HZ BOOKS
华章科技

从基本语法、应用架构、工具框架、编码风格、编程思想5个方面深入探讨
编写高质量JavaScript代码的技巧、禁忌和最佳实践

實戰

成林 著

Writing Solid JavaScript Code 188 Suggestions to improve Your JavaScript Program

编写高质量代码

改善JavaScript程序的188个建议

机械工业出版社
China Machine Press

實戰

51CTO.com
技术成就梦想 联合策划

作为一个Web开发工程师，你在编写JavaScript代码时是不是也被诸如以下这些问题所困扰：

- 如何减少全局变量的污染?
- 为什么不要使用类型构造器?
- 如何提高循环性能和条件性能的策略?
- 如何提高正则表达式的执行效率?
- 有哪些场景要避免使用正则表达式?
- 为什么要在循环体和异步回调中慎重使用闭包?
- 如何使用模块化规避缺陷?
- 为什么克隆节点比创建节点更好?
- 数据存储时如何考虑访问速度?
- 如何提高DOM的访问效率?

……

如果你曾经为类似于这样的一些问题感到疑惑不解或顿然大悟，说明你正在向JavaScript技术的巅峰攀登，正在成长为“振臂一呼，应者云集”的技术大牛，恭喜你！本书从不同的侧面出发，对JavaScript编码中的各种棘手的常见问题和“疑难杂症”奉献了真知灼见，相信你一定能从中受益。

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

上架指导：计算机\程序设计\JavaScript
ISBN 978-7-111-39905-6

9 787111 399056

定价：69.00元

實戰

Writing Solid JavaScript Code: 188 Suggestions to Improve Your JavaScript Program

编写高质量代码

改善JavaScript程序的188个建议

成林 著

机械工业出版社
China Machine Press

本书是 Web 前端工程师进阶修炼的必读之作，将为你通往“JavaScript 技术殿堂”指点迷津！内容全部由编写高质量的 JavaScript 代码的最佳实践组成，从基本语法、应用架构、工具框架、编码风格、编程思想等 5 大方面对 Web 前端工程师遇到的疑难问题给出了经验性的解决方案，为 Web 前端工程师如何编写更高质量的 JavaScript 代码提供了 188 条极为宝贵的建议。对于每一个问题，不仅以建议的方式给出了被实践证明为十分优秀的解决方案，而且还给出了经常被误用或被错误理解的不好的解决方案，从正反两个方面进行了分析和对比，犹如醍醐灌顶，让人豁然开朗。

本书针对每个问题所设计的应用场景都非常典型，给出的建议也都与实践紧密结合。书中的每一条建议都可能在你的下一行代码、下一个应用或下一个项目中被用到，建议你将此书放置在手边，随时查阅，一定能使你的学习和开发工作事半功倍。

图书在版编目（CIP）数据

编写高质量代码：改善 JavaScript 程序的 188 个建议 / 成林著 . —北京：机械工业出版社，2012.11

ISBN 978-7-111-39905-6

I. 编… II. 成… III. JAVA 语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2012）第 231084 号

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）
责任编辑：马 超
北京市荣盛彩色印刷有限公司印刷
2013 年 1 月第 1 版第 1 次印刷
186mm×240mm • 25.5 印张
标准书号：ISBN 978-7-111-39905-6
定价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

前　言

为什么要写这本书

JavaScript 是目前比较流行的 Web 开发语言。随着移动互联网、云计算、Web 3.0 和客户端开发概念的升温，JavaScript 语言不断成熟和普及，并被广泛应用于各种 B/S 架构的项目和不同类型的网站中。对于 JavaScript 初学者、网页设计爱好者以及 Web 应用开发者来说，熟练掌握 JavaScript 语言是必需的。

JavaScript 语言的最大优势在于灵活性好，适应能力强。借助各种扩展技术、开源库或框架，JavaScript 能够完成 Web 开发中各种复杂的任务，提升客户端用户体验。

作为资深的 Web 开发人员，笔者已经习惯了与高性能的编程语言和硬件打交道，因此一开始并没有对 JavaScript 编程有太高的期望。后来才发现，JavaScript 实际上是一种优秀且高效的编程语言，而且随着浏览器对其更好的支持、JavaScript 语言本身的性能提升，以及新的工具库加入，JavaScript 不断变得更好。JavaScript 结合 HTML 5 等为 Web 开发人员提供了真正可以发挥想象力的空间。Node.js 等新技术则为使用 JavaScript 对服务器进行编程描绘了非常美好的未来。

但是，在阅读网上大量散存的 JavaScript 代码时，笔者能明显感觉到很多用户正在误入“歧途”：编写的代码逻辑不清，结构混乱，缺乏编程人员应有的基本素养。这种现状一般都是用户轻视 JavaScript 语言所致。还有很多用户属于“半路出家”，误认为 JavaScript 就是一种“玩具语言”，没有以认真的态度对待和学习这门语言，书写代码也很随意。因此，笔者萌生了写一本以提高 JavaScript 代码编写质量为目的的书籍，在机械工业出版社华章公司杨福川编辑的鼓励和指导下，经过近半年的策划和准备，终于鼓起勇气动笔了。

本书特色

- **深**。本书不是一本语法书，它不会教读者怎么编写 JavaScript 代码，但它会告诉读者，为什么 Array 会比 String 类型效率高，闭包的自增是如何实现的，为什么要避免 DOM 迭代……不仅仅告诉读者 How（怎么做），而且还告诉读者 Why（为什么要这样做）。
- **广**。涉及面广。从编码规则到编程思想，从基本语法到系统框架，从函数式编程到面向对象编程，都有涉及，而且所有的建议都不是“纸上谈兵”，都与真实的场景相结合。
- **点**。从一个知识点展开讲解，比如继承，这里不提供继承的解决方案，而是告诉读者如何根据需要使用继承，如何设置原型，什么时候该用类继承，什么时候该用原型继承等。
- **精**。简明扼要。一个建议就是对一个问题的解释和说明，以及相关的解决方案，不拖泥带水，只针对一个问题进行讲解。
- **洁**。虽然笔者尽力把每个知识点写得生动，但代码就是代码，很多时候容不得深加工，最直接也就是最简洁的。

这是一本建议书。有这样一本书籍在手边，对如何编写出优雅而高效的代码提供指导，将是一件多么惬意的事情啊！

读者对象

本书适合以下读者阅读：

- 打算学习 JavaScript 的开发人员。
- 有意提升自己网站水平和 Web 应用程序开发能力的 Web 开发人员。
- 希望全面深入理解 JavaScript 语言的初学者。

此外，本书也适合熟悉下列相关技术的读者阅读：

- PHP/ASP/JSP
- HTML/ XML
- CSS

对于没有计算机基础知识的初学者，以及只想为网站添加简单特效和交互功能的读者，阅读本书前建议先阅读 JavaScript 基础教程类图书。

如何阅读本书

本书将改善 JavaScript 编程质量的 188 个建议以 9 章内容呈现：

❑第 1 章　JavaScript 语言基础

JavaScript 中存在大量的问题，这些问题会妨碍读者编写优秀的程序。应该避免 JavaScript 中那些糟糕的用法，因此本章主要就 JavaScript 语言的一些基本用法中容易犯错误的地方进行说明，希望能够引起读者的重视。

❑第 2 章　字符串、正则表达式和数组

JavaScript 程序与字符串操作紧密相连，在进行字符串处理时无时无刻不需要正则表达式的帮忙。如何提高字符串操作和正则表达式运行效率是很多开发者最易忽视的问题。同时，数组是所有数据序列中运算速度最快的一种类型，但很多初学者忽略了这个有用的工具。本章将就这 3 个技术话题展开讨论，通过阅读这些内容相信读者能够提高程序的执行效率。

❑第 3 章　函数式编程

函数式编程已经在实际应用中发挥了巨大作用，越来越多的语言不断地加入对诸如闭包、匿名函数等的支持。从某种程度上来讲，函数式编程正在逐步同化命令式编程。当然，用好函数并非易事，需要“吃透”函数式编程的本质，本章帮助读者解决在函数式编程中遇到的各种问题。

❑第 4 章　面向对象编程

JavaScript 采用的是以对象为基础，以函数为模型，以原型为继承机制的开发模式。因此，对于习惯于面向对象开发的用户来说，需要适应 JavaScript 语言的灵活性和特殊性。本章将就 JavaScript 类、对象、继承等抽象的问题进行探索，帮助读者走出“误区”。

❑第 5 章　DOM 编程

DOM 操作代价较高，在富网页应用中通常是一个性能瓶颈。因此，在 Web 开发中，需要特别注意性能问题，尽可能地降低性能损耗。本章将为读者提供一些好的建议，帮助读者优化自己的代码，让程序运行得更快。

❑第 6 章　客户端编程

在 JavaScript 开发中，很多交互效果都需要 CSS 的配合才能够实现，因此 CSS 的作用不容忽视。本章主要介绍 JavaScript+CSS 脚本化编程，以及 JavaScript 事件控制技巧。

❑第 7 章　数据交互和存储

数据交互和存储是 Web 开发中最重要的，也是最容易被忽视的问题，它也是高性能 JavaScript 的基石，是提升网站可用性的最大要素。本章主要介绍如何使用 JavaScript 提升数据交互的反应速度，以便更好地让数据在前、后台传递。

❑第 8 章　JavaScript 引擎与兼容性

JavaScript 兼容性是 Web 开发的一个重要问题。为了实现浏览器解析的一致性，需要找出不同引擎的分歧点在哪里。本章主要介绍各主流引擎在解析 JavaScript 代码时的分歧，使读者能够编写出兼容性很高的代码。

❑第 9 章　JavaScript 编程规范和应用

每种语言都存在缺陷。事实证明代码风格在编程中是非常重要的，好的风格促使代码能被更好地阅读，更为关键的是，它能够提高代码的执行效率。本章主要介绍如何提升 JavaScript 代码编写水平，主要包括风格、习惯、效率、协同性等问题，希望能够给读者带来帮助。

本书的期望

您是否曾经为了提供一个简单的应用解决方案而彻夜地查看源代码？

您是否曾经为了理解某个框架而冥思苦想、阅览群书？

您是否曾经为了提升 0.1s 的 DOM 性能而对多种实现方案进行严格测试和对比？

您是否曾经为了避免兼容问题而遍寻高手共同“诊治”？

……

在学习和使用 JavaScript 的过程中，您是否在原本可以很快掌握或解决的问题上耗费了大量的时间和精力？本书的很多内容都是笔者曾经付出代价换来的，希望它们能够给您带来一些帮助！

代码是一切的基石，一切都是以编码实现为前提的，通过阅读本书，期望为读者带来如下帮助：

❑能写出简单、清晰、高效的代码。

❑能架构一个稳定、健壮、快捷的应用框架。

❑能回答一个困扰很多人的技术问题。

❑能修复一个应用开发中遇到的大的 Bug。

❑能非常熟悉某个开源产品。

❑能提升客户端应用性能。

……

但是，“工欲善其事，必先利其器”，在“善其事”之前，先检查“器”是否已经磨得足够锋利了，是否能够在前进的路上披荆斩棘。无论将来的职业发展方向是架构师、设计师、分析师、管理者，还是其他职位，只要还与软件打交道，就有必要打好技术基础。本书所涉及的全部是核心的 JavaScript 编程技术，如果能全部理解并付诸实践，一定可以夯实 JavaScript 编程基础。

勘误和支持

除封面署名外，对本书编写提供帮助的还有：马本连、吴建华、江淑军、李斌、李经键、

郑伟、田蜜、陆颖、王慧明、张炜、陈锐、王幼平、杨龙贵、苏震巍、崔鹏飞等。由于作者的水平有限，加之编写时间仓促，书中难免会出现一些错误或不准确的地方，恳请读者批评指正。书中的全部源文件可以从华章网站（www.hzbook.com）下载。如果您有任何意见建议，欢迎发送邮件至邮箱 js_code@126.com，期待得到您的真挚反馈。

致谢

感谢机械工业出版社华章公司的杨福川编辑在这一年多的时间中始终支持我的写作，他的鼓励和帮助让我顺利完成了本书的编写工作。

最后感谢我的父母，感谢他们的养育之恩，感谢他们时时刻刻给我信心和力量！

谨以此书献给我最亲爱的家人，以及众多热爱 JavaScript 的朋友们！

成林

目　录

第1章 JavaScript 语言基础

对于任何语言来说，如何选用代码的写法和算法最终会影响到执行效率。与其他语言不同，由于 JavaScript 可用资源有限，所以规范和优化更为重要。代码结构是执行速度的决定因素之一：代码量少，运行速度不一定快；代码量多，运行速度也不一定慢。性能损失与代码的组织方式及具体问题的解决办法直接相关。

程序通常由很多部分组成，具体表现为函数、语句和表达式，它们必须准确无误地按照顺序排列。优秀的程序应该拥有前瞻性的结构，可以预见到未来所需要的修改。优秀的程序也有一种清晰的表达方式。如果一个程序被表达得很好，那么它更容易被理解，进而能够成功地被修改或修复。JavaScript 代码经常被直接发布，因此它应该自始至终具备发布质量。整洁是会带来价值的，通过在一个清晰且始终如一的风格下编写的程序会更易于阅读。

JavaScript 的弱类型和过度宽容特征，没有为程序质量带来安全编译时的保证，为了弥补这一点，我们应该按严格的规范进行编码。JavaScript 包含大量脆弱的或有问题的特性，这些会妨碍编写优秀的程序。我们应该避免 JavaScript 中那些糟糕的特性，还应该避免那些通常很有用但偶尔有害的特性。这样的特性让人既爱又恨，避免它们就能避免日后开发中潜在的错误。

建议 1：警惕 Unicode 乱码

ECMA 标准规定 JavaScript 语言基于 Unicode 标准进行开发，JavaScript 内核完全采用 UCS 字符集进行编写，因此在 JavaScript 代码中每个字符都使用两个字节来表示，这意味着可以使用中文来命名变量或函数名，例如：

```
var 人名 = "张三";
function 睡觉(谁){
    alert(谁 + ": 快睡觉！都半夜三更了。");
```

```
}
睡觉（人名）;
```

虽然 ECMAScript v3 标准允许 Unicode 字符出现在 JavaScript 程序的任何地方，但是在 v1 和 v2 中，ECMA 标准只允许 Unicode 字符出现在注释或引号包含的字符串直接量中，在其他地方必须使用 ASCII 字符集，在 ECMAScript 标准化之前，JavaScript 通常是不支持 Unicode 编码的。考虑到 JavaScript 版本的兼容性及开发习惯，不建议使用汉字来命名变量或函数名。

由于 JavaScript 脚本一般都“寄宿”在网页中，并最终由浏览器来解析和执行，因此在考虑到 JavaScript 语言编码的同时，还要顾及嵌入页面的字符编码，以及浏览器支持的编码。不过现在的浏览器一般都支持不同类型的字符集，只需要考虑页面字符编码与 JavaScript 语言编码一致即可，否则就会出现乱码现象。

当初设计 JavaScript 时，预计最多会有 65 536 个字符，从那以后慢慢增长到了一百万个字符。JavaScript 字符是 16 位的，这足够覆盖原有的 65 536 个字符，剩下的百万字符中的每一个都可以用一对字符来表示。

Unicode 把一对字符视为一个单一的字符，而 JavaScript 认为一对字符是两个不同的字符，这将会带来很多问题，考虑到代码的安全性，我们应该尽量使用基本字符进行编码。

建议 2：正确辨析 JavaScript 句法中的词、句和段

JavaScript 语法包含了合法的 JavaScript 代码的所有规则和特征，它主要分为词法和句法。词法包括字符编码、名词规则、特殊词规则等。词法侧重语言的底层实现（如语言编码问题等），以及基本规则的定义（如标识符、关键字、注释等）。它们都不是最小的语义单位，却是构成语义单位的组成要素。例如，规范字符编码集合、命名规则、标识符、关键字、注释规则、特殊字符用法等。

句法定义了语言的逻辑和结构，包括词、句和段的语法特性，其中段体现逻辑的结构，句表达可执行的命令，词演绎逻辑的精髓。

段落使用完整的结构封装独立的逻辑。在 JavaScript 程序中，常用大括号来划分结构，大括号拥有封装代码和逻辑的功能，由此形成一个独立的段落结构。例如，下面这些结构都可以形成独立的段落。

```
{
    // 对象
}
function (){
    // 函数
}
if (){
    // 条件
```

```
}
for (){
    // 循环
}
while (){
    // 循环
}
switch (){
    // 多条件
}
with (){
    // 作用域
}
try{
    // 异常处理
}
```

段落结构包含的内容可以是一条或多条语句。可以在段落起始标记（{）前面添加修饰词，如域谓词（with、catch）、逻辑谓词（if、while、for、switch 等）、函数谓词（function fn(arg)）等。

语句是由多个词构成的完整逻辑。在 JavaScript 中，常用分号（;）来划分语句，有时也可以省略分号，默认使用换行符表示完整的语句。

一条语句可以包含一个或多个词。例如，在下面两条语句中，第一条语句只有一个词，这是一个指令词，该指令只能位于循环体或 switch 结构体内。第二条语句包含 3 个词，alert 表示函数名（即变量），小括号表示运算符，而“"提示信息"”表示字符串直接量。

```
break;
alert("提示信息");
```

一条语句也可以包含一个或多个段落。例如，在下面这条语句中，直接把一个函数当做一个变量进行调用。

```
(function(i){
    alert(i);
})("提示信息");
```

还可以把函数包含在一个闭包中形成多个结构嵌套，这个嵌套结构体就构成了一个复杂的语句，例如：

```
(function(i){
    return function(){
        alert(i);
    };
})("提示信息")();
```

语句一般至少包含一个词或段落，但是语句也可以什么都不包含，仅由一个分号进行标识，这样的句子称为空语句。空语句常用做占位符。例如，在下面这个循环体内就包含了一个空语句。

```
for(var i; i<100;i++){
    ;
}
```

词语是 JavaScript 句法结构中的最小语义单位，包括指令（或称语句）、变量、直接量（或常量）、运算符等。在 JavaScript 中，词语之间必须使用分隔符进行分隔，否则 JavaScript 就会错误解析。下面的代码块是一个简单的求两个数平均值的方法。

```
var a = 34;
var b = 56;
function aver(c,d){
    return (c+d)/2;
}
alert(aver(a,b));
```

其中 var、function、return 是指令，这些指令也是 JavaScript 默认的关键字；a、b、c、d、aver、alert 为变量；34、56 是数值直接量；=、(、)、{、}、/、+、,是运算符。

建议 3：减少全局变量污染

定义全局变量有 3 种方式：

❑ 在任何函数外面直接执行 var 语句。

```
var f = 'value';
```

❑ 直接添加一个属性到全局对象上。全局对象是所有全局变量的容器。在 Web 浏览器中，全局对象名为 window。

```
window.f = 'value';
```

❑ 直接使用未经声明的变量，以这种方式定义的全局变量被称为隐式的全局变量。

```
 f = 'value';
```

为方便初学者在使用前无须声明变量而有意设计了隐式的全局变量，然而不幸的是忘记声明变量成了一个非常普遍的现象。JavaScript 的策略是让那些被忘记预先声明的变量成为全局变量，这导致在程序中查找 Bug 变得非常困难。

JavaScript 语言最为糟糕的就是它对全局变量的依赖性。全局变量就是在所有作用域中都可见的变量。全局变量在很小的程序中可能会带来方便，但随着程序越来越大，它很快变得难以处理。因为一个全局变量可以被程序的任何部分在任意时间改变，使得程序的行为被极大地复杂化。在程序中使用全局变量降低了程序的可靠性。

全局变量使在同一个程序中运行独立的子程序变得更难。如果某些全局变量的名称与子程序中的变量名称相同，那么它们将会相互冲突并可能导致程序无法运行，而且通常还使程序难以调试。

