。由于参与对象是松耦合的，我们可以向paper添加更多的订阅者而根本不需修改这些对象。

让我们将这个例子更进一步扩展并且让joe成为发布者（毕竟，使用博客和微博时任何人都可以是出版者）。因此，joe变成了一个发布者并且可在Twitter上分发状态更新：

```
makePublisher(joe);
joe.tweet = function (msg) {
    this.publish(msg);
};
```

现在想象一下，paper的公关部门决定读取读者的tweet[注1]，并且订阅joe的信息，那么需要提供方法readTweets()：

```
paper.readTweets = function (tweet) {
    alert('Call big meeting! Someone ' + tweet);
};
joe.subscribe(paper.readTweets);
```

现在，只要joe发出tweet消息，paper都会得到提醒：

```
joe.tweet("hated the paper today");
```

结果是一个提醒信息：“Call big meeting! Someone hated the paper today.”

可以在*http://jspatterns.com/book/7/observer.html*中看到完整的源代码，并且在控制台中现场演示。

示例#2：键盘按键游戏

让我们看另外一个例子。将重新实现与中介者模式中的键盘按键游戏完全相同的程序，但是这次使用了观察者模式。为了使它更先进一些，让我们接受无限数量的玩家，而不是只有两个玩家。仍然使用Player()构造函数创建player对象以及scoreboard对象。不过，mediator现在变成为一个game对象。

在中介者模式中，mediator对象至少所有其他参与对象并调用它们的方法。观察者模式中game对象并不会像那样做。相反，它会让对象订阅感兴趣的事件。比如，scoreboard对象将会订阅game的“scorechange”事件。

让我们先回顾一下通用publisher对象，然后略微调整它的接口，使之更接近于浏览器世界：

注1：　用户发到Twitter上的信息。

- 并不采用publish()、subscribe()以及unsubscribe()方法，我们采用以fire()、on()以及remove()命名的方法。
- 事件的type将一直被使用，因此它成为了上述三个函数的第一个参数。
- 除了订阅者的函数以外，还会提供一个额外的context，从而支持回调方法使用this以引用自己的对象。

新的publisher对象变为：

```
var publisher = {
    subscribers: {
        any: []
    },
    on: function (type, fn, context) {
        type = type || 'any';
        fn = typeof fn === "function" ? fn : context[fn];

        if (typeof this.subscribers[type] === "undefined") {
            this.subscribers[type] = [];
        }
        this.subscribers[type].push({fn: fn, context: context || this});
    },
    remove: function (type, fn, context) {
        this.visitSubscribers('unsubscribe', type, fn, context);
    },
    fire: function (type, publication) {
        this.visitSubscribers('publish', type, publication);
    },
    visitSubscribers: function (action, type, arg, context) {
        var pubtype = type || 'any',
            subscribers = this.subscribers[pubtype],
            i,
            max = subscribers ? subscribers.length : 0;

        for (i = 0; i < max; i += 1) {
            if (action === 'publish') {
                subscribers[i].fn.call(subscribers[i].context, arg);
            } else {
                if (subscribers[i].fn === arg && subscribers[i].context === context) {
                    subscribers.splice(i, 1);
                }
            }
        }
    }
};
```

新的Player()构造函数变为：

```
function Player(name, key) {
    this.points = 0;
    this.name = name;
    this.key  = key;
```

```
        this.fire('newplayer', this);
    }
    Player.prototype.play = function () {
        this.points += 1;
        this.fire('play', this);
    };
```

以上新的部分在于该构造函数接受key，即玩家用于得分所按的键盘的键（以前的代码中将键硬编码到程序中）。另外，每次当创建新的player对象时，一个名为“newplayer”的事件将被触发，每次当玩家玩游戏的时候，事件“play”将被触发。

scoreboard对象保持不变，它只是以当前分值更新其显示值。

新的game对象可以记录所有的player对象，因此它可以产生一个分数并且触发“scorechange”事件。它还将从浏览器中订阅所有的“keypress”事件，并且知道每个键所对应的玩家：

```
    var game = {

        keys: {},

        addPlayer: function (player) {
            var key = player.key.toString().charCodeAt(0);
            this.keys[key] = player;
        },

        handleKeypress: function (e) {
            e = e || window.event; // IE
            if (game.keys[e.which]) {
                game.keys[e.which].play();
            }
        },

        handlePlay: function (player) {
            var i,
                players = this.keys,
                score = {};

            for (i in players) {
                if (players.hasOwnProperty(i)) {
                    score[players[i].name] = players[i].points;
                }
            }
            this.fire('scorechange', score);
        }
    };
```

可以将任何对象转变成发行者的函数makePublisher()，仍然与前面报纸订阅的例子中的对应函数是相同的。game对象变成了一个发布者（因此，它能够触发“scorechange”事

件），并且Player.protoype也变成了发行者，以便每个player对象能够向任何决定监听的玩家触发“play”和“newplayer”事件：

```
makePublisher(Player.prototype);
makePublisher(game);
```

game对象订阅了“play”和“newplayer”事件(此外，还有浏览器中的“keypress”事件)，而scoreboard则订阅了“scorechange”事件。

```
Player.prototype.on("newplayer", "addPlayer",  game);
Player.prototype.on("play",      "handlePlay", game);
game.on("scorechange", scoreboard.update, scoreboard);
window.onkeypress = game.handleKeypress;
```

正如您在这里所看到的，on()方法使订阅者可以指定回调函数为函数引用（scoreboard.update）或字符串（"addPlayer"）的方式。只要提供了上下文环境（比如game对象[注2]，以字符串方式提供的回调函数就能正常运行。

设置的最后一个小部分是动态创建多个player对象(与他们对应的按键一起)，用户想创建多少个player对象都可以：

```
var playername, key;
while (1) {
    playername = prompt("Add player (name)");
    if (!playername) {
        break;
    }
    while (1) {
        key = prompt("Key for " + playername + "?");
        if (key) {
            break;
        }
    }
    new Player(playername,  key);
}
```

在这里该游戏程序开发结束。可以在网址*http://jspatterns.com/book/7/observer-game.html*查看其完整源代码并玩该游戏。

请注意，在中介者模式的实现中，mediator对象必须知道所有其他对象，以便在正确的时间调用正确的方法并且与整个游戏相协调。而在观察者模式中，game对象显得更缺乏智能，它主要依赖于对象观察某些事件并采取行动。比如，scoreboard监听“scorechange”事件。这导致了更为松散的耦合(越少的对象知道越少)，其代价是在记录谁监听什么事件时显得更困难一点。在本例的游戏中，所有订阅行为都出现在该代码

注2：　on()方法中的context参数。

的同一个位置，但是随着应用程序的增长，on()调用可能到处都是（例如，在每个对象的初始化代码中）。这会使得该程序难以调试，因为现在无法仅在单个位置查看代码并理解到底发生了什么事情。在观察者模式中，可以摆脱那种从开始一直跟随到最后的那种过程式顺序代码执行的程序。

小结

在本章中，学习了一些流行的设计模式，并且也看到如何在JavaScript实现这些模式。以下是我们所讨论的设计模式：

单体模式

针对一个“类”仅创建一个对象。如果您想以构造函数的方法替换类的思想并且还保持类似Java的语法，我们则为您考虑了多种方法。另外，从技术上来说，JavaScript中的所有对象都是单体。然而有时候程序开发员也会说“单体”，他们的本意是指以模块模式（module pattern）创建的对象。