实际上，这些全局变量削弱了程序的灵活性，应该避免使用全局变量。努力减少使用全局变量的方法：在应用程序中创建唯一一个全局变量，并定义该变量为当前应用的容器。

```
var My = {};
My.name = {
    "first-name" : " first ",
    "last-name" : " last "
};
My.work = {
    number : 123,
    one : {
        name : " one ",
        time : "2012-9-14 12:55",
        city : "beijing"
    },
    two : {
        name : "two",
        time : "2012-9-12 12:42",
        city : "shanghai"
    }
};
```

只要把多个全局变量都追加在一个名称空间下，将显著降低与其他应用程序产生冲突的概率，应用程序也会变得更容易阅读，因为 My.work 指向的是顶层结构。当然也可以使用闭包体将信息隐藏，它是另一种有效减少“全局污染”的方法。

在编程语言中，作用域控制着变量与参数的可见性及生命周期。这为程序开发提供了一个重要的帮助，因为它减少了名称冲突，并且提供了自动内存管理。

```
var foo = function() {
    var a = 1, b = 2;
    var bar = function() {
        var b = 3, c = 4;          // a=1, b =3, c=4
        a += b + c;                // a=8, b =3, c=4
    };                             // a=1, b =2, c=undefined
    bar();                         // a=21, b =2, c=undefined
};
```

大多数采用 C 语言语法的语言都拥有块级作用域。对于一个代码块，即包括在一对大括号中的语句，其中定义的所有变量在代码块的外部是不可见的。定义在代码块中的变量在代码块执行结束后会被释放掉。但是，对于 JavaScript 语言来说，虽然该语言支持代码块的语法形式，但是它并不支持块级作用域。

JavaScript 支持函数作用域，定义在函数中的参数和变量在函数外部是不可见的，并且在一个函数中的任何位置定义的变量在该函数中的任何地方都可见。

其他主流编程语言都推荐尽可能迟地声明变量，但是在 JavaScript 中就不能够这样，因为它缺少块级作用域，最好的做法是在函数体的顶部声明函数中可能用到的所有变量。

建议 4：注意 JavaScript 数据类型的特殊性

1. 防止浮点数溢出

二进制的浮点数不能正确地处理十进制的小数，因此 0.1+0.2 不等于 0.3。

```
num = 0.1+0.2;     //0.30000000000000004
```

这是 JavaScript 中最经常报告的 Bug，并且这是遵循二进制浮点数算术标准（IEEE 754）而导致的结果。这个标准适合很多应用，但它违背了数字基本常识。幸运的是，浮点数中的整数运算是精确的，所以小数表现出来的问题可以通过指定精度来避免。例如，针对上面的相加可以这样进行处理：

```
a = (1+2)/10;      //0.3
```

这种处理经常在货币计算中用到，在计算货币时当然期望得到精确的结果。例如，元可以通过乘以 100 而全部转成分，然后就可以准确地将每项相加，求和后的结果可以除以 100 转换回元。

2. 慎用 JavaScript 类型自动转换

在 JavaScript 中能够自动转换变量的数据类型，这种转换是一种隐性行为。在自动转换数据类型时，JavaScript 一般遵循：如果某个类型的值被用于需要其他类型的值的环境中，JavaScript 就自动将这个值转换成所需要的类型，具体说明见表 1.1。

表 1.1 数据类型自动转换

值（value）	字符串操作环境	数字运算环境	逻辑运算环境	对象操作环境
undefined	"undefined"	NaN	false	Error
null	"null"	0	false	Error
非空字符串	不转换	字符串对应的数字值		
NaN	true	String		
空字符串	不转换	0	false	String
0	"0"	不转换	false	Number
NaN	"NaN"	不转换	false	Number
Infinity	"Infinity"	不转换	true	Number
Number.POSITIVE_INFINITY	"Infinity"	不转换	true	Number
Number.NEGATIVE_INFINITY	"-Infinity"	不转换	true	Number
Number.MAX_VALUE	"1.7976931348623157e+308"	不转换	true	Number

（续）

值（value）	字符串操作环境	数字运算环境	逻辑运算环境	对象操作环境
Number.MIN_VALUE	"5e–324"	不转换	true	Number
其他所有数字	" 数字的字符串值 "	不转换	true	Number
true	"true"	1	不转换	Boolean
false	"false"	0	不转换	Boolean
对象	toString()	valueOf() 或 toString() 或 NaN	true	不转换

如果把非空对象用在逻辑运算环境中，则对象被转换为 true。此时的对象包括所有类型的对象，即使是值为 false 的包装对象也被转换为 true。

如果把对象用在数值运算环境中，则对象会被自动转换为数字，如果转换失败，则返回值为 NaN。

当数组被用在数值运算环境中时，数组将根据包含的元素来决定转换的值。如果数组为空数组，则被转换为数值 0。如果数组仅包含一个数字元素，则被转换为该数字的数值。如果数组包含多个元素，或者仅包含一个非数字元素，则返回 NaN。

当对象用于字符串环境中时，JavaScript 能够调用 toString() 方法把对象转换为字符串再进行相关计算。当对象与数值进行加号运算时，则会尝试将对象转换为数值，然后参与求和运算。如果不能够将对象转换为有效数值，则执行字符串连接操作。

3. 正确检测数据类型

使用 typeof 运算符返回一个用于识别其运算数类型的字符串。对于任何变量来说，使用 typeof 运算符总是以字符串的形式返回以下 6 种类型之一：

- "number"
- "string"
- "boolean"
- "object"
- "function"
- "undefined"

不幸的是，在使用 typeof 检测 null 值时，返回的是“object”，而不是“null”。更好的检测 null 的方式其实很简单。下面定义一个检测值类型的一般方法：

```
function type(o){
    return (o === null) ? "null" : (typeof o);
}
```

这样就可以避开因为 null 值影响基本数据的类型检测。注意：typeof 不能够检测复杂的

数据类型，以及各种特殊用途的对象，如正则表达式对象、日期对象、数学对象等。

对于对象或数组，可以使用 constructor 属性，该属性值引用的是原来构造该对象的函数。如果结合 typeof 运算符和 constructor 属性，基本能够完成数据类型的检测。表 1.2 所示列举了不同类型数据的检测结果。

表 1.2 数据类型检测

值（value）	typeof value（表达式返回值）	value.constructor（构造函数的属性值）
var value = 1	"number"	Number
var value = "a"	"string"	String
var value = true	"boolean"	Boolean
var value = {}	"object"	Object
var value = new Object()	"object"	Object
var value = []	"object"	Array
var value = new Array()	"object"	Array
var value = function(){}	"function"	Function
function className(){};	"object"	className

使用 constructor 属性可以判断绝大部分数据的类型。但是，对于 undefined 和 null 特殊值，就不能使用 constructor 属性，因为使用 JavaScript 解释器会抛出异常。此时可以先把值转换为布尔值，如果为 true，则说明不是 undefined 和 null 值，然后再调用 constructor 属性，例如：

```
var value = undefined;
alert(typeof value);                    //"undefined"
alert(value && value.constructor);      //undefined
var value = null;
alert(typeof value);                    //"object"
alert(value && value.constructor);      //null
```

对于数值直接量，也不能使用 constructor 属性，需要加上一个小括号，这是因为小括号运算符能够把数值转换为对象，例如：

```
alert((10).constructor);
```

使用 toString() 方法检测对象类型是最安全、最准确的。调用 toString() 方法把对象转换为字符串，然后通过检测字符串中是否包含数组所特有的标志字符可以确定对象的类型。toString() 方法返回的字符串形式如下：

```
[object class]
```

其中，object 表示对象的通用类型，class 表示对象的内部类型，内部类型的名称与该

对象的构造函数名对应。例如，Array 对象的 class 为“Array”，Function 对象的 class 为“Function”，Date 对象的 class 为“Date”，内部 Math 对象的 class 为“Math”，所有 Error 对象（包括各种 Error 子类的实例）的 class 为“Error”。

客户端 JavaScript 的对象和由 JavaScript 实现定义的其他所有对象都具有预定义的特定 class 值，如“Window”、“Document”和“Form”等。用户自定义对象的 class 值为“Object”。

class 值提供的信息与对象的 constructor 属性值相似，但是 class 值是以字符串的形式提供这些信息的，而不是以构造函数的形式提供这些信息的，所以在特定的环境中是非常有用的。如果使用 typeof 运算符来检测，则所有对象的 class 值都为“Object”或“Function”，所以此时的 class 值不能够提供有效信息。

但是，要获取对象的 class 值的唯一方法是必须调用 Object 对象定义的默认 toString() 方法，因为不同对象都会预定义自己的 toString() 方法，所以不能直接调用对象的 toString() 方法。例如，下面对象的 toString() 方法返回的就是当前 UTC 时间字符串，而不是字符串“[object Date]”。

```
var d = new Date();
alert(d.toString());        // 当前 UTC 时间字符串
```

要调用 Object 对象定义的默认 toString() 方法，可以先调用 Object.prototype.toString 对象的默认 toString() 函数，再调用该函数的 apply() 方法在想要检测的对象上执行。结合上面的对象 d，具体实现代码如下：

```
var d = new Date();
var m = Object.prototype.toString;
alert(m.apply(d));          //" [object Date] "
```

下面是一个比较完整的数据类型安全检测方法。

```
// 安全检测 JavaScript 基本数据类型和内置对象
// 参数：o 表示检测的值
/* 返回值：返回字符串 "undefined"、"number"、"boolean"、"string"、"function"、"regexp"、
"array"、"date"、"error"、"object" 或 "null" */
function typeOf(o){
    var _toString = Object.prototype.toString;
    // 获取对象的 toString() 方法引用
    // 列举基本数据类型和内置对象类型，可以进一步补充该数组的检测数据类型范围
    var _type ={
        "undefined" : "undefined",
        "number" : "number",
        "boolean" : "boolean",
        "string" : "string",
        "[object Function]" : "function",
        "[object RegExp]" : "regexp",
        "[object Array]" : "array",
        "[object Date]" : "date",
```

```
            "[object Error]" : "error"
        }
        return _type[typeof o] || _type[_toString.call(o)] || (o ? "object" : "null");
    }
```

应用示例：

```
var a = Math.abs;
alert(typeOf(a));          //"function"
```

上述方法适用于 JavaScript 基本数据类型和内置对象，而对于自定义对象是无效的。这是因为自定义对象被转换为字符串后，返回的值是没有规律的，并且不同浏览器的返回值也是不同的。因此，要检测非内置对象，只能够使用 constructor 属性和 instaceof 运算符来实现。

4. 避免误用 parseInt

parseInt 是一个将字符串转换为整数的函数，与 parseFloat（将字符串转换为浮点数）对应，这两种函数是 JavaScript 提供的两种静态函数，用于把非数字的原始值转换为数字。

在开始转换时，parseInt 会先查看位置 0 处的字符，如果该位置不是有效数字，则将返回 NaN，不再深入分析。如果位置 0 处的字符是数字，则将查看位置 1 处的字符，并重复前面的测试，依此类推，直到发现非数字字符为止，此时 parseInt() 函数将把前面分析合法的数字字符转换为数值并返回。

```
parseInt("123abc");        // 123
parseInt("1.73");          // 1
parseInt(".123");          // NaN
```

浮点数中的点号对于 parseInt 来说属于非法字符，因此它不会被转换并返回，这样，在使用 parseInt 时，就存在潜在的误用风险。例如，我们并不希望 parseInt("16") 与 parseInt("16 tons") 产生相同的结果。如果该函数能够提醒我们出现额外文本就好了，但它不会那么做。

对于以 0 为开头的数字字符串，parseInt() 函数会把它作为八进制数字处理，先把它转换为数值，然后再转换为十进制的数字返回。对于以 0x 开头的数字字符串，parseInt() 函数则会把它作为十六进制数字处理，先把它转换为数值，然后再转换为十进制的数字返回。例如：

```
var d = "010";     // 八进制
var e = "0x10";    // 十六进制
parseInt(d);       //8
parseInt(e);       //16
```

如果字符串的第一个字符是 0，那么该字符串将基于八进制而不是十进制来求值。在八进制中，8 和 9 不是数字，所以 parseInt("08") 和 parseInt("09") 的结果为 0，这个错误导致了在程序解析日期和时间时经常会出现问题。幸运的是，parseInt 可以接受一个基数作为参数，

这样 parseInt("08",10) 结果为 8，parseInt("09",10) 结果为 9。因此，建议读者在使用 parseInt 时，一定要提供这个基数参数。

通过在 parseInt 中提供基数参数，可以把二进制、八进制、十六进制等不同进制的数字字符串转换为整数。例如，下面把十六进制数字字符串 "123abc" 转换为十进制整数。

```
parseInt("123abc",16);    // 1194684
```

再如，把二进制、八进制和十进制数字字符串转换为整数：

```
parseInt("10",2);         // 把二进制数字 10 转换为十进制整数为 2
parseInt("10",8);         // 把八进制数字 10 转换为十进制整数为 8
parseInt("10" ,10);       // 把十进制数字 10 转换为十进制整数为 10
```

建议 5：防止 JavaScript 自动插入分号

JavaScript 语言有一个机制：在解析时，能够在一句话后面自动插入一个分号，用来修改语句末尾遗漏的分号分隔符。然而，由于这个自动插入的分号与 JavaScript 语言的另一个机制发生了冲突，即所有空格符都被忽略，因此程序可以利用空格格式化代码。

这两种机制的冲突，很容易掩盖更为严重的解析错误。有时会不合时宜地插入分号。例如，在 return 语句中自动插入分号将会导致这样的后果：如果 return 语句要返回一个值，这个值的表达式的开始部分必须和 return 在同一行上，例如：

```
var f = function(){
    return
    {
        status: true
    };
}
```

看起来这里要返回一个包含 status 成员元素的对象。不幸的是，JavaScript 自动插入分号让它返回了 undefined，从而导致下面真正要返回的对象被忽略。

当自动插入分号导致程序被误解时，并不会有任何警告提醒。如果把 { 放在上一行的尾部而不是下一行的头部，就可以避免该问题，例如：

```
var f = function(){
    return {
        status: true
    };
}
```

为了避免省略分号引起的错误，建议养成好的习惯，不管一行内语句是否完整，只要是完整的语句都必须增加分号以表示句子结束。

为了方便阅读，当长句子需要分行显示时，在分行时应确保一行内不能形成完整的逻辑语义。例如，下面代码是一条连续赋值的语句，通过分行显示可以更清楚地查看它们的关

系。这种分行显示，由于一行内不能形成独立的逻辑语义，因此 JavaScript 不会把每一行视为独立的句子，从而不会产生歧义。

```
var a =
   b =
   c =  4;
```

以上语句在一行内显示如下：

```
var a = b = c = 4;
```

对于下面这条语句，如果不能正确分行显示，就很容易产生歧义。该句子的含义：定义一个变量 i，然后为其赋值，如果变量 a 为 true，则赋值为 1，否则就判断变量 b，如果 b 为 true，则赋值为 2，否则就判断变量 c，如果 c 为 true，则赋值为 3，否则赋值为 4。

```
var i = a ? 1 : b ? 2 : c ? 3 : 4;
```

下面的分行显示就是错误的，因为表达式 a ? 1: b 能够形成独立的逻辑语义，所以 JavaScript 会自动在其后添加分号来表示一个独立的句子。

```
var i = a ? 1: b
       ? 2 : c
       ? 3 : 4;
```

安全的方法应该采用如下的分行显示，这样每一行都不能形成独立的语义。

```
var i = a ? 1
     : b ? 2
     : c ? 3
     : 4;
```

总之，在编写代码时，应养成使用分号结束句子的良好习惯，凡是完整的句子就应该使用分号进行分隔。分行显示的句子应该确保单行不容易形成独立的合法的逻辑语义。

建议 6：正确处理 JavaScript 特殊值

1. 正确使用 NaN 和 Infinity

NaN 是 IEEE 754 中定义的一个特殊的数量值。它不表示一个数字，尽管下面的表达式返回的是 true。

```
typeof NaN === 'number'   // true
```

该值可能会在试图将非数字形式的字符串转换为数字时产生，例如：

```
+ '0'      // 0
+ 'oops'   // NaN
```

如果 NaN 是数学运算中的一个运算数，那么它与其他运算数的运算结果就会是 NaN。如果有一个表达式产生出 NaN 的结果，那么至少其中一个运算数是 NaN，或者在某个地方产生了 NaN。

可以对 NaN 进行检测，但是 typeof 不能辨别数字和 NaN 的区别，并且 NaN 不等同于它自己，所以，下面的代码结果令人惊讶。

```
NaN === NaN        // false
NaN !== NaN        // true
```

为了方便检测 NaN 值，JavaScript 提供 isNaN 静态函数，以辨别数字与 NaN 区别。

```
isNaN(NaN)         // true
isNaN(0)           // false
isNaN('oops')      // true
isNaN('0')         // false
```

判断一个值是否可用做数字的最佳方法是使用 isFinite 函数，因为它会筛除掉 NaN 和 Infinity。Infinity 表示无穷大。当数值超过浮点数所能够表示的范围时，就要用 Infinity 表示。反之，负无穷大为 –Infinity。

使用 isFinite 函数能够检测 NaN、正负无穷大。如果是有限数值，或者可以转换为有限数值，那么将返回 true。如果只是 NaN、正负无穷大的数值，则返回 false。

不幸的是，isFinite 会试图把它的运算数转换为一个数字。因此，如果值不是一个数字，使用 isFinite 函数就不是一个有效的检测方法，这时不妨自定义 isNumber 函数。

```
var isNumber = function isNumber(value) {
    return typeof value === 'number' && isFinite(value);
}
```

2. 正确使用 null 和 undefined

JavaScript 有 5 种基本类型：String、Number、Boolean、Null 和 Undefined。前 3 种都比较好理解，后面两种就稍微复杂一点。Null 类型只有一个值，就是 null；Undefined 类型也只有一个值，即 undefined。 null 和 undefined 都可以作为字面量在 JavaScript 代码中直接使用。

null 与对象引用有关系，表示为空或不存在的对象引用。当声明一个变量却没有向它赋值的时候，它的值就是 undefined。undefined 的值会在如下情况中出现：

- 从一个对象中获取某个属性，如果该对象及其 prototype 链中的对象都没有该属性，该属性的值为 undefined。
- 一个函数如果没有显式通过 return 语句将返回值返回给其调用者，其返回值就是 undefined，但在使用 new 调用函数时例外。
- JavaScript 的函数可以声明任意多个形参，当该函数实际被调用时，传入的参数的个数如果小于声明的形式参数的个数，那么多余的形式参数的值为 undefined。

如果对值为 null 的变量使用 typeof 检测，得到的结果是“object”，而 typeof undefined 的值为“undefined”。null == undefined，null !== undefined。

与 null 不同，undefined 不是 JavaScript 的保留字，在 ECMAScript v3 标准中才定义 undefined 为全局变量，初始值为 undefined。因此，在使用 undefined 值时就存在一个兼容问题（早期浏览器可能不支持 undefined）。除了直接赋值和使用 typeof 运算符外，其他任何运算符对 undefined 的操作都会引发异常。不过，可以声明 undefined 变量，然后查看它的值，如果它的值为 undefined，则说明浏览器支持 undefined 值。例如：

```
var undefined;
alert(undefined);
```

如果浏览器不支持 undefined 关键字，可以自定义 undefined 变量，并将其赋值为 undefined。例如：

```
var undefined = void null;
```

声明变量为 undefined，将其初始化为表达式 void null 的值，由于运算符 void 在执行其后的表达式时会忽略表达式的结果值，而总是返回值 undefined，因此利用这种方法可以定义一个变量为 undefined，并将其赋值为 undefined。既然是将变量 undefined 赋值为 undefined，还可以使用如下方式：

```
var undefined = void 1;
```

或者使用没有返回值的函数：

```
var undefined = function(){}();
alert(undefined);  //"undefined"
```

可以使用 typeof 运算符来检测某个变量的值是否为 undefined：

```
var a;
if(typeof a == "undefined"){
}
```

3. 使用假值

JavaScript 的类型系统是非常混乱的，类型特性不明显，而且交叉错乱。JavaScript 语法系统拥有一大组假值，如以下代码所示。这些值的布尔值都是 false。

```
0                    //Number
NaN                  //Number
''                   //String
false                //Boolean
null                 //Object
undefined            //Undefined
```

这些值全部都等同于 false，但它们是不可互换的。例如，下面用法是错误的。

```
value = myObject[name];
if(value == null) {
}
```

这是在用一种错误的方式去确定一个对象是否缺少一个成员属性。undefined 是缺失的成员属性值，而上面代码片段用 null 来测试，使用了会强制类型转换的 == 运算符，而不是更可靠的 === 运算符。正确的用法如下：

```
value = myObject[name];
if(!value) {
}
```

undefined 和 NaN 并不是常见，它们是全局变量，还可以改变它们的值，虽然在程序设计中不应该采取这种做法，但可以改变它们的值。

建议 7：小心保留字的误用

JavaScript 语言中定义了很多备用或已经使用的保留字，按首字母顺序列出的保留字见表 1.3。

表 1.3　JavaScript 语言中定义的保留字

首字母	保留字
a	abstract
b	boolean、break、byte
c	case、catch、char、class、const、continue
d	debugger、default、delete、do、double
e	else、enum、export、extends
f	false 、final、finally、float、for、function
g	goto
i	if、implements 、import 、in 、instanceof 、int 、interface
l	long
n	native 、new 、null
p	package、private、protected、public
r	return
s	short、static、super、switch 、synchronized
t	this 、throw、throws、transient、true、try、typeof
v	var、volatile、void
w	while、with

这些单词中的大多数并没有在语言中使用，但是根据 JavaScript 语法规则，这些单词是不能用来命名变量或参数的。当保留字作为对象字面量的键值时，必须用引号括起来。保留字不能用在点语法中，所以有时必须使用中括号表示法。例如，下面的用法是合法的。

```
var method;
object = {box: value};
object = {'case': value};
object.box = value;
object['case'] = value;
```

但是，下面的用法就是非法的。

```
var class;
object = {case: value};
object.case = value;
```

各个浏览器对保留字的使用限制不同。例如，下面代码在 Firefox 中是合法的，而在其他浏览器中就是不合法的。

```
object = {case: value};
```

此外，不同的保留字也各不相同。例如，下面代码在 Firefox 和 Opera 9.5 中是合法的，但在 IE 和 Safari 中依然是不合法的。

```
object= {class:value};
```

对于 int、long、float 等保留字，它们在各浏览器中都可以用做变量名及对象字面量的键值。尽管如此，在这些场合依然不建议使用任何保留字。

建议 8：谨慎使用运算符

1. 用 ===，而不用 ==

JavaScript 有两组相等运算符：=== 和 !==、== 和 !=。=== 和 !== 这一组运算符会按照期望的方式工作。如果两个运算数类型一致且拥有相同的值，那么 === 返回 true，而 !== 返回 false。== 和 != 只有在两个运算数类型一致时才会做出正确的判断，如果两个运算数是不同的类型，会试图强制转换运算数的类型。转换的规则复杂且难以记忆，具体规则如下：

```
'' == '0'  // false
0 == ''    // true
0 == '0'   // true
false == 'false'  // false
false == '0'      // true
false == undefined // false
false == null     // false
null == undefined  // true
```

上面表达式如果全部使用 === 运算符，则都会返回 true。

== 和 != 运算符缺乏传递性，需要引起警惕。所谓传递性就是：如果 a==b 为 true，b==c 为 true，则 a==c 也为 true。因此，在 JavaScript 开发中，建议永远不要使用 == 和 !=，而选用 === 和 !== 运算符。

下面分别介绍一下 === 和 == 运算符的算法。

（1）=== 运算符的算法

在使用 === 来判断两个值是否相等时，如判断 x===y，会先比较两个值的类型是否相同，如果不相同，直接返回 false。如果两个值的类型相同，则接着根据 x 的类型展开不同的判断逻辑：

- 如果 x 的类型是 Undefined 或 Null，则返回 true。
- 如果 x 的类型是 Number，只要 x 或 y 中有一个值为 NaN，就返回 false；如果 x 和 y 的数字值相等，就返回 true；如果 x 或 y 中有一个是 +0，另外一个是 –0，则返回 true。
- 如果 x 的类型是 String，当 x 和 y 的字符序列完全相同时返回 true，否则返回 false。
- 如果 x 的类型是 Boolean，当 x 和 y 同为 true 或 false 时返回 true，否则返回 false。
- 当 x 和 y 引用相同的对象时返回 true，否则返回 false。

（2）== 运算符的算法

在使用 == 来判断两个值是否相等时，如判断 x==y，如果 x 和 y 的类型一样，判断逻辑与 === 一样；如果 x 和 y 的类型不一样，== 不是简单地返回 false，而是会进行一定的类型转换。

- 如果 x 和 y 中有一个是 null，另外一个是 undefined，返回 true，如 null == undefined。
- 如果 x 和 y 中有一个类型是 String，另外一个类型是 Number，会将 String 类型的值转换成 Number 来比较，如 3 == "3"。
- 如果 x 和 y 中有一个类型是 Boolean，会将 Boolean 类型的值转换成 Number 来比较，如 true == 1、true == "1"。
- 如果 x 和 y 中有一个类型是 String 或 Number，另外一个类型是 Object，会将 Object 类型的值转换成基本类型来比较，如 [3,4] == "3,4"。

2. 谨慎使用 ++ 和 --

递增（++）和递减（--）运算符使程序员可以用非常简洁的风格去编码，如在 C 语言中，它们使得通过一行代码实现字符串的复制成为可能，例如：

```
for (p = src, q = dest; !*p; p++, q++) *q = *p;
```

事实上，这两个运算符容易形成一种不谨慎的编程风格。大多数的缓冲区溢出错误所造成的安全漏洞都是由于这种编码导致的。

当使用 ++ 和 -- 时，代码往往变得过于紧密、复杂和隐晦。因此，在 JavaScript 程序设计中不建议使用它们，从而使代码风格变得更为整洁。

++ 和 -- 运算符只能够作用于变量、数组元素或对象属性。下面的用法是错误的。

```
4++;
```

正确的用法如下：

```
var n = 4;
n++;
```

++ 和 -- 运算符的位置不同所得运算结果也不同。例如，下面的递增运算符是先执行赋值运算，然后再执行递加运算。

```
var n = 4;
n++;        // 4
```

而下面的递增运算符是先执行递加运算，再进行赋值运算。

```
var n = 4;
++n;
```

3. 小心逗号运算符

逗号在 JavaScript 语言中表示连续运算，并返回最后运算的结果。例如，在下面这个示例中，JavaScript 先运算第一个数值直接量，再运算第二个数值直接量，然后运算第三个数值直接量，最后运算第四个数值直接量，并返回最后一个运算值 4。

```
var a = ( 1, 2, 3, 4);
alert(a);                          //4
```

再如：

```
a = 1, b = 2, c = 3;
```

等价于：

```
a = 1;
b = 2;
c = 3;
```

作为运算符，逗号一般用在特殊环境中，即在只允许出现一个句子的地方，把几个不同的表达式句子合并成一个长句。在 JavaScript 实际开发中，逗号运算符常与 for 循环语句联合使用。例如，在下面这个简单的 for 循环结构中，通过连续的运算符在参数表达式中运算多个表达式，以实现传递或运算多个变量或表达式。

```
for(var a = 10 , b = 0; a > b; a++ , b+=2){
    document.write("a = " + a +"  b = " + b + "<br>");
}
```

逗号运算符比较“怪异”，稍不留心就会出错。例如，在下面这个简单的示例中，变量 a 的返回值为 1，而不是连续运算后的返回值 4。

```
a = 1, 2, 3, 4;
alert(a);                    //1
```

第一个数值 1 先赋值给变量 a，然后 a 再参与连续运算，整个句子的返回值为 4，而变量 a 的返回值为 1，代码演示如下：

```
alert((a = 1, 2, 3, 4));          //4
alert(a = (1, 2, 3, 4));          //4
```

要确保变量 a 的值为连续运算，可以使用小括号逻辑分隔符强迫 4 个数值先进行连续运算，然后再赋值。

```
a = (1, 2, 3, 4);
alert(a);                    //4
```

当使用 var 关键字来定义变量时，会发现 a 最终没有返回值。

```
var a = 1, 2, 3, 4;
alert(a);                    //null
```

要确保 var 声明的变量正确返回连续运算的值，需要使用小括号先强迫数值进行计算，最后再把连续运算的值赋值给变量 a。

```
var a = (1, 2, 3, 4);
alert(a);                    //4
```

4. 警惕运算符的副作用

JavaScript 运算符一般不会对运算数本身产生影响，如算术运算符、比较运算符、条件运算符、取逆、“位与”等。例如，a = b + c，其中的运算数 b 和 c 不会因为加法运算而导致自身的值发生变化。

但在 JavaScript 中还有一些运算符能够改变运算数自身的值，如赋值、递增、递减运算等。由于这类运算符自身的值会发生变化，在使用时会具有一定的副作用，特别是在复杂表达式中，这种副作用更加明显，因此在使用时应该时刻保持警惕。例如，在下面代码中，变量 a 经过赋值运算和递加运算后，其值发生了两次变化。

```
var a;
a = 0;
a++;
alert(a);                    //1
```

再如：

```
var a;
a = 1;
```

```
a = (a++)+(++a)-(a++)-(++a);
alert(a);                          //-4
```

如果直观地去判断，会错误地认为返回值为0，实际上变量a在参与运算的过程中，变量a的值是不断发生变化的。这种变化很容易被误解。为了方便理解，进一步拆解(a++)+(++a)–(a++)–(++a)表达式：

```
var a;
a = 1;
b = a++;
c = ++a;
d = a++;
e = ++a;
alert(b+c-d-e);                    //-4
```

如果表达式中还包含其他能够引起自身值发生变化的运算，那么整个表达式的逻辑就无法用人的直观思维来描述了。因此，从代码可读性角度来考量：在一个表达式中不能够对相同操作数执行两次或多次引起自身值变化的运算，除非表达式必须这样执行，否则应该避免产生歧义。这种歧义在不同编译器中会产生完全不同的解析结果。例如，下面的代码虽然看起来让人头疼，但由于每个运算数仅执行了一次引起自身值变化的运算，所以不会产生歧义。

```
a = (b++)+(++c)-(d++)-(++e);
```

建议 9：不要信任 hasOwnProperty

hasOwnProperty 方法常被用做一个过滤器，用来消除 for in 语句在枚举对象属性时的弊端。考虑到 hasOwnProperty 是一个方法，而不是一个运算符，因此，在任何对象中，它可能会被一个不同的函数甚至一个非函数的值所替换。

例如，在下面代码中，obj 对象的 hasOwnProperty 成员被清空了，此时如果再利用这个方法来过滤出 obj 对象的本地属性就会失败。

```
var obj={}, name;
obj.hasOwnProperty = null;
for(name in obj) {
    if(obj .hasOwnProperty(name)) {
        document.writeln(name + ': ' + obj [name]);
    }
}
```

建议 10：谨记对象非空特性

JavaScript 从来没有真正的空对象，因为每个对象都可以从原型链中取得成员，这种机制

会带来很多麻烦。例如，在编写统计一段文本中每个单词的出现次数的代码时可以这样设计：先使用 toLowerCase 方法统一转换文本为小写格式，接着使用 split 方法以一个正则表达式为参数生成一个单词数组，然后可以遍历该数组中每个单词，并统计每个单词出现的次数。

```
var i, word;
var text = "A number of W3C staff will be on hand to discuss HTML5, CSS, and other
technologies of the Open Web Platform.";
var words = text.toLowerCase().split(/[\s,.]+/);
var count = {};
for( i = 0; i < words.length; i += 1) {
    word = words[i];
    if(count[word]) {
        count[word] += 1;
    } else {
        count[word] = 1;
    }
}
```

在执行结果中，count["on"] 的值为 1，count["of"] 的值是 3，而 count.constructor 却包含着一个看上去令人不可思议的字符串。在主流浏览器上，count.constructor 将会返回字符串：function Object (){[native code]}。其原因是 count 对象继承自 Object.prototype，而 Object.prototype 包含一个名为 constructor 的成员对象。count.constructor 的值是一个 Object。+= 运算符，就像 + 运算符一样，如果它的运算数不是数字时会执行字符串连接的操作而不是加法运算。因为 count 对象是一个函数，所以 += 运算符将其转换成一个莫名其妙的字符串，然后再把一个数字 1 加在它的后面。

采用与处理 for in 中问题相同的方法来避免类似的问题：用 hasOwnProperty 方法检测成员关系，或者查找特定的类型。在当前情形下，可以编写如下过滤条件：

```
if (typeof count[word] === 'number') {
}
```

建议 11：慎重使用伪数组

JavaScript 没有真正的数组，因此 typeof 运算符不能辨别数组和对象。伪数组在 JavaScript 中有很高的易用性，程序员不用给它设置维度，而且永远不用担心产生越界错误，但 JavaScript 数组的性能相比真正的数组可能更糟糕。要判断一个值是否为数组，必须使用 constructor 属性，例如：

```
if(value && typeof value === 'object' && value.constructor === Array) {
}
```

arguments 不是一个数组，它是一个带有 length 成员属性的对象，很多时候会把它理解为一个伪数组。使用上面的检测方法会将 arguments 识别为一个数组，有时候这是希望得到

的结果，尽管 arguments 不包含任何数组的方法。

建议 12：避免使用 with

with 语句的语法如下：

```
with ( Expression )
    Statement
```

with 会把由 Expression 计算出来的对象添加到当前执行上下文的作用域链的前面，然后使用这个扩大的作用域链来执行语句 Statement，最后恢复作用域链。不管其中的语句是否正常退出，作用域链都会被恢复。

由于 with 会把额外的对象添加到作用域链的前面，因此使用 with 可能会影响性能，并造成难以发现的错误。由于额外的对象在作用域链的前面，当执行到 with 语句，需要对标识符求值时，会先沿着该对象的 prototype 链查找。如果找不到，才会依次查找作用域链中原来的对象。因此，如果在 with 语句中频繁引用不在额外对象的 prototype 链中的变量，查找的速度会比不使用 with 慢，例如：

```
function A() {
    this.a = "A";
}
function B() {
    this.b = "B";
}
B.prototype = new A();
function C() {
    this.c = "C";
}
C.prototype = new B();
(function() {
    var myVar = "Hello World";
    alert(typeof a);      // "undefined"
    var a = 1;
    var obj = new C();
    with(obj) {
        alert(typeof a); //"string"
        alert(myVar);     // 查找速度比较慢
    }
    alert(typeof a);      // "number"
})();
```

在上面代码中，先通过 prototype 方式实现了继承。在 with 语句中，执行 alert(typeof a) 时需要查找变量 a，由于 obj 在作用域链的前面，而 obj 中也存在名为 a 的属性，因此 obj 中的 a 被找到。执行 alert(myVar) 需要查找变量 myVal，而 obj 中不存在名为 myVal 的属性，会继续查找作用域链中后面的对象，因此使用 with 比不使用 with 的速度慢。需要注意的是，最后

一条语句 alert(typeof a) 不在 with 中，因此查找到的 a 是之前声明的 number 型的变量。

使用 with 语句可以快捷地访问对象的属性，然而，得到的结果有时可能是不可预料的，所以应该避免使用它。例如：

```
with (obj) {
    a = b;
}
```

上面代码与下面的代码完成的是同样的事情。

```
if (obj.a === undefined) {
    a = obj.b === undefined ? b : obj.b;
} else {
    obj.a = obj.b === undefined ? b : obj.b;
}
```

因此，前面代码等价以下语句中的任何一条。

```
a = b;
a = obj.b;
obj.a = b;
obj.a = obj.b;
```

直接阅读代码不可能辨别出会得到这些语句中的哪一条。a 和 b 可能随着程序运行到下一步时发生变化，甚至可能在程序运行过程中就发生变化了。如果不能通过阅读程序来了解它将会做什么，就无法确信它是否会正确地执行我们要求的事情。

with 语句在 JavaScript 语言中存在，本身就严重影响了 JavaScript 处理器的速度，因为它阻止了变量名的词法作用域绑定。它的本意是好的，但如果没有它，JavaScript 语言可能会更好。

建议 13：养成优化表达式的思维方式

对同一个表达式稍加改动就会打乱表达式的逻辑运算顺序，因此我们应该学会优化表达式的结构，不改变表达式的运算顺序和结果即可提高代码的可读性。

1. 第一种方式——加小括号

例如，面对下面这个复杂表达式，可能被 && 和 || 的优先级所迷惑。

```
(a + b > c && a - b < c || a > b > c)
```

不过，如果进行如下优化，逻辑运算的顺序就会非常清晰了。

```
((a + b > c) && ((a - b < c) || (a > b > c)))
```

虽然增加这些小括号显得多余，但是这么写并没有影响表达式的实际运算，反而带来了

非常明显的好处。学会使用小括号分隔符来分隔表达式的逻辑层次，不失为一种高明之举。

使用小括号分隔符来优化表达式内部的逻辑层次，是一种很好的设计习惯。如果复杂表达式中存在一些与人的思维方式相悖的不良的逻辑结构，也会影响人们对代码的阅读和思考，这个时候就应该根据人的思维习惯来优化表达式的逻辑结构。

2. 第二种方式——改变表达式结构顺序

例如，想设计一个表达式来筛选学龄人群，即年龄大于或等于 6 岁且小于 18 岁的人：

```
if(age >= 6 && age < 18){
}
```

直观阅读，表达式 age>=6 && age<18 可以很容易被每一个人所理解。继续复杂化表达式：筛选所有弱势年龄人群，以便在购票时实施半价优惠，即年龄大于或等于 6 岁且小于 18 岁，或者年龄大于或等于 65 岁的人：

```
if(age >= 6 && age < 18 || age >= 65){
}
```

从逻辑上分析，上面表达式没有错误。但是从结构上分析就感觉比较模糊，为此我们可以使用小括号来分隔逻辑结构层次，以方便阅读。

```
if((age >= 6 && age < 18) || age >= 65){
}
```

但是，此时如果根据人的思维来思考条件表达式的逻辑顺序时，会发现它有些紊乱，与人的一般思维方式发生了错位。人的思维是一种线性的、有联系、有参照的一种方式，如图 1.1 所示。

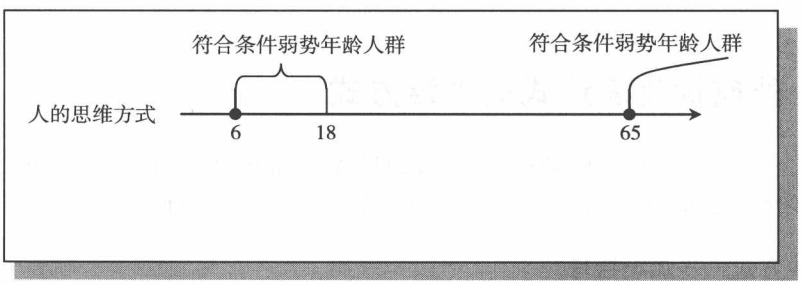


图 1.1 人的思维模型图

对于表达式（age >= 6 && age < 18）|| age >= 65 来说，其思维模型图如图 1.2 所示。通过对此模型图的直观分析，会发现该表达式的逻辑是非线性的，呈现多线思维的交叉型，这种思维结构对于机器计算来说基本上没有任何影响。但是对于人脑思维来说，就需要认真思考之后，才能把这个表达式中各个小的表达式逻辑单元串联在一起，形成一个完整的逻辑线。