工厂模式

根据字符串指定的类型在运行时创建对象的方法。

迭代器模式

提供一个API来遍历或操纵复杂的自定义数据结构。

装饰者模式

通过从预定义装饰者对象中添加功能，从而在运行时调整对象。

策略模式

在选择最佳策略以处理特定任务(上下文)的时候仍然保持相同的接口。

外观模式

通过把常用方法包装到一个新方法中，从而提供一个更为便利的API。

代理模式

通过包装一个对象以控制对它的访问，其主要方法是将访问聚集为组或仅当真正必要的时候才执行访问，从而避免了高昂的操作开销。

中介者模式

通过使您的对象之间相互并不直接“通话”，而是仅通过一个中介者对象进行通信，从而促进形成松散耦合。

观察者模式

通过创建“可观察的”对象，当发生一个感兴趣的事件时可将该事件通告给所有观察者，从而形成松散耦合。

第8章

DOM和浏览器模式

在本书的前面章节中，我们主要集中关注于核心JavaScript（ECMAScript），而并没有太多关注在浏览器中使用JavaScript的模式。本章将探索一些浏览器特定的模式，因为浏览器是使用JavaScript最为常见的环境。同时也是很多人不喜欢使用JavaScript的原因，他们认为JavaScript只是一种浏览器脚本。考虑到在浏览器中存在的很多前后矛盾的主机对象和DOM实现，这种想法也是可以理解的。很明显通过使用一些较好的可以减少客户端脚本负担的实践技巧，可以获益颇多。

在本章您将看到模式被划分为几类，包含DOM脚本、事件处理、远程脚本、页面载入JavaScript的策略和在产品网站上配置JavaScript的步骤等。

但是首先，让我们简单地并从哲学角度来探讨如何处理客户端的脚本。

关注分离

在网站应用程序开发过程中主要关心如下三个内容：

内容（Content）
: HTML的文档。

表现（Presentation）
: 指定文档外观的CSS样式。

行为（Behavior）
: 处理用户交互和文档各种动态变化的JavaScript。

将这三部分尽可能的相互独立，可以改进将应用程序交付给大量各种用户终端的效果，

图形化的浏览器、文本浏览器、针对残疾用户的辅助技术、移动设备等。关注分离（separation of concerns）也体现了渐进增强（progressive enhancement）的思想，最简单的用户终端可以具有基本的体现（仅能显示HTML文档），并随着用户终端能力的改进而获取更佳的用户体验。如果浏览器支持CSS，那么用户将可看到文档更好的表现方式。如果浏览器支持JavaScript，那么该文档更大程度上看起来像一个应用程序，并将获取更多增强用户体验的特性。

在实际中，关注分离意味着：

- 通过将CSS关闭来测试页面是否仍然可用，内容是否依然可读。
- 将JavaScript关闭来测试页面仍然可以执行其正常功能，所有的连接（不包含href="#"的实例）是否能够正常工作，所有的表单可以正常工作并正确提交信息。
- 不使用内联处理器（如onclick之类）和内联样式属性，因为这些都不属于内容层。
- 使用例如headings和lists这样语义上有意义的HTML元素。

JavaScript层（行为）应该是不引人注意的，也就是说，JavaScript层应该不会给用户造成不便，例如在不支持JavaScript的浏览器中不会造成网页不可用等问题，JavaScript应该是用来加强网页功能，而不能成为网页正常工作的必需组件。

常见的用于处理浏览器差异性的技术是特性检测技术（capability detection）。该技术建议不要使用用户代理来嗅探代码路径，而应该在运行环境中检查是否有所需的属性或方法。通常将使用代理嗅探这种方法看做一种反模式。有时候这是不可避免的，但是应该在使用特性检测技术无法获得确定性结论时（或者会导致极大的性能损失时），不得已才使用代理嗅探。

```
// 反模式
if (navigator.userAgent.indexOf('MSIE') !== -1) {
    document.attachEvent('onclick', console.log);
}

// 比较好的做法
if (document.attachEvent) {
    document.attachEvent('onclick', console.log);
}

// 更具体的做法
if (typeof document.attachEvent !== "undefined") {
    document.attachEvent('onclick', console.log);
}
```

采用关注分离还有助于开发、维护、和升级现有Web应用程序，因为当发生故障时，可以知道去什么地方排错。当是JavaScript发生错误时，无需查看HTML代码和CSS代码来查错。

DOM脚本

使用页面的DOM树是客户端JavaScript最常用的任务。这也是头痛的主要原因（JavaScript因此获得一些不好的名声），因为不同的浏览器在DOM方法的实现方面并不一致。这也是为什么使用一个好的JavaScript类库（该类库可以抽象出不同浏览器的区别）可以显著加快开发进度。

让我们来看看在访问和修改DOM树时推荐的一些模式（主要是出于性能方面考虑）。

DOM访问

DOM访问的代价是昂贵的，它是制约JavaScript性能的主要瓶颈。这是因为DOM通常是独立于JavaScript引擎而实现的。从浏览器的视角看，采用该方法是有意义的，因为有的JavaScript应用程序可能根本就不需要DOM。而且除JavaScript以外的其他程序（例如IE中的VBScript）也可以用来和页面的DOM共同工作。

总之，DOM的访问应该减少到最低。这意味着：

- 避免在循环中使用DOM访问。
- 将DOM引用分配给局部变量，并使用这些局部变量。
- 在可能的情况下使用selector API。
- 当在HTML容器中重复使用时，缓存重复的次数（参考第2章）。

请看如下范例，尽管第二种方式循环语句更长，但针对不同的浏览器，它会比第一种方式快上几十倍到几百倍。

```
//反模式
for (var i=0;i<100;i+=1){
    Document.getElementById( "result" ).innerHTML+=i+" ," ;
}

//更好的方式，使用了局部变量
Var i,content=" " ;
For(i=0;i<100;i+=1){
Content+=i+" ," ;
}
Document.getElementById( "result" ).innerHTML+=content;
```

接下来的一个片段中第二个范例是更好的使用方法（使用了局部变量风格），尽管其需要额外的一行代码和一个变量：

```
// 反模式
var padding = document.getElementById("result").style.padding,
```

```
    margin  = document.getElementById("result").style.margin;

// 更好的做法
var style   = document.getElementById("result").style,
padding = style.padding,
 margin  = style.margin;
```

可以采用如下方法来使用selector API：

```
document.querySelector("ul .selected");
document.querySelectorAll("#widget .class");
```

这些方法接受一个CSS选择字符串并返回一个匹配该选择的DOM结点列表。该选择方法在现在主流的浏览器（IE从8.0以后都支持）中都是支持的，并且会比使用其他DOM方法来自己实现选择要快得多。最近一些最新版本的流行JavaScript库利用了selector API，因此最好是使用个人喜好的最新版本的JavaScript库。

为经常访问的元素增加id属性是一个很好的做法，因为document.getElementById(myid)是最简单快捷查找节点的方法。

操纵DOM

除了访问DOM元素以外，通常还需要修改、删除或者增加DOM元素。更新DOM会导致浏览器重新绘制屏幕，也会经常导致reflow（也就是重新计算元素的几何位置），这样会带来巨大的开销。

通常的经验法则是尽量减少更新DOM，这也就意味着将DOM的改变分批处理，并在“活动”文档树之外执行这些更新。

当需要创建一个相对比较大的子树，应该在子树完全创建之后再将子树添加到DOM树中。这时可以采用文档碎片（document fragment）技术来容纳所有节点。

下面将介绍如何不立即添加节点：

```
// 反模式
// 在创建时立即添加节点

var p, t;

p = document.createElement('p');
t = document.createTextNode('first paragraph');
p.appendChild(t);
document.body.appendChild(p);

p = document.createElement('p');
t = document.createTextNode('second paragraph');
p.appendChild(t);
```

```
document.body.appendChild(p);
```