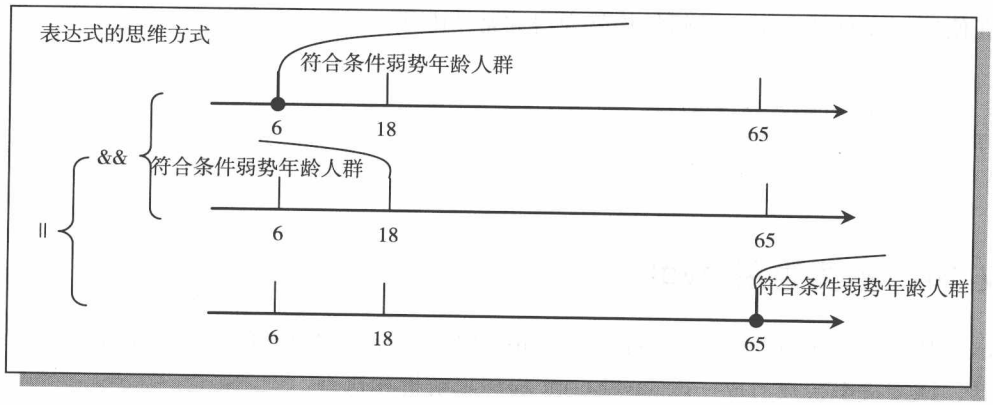


图 1.2　表达式的思维模型图

直观分析，这个逻辑结构的错乱是因为随意混用大于号和小于号等运算符造成的。如果调整一下表达式的结构顺序，阅读起来就非常清晰了。

```
if(( 6 <= age && age < 18) || 65 <= age ){
}
```

这里采用了统一的小于号方式，即所有参与比较的项都按着从左到右、从小到大的思维顺序进行排列，而不再遵循变量始终居左、比较值始终居右的传统编写习惯。

3. 第三种方式——避免布尔表达式的叠加

表达式中的另一个“顽疾”是布尔型表达式的重叠所产生的迷宫。在复杂表达式中，这种多重叠加的布尔表达式足以“令人生畏”。例如，对于下面这个条件表达式：

```
if !(!isA || !isB) {
}
```

如果采用“剥洋葱皮”的方法逐层分析，也可以找到结果。当然，也可以对这样的表达式进行优化，以方便阅读。例如：

```
( ! isA || ! isB) = ! (isA && isB)
( ! isA && ! isB) = ! (isA || isB)
```

?运算符为很多程序员所喜爱，特别是在 JavaScript 函数式编程中，这个运算符的使用频率会更高。但对于很多开发者来说，由于难以适应这种连续思维，会在阅读代码时产生障碍。为此，可以适当采用 if 条件语句对 ?运算符表达式进行分解。

4. 第四种方式——if 语句分解

例如，对于下面这个复杂表达式，如果不仔细进行分析，很难理清它的逻辑顺序。

```
var a.b = new c(a.d ? a.e(1) : a.f(1))
```

在使用 if 条件语句后，逻辑结构就变得非常清晰了。

```
if(a.d){
   var a.b = new c(a.e(1));
}else{
   var a.b = new c(a.f(0));
}
```

建议 14：不要滥用 eval

eval 是一个被滥用得很严重的 JavaScript 特性。eval 函数传递一个字符串给 JavaScript 编译器，该字符串会被当成一段 JavaScript 程序来解析和执行。

很多开发者对 JavaScript 语言一知半解，却喜欢使用 eval。例如，如果只知道点表示法，却不知道下标表示法，就会按如下方法编写代码：

```
eval("value = obj." + key + ";");
```

而不是按如下方法编写：

```
value = obj[key];
```

使用 eval 形式的代码会更加难以阅读。这种形式将使代码性能显著降低，因为 eval 必须运行编译器，同时这种形式减弱了 Web 应用的安全性，因为它向被求值的文本授予了太多的权限。使用 eval 与使用 with 语句一样，降低了语言的性能。

除了显式调用 eval 外，JavaScript 还支持隐式调用 eval。Function 构造器是 eval 的另一种形式，所以也应该避免使用它。当传递的是字符串参数时，setTimeout 和 setInterval 函数（浏览器提供的函数，能接受字符串参数或函数参数）会像 eval 那样去处理，因此也应该避免使用字符串参数形式。例如，下面是使用函数参数形式进行的处理。

```
var obj = {
    show1 : function() {
        alert(" 时间到！ ");
    },
    show2 : function() {
        alert("10 秒一次的提醒！ ");
    };
};
setTimeout(obj.show1, 1000);
setTimeout("obj.show1();", 2000);
setInterval(obj.show2, 10000);
setInterval("obj.show2();", 10000);
```

在 Ajax 应用中，JSON 是一种流行的浏览器端与服务器端之间传输数据的格式。服务器端传过来的数据在浏览器端通过 JavaScript 的 eval 方法转换成可以直接使用的对象。然而，在浏览器端执行任意的 JavaScript 会带来潜在的安全风险，恶意的 JavaScript 代码可能会破

坏应用。对于这个问题，有两种解决方法：

- 带注释的 JSON（JSON comments filtering）。
- 带前缀的 JSON（JSON prefixing）。

这两种方法都是在 Dojo 中用来避免 JSON 劫持（JSON hijacking）的方法。带注释的 JSON 指的是从服务器端返回的 JSON 数据都是带有注释的，浏览器端的 JavaScript 代码需要先去掉注释的标记，再通过 eval 来获得 JSON 数据。这种方法一度被广泛使用，后来被证明并不安全，还会引入其他的安全漏洞。带前缀的 JSON 是目前推荐使用的方法，这种方法的使用非常简单，只需要在从服务器端返回的 JSON 字符串之前加上 {} &&，再调用 eval 方法。关于这两种方法的细节，可参考 http://www.ibm.com/developerworks/cn/web/wa-lo-dojoajax1/?S_TACT=105AGX52&S_CMP=tec-csdn#resources 中的内容。对 JSON 字符串进行语法检查，安全的 JSON 应该是不包含赋值和方法调用的。在 JSON 的 RFC 4627 中，给出了判断 JSON 字符串是否安全的方法，此方法通过两个正则表达式来实现（代码如下）。

```
    var my_JSON_object = !(/[^,:{}\[\]0-9.\-+Eaeflnr-u \n\r\t]/.test(text.
replace(/"(\\.|[^"\\])*"/g, ''))) && eval('(' + text + ')');
```

建议 15：避免使用 continue

continue 语句与 break 语句用法相似，在循环结构中用于控制逻辑的执行方向。break 语句用于停止循环，而 continue 语句却用于再次执行循环。与 break 语句语法相同，continue 语句可以跟随一个标签名，用来指定继续执行的循环结构的起始位置。

```
continue label;
```

例如，在下面的这个示例中，当循环变量等于 4 时，会停止循环体内最后一句的执行，返回 for 语句起始位置继续执行下一次循环。

```
for(var i=0; i<10;i++){
    alert(i);
    if (i==4) continue;
    alert(i);
}
```

不管 continue 语句是否带有标签，都只能在循环结构（如 while、do/while、for、for/in）体内使用，在其他地方都会引发编译错误。当执行 continue 语句时，会停止当前循环过程，开始执行下一次的循环。但对于不同的结构体，continue 语句继续执行的位置会略有不同。

在实践中，通过代码重构移除 continue 语句会使性能得到改善。因此，在非必要条件下，建议不要使用 continue 语句。

建议 16：防止 switch 贯穿

JavaScript 语言中那些显而易见的危险或无用的特性不是最糟糕的，这些特性很容易被避免。最糟糕的特性像“带刺的玫瑰”，它们是有用的，但也是危险的。

switch 语句的由来可以追溯到 FORTRAN IV 的 go to 语句。除非明确地中断流程，否则每次条件判断后都贯穿到下一个 case 条件。switch 语句的基本语法格式如下：

```
switch (expression ){
    statements
}
```

完全扩展后的 switch 结构如下：

```
switch ( expression ){
    case label:
        statementList
    case label:
        statementList
    ...
    default:
        statementList
}
```

当执行 switch 语句时，JavaScript 解释器首先计算 expression 表达式的值，然后使用这个值与每个 case 从句中 label 标签值进行比较，如果相同则执行该标签下的语句。在执行时如果遇到跳转语句，则会跳出 switch 结构，否则按顺序向下执行，直到 switch 语句末尾。如果没有匹配的标签，则会执行 default 从句中的语句。如果没有 default 从句，则跳出 switch 结构，执行其后的句子。从 ECMAScript v3 版本开始允许 case 从句中可以是任意的表达式，这在 C/C++ 和 Java 语言中是不允许的。switch 语句的示例如下：

```
switch (a = 3){
    case 3-2:
        alert(1);
        break;
    case 1+1:
        alert(2);
        break;
    case b=3:
        alert(3);
}
```

在 switch 语句中，case 从句只是指明了想要执行代码的起点，并没有指明终点，如果没有向 case 从句中添加 break 语句，则会发生连续贯穿现象，从而忽略后面 case 从句，这样就会造成 switch 结构的逻辑混乱。不过，如果是在函数中使用 switch 语句，还可以使用 return 语句来代替 break 语句，这两个语句都可以终止 switch 语句，防止 case 从句之间发生逻辑贯穿。

建议 17：块标志并非多余

if、while、do 或 for 语句可以接受一个括在大括号中的代码块，也可以接受单行语句。单行语句的形式是另一种“带刺的玫瑰”。它的好处是可以节省两个字节，但是它模糊了程序的结构，在随后的操作中可能产生问题，例如：

```
if(0)
    if(1)
        alert(1);
else
    alert(0);
```

如果不借助代码版式，很难明白以上代码的逻辑结构。而 JavaScript 解释器会根据 if 关键字与 else 关键字最近原则按如下结构进行解释。

```
if(0)
    if(1)
        alert(1);
    else
        alert(0);
```

如果其中子结构中包含多行语句，这个问题就比较麻烦了，甚至会出现执行错误的情况。因此，为了避免嵌套的条件结构发生混乱，应该使用大括号语法来分隔代码块，例如：

```
if(0) {
    if(1) {
        alert(1);
    }
}
else{
    alert(0);
}
```

严格遵循规范，并始终使用代码块，会使代码更容易理解。

建议 18：比较 function 语句和 function 表达式

在 JavaScript 语言中，既有 function 语句，也有函数表达式，这是令人困惑的，因为它们看起来是相同的。一个 function 语句就是值为一个函数的 var 语句的简写形式。

下面的语句：

```
function f() {}
```

相当于：

```
var f=function() {}
```

这里建议使用第二种形式，因为它能明确表示 f 是一个包含一个函数值的变量。要用好 JavaScript 这门语言，理解函数就是数值是很重要的。

function 语句在解析时会被提升，这意味着不管 function 被放置在哪里，它都会被移动到定义时所在作用域的顶层。这放宽了函数必须先声明后使用的要求，当然这也会造成混乱。

在 if 语句中也是禁止使用 function 语句的。大多数的浏览器都允许在 if 语句中使用 function 语句，但它们在解析 function 语句的处理上各不相同，因此造成了可移植性方面的问题。

根据官方的语法约定，一个语句不能够以一个函数表达式开头，而以单词 function 开头的语句是一个 function 语句。解决这个问题的方法就是把函数表达式括在一个圆括号之中。

```
(function () {
    //…
}());
```

建议 19：不要使用类型构造器

在默认状态下，JavaScript 预定义了很多构造函数，如 Function()、Array()、Date()、string() 等，如果去掉小括号，它们就是 JavaScript 内置对象。在 JavaScript 中，构造函数实际上就是类的一种抽象结构。

利用 new 运算符调用构造函数，可以快速生成很多实例对象。例如：

```
var f = new Function(p1, p2, ..., pn, body);
```

其中构造函数 Function() 的参数类型都是字符串，p1 ～ pn 表示所创建函数的参数名称列表，body 表示所创建函数的函数结构体语句，在 body 语句之间通过分号进行分隔。可以完全省略所有参数，仅为构造函数传递一个字符串，用来表示函数的具体结构。f 就是所创建函数的名称。例如：

```
var f = new Function("a", "b", "return a+b");
```

使用 new Function() 的形式来创建一个函数不是很常见，因为一个函数体通常会包括多条语句，如果将这些语句以一个字符串的形式作为参数来传递，代码的可读性会很差。类似的用法还有：

```
new Boolean(false)
```

new 运算符调用函数会返回一个对象，该对象有一个 valueof 方法，同时返回被包装的函数，其实这是完全没有必要的，并且有时还令人困惑。不要使用 new Boolean、new Number 或 new String。此外，应该避免使用 new Object 和 new Array，可以使用 {} 和 [] 来代替。

在其他语言中，构造函数一般没有返回值，它只是初始化由 this 关键字所指代的对象，

并且什么都不返回。但是，JavaScript 构造函数可以返回一个对象，返回的对象将成为 new 运算符的运算值，此时 this 所引用的对象就会被覆盖。

```
function F(){
    this.x = 1;
    return { y : 2 };
}
var f = new F();
f.y;                        // 2
```

上面示例演示了如何使用返回的对象覆盖构造函数的实例对象，但是，如果返回值是原始值时，就不会覆盖实例化对象，此时按着普通函数的方式调用构造函数就可以得到返回值。 例如：

```
function F(){
    this.x = 1;
    return true;
}
var f = new F();
f.x;        // 1
F();        // true
```

因此，如果构造函数的返回值为对象，可以直接调用构造函数来引用该返回值对象，而不需要使用 new 运算符来运算。

建议 20：不要使用 new

通过 new 运算符将创建一个继承于其运算数的原型的新对象，然后调用该运算数，把新创建的对象绑定给 this。这给了运算数（它应该是一个构造器函数）一个机会，在返回给请求者前去自定义新创建的对象。

如果忘记使用 new 运算符，得到的就是一个普通的函数调用，并且 this 被绑定到全局对象，而不是新创建的对象。这意味着当函数尝试去初始化新成员时，它将会“污染”全局变量，这是一件非常糟糕的事情，既没有编译时警告，也没有运行时警告。

按照惯例，结合 new 运算符使用的函数应该被命名为首字母大写的形式，并且首字母大写的形式应该只用来命名那些构造器函数。这个约定提供了一个视觉线索，以帮助发现那些 JavaScript 语言自身经常忽略却需要付出昂贵代价的错误。一个更好的应对策略就是根本不使用 new 运算符。

建议 21：推荐提高循环性能的策略

每次运行循环体时都会产生性能开销，增加总的运行时间，即使是循环体中最快的代

码，累计迭代上千次，也将带来不小的负担。因此，减少循环的迭代次数可获得显著的性能提升。例如：

```
var iterations = Math.floor(items.length / 8), startAt = items.length % 8, i = 0;
do {
    switch(startAt) {
        case 0:
            process(items[i++]);
        case 7:
            process(items[i++]);
        case 6:
            process(items[i++]);
        case 5:
            process(items[i++]);
        case 4:
            process(items[i++]);
        case 3:
            process(items[i++]);
        case 2:
            process(items[i++]);
        case 1:
            process(items[i++]);
    }
    startAt = 0;
} while (--iterations);
```

在上面代码中，每次循环最多可调用 process() 函数 8 次。循环迭代次数为元素总数除以 8。因为总数不一定是 8 的整数倍，所以 startAt 变量存放余数，指明第一次循环中应当执行多少次 process()。如果现在有 12 个元素，那么第一次循环将调用 process() 4 次，第二次循环调用 process() 8 次，用两次循环代替了 12 次循环。在下面的代码中取消 switch 表达式，将余数处理与主循环分开。

```
var i = items.length % 8;
while(i) {
    process(items[i--]);
}
i = Math.floor(items.length / 8);
while(i) {
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
}
```

虽然上面代码使用两个循环替代了先前的一个循环，但是在循环体中去掉了 switch 表达

式，速度更快。如果循环的迭代次数少于 1000 次，减少迭代次数的循环与普通循环相比可能只有微不足道的性能提升。如果迭代次数超过 1000 次，比如在 500 000 次迭代中，合理地减少循环的迭代次数可以使运行时间减少到普通循环的 70%。

有两个因素影响到循环的性能：

- 每次迭代干什么。
- 迭代的次数。

通过减少这两者中的一个或全部（的执行时间），可以提高循环的整体性能。如果一次循环需要较长时间来执行，那么多次循环将需要更长时间。限制在循环体内进行耗时操作的数量是一个加快循环的好方法。一个典型的数组处理循环可采用 3 种循环的任何一种。最常用的代码如下：

```
// 方法 1
for (var i=0; i < items.length; i++){
    process(items[i]);
}
// 方法 2
var j=0;
while (j < items.length){
    process(items[j++]);
}
// 方法 3
var k=0;
do {
    process(items[k++]);
} while (k < items.length);
```

在每个循环中，每次运行循环体都要发生如下操作：

第 1 步，在控制条件中读一次属性（items.length）。

第 2 步，在控制条件中执行一次比较（i < items.length）。

第 3 步，比较操作，观察条件控制体的运算结果是不是 true（i < items.length == true）。

第 4 步，一次自加操作（i++）。

第 5 步，一次数组查找（items[i]）。

第 6 步，一次函数调用（process(items[i])）。

在这些简单的循环中，即使没有太多的代码，每次迭代都要进行这 6 步操作。代码运行速度很大程度上由 process() 对每个项目的操作所决定，即便如此，减少每次迭代中操作的总数也可以大幅度提高循环的整体性能。

优化循环的第一步是减少对象成员和数组项查找的次数。在大多数浏览器上，这些操作比访问局部变量或直接量需要更长时间。例如，在上面代码中，每次循环都查找 items.length，这是一种浪费，因为该值在循环体执行过程中不会改变，因此产生了不必要的性能损失。我们可以简单地将此值存入一个局部变量中，在控制条件中使用这个局部变量，从而提高了循环性能，例如：

```
for (var i=0, len=items.length; i < len; i++){
    process(items[i]);
}
var j=0, count = items.length;
while (j < count){
    process(items[j++]);
}
var k=0, num = items.length;
do {
    process(items[k++]);
} while (k < num);
```

这些重写后的循环只在循环执行之前对数组长度进行一次属性查询，使控制条件中只有局部变量参与运算，因此速度更快。根据数组的长度，在大多数浏览器上总循环时间可以节省大约 25%，在 IE 浏览器中可节省 50%。

还可以通过改变循环的顺序来提高循环性能。通常，数组元素的处理顺序与任务无关，可以从最后一个开始，直到处理完第一个元素。倒序循环是编程语言中常用的性能优化方法，不过一般不太容易理解。在 JavaScript 中，倒序循环可以略微提高循环性能，例如：

```
for (var i=items.length; i--; ){
    process(items[i]);
}
var j = items.length;
while (j--){
    process(items[j]);
}
var k = items.length-1;
do {
    process(items[k]);
} while (k--);
```

在上面代码中使用了倒序循环，并且在控制条件中使用了减法。每个控制条件只是简单地与零进行比较。控制条件与 true 值进行比较，任何非零数字自动强制转换为 true，而零等同于 false。实际上，控制条件已经从两次比较减少到一次比较。将每次迭代中两次比较减少到一次可以大幅度提高循环速度。通过倒序循环和最小化属性查询，可以看到执行速度比原始版本提升了 50% ～ 60%。

与原始版本相比，每次迭代中只进行如下操作：

第 1 步，在控制条件中进行一次比较（i == true）。

第 2 步，一次减法操作（i--）。

第 3 步，一次数组查询（items[i]）。

第 4 步，一次函数调用（process(items[i])）。

新循环的每次迭代中减少两个操作，随着迭代次数的增长，性能将显著提升。

建议 22：少用函数迭代

ECMA-262v4 为本地数组对象新增加了一个 forEach 方法。此方法遍历一个数组的所有成员，并且在每个成员上执行一个函数。在每个元素上执行的函数作为 forEach() 的参数传进去，并在调用函数时接收 3 个参数：数组项的值、数组项的索引、数组自身。例如：

```
items.forEach(function(value, index, array){
    process(value);
});
```

forEach 在 Firefox、Chrome 和 Safari 等浏览器中为原生函数。另外，forEach 在大多数 JavaScript 库中都有等价实现，例如：

```
//YUI 3
Y.Array.each(items, function(value, index, array){
    process(value);
});
//jQuery
jQuery.each(items, function(index, value){
    process(value);
});
//Dojo
dojo.forEach(items, function(value, index, array){
    process(value);
});
//Prototype
items.each(function(value, index){
    process(value);
});
```

尽管基于函数的迭代使用起来非常便利，但是比基于循环的迭代要慢一些。每个数组项要关联额外的函数调用是造成速度慢的主要原因。在所有情况下，基于函数的迭代占用时间是基于循环的迭代的 8 倍，因此在非特殊需求下，不建议使用函数迭代。

建议 23：推荐提高条件性能的策略

与循环相似，条件表达式决定 JavaScript 运行流的走向。与其他语言一样，JavaScript 也采用了 if 和 switch 两种条件结构。由于不同浏览器针对流程控制进行了不同的优化，因此两者在性能上并没有特别大的差异，主要还是根据需求形式进行分析和选择：条件数量较大，建议选择 switch 结构，而不是 if 结构，这样可以使代码更易读；如果条件较少时，建议选择 if 结构。

```
// 条件少
if(found) {
    // 执行代码
```

```
} else {
    // 执行代码
}
// 条件多
switch (color) {
    case "red":
        // 执行代码
        break;
    case "blue":
        // 执行代码
        break;
    case "brown":
        // 执行代码
        break;
    case "black":
        // 执行代码
        break;
    default:
        // 执行代码
}
```

事实证明，在大多数情况下，switch 比 if 运行更快，但是只有当条件体数量很大时才明显更快。switch 与 if 的主要性能区别在于：当条件体增加时，if 性能负担增加的程度比 switch 更大。因此，从性能方面考虑，如果条件体较少，应使用 if；如果条件体较多，应使用 switch。

一般来说，if 适用于判断两个离散的值或几个不同的值域。如果判断多于两个离散值，那么 switch 将是更理想的选择。

建议 24：优化 if 逻辑

逻辑顺序体现了人的思维的条理性和严密性。合理的顺序可以提升解决问题的品质，相反，混乱的顺序很容易导致各种错误的发生。在分支结构中经常需要面临各种优化逻辑顺序的问题。

人在思考问题时，一般总会对各种最可能发生的情况做好准备，这叫做“有备而来”。分支结构中各种条件根据情况的先后、轻重来排定顺序。如果把最可能的条件放在前面，把最不可能的条件放在后面，这样程序在执行时总会按照代码先后顺序逐一检测所有条件，直到发现匹配的条件时才停止继续检测。如果把最可能的条件放在前面，就等于降低了程序的检测次数，自然也就提升了分支结构的执行效率，避免空转，这在大批量数据检测中效果非常明显。例如，对于一个论坛系统来说，普通会员的数量要远远大于版主和管理员的数量。换句话说，大部分登录的用户都是普通会员，如果把普通会员的检测放在分支结构的前面，就会减少计算机检测的次数。

if 优化目标：最小化找到正确分支之前所判断条件体的数量。if 优化方法：将最常见的

条件体放在首位。例如：

```
if(value < 5) {
    //do something
} else if(value > 5 && value < 10) {
    //do something
} else {
    //do something
}
```

这段代码只有在 value 值经常小于 5 时才是最优的。如果 value 经常大于或等于 10，那么在进入正确分支之前，必须两次运算条件体，导致表达式的平均运行时间增加。if 中的条件体应当总是按照从最大概率到最小概率的顺序排列，以保证理论运行速度最快。

另外一种减少条件判断数量的方法：将 if 编写成一系列嵌套结构。使用一个单独的一长串的 if 结构通常导致运行缓慢，因为每个条件体都要被计算，例如：

```
if(value == 0) {
    return result0;
} else if(value == 1) {
    return result1;
} else if(value == 2) {
    return result2;
} else if(value == 3) {
    return result3;
} else if(value == 4) {
    return result4;
} else if(value == 5) {
    return result5;
} else if(value == 6) {
    return result6;
} else if(value == 7) {
    return result7;
} else if(value == 8) {
    return result8;
} else if(value == 9) {
    return result9;
} else {
    return result10;
}
```

在这个 if 结构中，所计算的条件体的最大数目是 10。如果假设 value 的值在 0 ～ 10 之间均匀分布，那么会增加平均运行时间。为了减少条件判断的数量，可重写为一系列嵌套结构，例如：

```
if(value < 6) {
    if(value < 3) {
        if(value == 0) {
            return result0;
        } else if(value == 1) {
            return result1;
```

```
            } else {
                return result2;
            }
        } else {
            if(value == 3) {
                return result3;
            } else if(value == 4) {
                return result4;
            } else {
                return result5;
            }
        }
    } else {
        if(value < 8) {
            if(value == 6) {
                return result6;
            } else {
                return result7;
            }
        } else {
            if(value == 8) {
                return result8;
            } else if(value == 9) {
                return result9;
            } else {
                return result10;
            }
        }
    }
```

重写 if 结构后，每次抵达正确分支时最多通过 4 个条件判断。新的 if 结构使用二分搜索法将值域分成了一系列区间，然后逐步缩小范围。当数值范围分布在 0 ～ 10 时，此代码的平均运行时间大约是前面代码的一半。此方法适用于需要测试大量数值的情况，而相对离散值来说 switch 更合适。

当然，在性能影响不是很大的情况下，遵循条件检测的自然顺序会更容易阅读。例如，对于检测周一到周五值日任务安排的分支结构来说，虽然周五的任务比较重要，但是这类任务有着明显的顺序，安排顺序结构还是遵循它的自然逻辑比较好。如果打乱条件的顺序，把周五的任务安排在前面，对于整个分支结构的执行性能没有太大的帮助，并且不方便阅读代码。考虑到这一点，按自然顺序来安排结构会更富可读性。

应注意分支之间的顺序优化，当然在同一个条件表达式内部也应该考虑逻辑顺序问题。在执行逻辑“与”或逻辑“或”运算时，有可能会省略右侧表达式的计算，如果希望不管右侧表达式是否成立都进行计算，就应该考虑逻辑顺序问题。例如，有两个条件 a 和 b，其中条件 a 多为 true，b 就是一个必须执行的表达式，那么下面的逻辑顺序设计就欠妥当：

```
if(a && b){
}
```

如果条件 a 为 false，则 JavaScript 会忽略表达式 b 的计算。如果 b 表达式影响到后面的运算，那么不执行表达式 b 自然会对后面的逻辑产生影响，这时可以采用下面的逻辑结构，在 if 结构前先执行表达式 b，这样即使条件 a 的返回值为 false，也能够保证表达式 b 被计算。

```
var c = b;
if(a && b){
}
```

建议 25：恰当选用 if 和 switch

switch 结构中存在很多限制，存在这些限制的主要目的是提高多重分支结构的执行效率。因此，如果能够使用 switch 结构，就不要选择 if 结构。

无论是使用 if 结构，还是使用 switch 结构，应该确保下面 3 个目标的基本实现：

- 准确表现事物内在的、固有的逻辑关系。不能为了结构而破坏事物的逻辑关系。
- 优化逻辑的执行效率。执行效率是程序设计的重要目标，不能为了省事而随意耗费资源。
- 简化代码的结构层次，使代码更方便阅读。

相对来说，下面几种情况更适合使用 switch 结构：

- 枚举表达式的值。这种枚举是可以期望的、平行逻辑关系的。
- 表达式的值具有离散性，不具有线性的非连续的区间值。
- 表达式的值是固定的，不是动态变化的。
- 表达式的值是有限的，而不是无限的，一般情况下表达式应该比较少。
- 表达式的值一般为整数、字符串等类型的数据。

而 if 结构则更适合下面的一些情况：

- 具有复杂的逻辑关系。
- 表达式的值具有线性特征，如对连续的区间值进行判断。
- 表达式的值是动态的。
- 测试任意类型的数据。

例如，针对学生分数进行不同的判断，如果分数小于 60，则评定为不及格；如果分数在 60 ～ 75（不包含 75）之间，则评定为合格；如果分数在 75 ～ 85（不包含 85）之间，则评定为良好；如果分数在 85 ～ 100 之间，则评定为优秀。针对这种情况，表达式的值是连续的线性判断，显然使用 if 结构会更合适一些。

```
if(score < 60){
    alert(" 不及格 ");
}
else if(60 <= score < 75){
```

```
    alert("合格");
}
else if(75 <= score < 85){
    alert("良好");
}
else if(85 <= score <= 100){
    alert("优秀");
}
```

如果使用 switch 结构，则需要枚举 100 种可能，如果分数值还包括小数，情况就更加复杂了，此时使用 switch 结构就不是明智之举。但是，对于有限制的枚举数据，比如性别，使用 switch 结构会更高效，例如：

```
switch(sex){
    case "女":
        alert("女士");
        break;
    case "男":
        alert("先生");
        break;
    default:
        alert("请选择性别");
}
```

建议 26：小心 if 嵌套的思维陷阱

人的思维是非常复杂的，这在一定程度上会增加 if 结构嵌套的复杂性。假设有 4 个条件，只有当这些条件全部成立时，才允许执行某件事情。遵循人的一般思维习惯，在检测这些条件时，常常会沿用下面这种结构嵌套：

```
if(a){
    if(b){
        if(c){
            if(d){
                alert("所有条件都成立！");
            }
            else{
                alert("条件 d 不成立！");
            }
        }
        else{
            alert("条件 c 不成立！");
        }
    }
    else{
        alert("条件 b 不成立！");
    }
}
else{
```

```
    alert(" 条件 a 不成立！ ");
}
```

从思维的方向性上来考虑，这种结构嵌套并没有错误，使用下面这个 if 结构来表示更为简单。

```
if(a && b && c && d){
    alert(" 所有条件都成立！ ");
}
```

从设计时的本意来考虑：使用 if 语句逐个验证每个条件的合法性，并且对某个条件是否成立进行提示，以方便跟踪每个条件。但是，如果使用了上面的 if 结构多重嵌套，就会出现另一种可能：a 条件不成立，程序会自动退出整个嵌套结构，而不管 b、c 和 d 的条件是否成立。这种“武断”很容易给测试带来“伤害”。如果核心的处理过程包含多条语句，或者出错的情况处理更为复杂，层层包裹的 if 结构会使代码嵌套过深，难以编辑。

为避免上述情况的发生，一般采取排除法，即对每个条件进行排除，条件全部成立再执行特定的操作。为了能够把条件有机地联系在一起，这里使用了一个布尔型变量作为钩子把每个 if 条件结构串在一起。

```
var t = true; // 初始化行为变量为 true
if(!a){
    alert(" 条件 a 不成立！ ");
    t = false;
}
if(!b){
    alert(" 条件 b 不成立！ ");
    t = false;
}
if(!c){
    alert(" 条件 c 不成立！ ");
    t = false;
}
if(!d){
    alert(" 条件 d 不成立！ ");
    t = false;
}
if(t){
    alert(" 所有条件都成立！ ");
}
```

排除法有效地避免了条件结构的多重嵌套，并且更加符合人的思维模式。当然，这种设计方法也存在一定的局限性，一旦发生错误，就要放弃后面的操作。如果仅为了检查某个值的合法性，也就无所谓了。如果为了改变变量值和数据操作等，那么直接放弃就会导致后面的数据操作无法进行，为了防止此类问题的发生，不妨再设计一个标志变量来跟踪整个操作行为。

建议 27：小心 if 隐藏的 Bug

很多程序员都犯过这样低级的错误：

```
if(a = 1){
    alert(a);
}
```

把比较运算符（==）错写为赋值运算符（=）。这样的 Bug 一般很难发现，由于它是一个合法的表达式，不会导致编译错误。由于此表达的返回值为非 0 数值，JavaScript 会自动把它转换为 true，因此这样的分支结构的条件永远成立。

为了防止出现这样低级而又令人讨厌的错误，建议在条件表达式的比较运算中，把常量写在左侧，把变量写在右侧，这样即使把比较运算符（==）错写为赋值运算符（=），也会导致编译错误，因为常量是不能够被赋值的，从而能够即时发现这个 Bug。例如：

```
if(1 == a){
    alert(a);
}
```

下面这个错误也是很容易发生的：

```
var a=2;
if(1 == a);
{
    alert(a);
}
```

当在条件表达式后错误地附加一个分号时，整个条件结构的逻辑就发生了根本的变化。用代码来描述上面结构的逻辑如下：

```
var a=2;
if(1 == a)
    ;
{
    alert(a);
}
```

也就是说，JavaScript 会把条件表达式之后的分号视为一个空语句，从而改变了原来设想的逻辑。因此，要避免这样的低级错误，应该牢记条件表达式之后不允许使用分号，当然也可以通过把大括号与条件表达式书写在一行内来防止疏忽。

```
var a=2;
if(1 == a){
    alert(a);
}
```

建议 28：使用查表法提高条件检测的性能

当有大量离散值需要测试时，使用 if 和 switch 都比使用查表法要慢得多。在 JavaScript 中查表法可通过数组或普通对象实现，查表法访问数据比 if 和 switch 更快，特别是当条件体的数目很大时。与 if 和 switch 相比，查表法不仅非常快，而且当需要测试的离散值数量非常大时，也有助于保持代码的可读性。

例如，在下面代码中，使用 switch 检测 value 值。

```
switch(value) {
    case 0:
        return result0;
    case 1:
        return result1;
    case 2:
        return result2;
    case 3:
        return result3;
    case 4:
        return result4;
    case 5:
        return result5;
    case 6:
        return result6;
    case 7:
        return result7;
    case 8:
        return result8;
    case 9:
        return result9;
    default:
        return result10;
}
```

使用 switch 结构检测 value 值的代码所占的空间可能与 switch 的重要性不成比例，代码很笨重。整个结构可以用一个数组查询替代：

```
    var results = [result0, result1, result2, result3, result4, result5, result6,
result7, result8, result9, result10]
    return results[value];
```

当使用查表法时，必须完全消除所有条件判断。操作转换成一个数组项查询或一个对象成员查询。使用查表法的一个主要优点：由于没有条件判断，当候选值数量增加时，基本上不会增加额外的性能开销。查表法常用于一个键和一个值形成逻辑映射的领域，而 switch 更适合于每个键需要一个独特的动作或一系列动作的场合。

建议 29：准确使用循环体

1. 选择正确的循环体

在大多数编程语言中，代码执行时间多数消耗在循环的执行上。在一系列编程模式中，循环是最常用的模式之一，因此也是提高性能必须关注的地方之一。理解 JavaScript 中循环对性能的影响至关重要，因为死循环或长时间运行的循环会严重影响用户体验。JavaScript 定义了 4 种类型的循环：

第一种循环是标准的 for 循环，与 C 语言使用同样的语法：

```
for (var i=0; i < 10; i++){
    // 循环体
}
```

for 循环是最常用的循环结构，它由 4 部分组成：初始化体、前测条件、后执行体、循环体。当遇到一个 for 循环时，初始化体首先执行，然后进入前测条件。如果前测条件的计算结果为 true，则执行循环体，然后运行后执行体。for 循环在封装上的直接性是开发者喜欢使用它的原因之一。

第二种循环是 while 循环。while 循环是一个简单的前测循环，由一个前测条件和一个循环体构成：

```
var i = 0;
while(i < 10){
    // 循环体
    i++;
}
```

在执行循环体之前，先对前测条件进行计算。如果计算结果为 true，那么就执行循环体；否则将跳过循环体。任何 for 循环都可以写成 while 循环，反之亦然。

第三种循环是 do while 循环。do while 循环是 JavaScript 中唯一一种后测试的循环，它包括两部分：循环体和后测条件。

```
var i = 0;
do {
    // 循环体
} while (i++ < 10);
```

在一个 do while 循环中，循环体至少运行一次，后测条件决定循环体是否应再次执行。

第四种循环是 for in 循环。此循环有一个非常特殊的用途：可以枚举任何对象的命名属性。

```
for (var prop in object){
    // 循环体
}
```

每次执行循环，属性都被对象属性的名字（一个字符串）填充，直到所有的对象属性遍

历完成才返回。返回的属性包括对象的实例属性和对象从原型链继承而来的属性。

提高循环性能的起点是选用哪种循环。在 JavaScript 提供的 4 种循环类型中，只有 for in 循环执行速度比其他循环明显要慢。

由于每次迭代操作要搜索实例或原型的属性，for in 循环每次迭代都要付出更多开销，因此它比其他类型循环执行速度慢一些。在同样的循环迭代操作中，其他类型循环执行速度比 for in 循环快 7 倍之多，因此推荐这样做：除非需要对数目不详的对象属性进行操作，否则避免使用 for in 循环。例如，迭代遍历一个有限的、已知的属性列表，使用其他循环类型更快，具体的使用模式如下：

```
var props = ["prop1", "prop2"],
i = 0;
while (i < props.length){
   process(object[props[i]]);
}
```

此代码创建一个由成员和属性名构成的队列。while 循环用于遍历这几个属性并处理所对应的对象成员，而不是遍历对象的每个属性。此代码只关注感兴趣的属性，节约了循环时间。

2. 比较 for 和 while

除 for in 循环外，其他循环类型的性能相当，难以确定哪种循环执行速度更快。选择循环类型应基于需求而不是性能。

可以通过设计 for 和 while 循环来完成特定动作的重复性操作。下面分别从语义性、思维模式、达成目标这 3 个角度来分析如何正确选用 while 和 for 循环。

（1）从语义性角度比较

for 和 while 循环可以按如下模式进行相互转换：

```
for (initialization ; test ; increment ) // 声明并初始化循环变量、循环条件、递增循环变量
    statements                           // 可执行的循环语句
```

相当于：

```
initialization;    // 声明并初始化循环变量
while(test) {      // 循环条件
    statement      // 可执行的循环语句
    increment;     // 递增循环变量
}
```

for 循环是以循环变量的变化来控制循环进程的，即 for 循环的整个循环流程是预先计划好的，虽然中途可能会因存在异常或特别情况而退出循环，但是循环的规律性是有章可循的。这样我们能够很容易地预知循环的次数、每次循环的状态等信息。

while 循环根据特定条件来决定循环操作，由于这个条件是动态的，无法预知条件何时

为 true 或 false，因此该循环的循环操作就具有很大的不确定性，每一次循环时都不知道下一次循环的状态如何，只能通过条件的动态变化来确定。

因此，for 结构常常被用于有规律的重复操作中，如对数组、对象、集合等的操作。当然，对于这些对象的迭代操作，更适合使用 for in 这种特殊的 for 循环来操作，因为它提供了更大的便利，可以防止错误的发生。while 循环更适合用于待定条件的重复操作，以及依据特定事件控制的循环等操作。

（2）从思维模式角度比较

for 循环和 while 循环在思维模式上也存在差异。在 for 循环中，将循环的三要素（起始值、终止值和步长）定义为 3 个基本表达式作为结构语法的一部分固定在 for 语句内，使用小括号进行语法分隔，这与 while 循环中 while 语句内仅是条件检测的表达式截然不同，这样更有利于 JavaScript 进行快速预编译。

因此，当阅读到 for 结构的第一行代码时，就能够获取整个循环结构的控制方式，然后再根据上面的表达式来决定是否执行下面的循环体内的语句。可以这样概括，for 结构适合简单的数值迭代操作。例如，快速阅读下面示例的代码：

```
for(var n = 1; n < 10; n ++ ) {// 循环操作的环境条件
    alert(n); // 循环操作的语句
}
```

之后可以按以下方式对逻辑思维进行总结。

执行循环条件：1 < n < 10、步长为 n++。

执行循环语句：alert(n)。

这种把循环操作的环境条件和循环操作语句分离开的设计模式能够提高程序的执行效率，同时也避免了因为把循环条件与循环语句混在一起而造成的遗漏或错误。描述这种思维模式的简化示意图如图 1.3 所示。