创建文档碎片来离线升级节点信息是更好的做法。当将文档碎片添加到DOM树时，不是将碎片本身添加到DOM树中，而是将文档碎片的内容添加进DOM树中。该操作是十分方便的。文档碎片是一种很好的方法，可以用来封装许多节点信息，甚至这些节点并没有合适的父节点（例如，文章不在div元素范围内）。

接下来是一个使用文档碎片的范例：

```
var p, t, frag;

frag = document.createDocumentFragment();

p = document.createElement('p');
t = document.createTextNode('first paragraph');
p.appendChild(t);
frag.appendChild(p);

p = document.createElement('p');
t = document.createTextNode('second paragraph');
p.appendChild(t);
frag.appendChild(p);

document.body.appendChild(frag);
```

在这个范例中活动的文档仅仅更新了一次并触发一次屏幕重绘。而如果采用之前的反模式，每执行一个段落都会重绘一次。

在为DOM树添加新结点时文档碎片是非常有用的。但在更新DOM现有的部分时，仍然可以批处理提交修改。具体方法是：为需要修改的子树的根节点建立一个克隆镜像，然后对该克隆镜像做所有的修改操作，在完成修改操作后用克隆镜像替换原来的子树。

```
var oldnode = document.getElementById('result'),
    clone = oldnode.cloneNode(true);

// 处理克隆对象...

// 完成后:
oldnode.parentNode.replaceChild(clone, oldnode);
```

事件

处理浏览器事件（例如单击、鼠标移动等）是浏览器脚本领域中一个有许多不一致性并导致工作失败的源头。JavaScript库可以减少为了支持IE（在IE 9.0之前的版本）和符合W3C规范的实现所做的双重工作。

让我们重温关于浏览器事件的要点，因为可能并不总是为简单的网页使用某个现有的库，有可能还会创建自己库。

事件处理

通常事件处理是通过为元素附加事件监听器来实现的，例如有一个按钮，该按钮在每次单击后都会增加一次计数。可以增加一个内联的onclick属性，该属性在所有的浏览器中都可以正常工作，但是该属性会和关注分离和渐进增强有冲突。因此，应该争取在JavaScript中附加监听器，并放置于所有标记之外。

假定有如下标记：

```
<button id="clickme">Click me: 0</button>
```

可以为该节点的onclick属性分配一个函数，但这种做法只能指定一个函数：

```
// 次优解决方案
var b = document.getElementById('clickme'),
    count = 0;
b.onclick = function () {
    count += 1;
    b.innerHTML = "Click me: " + count;
};
```

如果希望在一次单击后执行多个函数功能，仍然维持采用现在的松耦合模式是无法做到的。技术上来说，可以检查onclick是否已经包含一个函数，如果包含了一个函数，那么就将现有的函数功能加到新函数中，并用新函数替换onclick中的原有函数的属性。但更清晰的方法是使用addEventListener()方法。在IE 8.0之前的版本中没有该方法，在这些老版本浏览器中应该使用attachEvent()。

让我们回顾一下初始化分支模式（参考第4章），可以看到定义跨浏览器事件监听器工具的一种比较好的实现范例。现在无需探究所有的细节，让我们先尝试为按钮添加一个监听器：

```
var b = document.getElementById('clickme');
if (document.addEventListener) { // W3C
    b.addEventListener('click', myHandler, false);
} else if (document.attachEvent) { // IE
    b.attachEvent('onclick', myHandler);
} else { // 终极手段
    b.onclick = myHandler;
}
```

现在一旦按钮被点击，myHandler()函数将会执行，该函数会增加按钮上面“click me:0”中的数值。让我们假定有多个按钮，并且这些按钮共享同一个myHandler()函

数。考虑到可以从每次点击时创建的事件对象中获取数值，因此为每个数值维持按钮节点和计数器之间引用是十分低效的。

让我们先来看看对此的解决方案，然后在加以评论：

```
function myHandler(e) {

    var src, parts;

    // 获取事件和源元素
    e = e || window.event;
    src = e.target || e.srcElement;

    // 实际工作：升级标签
    parts = src.innerHTML.split(": ");
    parts[1] = parseInt(parts[1], 10) + 1;
    src.innerHTML = parts[0] + ": " + parts[1];
    // 无冒泡
    if (typeof e.stopPropagation === "function") {
        e.stopPropagation();
    }
    if (typeof e.cancelBubble !== "undefined") {
        e.cancelBubble = true;
    }

    // 阻止默认操作
    if (typeof e.preventDefault === "function") {
        e.preventDefault();
    }
    if (typeof e.returnValue !== "undefined") {
        e.returnValue = false;
    }
}
```

在*http://jspatterns.com/book/8/click.html*中有生动的范例。

这个事件处理函数分为四个部分：

- 首先需要获取对事件对象的访问权，该事件对象包含了关于事件和触发该事件的网页元素的信息。事件对象被传递给回调事件处理器，而不是使用onclick属性（可以通过全局属性windows.event来获取访问权）。
- 第二部分是处理升级标签的实际工作。
- 接下来第三部分是取消事件的传播。在当前特定的范例中，这一部分可以省略，不是必须的。但是通常如果不这样做，会导致事件传播到文档根，甚至是传播到window对象中。在这个部分需要采用两种方法实现：一种是W3C的标准方法（stopPropagation()）；另外一种是IE特有的方法（cancelBubble）。
- 最后，如果需要时，要阻止执行默认操作。一些事件拥有默认操作，但可以使用

preventDefault()来阻止默认操作（在IE中，通过将returnValue设置为false来实现）。

如您所见，这样的做法包含有很多重复性工作，因此按照第7章讨论的那样使用正面方法创建自己的事件工具是十分有意义的。

事件授权

事件授权模式得益于事件起泡，会减少为每个节点附加的事件监听器数量。如果在div元素汇总有10个按钮，只需要为该div元素附加一个事件监听器就可以实现为每个按钮分别附加一个监听器的效果。

下面是一个在div元素中有三个按钮的范例（见图8-1）。详细的关于事件授权的演示请参见*http://jspatterns.com/book/8/click-delegate.html*。

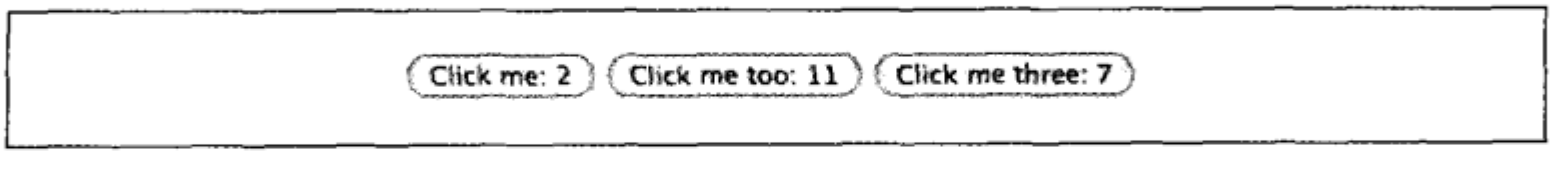

图8-1：事件授权范例：三个会增加标签中数值的按钮

因此，可以使用如下标记：

```
<div id="click-wrap">
  <button>Click me: 0</button>
  <button>Click me too: 0</button>
  <button>Click me three: 0</button>
</div>
```

可以通过为“click-wrap” div附加监听器来代替为每一个按钮就附加监听器。然后只需要对之前范例中使用的myHandler()函数做微小修改（需要过滤不感兴趣的点击事件），就可以直接使用。在这种情况下，只需寻找按钮的点击事件，而同一个div元素中其他点击事件都会被忽略。

对myHandler()需要做的修改就是判断事件的nodeName是否为“button”，如果是，则执行函数功能：

```
// ...
// 获取事件和源元素
e = e || window.event;
src = e.target || e.srcElement;

if (src.nodeName.toLowerCase() !== "button") {
    return;
}
```

```
// ...
```