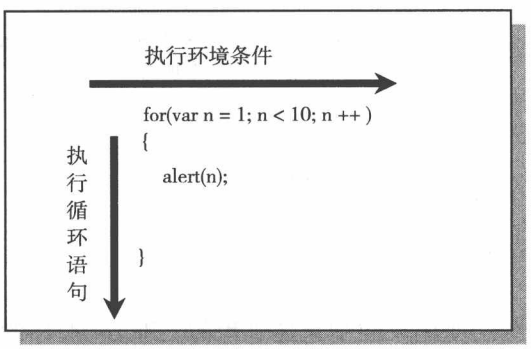


图 1.3　for 结构的数值迭代计算

如果 for 循环的循环条件比较复杂，不是简单的数值迭代，这时 for 语句就必须考虑如何把循环条件和循环语句联系起来才可以正确地执行整个 for 循环。因此，根据 for 结构的运

算顺序，for 语句首先计算第一个和第二个表达式，然后执行循环体语句，最后返回执行 for 语句的第三个表达式，如此循环执行。例如：

```
for(var a = true, b = 1; a; b ++ ){
    if(b > 9)                          // 在循环体内间接计算迭代的步长
    a = false;
    alert(b);
}
```

在上面的这个示例中，for 语句的第三个表达式不是直接计算步长的，整个 for 循环也没有明确告知循环步长的表达式，如果要确定迭代的步长，就必须依据循环体内的语句。因此，整个 for 结构的逻辑思维就存在一个回旋的过程，如图 1.4 所示。

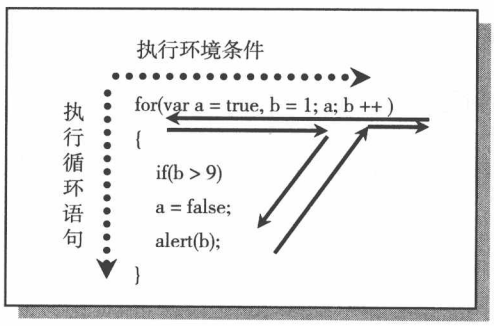


图 1.4　for 结构的条件迭代计算

for 循环的特异性导致在执行复杂条件时效率会大大降低。相对而言，while 循环天生就是为复杂的条件而设计的，它将复杂的循环控制放在循环体内执行，而 while 语句自身仅用于测试循环条件，这样就避免了结构的分隔和逻辑的跳跃。例如，使用 while 结构来表示这种复杂的条件循环的代码如下，这种思维变化的示意图如图 1.5 所示。

```
var a = true, b = 1; while(a) {// 在循环体内间接计算迭代
    if(b > 9)
    a = false;
    alert(b);
    b ++;
}
```

（3）从达成目标的角度比较

有些循环的循环次数在循环之前就可以预测，如计算 1 ～ 100 的数字和。而有些循环具有不可预测性，无法事先确定循环的次数，甚至无法预知循环操作的趋向，这些构成了在设计循环结构时必须考虑的达成目标需要解决的问题。即使是相同的操作，如果达成目标的角度不同，可能重复操作的设计也就不同。例如，统计全班学生的成绩和统计合格学生的成绩就是两个不同的达成目标。一般来说，在循环结构中动态改变循环变量的值时建议使用 while 结构，而对于静态的循环变量，则可以考虑使用 for 结构。

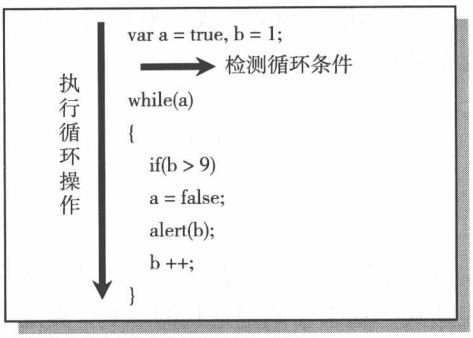


图 1.5　while 结构的条件计算

建议 30：使用递归模式

复杂算法通常比较容易使用递归实现。很多传统算法正是通过递归实现的，如阶乘函数。

```
function factorial(n) {
    if(n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

递归函数的问题：错误定义或缺少终结条件会导致函数长时间运行，使浏览器出现假死现象。此外，递归函数还会受到浏览器调用栈大小的限制。

JavaScript 引擎所支持的递归数量与 JavaScript 调用栈大小直接相关。只有 IE 例外，因为它的调用栈与可用系统内存相关，而其他浏览器有固定的调用栈限制。当使用了太多的递归，超过最大调用栈尺寸时，浏览器会弹出错误信息。

调用栈溢出错误可以用 try catch 语句捕获。异常类型因浏览器而不同：在 Firefox 中，它是一个 InternalError 错误；在 Safari 和 Chrome 中，它是一个 RangeError 错误；在 IE 中会抛出一个一般性的 Error 类型；在 Opera 中不抛出错误，但会终止 JavaScript 引擎。正确处理这些错误的方法如下：

```
try {
    recurse();
} catch (ex){
    alert("errorInfo");
}
```

当出现调用栈尺寸限制的问题时，第一步应该定位在代码中的递归实例上。为此，有两

个递归模式可供选择。

第一种是直接递归模式，即一个函数调用自身。当发生错误时，这种模式比较容易定位。其一般模式如下：

```
function recurse(){
    recurse();
}
recurse();
```

第二种是精巧模式，它包含两个函数：

```
function first(){
    second();
}
function second(){
    first();
}
first();
```

在这种递归模式中，两个函数互相调用对方，形成一个无限循环。大多数调用栈错误与这两种模式有关。常见的栈溢出原因是一个不正确的终止条件，因此定位模式错误的第一步是验证终止条件。如果终止条件是正确的，那么算法包含了太多层递归，为了能够安全地在浏览器中运行，应改用迭代、制表或混合模式。

建议 31：使用迭代

任何可以用递归实现的算法都可以用迭代实现。迭代算法通常包括几个不同的循环，分别对应算法过程的不同方面。虽然迭代也会导致性能问题，但是使用优化的循环替代长时间运行的递归函数可以提高性能，因为运行一个循环比反复调用一个函数的开销要低。

例如，合并排序算法是最常用的以递归实现的算法。下面是一个简单的合并排序算法：

```
function merge(left, right) {
    var result = [];
    while(left.length > 0 && right.length > 0) {
        if(left[0] < right[0]) {
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }
    return result.concat(left).concat(right);
}
function mergeSort(items) {
    if(items.length == 1) {
        return items;
    }
```

```
        var middle = Math.floor(items.length / 2), left = items.slice(0, middle), right
= items.slice(middle);
        return merge(mergeSort(left), mergeSort(right));
    }
```

这个合并排序代码相当简单直接，其中 mergeSort() 函数被调用得非常频繁。对一个超过 1500 项的数组进行操作，就可能在 Firefox 上导致栈溢出。程序陷入栈溢出错误并不一定要修改整个算法，这说明递归不是最好的实现方法。合并排序算法还可以用迭代实现，例如：

```
function mergeSort(items) {
    if(items.length == 1) {
        return items;
    }
    var work = [];
    for(var i = 0, len = items.length; i < len; i++) {
        work.push([items[i]]);
    }
    work.push([]);
    for(var lim = len; lim > 1; lim = (lim + 1) / 2) {
        for(var j = 0, k = 0; k < lim; j++, k += 2) {
            work[j] = merge(work[k], work[k + 1]);
        }
        work[j] = [];
    }
    return work[0];
}
```

在上面代码中，mergeSort() 实现与上一个 merge 函数同样的功能却没有使用递归。虽然迭代可能比递归合并排序慢一些，但是它不会像递归那样影响调用栈。将递归算法切换为迭代是避免栈溢出错误的方法之一。

建议 32：使用制表

代码所做的事情越少，它的运行速度就越快，因此，避免重复工作很有意义。多次执行相同的任务也在浪费时间。制表法通过缓存先前计算结果为后续计算所使用，避免了重复工作，这使得制表成为递归算法中最有用的技术。

当递归函数被多次调用时，重复工作很多。以下 factorial() 函数是一个递归函数重复多次的典型例子。

```
var fact6 = factorial(6);
var fact5 = factorial(5);
var fact4 = factorial(4);
```

此代码生成 3 个阶乘结果，factorial() 函数总共被调用了 18 次。此代码中最糟糕的部分是，所有必要的计算已经在第一行代码中执行过了。因为 6 的阶乘等于 6 乘以 5 的阶乘，所以 5 的阶乘被计算了两次。更糟糕的是，4 的阶乘被计算了 3 次。更为明智的方法是保存并

重利用它们的计算结果，而不是每次都重新计算整个函数。使用制表法重写 factorial() 函数：

```
function memfactorial(n) {
    if(!memfactorial.cache) {
        memfactorial.cache = {
            "0" : 1,
            "1" : 1
        };
    }
    if(!memfactorial.cache.hasOwnProperty(n)) {
        memfactorial.cache[n] = n * memfactorial(n - 1);
    }
    return memfactorial.cache[n];
}
```

使用制表法设计阶乘函数的关键是建立一个缓存对象，此对象位于函数内部，其中预置了两个最简单的阶乘：0 和 1。在计算阶乘之前，先检查缓存中是否已经存在相应的计算结果。没有对应的缓冲值说明这是第一次计算此数值的阶乘，计算完成之后结果被存入缓存之中，以备今后使用。

```
var fact6 = memfactorial(6);
var fact5 = memfactorial(5);
var fact4 = memfactorial(4);
```

上面代码返回 3 个不同的阶乘值，总共只调用 memfactorial() 函数 8 次。既然所有必要的计算都在第一行代码中完成了，那么后两行代码不会产生递归运算，因为直接返回了缓存中的数值。制表过程会因递归函数的种类不同而略有不同，但总体上具有相同的模式。

建议 33：优化循环结构

循环是最浪费资源的一种流程。循环结构中一点小小的损耗都会被成倍放大，从而影响程序运行的效率。下面从以下几个方面介绍如何优化循环结构，从而提高循环结构的执行效率。

（1）优化结构

循环结构常常与分支结构混用在一起，因此如何嵌套就非常讲究了。例如，设计一个循环结构，结构内的循环语句只有在特定条件下才被执行。使用一个简单的例子来演示，其正常思维结构如下：

```
var a = true;
for(var b = 1; b < 10; b ++ ) {   // 循环结构
    if(a == true) { // 条件判断
    }
}
```

很明显，在这个循环结构中 if 语句会被反复执行。如果这个 if 语句是一个固定的条件检测表

达式，也就是说，如果 if 语句的条件不会受循环结构的影响，那么不妨采用如下的结构来设计：

```
if(a == true) { // 条件判断
    for(var b = 1; b < 10; b ++ ) {// 循环结构
    }
}
```

这样，if 语句只被执行一次，如果 if 条件不成立，则直接省略 for 语句的执行，从而使程序的执行效率大大提高。但是，如果 if 条件表达式受循环结构的制约，就不能够采用这种结构嵌套了。

（2）避免不必要的重复操作

在循环体内经常会存在不必要的损耗。例如，在下面的这个示例中，在循环内声明数组，然后读取数组元素的值。

```
for(var b = 0; b < 10; b ++ ) {
    var a = new Array(1,2,3,4,5,6,7,8,9,10);
    alert(a[b]);
}
```

显然，在这个循环结构中，每循环一次都会重新定义数组，这样的设计极大地浪费了资源。如果把这个数组放在循环体外会更加高效，例如：

```
var a = new Array(1,2,3,4,5,6,7,8,9,10);
for(var b = 0; b < 10; b ++ ){
    alert(a[b]);
}
```

在日常开发中，类似这样不必要且重复的事情常常会浪费大量的系统资源，其实只要稍微留意，此类问题就会避免。

（3）妥善定义循环变量

对于 for 循环来说，它主要利用循环变量来控制整个结构的运行。当循环变量仅用于结构内部时，不妨在 for 语句中定义循环变量，这样能够优化循环结构。例如，计算 1 ～ 100 数字的和：

```
var s = 0;
for(var i = 0; i <= 100; i ++ ) {
    s += i;
}
alert(s);
```

显然下面的做法就不是很妥当，因为单独定义循环变量，实际上增大了系统开销。

```
var i = 0;
var s = 0;
for(i = 0; i <= 100; i ++ ) {
    s += i;
}
alert(s);
```

第 2 章

字符串、正则表达式和数组

所有 JavaScript 程序都与字符串操作紧密相连。例如，许多应用程序利用 Ajax 从服务器获取字符串，将这些字符串转换成更易用的 JavaScript 对象，然后从数据中生成 HTML 字符串。一个典型的程序需要处理若干这样的任务：合并、分解、重新排列、搜索、遍历，以及其他方法处理字符串。

在 JavaScript 中，正则表达式是必不可少的，它的重要性远超过琐碎的字符串处理。提高正则表达式效率是很多开发者最易忽视的问题之一。使正则表达式更快地找到匹配，以及在非匹配位置上花费更少时间是开发者必须重视的问题之一。

密集的字符串操作和粗浅地编写正则表达式是造成 JavaScript 程序性能问题的主要原因，本章的建议可帮助读者避免这些常见问题。同时，由于数组是所有数据序列中运算速度最快的一种类型，且 JavaScript 提供了大量的数组操作方法，这些因素都值得我们去认真研究数组的内部工作机制和应用技巧。

建议 34：字符串是非值操作

在字符串的复制和传递过程中，JavaScript 解释器以引用方式来实现对字符串的操作。将字符串数据存储到堆区，然后把字符串的引用地址存储在字符串变量中。同时为了避免错误操作，JavaScript 解释器强制约定字符串在堆区存储的数据是不可变的。这相当于设置字符串在堆区存储的数据为“只读”内容。因此，我们会发现没有一种 JavaScript 语法、方法或属性可以改变字符串中的原字符。

当进行字符串的复制和传递时，只是在栈区复制和传递字符串的引用地址，这种模拟使用引用的方法进行操作加快了内存的计算速度，不必把所有字符串都读取到栈区进行操作，这样节省了大量时间，提高了运行效率，如图 2.1 所示。

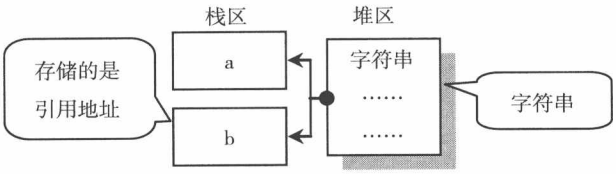


图 2.1　引用的字符串

例如，把变量 a 中的字符串复制给变量 b，那么在 b 中修改字符串内容，不会影响到 a 中包含的内容，字符串的复制和传递操作像是为字符串建立了独立的副本。

```
var a = "javascript";
var b = a;
b = b.toUpperCase();
alert(a);
alert(b);
```

在上面代码中，最终变量 a 和 b 的值是不同的，虽然它们都引用同一个字符串。JavaScript 对于字符串的复制和传递仅是简单地采用引用的方法，操作对象为堆区字符串的地址，即复制和传递地址。但是，一旦编辑字符串本身的值，JavaScript 就会把堆区的字符串读取到栈区进行独立操作。

操作完毕，要把结果赋值给原变量，JavaScript 需要再次把字符串数据写回堆区，但没有覆盖原值所在的区域，而是新开辟一个区域进行存储，并把新空间的地址传递给栈区的变量进行存储，也就是说，在堆区新建一个副本。如果不把结果赋值给变量，就待在栈区等待 JavaScript 垃圾回收，而原来变量的值并没有改变。所以，在上面代码中，修改变量 b 的字符串后，还要把结果字符串赋值给变量 b。用示意图进行演示，如图 2.2 所示。

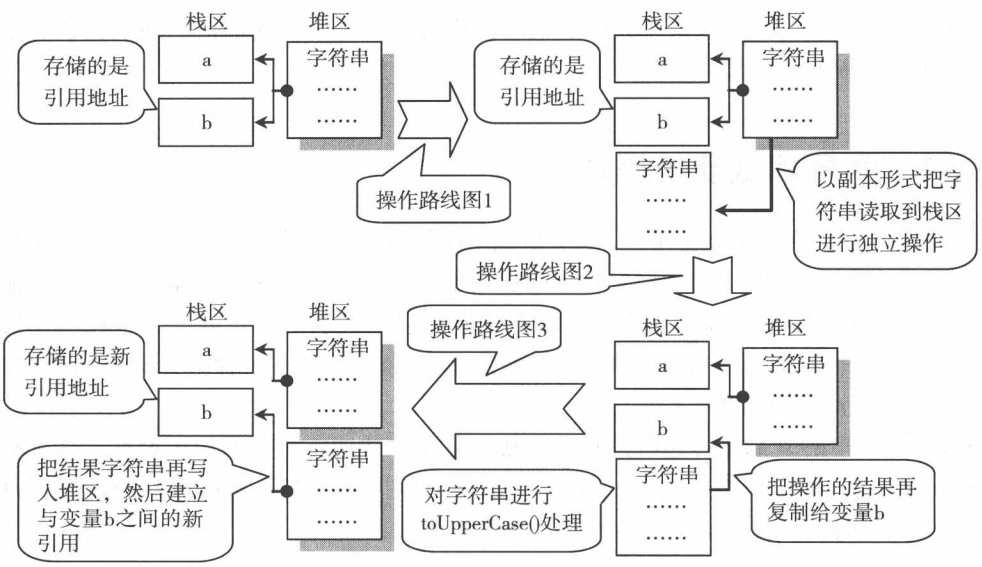


图 2.2　JavaScript 对于字符串的操作过程演示

因此，在操作字符串时，读者应该注意以下问题：

- 字符串的复制（即赋值）、传递（参数传递）仅是对字符串的引用进行操作，而不是对字符串本身的值进行操作。
- 修改字符串的值，需要使用值的方法进行操作，而不用修改对字符串的引用。
- 修改字符串的值，不是在堆区原值本身上进行修改，而是通过副本进行修改。
- 修改的字符串副本与原值没有任何联系，如果不把修改值复制给原值变量，则不会对原值产生影响。
- 当把修改的字符串复制给原值变量时，会重新建立一个新的引用，并把修改值存储到堆区新的位置。
- 原值引用的区域，如果还被其他变量引用，则继续保留，否则会被 JavaScript 回收程序回收。

建议 35：获取字节长度

String 对象的 length 属性能够返回字符串的长度，不管字符是单字节，还是双字节，都作为一个来计算。因此，要获取字符串的字节长度，必须通过手工计算获取，这里介绍两种方法。

1）第一种方法是利用循环结构枚举每个字符，并根据字符的字符编码，判断当前字符是单字节还是双字节，然后递加字符串的字节数。

```
String.prototype.lengthB = function( ){
    var b = 0, l = this.length;
    if( l ){
        for( var i = 0; i < l; i ++ ){
            if(this.charCodeAt( i ) > 255 ){
                b += 2;
            }else{
                b ++ ;
            }
        }
        return b;
    }else{
        return 0;
    }
}
var s = "String对象长度";
alert(s.lengthB());        // 14
```

在检测字符是否为双字节或单字节时，方法也有多种，这里提供两种思路（代码如下）：

```
for( var i = 0; i < l; i ++ ){
    var c = this.charAt( i );
    if ( escape( c ).length > 4 ) {
```

```
        b += 2;
    }else if( c != "\r" ) {
        b ++ ;
    }
}
```

或者使用正则表达式进行字符编码验证：

```
for( var i = 0; i < l; i ++ ){
    var c = this.charAt( i );
    if ( /^[\u0000-\u00ff]$/.test(c) ) {
        b ++ ;
    }else {
        b += 2;
    }
}
```

2）第二种方法是利用正则表达式把字符串中双字节字符临时替换为两个字符，然后调用 length 属性获取临时字符串的长度。