事件授权的缺点在于如果碰巧没有感兴趣的事件发生，那么增加的小部分代码就显得没用了。但是采用该模式所获得的收益（性能和更为清晰的代码）远远大于缺点，因此强烈推荐使用该模式。

最近的JavaScript库通过API，使得事件授权更为简便。举例来说，YUI3有一个Y.delegate()方法，该方法可以指定一个CSS选择器来匹配封装，并使用另外一个选择器来匹配感兴趣的节点。这是十分方便的，因为当事件在关注的节点之外发生时，回调事件函数实际上并没有被调用。在这种情形下，附加一个事件监听器的代码是十分简便的，如下所示：

```
Y.delegate('click', myHandler, "#click-wrap", "button");
```

由于YUI将各种浏览器的区别抽象出来了，可以由用户决定事件的来源，因此回调函数将变得更为简便：

```
function myHandler(e) {

    var src = e.currentTarget,
        parts;

    parts = src.get('innerHTML').split(": ");
    parts[1] = parseInt(parts[1], 10) + 1;
    src.set('innerHTML', parts[0] + ": " + parts[1]);

    e.halt();
}
```

如需更完整的范例，请参考*http://jspatterns.com/book/8/click-y-delegate.html*

长期运行脚本

可能会注意到有时候浏览器会提示某个脚本已经运行了很长时间，是否应该停止该脚本。实际上无论要处理多么复杂的任务，都不希望应用程序发生上述事情。

而且，如果该脚本的工作十分繁重，那么浏览器的UI将会无法响应用户的任何操作。这将给用户带来十分不好的体验，应该尽量避免。

在JavaScript中没有线程，但是可以在浏览器中使用setTimeout()来模拟线程，在最新版本的浏览器中可以使用Web Workers。

setTimeout()

这样做的思想是将一个大任务分解为多个小任务，并为每一个小任务设置超时时间为1

毫秒。通过为每一个小任务设置超时为1毫秒，会导致完成整个任务需要耗费更长的时间，但是通过这样做，可以使得用户接口保持响应，用户体验更好。

注意： 超时时间设置为1毫秒（或者设置为0毫秒）实际上是与浏览器和操作系统相关的。将超时时间设置为0并不意味着没有超时，而是指尽可能快的处理。例如在IE中，最快的时钟周期是15毫秒。

Web Workers

最近的浏览器为长期运行的脚本提供了另外一个解决方案：Web Workers。Web Workers为浏览器提供了背景线程支持。可以将任务比较繁重的计算放在单独一个文件中，例如*my_web_worker.js*。从主程序（网页）中调用该文件，如下所示：

```
var ww = new Worker('my_web_worker.js');
ww.onmessage = function (event) {
    document.body.innerHTML +=
        "<p>message from the background thread: " + event.data + "</p>";
};
```

下面示例中Web Workers做了1e8（100000000）次简单算术操作：

```
var end = 1e8, tmp = 1;
postMessage('hello there');

while (end) {
    end -= 1;
    tmp += end;
    if (end === 5e7) { // 5e7 是 1e8的一半
        postMessage('halfway there, `tmp` is now ' + tmp);
    }
}

postMessage('all done');
```

Web Workers使用postMessage()来与调用者通信，并且调用者订阅onmessage事件来接收更新。onmessage回调函数接收事件对象作为参数，并且该对象包含data属性。类似地，调用者可以使用ww.postMessage()将数据传递给Web Workers，Web Worker会使用onmessage回调函数来订阅这些信息。

之前的范例将在浏览器中打印出如下内容：

```
message from the background thread: hello there
message from the background thread: halfway there, `tmp` is now 3749999975000001
message from the background thread: all done
```

远程脚本

当今Web应用程序经常使用远程脚本来在无需重新载入当前页面时与服务器通信。该方法可以获取更多响应，并使得类似桌面的网页应用程序成为可能。现在我们讨论一些使用JavaScript与远程服务器通信的方法。

XMLHttpRequest

当今XMLHttpRequest是一个在大多数浏览器中都支持的特殊对象，该对象可以让您采用JavaScript建立HTTP请求。建立一个HTTP请求分为如下三个步骤：

1. 建立一个XMLHttpRequest对象（简写为XHR）。
2. 提供一个回调函数来告知请求对象改变状态。
3. 发送请求。

第一步操作十分简单：

```
var xhr =  new XMLHttpRequest();
```

但是在IE浏览器在7.0之前的版本中，XHR功能性是以ActiveX对象的方式实现的，因此对于那些版本需要做一些特殊处理。

第二步是为readystatechange事件提供一个回调函数：

```
xhr.onreadystatechange = handleResponse;
```

最后一步是使用open()和send()两个方法来启动该请求。先使用open()方法指定HTTP请求方法（例如GET和POST）和URL。然后使用send()方法传递POST的数据或者仅仅一个空白字符串（在GET模式下）。open()方法的最后一个参数指定该请求是否是异步的。异步模式意味着浏览器将不会停下来以等待回应。这当然会给用户更佳的用户体验，因此除非在有特点的理由以外，其他情况都应该将异步参数设置为true:

```
xhr.open("GET", "page.html", true);
xhr.send();
```

下面是一个完整的范例，展示了获取网页内容，并采用新的内容更新当前网页的过程（演示文件详见*http://jspatterns.com/book/8/xhr.html*）：

```
var i, xhr, activeXids = [
    'MSXML2.XMLHTTP.3.0',
    'MSXML2.XMLHTTP',
    'Microsoft.XMLHTTP'
];
```

```
if (typeof XMLHttpRequest === "function") { // 原生 XHR
    xhr =  new XMLHttpRequest();
} else { // 在IE 7之前版本
    for (i = 0; i < activeXids.length; i += 1) {
        try {
            xhr = new ActiveXObject(activeXids[i]);
            break;
        } catch (e) {}
    }
}
xhr.onreadystatechange = function () {
    if (xhr.readyState !== 4) {
        return false;
    }
    if (xhr.status !== 200) {
        alert("Error, status code: " + xhr.status);
        return false;
    }
    document.body.innerHTML += "<pre>" + xhr.responseText + "<\/pre>";
};

xhr.open("GET", "page.html", true);
xhr.send("");
```

下面是对该范例的一些注释：

- 对于IE来说，在IE6.0以及之前的版本中新建XHR对象的过程有一些复杂。范例中依次通过一个ActiveX标识符列表（从最新版本到早期版本）来尝试创建新对象来确定IE的版本，并将这部分操作封装在try-catch块中。
- 回调函数检查xhr对象的readyState属性。该属性取值范围为从0～4，共5个可能的属性值，其中属性值为4意味着“完成”。如果xhr对象的状态不是完整状态，那么继续等待下一个readystatechange事件。
- 回调函数也会检查xhr对象的status属性。该属性对应于HTTP的状态码，例如200就对应于（OK），而404对应于（Not found）。这里只关心状态码为200的情况，而将其他状态码都按照错误处理（这是出于简便起见，否则就需要检查其他的有效状态）。
- 以上列出来的代码将会在每次创建请求的时候，就检查浏览器支持的方法来创建XHR对象。由于已经在之前的章节学习了一些模式（例如初始化分支模式），可以重写该段代码，以使得只需要检查一次浏览器可以支持的方法。