```
String.prototype.lengthB = function(){
    var s = this.replace( /[^\x00-\xff]/g, "**" );
    return s.length;
}
```

这种方法比较简洁，但执行速度相对较慢，因为需要两次遍历字符串，即调用 replace() 方法时一次，使用 length 属性时一次。而第一种方法只进行一次字符串遍历。

提示：String 对象的 length 属性是只读属性，这与 Array 的 length 属性不同。不过，与数组相同，字符串可以使用下标来定位单个字符在字符串中的位置，其中第一个字符的下标值为 0，最后一个字符的下标值为 length–1。字符串中的字符是不能够被 for in 循环枚举的。运算符 delete 也不能删除字符串中指定位置的字符。

建议 36：警惕字符串连接操作

字符串连接表现出惊人的“性能紧张”。一个任务通过一个循环向字符串末尾不断地添加内容，以创建一个字符串。例如，创建一个 HTML 表或一个 XML 文档。此类处理在一些浏览器上表现得非常糟糕。

当连接少量字符串时，这些问题都可以忽略，临时使用可选择最熟悉的操作。当合并字符串的长度和数量增加之后，有些函数开始显示出“威力”。

（1）+、+=

+、+= 运算符提供了连接字符串的最简单方法。除 IE 7 及其以前版本外，当前所有浏览器都对这种方法优化得很好，因此不需要使用其他方法。当然，还可以提高这些操作的效

率。例如，下面这行代码是字符串连接的常用方法：

```
str += "one" + "two";
```

JavaScript 在执行这行代码时，会进行以下 4 个步骤：

第 1 步，在内存中创建一个临时字符串。

第 2 步，临时字符串的值被赋予“onetwo”。

第 3 步，临时字符串与 str 的值进行连接。

第 4 步，把结果赋予 str。

不过，通过下面代码进行优化能够提高执行效率：两个离散表达式直接将内容附加到 str 上，避免了临时字符串（第 1 步和第 2 步）。在大多数浏览器中，这样做可以使执行速度提升 10% ～ 40%。

```
str += "one";
str += "two";
```

实际上，也可以用以下一行代码实现同样的性能提升。

```
str = str + "one" + "two";
```

这就避免了使用临时字符串，因为赋值表达式开头以 str 为基础，一次追加一个字符串，从左至右依次连接。如果改变连接顺序，如下所示：

```
str = "one" + str + "two";
```

就会失去这种优化性能。这与浏览器合并字符串时分配内存的方法有关。除 IE 以外，浏览器尝试扩展表达式左端字符串的内存，然后简单地将第二个字符串复制到它的尾部。在一个循环中，如果基本字符串位于最左端，就可以避免多次复制一个越来越大的基本字符串。

然而，上面的方法并不适用于 IE。对于 IE 来说，这种优化几乎没有任何作用，在 IE 8 上甚至比 IE 7 和早期版本更慢，这与 IE 执行连接操作的机制有关。

在 IE 8 中，连接字符串只是记录下构成新字符串的各部分字符串的引用。在最后时刻，各部分字符串才被逐个复制到一个新的“真正的”字符串中，然后用它取代先前的字符串引用，因此并非每次使用字符串时都发生合并操作。

IE 7 和更早的浏览器在连接字符串时使用更糟糕的实现方法，每连接一对字符串都要将其复制到一块新分配的内存中。使用上述方法反而会使代码执行速度更慢，因为合并多个短字符串比连接一个大字符串更快，因此要避免多次复制那些大字符串。例如：

```
largeStr = largeStr + s1 + s2;
```

在 IE 7 和更早的版本中，必须将这个大字符串复制两次。首先与 s1 合并，然后再与 s2 合并。相反，对于下面代码：

```
largeStr = s1 + s2;
```

先将两个小字符串合并起来，然后将结果返回给大字符串。创建中间字符串 s1 + s2 与两次复制大字符串相比，对性能的“冲击”要轻得多。

（2）编译期合并

在赋值表达式中所有字符串连接都属于编译期常量，Firefox 自动地在编译过程中合并它们。在以下这个方法中可看到这一过程：

```
function foldingDemo() {
    var str = "compile" + "time" + "folding";
    str += "this" + "works" + "too";
    str = str + "but" + "not" + "this";
}
alert(foldingDemo.toString());alert(foldingDemo.toString());
```

在 Firefox 中我们经常看到这种形式：

```
function foldingDemo() {
    var str = "compiletimefolding";
    str += "thisworkstoo";
    str = str + "but" + "not" + "this";
}
alert(foldingDemo.toString());alert(foldingDemo.toString());
```

当字符串是这样合并在一起时，由于运行时没有中间字符串，因此连接它们的时间和内存可以减少到零。这种功能非常了不起，但它并不经常起作用。

（3）数组联结

Array.prototype.join 方法将数组的所有元素合并成一个字符串，并在每个元素之间插入一个分隔符字符串。如果传递一个空字符串作为分隔符，可以简单地将数组的所有元素连接起来。

在大多数浏览器上，数组联结比连接字符串的其他方法更慢，但事实上，作为一种补偿方法，在 IE 7 和更早的浏览器上它是连接大量字符串的唯一的高效途径。例如，下面的示例代码演示了可用数组联结解决的性能问题：

```
var str = "I'm a thirty-five character string.", newStr = "", appends = 5000;
while(appends--) {
    newStr += str;
}
```

此代码连接 5000 个长度为 35 的字符串。执行以上代码后显示在 IE 7 中执行此测试所需的时间，从 5000 次连接开始，然后逐步增加连接数量。IE 7 的连接算法要求浏览器在循环过程中反复地为越来越大的字符串复制和分配内存，结果是出现以平方关系递增的运行时间和内存消耗。

目前所有其他的浏览器（包括 IE 8 及其以上版本）在这个测试中表现良好，不会呈现平方关系的复杂性递增，这是真正的改善。然而，此程序演示了看似简单的字符串连接所产生的影响。5000 次连接用去 226ms 已经是一个显著的性能冲击了，应当尽可能地缩减这一时

间。锁定用户浏览器长达 32 s，只是为了连接 20 000 个短字符串，这对任何应用程序来说都是不能接受的。

如果使用数组联结生成同样的字符串，则代码如下：

```
var str = "I'm a thirty-five character string.", strs = [], newStr, appends = 5000;
while(appends--) {
    strs[strs.length] = str;
}
newStr = strs.join("");
```

上面代码优化的核心是避免重复的内存分配和复制越来越大的字符串。当联结一个数组时，浏览器宁愿分配足够大的内存用于存放整个字符串，也不会超过一次地复制最终字符串的同一部分。

原生字符串连接函数接受任意数目的参数，并将每一个参数都追加到调用函数的字符串，这是连接字符串最灵活的方法，因为可以利用它追加一个字符串，或者一次追加几个字符串，或者追加一个完整的字符串数组。

```
str = str.concat(s1);
str = str.concat(s1, s2, s3);
str = String.prototype.concat.apply(str, array);
```

在大多数情况下，concat 执行速度比简单的“+”和“+=”慢一些，而且在 IE、Opera 和 Chrome 浏览器上会大幅变慢。此外，虽然使用 concat 合并数组中的所有字符串看起来和前面讨论的数组联结差不多，但是通常它更慢一些（在 Opera 浏览器上除外），而且它还潜伏着严重的性能问题，这与在 IE 7 和更早版本中使用“+”和“+=”创建大字符串情况类似。

建议 37：推荐使用 replace

String 对象的 replace 方法包含两个参数，第一个参数表示执行匹配的正则表达式，也可以传递字符串，第二个参数表示准备代替匹配的子字符串，例如，把字符串 html 替换为 htm。

```
var b = s.replace("html", "htm" );
```

与 search 和 match 方法不同，replace 方法不会把字符串转换为正则表达式对象，而是以字符串直接量的文本模式进行匹配。第二个参数可以是替换的文本，或者是生成替换文本的函数，把函数返回值作为替换文本来替换匹配文本。

replace 方法同时执行查找和替换两个操作。该方法将在字符串中查找与正则表达式相匹配的子字符串，然后调用第二个参数值或替换函数替换这些子字符串。如果正则表达式具有全局性质，那么将替换所有的匹配子字符串，否则，只替换第一个匹配子字符串。

在 replace 方法中约定了一个特殊的字符“$”，如果这个美元符号附加了一个序号，就表示引用正则表达式中匹配的子表达式存储的字符串。例如：

```
var s = "javascript";
var b = s.replace( /(java)(script)/, "$2-$1");
alert( b );        //"script-java"
```

在上面的代码中，正则表达式 /(java)(script)/ 中包含两对小括号，按顺序排列，其中第一对小括号表示第一个子表达式，第二对小括号表示第二个子表达式，在 replace 方法的参数中可以分别使用字符串 "$1" 和 "$2" 来表示对它们匹配文本的引用，当然它们不是标识符，仅是一个标记，所以不可以作为变量参与计算。除了上面约定之外，美元符号与其他特殊字符组合还可以包含更多的语义，详细说明如下：

- $1、$2、…、$99：与正则表达式中的第 1 ～ 99 个子表达式相匹配的文本。
- $&（美元符号 + 连字符）：与正则表达式相匹配的子字符串。
- $`（美元符号 + 切换技能键）：位于匹配子字符串左侧的文本。
- $'（美元符号 + 单引号）：位于匹配子字符串右侧的文本。
- $$：表示 $ 符号。

```
var s = "javascript";
var b = s.replace( /.*/, "$&$&");//" javascriptjavascript "
```

由于字符串“$&”在 replace 方法中被约定为正则表达式所匹配的文本，因此利用它可以重复引用匹配的文本，从而实现字符串重复显示效果。其中正则表达式“/.*/”表示完全匹配字符串。

```
var s = "javascript";
var b = s.replace( /script/, "$& != $`");        //"javascript != java"
```

其中字符“$&”代表匹配子字符串“script”，字符“$`”代表匹配文本左侧文本“java”。

```
var s = "javascript";
var b = s.replace( /java/, "$&$' is "); //"javascript is script"
```

其中字符“$&”代表匹配子字符串“java”，字符“$'”代表匹配文本右侧文本“script”。然后用“$&$' is”所代表的字符串“javascript is”替换原字符串中的“java”子字符串，即组成一个新的字符串“javascript is script”。

在 ECMAScript v3 中明确规定，replace 方法的第二个参数建议使用函数，而不是字符串（当然不是禁止使用），JavaScript 1.2 实现了对这个特性的支持。这样，当 replace 方法执行匹配时，每次都会调用该函数，函数的返回值将作为替换文本执行匹配操作，同时函数可以接收以 $ 为前缀的特殊字符组合，用来对匹配文本的相关信息进行引用。

```
var s = 'script language = "javascript" type= " text / javascript"';
var f = function($1){
    return $1.substring(0,1).toUpperCase() + $1.substring(1);
}
var a = s.replace( /(\b\w+\b)/g, f );
alert(a);  //Script Language = "JavaScript" Type = " Text /JavaScript"
```

在上面的示例代码中，函数 f() 的参数为特殊字符“$1”，它表示正则表达式 /(\b\w+\b)/ 中小括号每次匹配的文本。然后在函数体内对这个匹配文本进行处理，截取其首字母并转换为大写形式，之后返回新处理的字符串。replace 方法能够在原文本中使用这个返回的新字符串替换每次匹配的子字符串。

对于上面的示例，可以使用小括号来获取更多匹配文本的信息。例如，直接利用小括号传递单词的首字母，然后进行大小写转换处理：

```
var s = 'script language = "javaScript" type= " text / javaScript"';
var f = function($1,$2,$3){
    return $2.toUpperCase()+$3;
}
var a = s.replace( /\b(\w)(\w*)\b/g, f ); //Script Language = "JavaScript" Type =
" Text /JavaScript"
```

在函数 f() 中，第一个参数表示每次匹配的文本，第二个参数表示第一个小括号的子表达式所匹配的文本，即单词的首字母，第二个参数表示第二个小括号的子表达式所匹配的文本。

实际上，replace 方法的第二个参数（函数式参数）不需要传递任何形参，replace 方法依然会向它传递多个实参，这些实参都包含一定的意思，具体说明如下：

- 第一个参数表示与匹配模式相匹配的文本，如上面示例中每次匹配的单词字符串。
- 其后的参数是与匹配模式中子表达式相匹配的字符串，参数个数不限，根据子表达式数而定。
- 后面的参数是一个整数，表示匹配文本在字符串中的下标位置。
- 最后一个参数表示字符串自身。

例如，将上面示例中替换文本函数改为如下形式。

```
var f = function(){
    return arguments[1].toUpperCase()+arguments[2];
}
```

如果不为函数传递形参，直接调用函数的 arguments 属性，同样能够读取到正则表达式中相关匹配文本的信息。

- arguments[0] 表示每次匹配的单词。
- arguments[1] 表示第一个子表达式匹配的文本，即单词的首字母。
- arguments[2] 表示第二个子表达式匹配的文本，即单词的余下字母。
- arguments[3] 表示匹配文本的下标位置，如第一个匹配单词“script”的下标位置就是 0，依此类推。
- arguments[4] 表示要执行匹配的字符串，这里表示“script language = "javascript" type= " text / javascript"”。

```
var s = 'script language = "javascript" type= " text / javascript"';
var f = function(){
```

```
        for( var i = 0; i < arguments.length; i ++ ){
            alert(" 第 "+(i+1)+" 个参数的值：  "+arguments[i]);
        }
    }
    var a = s.replace( /\b(\w)(\w*)\b/g, f );
```

在函数体中，使用 for 循环结构遍历 argumnets 属性，每次匹配单词时，都会弹出 5 次提示信息，分别显示上面所列的匹配文本信息。其中，arguments[1]、arguments[2] 会根据每次匹配文本不同，分别显示当前匹配文本中子表达式匹配的信息，arguments[3] 显示当前匹配单词的下标位置。而 arguments[0] 总是显示每次匹配的单词，arguments[4] 总是显示被操作的字符串。

例如，下面代码能够自动提取字符串中的分数，进行汇总后算出平均分，然后利用 replace 方法提取每个分值，与平均分进行比较以决定替换文本的具体信息。

```
var s = " 张三 56 分，李四 74 分，王五 92 分，赵六 84 分 ";
var a = s.match( /\d+/g ), sum = 0;
for( var i= 0 ; i<a.length ; i++){
    sum += parseFloat(a[i]);
};
var avg = sum / a.length;
function f(){
    var n = parseFloat(arguments[1]);
    return n + " 分 " + " ( " + (( n > avg ) ? ( " 超出平均分 " + ( n - avg ) ) :
( " 低于平均分 " + ( avg - n ) )) + " 分 ) ";
}
var s1 = s.replace( /(\d+) 分 /g, f );
alert( s1 );/* " 张三 56 分（低于平均分 20.5 分），李四 74 分（低于平均分 2.5 分），王五 92 分
（超出平均分 15.5 分），赵六 84 分 （ 超出平均分 7.5 分）"*/
```

在上面的示例中，遍历数组时不能够使用 for in 语句，因为这个数组中还存储着其他相关的匹配文本信息。应该使用 for 结构来实现。由于截取的数字都是字符串类型，所以应该把它们都转换为数值类型，否则会被误解，如把数字连接在一起，或者按字母顺序进行比较等。

建议 38：正确认识正则表达式工作机制

有很多因素影响正则表达式的效率。首先，正则表达式适配的文本千差万别，部分匹配时比完全不匹配所用的时间要长。其次，每种浏览器的正则表达式引擎也有不同的内部优化。要有效使用正则表达式，重要的是理解它们的工作机制。一个正则表达式处理的基本步骤如下：

第 1 步，编译。在创建了一个正则表达式对象后，浏览器先要检查模板有没有错误，然后将它转换成一个本机代码例程，用于执行匹配工作。如果将正则表达式赋给一个变量，就可以避免重复执行此步骤。

第 2 步，设置起始位置。当一个正则表达式投入使用时，要先确定目标字符串中开始搜索的位置。它是字符串的起始位置，或者由正则表达式的 lastIndex 属性指定，但当它从第 4 步返回到这里的时候，此位置将位于最后一次尝试起始位置推后一个字符的位置上。

浏览器厂商优化正则表达式引擎的方法：在这一阶段中通过早期预测跳过一些不必要的工作。例如，如果一个正则表达式以 ^ 开头，IE 和 Chrome 浏览器通常判断在字符串起始位置上是否能够匹配，进而避免不明智地搜索后续位置。另一个例子是匹配第三个字母是 x 的字符串，聪明的方法是先找到 x，然后再将起始位置回溯两个字符。

第 3 步，匹配每个正则表达式的字元。正则表达式一旦找好起始位置，将会一个个地扫描目标文本和正则表达式模板。当一个特定字元匹配失败时，正则表达式将试图回溯到扫描之前的位置上，然后进入正则表达式其他可能的路径。

第 4 步，匹配成功或失败。如果在字符串的当前位置上发现一个完全匹配的字符，那么正则表达式宣布成功。如果正则表达式的所有可能路径都尝试过了，但没有成功匹配，那么正则表达式引擎回到第 2 步，从字符串的下一个字符重新尝试。只有字符串中的每个字符（以及最后一个字符后面的位置）都经历了这样的过程之后还没有成功匹配，正则表达式才会宣布彻底失败。

牢记这一过程将有助于判别那些影响正则表达式性能问题的类型。

建议 39：正确理解正则表达式回溯

在正则表达式实现中，回溯是匹配过程的基本组成部分，它是正则表达式如此好用和强大的根源。然而，回溯计算代价很高，如果设计失误，将导致失控。回溯是影响整体性能的唯一因素，理解它的工作原理，以及如何减小使用频率，可能是编写高效正则表达式的关键点。

当一个正则表达式扫描目标字符串时，从左到右逐个扫描正则表达式的组成部分，在每个位置上测试能不能找到一个匹配。对于每一个量词和分支，都必须确定如何继续进行。如果是一个量词（如 *、+? 或者 {2,}），那么正则表达式必须确定何时尝试匹配更多的字符；如果遇到分支（通过 | 操作符），那么正则表达式必须从这些选项中选择一个进行尝试。

当正则表达式做出这样的决定时，如果有必要，它会记住另一个选项，以备返回后使用。如果所选方案匹配成功，正则表达式将继续扫描正则表达式模板，如果其余部分匹配也成功了，那么匹配就结束了。但是，如果所选择的方案未能发现相应匹配，或者后来的匹配也失败了，正则表达式将回溯到最后一个决策点，然后在剩余的选项中选择一个。继续这样，直到找到一个匹配，或者量词和分支选项的所有可能的排列组合都尝试失败后放弃这一过程，然后移动到此过程开始位置的下一个字符上，重复此过程。

例如，下面的代码演示了这一过程是如何通过回溯处理分支的。

```
/h(ello|appy) hippo/.test("hello there, happy hippo");
```

上面一行正则表达式用于匹配“hello hippo”或“happy hippo”。测试一开始要查找一个h，目标字符串的第一个字母恰好就是 h，立刻就找到了。接下来，子表达式（ello|appy）提供了两个处理选项。正则表达式选择最左边的选项（分支选择总是从左到右进行），检查 ello 是否匹配字符串的下一个字符，确实匹配，然后正则表达式又匹配了后面的空格。

然而，在接下来的匹配中正则表达式“走进了死胡同”，因为 hippo 中的 h 不能匹配字符串中的下一个字母 t。此时正则表达式还不能放弃，因为它还没有尝试过所有的选择，随后它回溯到最后一个检查点（在匹配了首字母 h 之后的那个位置上）并尝试匹配第二个分支选项。但由于匹配没有成功，而且也没有更多的选项了，正则表达式认为从字符串的第一个字符开始匹配是不能成功的，因此它从第二个字符开始重新进行查找。正则表达式没有找到 h，继续向后找，直到第 14 个字母才找到，它匹配 happy 的那个 h。随后正则表达式再次进入分支过程，这次 ello 未能匹配，但在回溯之后的第二次分支中，它匹配了整个字符串“happy hippo”，匹配成功了。

再如，下面代码演示了带重复量词的回溯。