JSONP

JSONP（有填充的JSON）是另外一种创建远程请求的方法。和XHR有所不同，它不受同一个域浏览器策略的限制，出于从第三方网站载入数据的安全考虑，需要小心使用。

对应于XHR请求，JSONP的请求可以是任意类型的文档：

- XML文档（过去常用的）。
- HTML块（现在十分常见的）。
- JSON数据（轻量级，并且方便）。
- 简单文本文件或者其他文档。

对于JSONP，最常见的是用函数调用封装的JSON，函数名由请求来提供。

JSONP请求的URL通常格式如下所示：

http://example.org/getdata.php?callback=myHandler

*getdata.php*可以是任意类型的网页，`callback`参数指定采用哪个JavaScript函数来处理该请求。

然后像下面这样将URL载入到动态的`<script>`元素：

```
var script = document.createElement("script");
script.src = url;
document.body.appendChild(script);
```

服务器响应一些JSONP数据，这些数据作为回调函数的参数。最终的结果是在网页中包含了一个新的脚本，该脚本碰巧是一个函数调用，例如：

```
myHandler({"hello": "world"});
```

JSONP 范例: 字棋游戏（Tic-tac-toe）

下面展示一个使用JSONP的范例，一个字棋游戏，这里玩家既是客户端（浏览器），也是服务器。客户端和服务器都会生成一个1～9的随机数，并使用JSONP来获取服务器的值（见图8-2）。

可以在*http://jspatterns.com/book/8/ttt.html*这个网址在线玩该游戏。

这里有两个按钮：一个新建游戏，另外一个按钮切换为服务器方（在一定超时后，会自动切换为客户方）：

```
<button id="new">New game</button>
<button id="server">Server play</button>
```

面板中包含9个小格子，分别对应于不同id属性。例如：

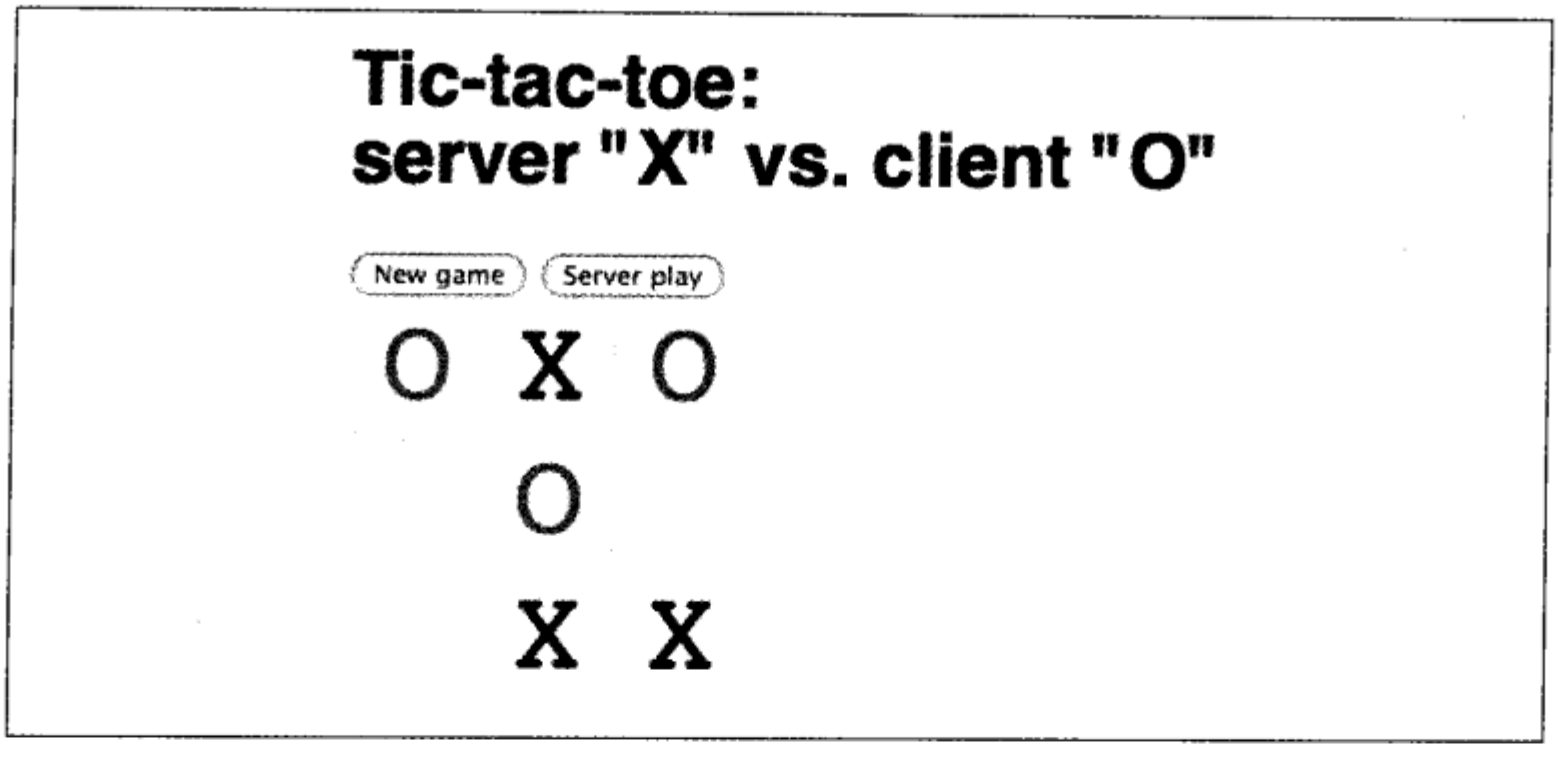

图8-2：字棋游戏JSONP挑战

```
<td id="cell-1"> </td>
<td id="cell-2"> </td>
<td id="cell-3"> </td>
...
```

完整游戏代码在ttt全局对象中实现：

```
var ttt = {
    // 目前已经玩过的空格
    played: [],

    // 速记
    get: function (id) {
        return document.getElementById(id);
    },

    // 处理单击事件
    setup: function () {
        this.get('new').onclick = this.newGame;
        this.get('server').onclick = this.remoteRequest;
    },

    // 清理面板
    newGame: function () {
        var tds = document.getElementsByTagName("td"),
            max = tds.length,
            i;
        for (i = 0; i < max; i += 1) {
            tds[i].innerHTML = " ";
        }
        ttt.played = [];
    },

    // 建立请求
```

```
remoteRequest: function () {
    var script = document.createElement("script");
    script.src = "server.php?callback=ttt.serverPlay&played=" + ttt.played.join(',');
    document.body.appendChild(script);
},

// 回调函数，轮到电脑玩游戏
serverPlay: function (data) {
    if (data.error) {
        alert(data.error);
        return;
    }
    data = parseInt(data, 10);
    this.played.push(data);

    this.get('cell-' + data).innerHTML = '<span class="server">X<\/span>';

    setTimeout(function () {
        ttt.clientPlay();
    }, 300); // 仿佛在沉思
},

// 轮到玩家玩游戏
clientPlay: function () {
    var data = 5;

    if (this.played.length === 9) {
        alert("Game over");
        return;
    }

    // 生成 1~9之间的随机数
    // 直至找到一个没有选择的空格
    while (this.get('cell-' + data).innerHTML !== " ") {
        data = Math.ceil(Math.random() * 9);
    }
    this.get('cell-' + data).innerHTML = 'O';
    this.played.push(data);

}

};
```

ttt对象维持一个目前为止已经选择的空格列表，并将其发给服务器，因此服务器可以排除已经选择的数字来返回一个新数字。如果有错误发生，服务器将会返回类似如下信息：

```
ttt.serverPlay({"error": "Error description here"});
```

正如您所看到的那样，JSONP中的回调函数必须是一个公有的和全局有效的函数。该回调函数可以不必是一个全局函数，但是必须是全局对象的一个方法。如果这里没有错误的话，服务器将会返回如下函数调用：

```
ttt.serverPlay(3);
```

这里3意味着服务器给出的随机选择第三个空格。在这种情形下，由于数据十分简单，甚至不需要使用JSON格式，只需要使用一个数值表示就行。

框架和图像灯塔

使用框架也是一种处理远程脚本的备选方案。可以使用JavaScript创建一个iframe元素，并修改其src属性的URL。新的URL可以包含更新调用者（在iframe之外的父页面）的数据和函数调用。

使用远程脚本最简单的场景是在只需要向服务器发送数据，而无需服务器回应的时候。在这种情形下，可以创建一个新图像，并将其src属性设置为服务器上的脚本文件，如下所示：

```
new Image().src = "http://example.org/some/page.php";
```

这种模式称为图像灯塔（image beacon），这在希望向服务器发送日志数据时是非常有用的。举例来说，该模式可以用于收集访问者统计信息。因为用户并不需要使用服务器对这些日志数据的响应，通常的做法是服务器用一个1×1像素的GIF图片来作为响应（这是一种不好的模式）。使用“204 Not Content”这样的HTTP响应是更好的选择。该HTTP响应的意思是指仅向客户端发送HTTP报头文件，而不发送HTTP内容体。

配置JavaScript

在采用JavaScript时，还有一些性能上需要考虑的因素。这里将在比较高的层面上讨论一下这方面最重要的问题，如需要了解更为详细的内容，请参考《High Performance Web Sites》和《Even Faster Web Sites》，都是由O'Reilly公司出版。

合并脚本文件

构建快速载入页面的第一条规则就是尽可能少地使用外部组件，因为HTTP请求是十分消耗资源的。对于JavaScript来说，可以通过合并外部脚本文件来明显提高页面载入速度。

假定网页使用了jQuery库，这是一个.js文件。然后还需要使用一些jQuery插件，每个插件都是一个独立的文件。这样在编写代码前，就拥有了4～5个文件。将这些文件合并为一个文件是十分有意义的，特别是考虑到这些文件通常都十分小（2～3Kb），因而导致HTTP开销比实际下载文件的开销大得多。将这些脚本文件合并的方法很简单，只需要创建一个新文件，并将这些脚本文件的内容复制进去就行。

当然，应该在编写代码之前合并这些文件，而不能在开始开发的过程中合并文件，因为那样做会导致很多调试的开销。

合并脚本文件的做法也有一些缺点，如下所示：

- 尽管在正式开始编写代码前需要增加一个合并脚本文件的步骤，但该操作可以采用命令行自动完成，例如在Linux/Unix可以使用cat命令来合并，如下所示：

```
$ cat jquery.js jquery.quickselect.js jquery.limit.js > all.js
```

- 丢失一些缓存效益。当对其中某一个脚本文件进行修改后，该修改并不会体现到整个合并后的文件中。这就是为什么对于大型项目需要有发布规划，或者是采用两个脚本文件包：一个包含那些可能会改变的文件；另外一个包含那些不会发生修改的文件。
- 对于文件包最好是使用版本号或者其他内容来命名。例如使用时间戳：*all_20100426.js*，也可以使用文件内容的哈希值来命名。

以上这些缺点可以归纳为主要在于不方便，但是使用合并脚本文件的方法带来的收益远大于带来的不便性。

精简和压缩脚本文件

在第2章中已经涉及了代码精简。将代码精简作为构建JavaScript脚本的一部分是十分重要的。

当从用户视角考虑时，用户没有必要下载所有的注释语句，删除这些注释语句对应用程序正常运行没有影响。

精简脚本文件带来的收益依赖于使用的注释语句和空格的数量，也和具体精简工具有关。但通常来说，可以精简大约50%的文件大小。

应该经常维护对脚本文件的压缩，这只需要在服务器配置中启用gzip压缩支持就可以实现，这样的配置会立即提高速度。如果使用了共享主机的服务，无法获取足够的自由来对服务器进行配置，大部分服务提供商至少会允许您使用Apache的*.htaccess*配置文件。因此应该在Web根目录中，将下列代码添加到*.htacess*文件中：

```
AddOutputFilterByType DEFLATE text/html text/css text/plain text/xml
application/javascript application/json
```

通常这样的压缩配置会减少70%的文件大小。将精简和压缩两种操作相结合，最后只需要下载的文件大小仅有未精简、压缩之前的文件的15%。

Expires报头

与通常人们的想法相反，文件并不会在浏览器缓存中保存太久时间。可以通过使用expires报头来增加重复访问时，请求的文件依然在缓存中的概率。

该操作也仅需要在*.htaccess*文件中增加如下代码：

```
ExpiresActive On
ExpiresByType application/x-javascript "access plus 10 years"
```

这样做的缺点在于如果希望修改文件，就需要重命名该文件。但可能已经为合并后的文件确定了一个命名约定。

使用CDN

CDN是内容分发网络（Content Delivery Network）的缩写。CDN提供付费的主机服务，它允许您将文件副本放置于全球各个数据中心，以便用户可以选择速度最快的服务器进行连接，而您文件代码中的URL地址不需要修改。

如果不希望为使用CDN付一定的开支，还有一些免费的选择：

- Google提供了许多流行的开源的库，可以免费链接到这些库，并从Google的CDN中获益。
- Microsoft提供了jQuery和其自身的Ajax库。
- Yahoo！在它自己的CDN上提供了YUI库。

载入策略

乍看一样，如何将脚本文件包含到网页文件中是一个十分简单直白的问题。只需要使用`<script>`元素，要么直接使用内联的JavaScript代码，要么在`src`属性中使用到单独文件的连接，如下所示：

```
// 选项1
<script>
console.log("hello world");
</script>
// 选项2
<script src="external.js"></script>
```

但是当希望构建高性能网页应用程序时，需要意识到还有更多的模式可以考虑。

作为边注，有一些开发者倾向使用的一些常见的属性：

language="JavaScript"

大部分情况下是使用“JavaScript”，有时候会附上版本号。language属性可以不使用，因为默认语言就是JavaScript。在老版本浏览器中版本号不能正常工作，会提示报错。

type="text/javascript"

在HTML4和XHTML1标准中该属性是必须的，但是这应该不需要成为一个必须属性，因为浏览器默认就是以JavaScript处理的。在HTML5中该属性不是必须的属性。除非为了满足标记器，没有其他理由来使用type属性。

defer

defer也是没有得到广泛支持的一种方法（在HTML5中更好的是使用async），指定下载外部脚本文件不会阻止下载网页的其他内容。

<script>元素的位置

脚本元素会阻止下载网页内容。浏览器可以同时下载多个组件，但一旦遇到一个外部脚本文件后，浏览器会停止进一步下载，直到这个脚本文件下载、解析并执行完毕。这会严重影响网页载入的总时间，特别是在网页载入时会发生多次这类事情。

为了最小化阻止的影响，可以将脚本元素放置于网页的最后部分，刚好在</body>标签之前。在这个位置脚本文件不会阻止其他任何文件块。网页组件的其他部分将会被下载并执行。

在文档抬头使用单独文件是最坏的模式：

```
<!doctype html>
<html>
<head>
    <title>My App</title>
    <!-- ANTIPATTERN -->
    <script src="jquery.js"></script>
    <script src="jquery.quickselect.js"></scri
    <script src="jquery.lightbox.js"></script>
    <script src="myapp.js"></script>
</head>
<body>
    ...
</body>
</html>
```

将所有文件合并是更好的做法：

```
<!doctype html>
<html>
```

```
<head>
    <title>My App</title>
    <script src="all_20100426.js"></script>
</head>
<body>
    ...
</body>
</html>
```

最好的做法是将合并后的脚本放于网页的最后部分：

```
<!doctype html>
<html>
<head>
    <title>My App</title>
</head>
<body>
    ...
    <script src="all_20100426.js"></script>
</body>
</html>
```

HTTP块

HTTP支持所谓的块编码，该技术允许分片发送网页。因此如果有一个很复杂的网页，不需等待服务器完成所有运算工作，就可以提前将一些静态页面报头先发送给用户。

最简单的策略是将<head>部分内容作为HTTP的第一个块，而将网页中其他部分内容作为第二个块。换句话说，网页的分块类似下面范例：

```
<!doctype html>
<html>
<head>
    <title>My App</title>
</head>
<!-- end of chunk #1 -->
<body>
    ...
    <script src="all_20100426.js"></script>
</body>
</html>
<!-- end of chunk #2 -->
```

一个简单的改进是将第二块中的JavaScript代码移到第一块的<head>中。这样做使得浏览器可以在服务器没有准备好第二块的时候，就开始下载脚本文件：

```
<!doctype html>
<html>
<head>
    <title>My App</title>
```

```
</head>
<!-- end of chunk #1 -->
<body>
    ...
    <script src="all_20100426.js"></script>
</body>
</html>
<!-- end of chunk #2 -->
```

还有一个更好的做法是在网页文件的底部建立一个仅包含脚本文件的第三个块。如果在每个页面的顶部都有一些静态报头，可以将这部分内容放置在第一个块中：

```
<!doctype html>
<html>
<head>
    <title>My App</title>
</head>
<body>
    <div id="header">
       <img src="logo.png" />
       ...
    </div>
    <!-- end of chunk #1 -->
    ... The full body of the page ...

    <!-- end of chunk #2 -->
    <script src="all_20100426.js"></script
</body>
</html>
<!-- end of chunk #3 -->
```

这种方法非常适合渐进式增强的思想，并且不会影响到JavaScript代码的执行。一旦下载完HTML文件的第二部分后，就已经拥有一个完全载入、显示和可用的网页了，给用户看到的效果就好像JavaScript已经在浏览器中禁用了一样。等到JavaScript代码下载完毕后，它会增强网页的功能，并增加所有附加功能。

使用动态<script>元素来无阻塞地下载

如上所述，JavaScript会阻止所有后续文件的下载，但是有一些模式可以防范这个问题：

- 使用XHR请求载入脚本，并使用eval()将其转换为字符串。该方法受同一个域的限制，并且使用了eval()这种不好的模式。
- 使用defer和async属性，但是这种方法并不能在所有的浏览器上有效。
- 使用动态的<script>元素。

最后一种方法是一个比较好的、可实现的模式。类似于JSONP中所示，需要创建一个新的脚本元素，设置该元素的src属性，最后将该元素添加到网页文件中。

下面是一个异步载入JavaScript文件的范例，该过程不会阻塞网页文件中其他部分的下载：

```
var script = document.createElement("script");
script.src = "all_20100426.js";
document.documentElement.firstChild.appendChild(script);
```

该模式的缺点在于如果JavaScript脚本依赖于载入主.js文件，那么采用该模式后不能有其他脚本元素。主.*js*文件是异步载入的，因此无法保证该文件什么时候能够载入完毕，所以紧跟着主.*js*文件的脚本可能要假定对象都还未定义。

为了解决该缺点，可以让所有内联的脚本都不要立即执行，而是将这些脚本都收集起来放在一个数组里面。然后当主脚本文件载入完毕后，就可以执行说有缓存数组中收集的函数了。为了实现该目的，需要有三步：

首先，创建一个数组来存储所有内联代码，这部分代码应该放在页面文件尽可能前面的位置：

```
var mynamespace = {
    inline_scripts: []
};
```

然后，需要将所有单独的内联脚本封装到一个函数中，并将每个函数增加到inline_scripts数组中，如下所示：

```
// 过去是:
// <script>console.log("I am inline");</script>

// 修改为:
<script>
mynamespace.inline_scripts.push(function () {
    console.log("I am inline");
});
</script>
```

最后，循环执行缓存中的所有内联脚本：

```
var i, scripts = mynamespace.inline_scripts, max = scripts.length;
for (i = 0; i < max; max += 1) {
    scripts[i]();
}
```

增加<script>元素

通常来说，脚本是放置于文档的<head>区域，但是也可以将脚本文件放置于任何元素之内，包含body区域（和JSONP范例中类似）。

在之前的范例中我们使用documentElement来添加<head>，这是因为documentElement是指<html>，而它的第一个子元素就是<head>：

```
document.documentElement.firstChild.appendChild(script);
```

通常也可以这样写：

```
document.getElementsByTagName("head")[0].appendChild(script);
```

在能够掌控标记的时候，这样写是没有问题的。但是如果是创建一个小部件或者是一个广告，无法确定网页的类型该如何办呢？从技术上来说，可以在网页中不使用<head>和<body>，尽管document.body通常能够在没有<body>标签后正常运作：

```
document.body.appendChild(script);
```

但是实际上有一个标签一直会在脚本运行的网页中存在——<script>标签。如果没有<script>标签（用于内联或者外联文件），那么里面的JavaScript代码就不会运行。基于上述事实，可以在网页中使用insertBefore()来在第一个有效的元素之前插入元素：

```
var first_script = document.getElementsByTagName('script')[0];
first_script.parentNode.insertBefore(script, first_script);
```

在这里first_script是脚本元素，开发人员保证该脚本元素位于网页中，并且script是创建的最新脚本。

延迟加载

关于在页面载入完成后，载入外部文件的这种技术称为延迟加载。通常将一大段代码切分为两部分是十分有益的：

- 一部分代码是用于初始化页面并将事件处理器附加到UI元素上的。
- 第二部分代码只在用户交互或者其他条件下才用得上。

这样做的目的是希望渐进式地载入页面，尽可能快地提供目前需要使用的信息，而其余的内容可以在用户浏览该页面时在后台载入。

载入第二部分JavaScript代码的方法非常简单，只需要再一次为head或者body添加动态脚本元素：

```
... The full body of the page ...

<!-- end of chunk #2 -->
<script src="all_20100426.js"></script>
<script>
```

```
    window.onload = function () {
        var script = document.createElement("script");
        script.src = "all_lazy_20100426.js";
        document.documentElement.firstChild.appendChild(script);
    };
    </script>
</body>
</html>
<!-- end of chunk #3 -->
```

对于许多应用程序来说，延迟加载的代码部分远远大于立即加载的核心部分，因为很多有趣的“操作”（例如拖放操作、XHR和动画等）只在用户出发后发生。

按需加载

之前的模式在页面载入后，无条件地载入附加的JavaScript脚本，假定这些代码极有可能用得上。但是有没有办法可以设法只载入那部分确实需要的代码呢？

想象一下，在网页上有一个具有多个不同标签的侧边栏。单击一次标签会发出一个XHR请求来获取内容、更新标签内容，并且更新过程中标签颜色还有动画变化。假使这是唯一的一个需要XHR和动画库地方呢？又假使如果用户从未单击该标签呢？

这时请使用按需加载模式。可以创建一个`require()`函数或方法，该函数包含需要加载的脚本的名称和当附加脚本加载后需要执行的回调函数。

`require()`函数的用法如下：

```
require("extra.js", function () {
    functionDefinedInExtraJS();
});
```

让我们来看看该如何实现该函数。很明显，需要请求附加脚本，只需要按照动态`<script>`元素模式即可。根据不同的浏览器，计算出脚本加载的时间需要一些小技巧：

```
function require(file, callback) {

    var script = document.getElementsByTagName('script')[0],
        newjs = document.createElement('script');

    // IE浏览器
    newjs.onreadystatechange = function () {
        if (newjs.readyState === 'loaded' || newjs.readyState === 'complete') {
            newjs.onreadystatechange = null;
            callback();
        }
    };

    // 其他
    newjs.onload = function () {
```

```
            callback();
        };

        newjs.src = file;
        script.parentNode.insertBefore(newjs, script);
    }
```

下面是对于上述实现的一些解释：

- 在IE中订阅readystatechange事件，并寻找readyState状态为”loaded”或”complete”的状态。其他浏览器将会忽略这部分代码。
- 在Firefox、Safari和Opera中，需要通过onload属性订阅load事件。
- 这种方法不适用于Safari 2。如果确实需要支持该版本浏览器，请创建一个时间间隔来定期检查是否指定变量（在附加文件中定义的变量）已经定义。当该变量被定义以后，就意味着新脚本已经加载并执行了。

可以创建一个人为的延迟脚本（用于模拟网络延迟）来测试上述实现，为其命名为*ondemand.js.php*，该文件内容如下：

```
<?php
header('Content-Type: application/javascript');
sleep(1);
?>
function extraFunction(logthis) {
    console.log('loaded and executed');
    console.log(logthis);
}
```

现在测试一下require()函数：

```
require('ondemand.js.php', function () {
    extraFunction('loaded from the parent page');
    document.body.appendChild(document.createTextNode('done!'));
});
```

这段代码将会在控制台打印出两条直线，并更新网页，显示“done!”。如需完整的代码请参见*http://jspatterns.com/book/7/ondemand.html*

预加载JavaScript

在延迟加载模式和按需加载模式中，我们延迟加载当前页面需要的脚本。此外，还可以延迟加载当前页面不需要，但是在后续页面中可能需要的脚本。如此，当用户打开接下来的网页后，所需要的脚本已经预先加载了，进而用户感觉速度会加快了很多。

预加载可以使用动态脚本模式来实现。但是这意味着该脚本将被解析和执行。解析仅仅

会增加预加载的时间，而执行脚本可能会导致JavaScript错误，因为这些脚本本应该是在第二个页面执行的。例如寻找某个特定的DOM节点。

这是可以加载脚本而并不解析和执行这些脚本的。该方法对CSS和图像也同样有效。

在IE中可以使用熟悉的图像灯塔模式来发出请求：

```
new Image().src = "preloadme.js";
```

在所有其他浏览器中可以使用一个<object>来代替脚本元素，并将其data属性指向脚本的URL：

```
var obj = document.createElement('object');
obj.data = "preloadme.js";
document.body.appendChild(obj);
```

为了避免显示出该对象，应该将该对象的width和height属性都设置为0。

可以创建一个通用的preload()函数或者方法，并使用初始化分支模式（参考第4章）来处理浏览器差异：

```
var preload;
if (/*@cc_on!@*/false) { // 使用条件注释的IE 嗅探
    preload = function (file) {
        new Image().src = file;
    };
} else {
    preload = function (file) {
        var obj = document.createElement('object'),
            body = document.body;
        obj.width = 0;
        obj.height = 0;
        obj.data = file;
        body.appendChild(obj);
    };
}
```

这样就可以使用新函数了：

```
preload('my_web_worker.js');
```

这种模式的缺点在于使用了用户代理嗅探，但是这是无法避免的。因为在这种情形下，使用特性检测技术无法告知关于浏览器行为的足够信息。举例来说，在这种模式下如果typeof Image是一个函数，那么理论上可以用该函数来代替嗅探来进行测试。然而在这里该方法没有作用，因为所有的浏览器都支持new Image()；区别仅仅在于有的浏览器为图像有独立的缓存，这也就意味着作为图像预加载的组件不会被用做缓存中的脚本，因此下一个页面会再次下载该图像。

注意： 浏览器嗅探使用了分支注释是十分有趣的。该方法比在navigator.userAgent中寻找字符串要安全一些，因为那些字符串很容易被用户修改。

例如：

```
var isIE = /*@cc_on!@*/false;
```

上述语句会在除IE外其他所有浏览器中将isIE设置为false（因为这些浏览器会忽略注释语句）。但是在IE中isIE值为true，因为在注释语句中有一个“！”。因此在IE中该语句为：

```
var isIE = !false; // true
```

预加载模式可以用与各种类型组件，而不限于脚本。举例来说，这在登录页面就十分有用。当用户开始输入用户名时，可以使用输入的时间来启动预加载，因为用户下一步极有可能进入登录后的界面。

小结

鉴于本书之前的章节已经介绍了JavaScript的大部分核心模式，这些模式是和环境无关的。本章主要集中讨论在特定客户端浏览器环境下的模式。

本章讨论了如下内容：

- 关注点分离的思想（HTML：内容；CSS：表现；JavaScript：行为）、不引人注目的 JavaScript、以及与浏览器嗅探相对的特性检测（尽管在本章最后一部分介绍了如何打破该模式）。
- DOM脚本，加速DOM访问和处理的模式。主要包括批处理DOM操作，因为每次处理DOM都会消耗不少开销。
- 事件，跨浏览器事件处理和使用事件授权来减少事件监听器的数量，以改进性能。
- 两种处理长期高运算量脚本的模式。使用set Timeout()来将长期操作分解为更小的操作和在新款浏览器中使用Web Worker。
- 多种用于远程脚本的模式，这些远程脚本实现服务器和客户端直接的通信，主要包括XHR、JSONP、框架和图像灯塔。
- 在产品环境配置JavaScript。确保将脚本合并为较少的文件、精简并压缩（减少大约85%的文件大小）、将内容放置在CDN中和设置Expires报头来改善缓存。
- 如何将脚本合理地放置在网页中，以改进性能的模式，包含放置<script>元素的各种位置和从HTTP分块中获益。在加载大脚本文件时为了提高命中率，介绍了各种模式，包括延迟加载、预加载和按需加载JavaScript。

JavaScript模式

什么是使用JavaScript开发应用程序最好的方法呢？本书将使用大量JavaScript编码模式和最佳实践来帮您回答该问题。如果您是一名有经验的开发人员，正在寻找与对象、函数、继承以及其他特定语言分类相关的解决方案，那么本书中的抽象方案和代码模板将是十分理想的指南，无论您正在使用JavaScript编写客户端、服务端抑或是桌面应用程序。

《JavaScript模式》是由JavaScript专家Stoyan Stefanov撰写，Stoyan是Yahoo公司的资深技术员，他还是YSlow 2.0（一个Web性能优化工具）的技术架构师。本书包含了实现每个讨论的模式的实践建议，并附有数个可以立即上手的范例。同时还可以学到一些反模式，一些常见的编程方式，这些编程方式引发的问题比解决的还要多。

- **探索有用的习惯来编写高质量JavaScript代码，例如避免使用全局变量、使用单var声明等。**
- **学习为什么字面量表示模式是比构造函数更简洁的选择。**
- **探索在JavaScript中不同的定义函数的方法。**
- **创建超越基本模式对象的对象，基本模式对象通常使用对象字面量和构造函数。**
- **学习在JavaScript中使用代码重用和继承的一些有效选择。**
- **学习常见设计模式（单体模式、工厂模式、装饰模式）在JavaScript中的方法。**
- **检视应用于特定客户端浏览器环境下的模式。**

“Stoyan为JavaScript开发者编写了一本开发大规模Web应用程序的指南。”

——Ryan Grove
Yahoo公司YUI项目工程师

Stoyan Stefanov是Yahoo的Web开发员、YUI的合作者、演讲师和博客博主。他还是多本O'Reilly书籍的作者、贡献者和技术评审。Stoyan是smush.it图像优化工具的创建者和YSlow 2.0（一种Yahoo的性能优化工具）的体系架构师。

O'REILLY®
oreilly.com.cn

www.oreilly.com
O'Reilly Media, Inc.授权中国电力出版社出版

ISBN 978-7-5123-2923-2
9 787512 329232 >
定价：38.00元