```
    var str = "<p>Para 1.</p>" +"<img src='smiley.jpg'>" +"<p>Para 2.</p>"
+"<div>Div.</div>";
    /<p>.*<\/p>/i.test(str);
```

正则表达式先匹配了字符串开始的 3 个字母 ，然后是 .*。点号表示匹配除换行符以外的任意字符，星号这个“贪婪”量词表示重复零次或多次，匹配尽量多的次数。因为目标字符串中没有换行符，正则表达式将匹配剩下的全部字符串！不过由于正则表达式模板中还有更多内容需要匹配，所以正则表达式尝试匹配 <。由于在字符串末尾匹配不成功，因此每次回溯一个字符，继续尝试匹配 <，直到正则表达式回到 标签的 < 位置。接下来尝试匹配 \/（转义反斜杠），匹配成功，然后匹配 p，匹配不成功。正则表达式继续回溯，重复此过程，直到第二段末尾时终于匹配了 。匹配返回成功需要从第一段头部一直扫描到最后一个的末尾，这可能不是我们想要的结果。

将正则表达式中的“贪婪”量词 * 改为“懒惰”（又名“非贪婪”）量词 *?，以匹配单个段落。“懒惰”量词的回溯工作以相反方式进行。当正则表达式 / .*?<\/p>/ 推进到 .*? 时，首先尝试全部跳过，然后继续匹配 <\/p>。

这样做是因为 *? 匹配零次或多次，尽可能少重复，尽可能少意味着可以重复零次。但是，当随后的 < 在字符串的这一点上匹配失败时，正则表达式回溯并尝试下一个最小的字符数：1 个。正则表达式继续像这样向前回溯到第一段的末尾，在那里量词后面的 <\/p> 得到完全匹配。

如果目标字符串只有一个段落，那么此正则表达式的“贪婪”版本和“懒惰”版本是等价的，但尝试匹配的过程不同。

当一个正则表达式占用浏览器几秒甚至更长时间时，问题原因很可能是回溯失控。为说明此问题，给出下面的正则表达式，它的目标是匹配整个 HTML 文件。此表达式被拆分成多

行是为了适合页面显示。与其他正则表达式不同，JavaScript 在没有选项时可使点号匹配任意字符，包括换行符，所以此例中以 [\s\S] 匹配任意字符。

```
/<html>[\s\S]*?<head>[\s\S]*?<title>[\s\S]*?<\/title>[\s\S]*?<\/head>
[\s\S]*?<body>[\s\S]*?<\/body>[\s\S]*?<\/html>/
```

此正则表达式匹配在正常 HTML 字符串时工作良好，但当目标字符串缺少一个或多个标签时，就会变得十分糟糕。例如 </html> 标签缺失，最后一个 [\s\S]*? 将扩展到字符串的末尾，因为在那里没有发现 </html> 标签，然后正则表达式将查看此前的 [\s\S]*? 队列记录的回溯位置，使它们进一步扩大。正则表达式尝试扩展倒数第二个 [\s\S]*?——用它匹配 </body> 标签，就是此前匹配过正则表达式模板 <\/body> 的那个标签，然后继续查找第二个 </body> 标签，直到字符串的末尾。当所有这些步骤都失败时，倒数第三个 [\s\S]*? 将被扩展，直至字符串的末尾，依此类推。

此类问题的解决办法在于尽可能具体地指出分隔符之间的字符匹配形式，如模板 ".*?" 用于匹配双引号包围的一个字符串。用更具体的 [^"\rn]* 取代过于宽泛的 .*? 就去除了回溯时可能发生的几种情况，如尝试用点号匹配引号，或者扩展搜索超出预期范围。

在 HTML 的例子中解决办法不是那么简单。不能使用否定字符类型，如用 [^<] 替代 [\s\S]，因为在搜索过程中可能会遇到其他类型的标签。但是，可以通过重复一个非捕获组来达到同样效果，它包含一个回溯（阻塞下一个所需的标签）和 [\s\S]（任意字符）元序列。这样可以确保中间位置上查找的每个标签都会失败。然后，更重要的是，[\s\S] 模板在回溯过程中阻塞的标签在被发现之前不能被扩展。应用此方法后对正则表达式的最终修改如下：

```
/<html>(?:(?!<head>)[\s\S])*<head>(?:(?!<title>)[\s\S])*<title>
(?:(?!<\/title>)[\s\S])*<\/title>(?:(?!<\/head>)[\s\S])*<\/head>
(?:(?!<body>)[\s\S])*<body>(?:(?!<\/body>)[\s\S])*<\/body>
(?:(?!<\/html>)[\s\S])*<\/html>/
```

虽然这样做消除了潜在的回溯失控，并允许正则表达式在匹配不完整 HTML 字符串失败时的使用时间与文本长度呈线性关系，但是正则表达式的效率并没有提高。像这样为每个匹配字符进行多次前瞻，缺乏效率，而且成功匹配过程也相当慢。匹配较短字符串时使用此方法相当不错，而匹配一个 HTML 文件可能需要前瞻并测试上千次。

建议 40：正确使用正则表达式分组

所谓分组，就是通过使用小括号语法分隔符来包含一系列字符、字符类，或者重复类量词，以实现处理各种特殊的字符序列。例如，针对下面的字符串，希望分拣出每个标签：

```
var s ="<html><head><title></title></head><body></body></html>";
```

如果使用贪婪模式进行匹配，虽然可以抓取所有标签，但是并没有分开每个标签：

```
var r = /<.*>/
var a = s.match(r); /* 单元素数组 ["<html><head><title></title></head> <body>
</body></html>"] */
```

如果使用惰性模式进行匹配，每次仅能够抓取一个标签：

```
var r = /<.*?>/
var a = s.match(r);        // ["<html>"]
```

但是，如果利用分组来进行匹配，则可以获取每个标签的名称：

```
var r = /(<.*?>)/g;        // 分组模式
var a = s.match(r);        // 全局匹配标签，并存储到数组 a 中
for(var i = 0 ; i < a.length; i ++ ){    // 遍历数组 a，获取每个标签的名称
    alert(a[i]);
}
```

在上面的示例中，通过小括号逻辑分隔符，实现分别存储每个被匹配的标签，最后通过这个数组来获取每个标签的名称。注意，对正则表达式来说，小括号表示一个独立的逻辑域，其匹配的内容将被独立存储，这样就可以以数组形式读取每个子表达式所匹配的信息。

再看一个示例，假设准备匹配下面这个长字符串：

```
var s = "abcdef-abcdef-abcdef-abcdef-abcdef";
```

如果不使用分组，那么可能实现的正则表达式如下：

```
var r = /abcdef-abcdef-abcdef-abcdef-abcdef/;
var a = s.match(r);        // 单元素数组 ["abcdef-abcdef-abcdef-abcdef-abcdef"]
```

尽管这是可以的，还是有点低效。如果不知道该字符串中到底出现几次重复，这时候可以使用分组来重写这个表达式：

```
var r = /(abcdef-?)*/;     // 分组模式进行匹配
var a = s.match(r);        //["abcdef-abcdef-abcdef-abcdef-abcdef", "abcdef"]
```

在小括号内的匹配模式表示正则表达式的子表达式，而跟随在小括号后的重复类数量词将会作用于子表达式，而不是字符“)”。因此，在上面的示例中通过小括号把每个标签内容作为匹配对象，然后通过重复类星号不但能进行迭代匹配，最终能够快速实现匹配的目的。

当然，并不限制在分组后使用星号，还可以使用任意重复类数量词：

```
var r = /(abcdef-?){5}/;          // 连续匹配 5 次子表达式
var r = /(abcdef-?){1,5}/;        // 最多匹配 5 次子表达式
var r = /(abcdef-?){0,}/;         // 匹配任意次子表达式
var r = /(abcdef-?)?/;            // 最多匹配一次子表达式
var r = /(abcdef-?)+/;            // 最小匹配一次子表达式
```

如果混合使用字符、字符类和量词，甚至可以实现一些相当复杂的分组，例如：

```
var s = "<html>< html><html >< html ></html>< /html></ html>< / html >";
var r = /<([\/\s]*?)html(\s)*?>/g;
```

```
    var a = s.match(r);        //["<html >","< html >","< html  >","<  html  >","</html
>","<  /html >","< / html >","<  / html  >"]
```

在上面的正则表达式中，使用了两个分组：第一个分组中包含了一个字符范围类，其中可以任意匹配空格或斜杠，文本范围类附加了一个量词“*”，表示空格或斜杠可以出现任意次数，为了避免正则表达式的贪婪性，在重复类量词后面附加了问号（?），使其以惰性模式进行匹配；第二个分组是一个简单的任意空格匹配。当然可以不分组，但是第一个分组是必需的，因为数量词作用于子表达式，而不是某个特定的字符。通过上面正则表达式，可以匹配任意形式的 <html> 标签，这样就不用担心空格或斜杠对匹配语义的影响。在正则表达式中，分组具有极高的应用价值，下面进行简要说明。

- 把单独的项目进行分组，以便合成子表达式，这样就可以像处理一个独立的字符那样，使用 |、+、* 或 ? 等元字符来处理它们。例如：

```
var s ="javascript is not java";
var r = /java(script)?/g;
var a = s.match(r);                    //["javascript","java"]
```

上面的正则表达式可以匹配字符串“javascript”，也可以匹配字符串“java”，因为在匹配模式中通过分组可以使用量词“?”来修饰该子表达式，这样匹配字符串时，其后既可以有“script”，也可以没有。

- 在正则表达式中，通过分组可以在一个完整的模式中定义子模式。当一个正则表达式成功地与目标字符串相匹配时，也可以从目标字符串中抽出与小括号中的子模式相匹配的部分。例如：

```
var s ="ab=21,bc=45,cd=43";
var r = /(\w+)=(\d*)/;
var a = s.match(r);        //["ab=21" , "ab","21"]
```

在上面的示例中，不仅要匹配出每个变量声明，还想知道每个变量的名称及其值。这时如果使用小括号进行分组，把需要独立获取的信息作为子表达式，就不仅可以抽出声明，还可以提取更多的有用的信息。

- 在同一个正则表达式的后部可以引用前面的子表达式。这是通过在字符“\”后加一位或多位数字实现的。数字指定了带括号的子表达式在正则表达式中的位置，如“\1”引用的是第一个带括号的子表达式，“\2”引用的是第二个带小括号的子表达式。例如：

```
var s ="<h1>title<h1><p>text<p>";
var r = /(<\/?\w+>).*\1/g;
var a = s.match(r);        //["<h1>title<h1>" , "<p>text<p>"]
```

在上面的示例中，通过引用前面子表达式匹配的文本来实现成组匹配字符串。

由于子表达式可以嵌套在其他子表达式中，因此它的位置编号是根据左括号的顺序来定

的。例如，在下面的正则表达式中，嵌套的子表达式 (<\/?\w+>) 被指定为“\2”。

```
var s ="<h1>title<h1><p>text<p>";
var r = /((<\/?\w+>).*\2)/g;
var a = s.match(r);          //["<h1>title<h1>" , "<p>text<p>"]
```

注意，对正则表达式中前面子表达式的引用，指的并不是那个子表达式的模式，而是与模式相匹配的文本。例如，下面这个字符串就无法实现匹配。

```
var s ="<h1>title</h1><p>text</p>";
var r = /((<\/?\w+>).*\2)/g;
var a = s.match(r);          //null
```

虽然子表达式 (<\/?\w+>) 可以匹配“<h1>”，也可以匹配“</h1>”，但是对于“\2”来说，它引用的是前面子表达式匹配的文本，而不是它的匹配模式。如果要引用前面子表达式的匹配模式，则必须使用下面正则表达式。

```
var r = /((<\/?\w+>).*((<\/?\w+>))/g;
var a = s.match(r);          //["<h1>title</h1>","<p>text</p>"]
```

建议 41：正确使用正则表达式引用

正则表达式在执行匹配运算时会自动把每个分组（子表达式）匹配的文本都存储在一个特殊的地方以备将来使用。这些存储在分组中的特殊值被称为反向引用。反向引用将遵循从左到右的顺序，根据表达式中左括号字符的顺序进行创建和编号。

```
var s = "abcdefghijklmn";
var r = /(a(b(c)))/;
var a = s.match(r);          //["abc", "abc" , "bc" , "c"]
```

在这个分组匹配模式中，共产生了 3 个反向引用，第一个是“(a(b(c)))”，第二个是“(b(c))”，第三个是“(c)”。它们引用的匹配文本分别是字符串“abc”、“bc”和“c”。

反向引用在应用开发中主要有以下几种常规用法。

1）在正则表达式对象的 test() 方法，以及字符串对象的 match() 和 search() 等方法中使用。在这些方法中，反向引用的值可以从 RegExp() 构造函数中获得。例如：

```
var s = "abcdefghijklmn";
var r = /(\w)(\w)(\w)/;
r.test(s);
alert(RegExp.$1);            // 第 1 个子表达式匹配的字符 a
alert(RegExp.$2);            // 第 2 个子表达式匹配的字符 b
alert(RegExp.$3);            // 第 3 个子表达式匹配的字符 c
```

在正则表达式执行匹配测试后，所有子表达式匹配的文本都被分组存储在 RegExp() 构造函数的属性内，通过前缀符号 $ 与正则表达式中子表达式的编号来引用这些临时属性，其中属性 $1 标识符指向第一个值引用，属性 $2 标识符指向第二个值引用，依此类推。

2）可以直接在定义分组的表达式中包含反向引用。这可以通过使用特殊转义序列（如\1、\2 等）来实现。例如：

```
var s = "abcbcacba";
var r = /(\w)(\w)(\w)\2\3\1\3\2\1/;
var b = r.test(s);          // 验证正则表达式是否匹配该字符串
alert(b);                   //true
```

在上面的正则表达式中，“\1”表示对第一个反向引用（\w）所匹配的字符 a 的引用，“\2”表示对第二个反向引用（\w）所匹配的字符 b 的引用，“\3”表示对第二个反向引用（\w）所匹配的字符 c 的引用。

3）可以在字符串对象的 replace() 方法中使用。通过使用特殊字符序列 $1、$2、$3 等来实现。例如，在下面的示例中将颠倒相邻字母和数字的位置。

```
var s = "aa11bb22c3d4e5f6";
var r = /(\w+?)(\d+)/g;
var b = s.replace(r,"$2$1");
alert(b);                   //"11aa22bb3c 4d5e6f"
```

在这个示例中，正则表达式包括两个分组，第一个分组匹配任意连续的字母，第二个分组匹配任意连续的数字。在 replace() 方法的第二个参数中，$1 表示对正则表达式中第一个子表达式匹配文本的引用，而 $2 表示对正则表达式中第二个子表达式匹配文本的引用，通过颠倒 $1 和 $2 标识符的位置即可实现字符串的颠倒以替换原字符串。

建议 42：用好正则表达式静态值

正则表达式的静态属性比较特殊，有两个名字：长名（全称）和短名（简称，以美元符号开头表示），详细说明见表 2.1。

表 2.1　RegExp 的静态属性

长名	短名	说明
input	$_	最后用于匹配的字符串，即传递给 exec() 或 test() 方法的字符串
lastMatch	$&	最后匹配的字符
lastParen	$+	最后匹配的分组
leftContext	$`	在上次匹配之前的子字符串
multiline	$*	用于指定是否所有表达式都使用多行模式的布尔值
rightContext	$'	在上次匹配之后的子字符串

在下面的这个示例中借助正则表达式的静态属性，匹配字符串“Javascript”，不区分大小写：

```
var s = "Javascript,not Javascript";
var r = /(Java)Script/gi;
var a = r.exec(s);
alert(RegExp.input);       //"Javascript,not Javascript"
alert(RegExp.leftContext);        // 空字符串，因为在第一次匹配操作时，左侧没有内容
alert(RegExp.rightContext);       //",not Javascript"
alert(RegExp.lastMatch);  //"Javascript "
alert(RegExp.lastParen);  //"Java"
```

上面示例演示了正则表达式的几个静态属性的用法。

- input 属性实际上存储的是被执行匹配的字符串，即整个字符串“Javascript,not Javascript”。
- leftContext 属性存储的是执行第一次匹配之前的子字符串，这里为空，因为在第一次匹配时文本“Javascript”左侧为空，而 rightContext 属性存储的是执行第一次匹配之后的子字符串，即为“,not Javascript”。
- lastMatch 属性包含的是第一次匹配的子字符串，即为“Javascript”。
- lastParen 属性包含的是第一次匹配的分组，即为“Java”。如果模式中包含多个分组，则会显示最后一个分组所匹配的字符。例如：

```
var r = /(Java)(Script)/gi;
var a = r.exec(s);                  // 执行匹配操作
alert(RegExp.lastParen);            // 返回 "Script"，而不再是 "Java"
```

也可以使用短名来读取这些属性所包含的值，考虑到这些短名不符合 JavaScript 语法规范，因此必须使用中括号运算符来进行读取操作。不过对于 $_ 属性来说，由于它符合 JavaScript 标识符语法规范，因此可以直接使用。例如，针对上面示例也可以这样设计：

```
var s = "Javascript,not Javascript";
var r = /(Java)(Script)/gi;
var a = r.exec(s);
alert(RegExp.$_);          //"Javascript,not Javascript"
alert(RegExp["$`"]);       // 空字符串
alert(RegExp["$'"]);       //",not Javascript"
alert(RegExp["$&"]);       //"Javascript "
alert(RegExp["$+"]);       //"Java"
```

这些属性的值都是动态的，每次执行 exec() 或 test() 方法时，所有属性值都会被重新设置。当在下面示例中执行第一次匹配和第二次匹配时，这些静态属性值都会实时动态更新。

```
var s = "Javascript,not Javascript";
var r = /Scrip(t)/gi;      // 第一次定义的匹配模式
var a = r.exec(s);         // 执行第一次匹配
alert(RegExp.$_); //"Javascript,not Javascript"
alert(RegExp["$`"]);       //"Java"
alert(RegExp["$'"]);       //",not Javascript"
alert(RegExp["$&"]);       //"Script"
alert(RegExp["$+"]);       //"t"
```

```
var r = /Jav(a)/gi;          // 第二次定义的匹配模式
var a = r.exec(s);           // 执行第二次匹配
alert(RegExp.$_);            //"Javascript,not Javascript"
alert(RegExp["$`"]);         // 空字符串
alert(RegExp["$'"]);         //"Script,not Javascript"
alert(RegExp["$&"]);         //"Java"
alert(RegExp["$+"]);         //"a"
```

通过上面的示例可以看出，RegExp 对象的静态属性是公共的，对于所有正则表达式来说是可以共享的，因此这些静态属性的值也是实时变化的。

multiline 属性与上面几个属性不同，它不会根据每次执行的操作进行实时更新，并且还可以控制所有正则表达式的 m 标志项。例如：

```
var s = "a\nb\nc";
var r = /\w+$/g;             // 定义匹配模式
var a = s.match(r);          // 执行默认匹配，返回数组 ["c"]
RegExp.multiline = true;     // 动态设置模式为多行匹配
var a = s.match(r);          //["a", "b", "c"]
```

提示：IE 和 Opera 浏览器不支持 RegExp.multiline 属性，考虑到浏览器的兼容性，不建议读者使用这种动态方式设置正则表达式的多行匹配模式。

建议 43：使用 exec 增强正则表达式功能

RegExp 对象定义了两个用于执行模式匹配操作的方法，它们的行为与 String 对象的正则表达式操作方法类似。例如，RegExp 对象的 exec 方法与 String 对象的 match 方法相似，只不过 exec 是以字符串为参数的 RegExp 对象方法，而 match 方法是以正则表达式为参数的 String 对象方法。在非全局模式下，它们的返回值是相同的。

在所有 RegExp 模式匹配方法和 String 模式匹配方法中，exec 方法的功能最强大。作为正则表达式的通用匹配方法，exec 方法比 RegExp.test()、String.search()、String.replace() 和 String.match() 都复杂。该方法需要一个参数，用来执行要执行操作的字符串并返回一个数组，此数组中存放的是匹配结果。如果没有找到匹配的文本，返回值为 null。例如：

```
var s = "javascript";
var r = /java/g;
var a = r.exec(s);           //["java"]
```

exec 方法的工作机制是这样的：当调用方法时，先检索字符串参数，从中获取与正则表达式相匹配的文本。如果找到了匹配的文本，就会返回一个数组；否则，返回 null。对返回数组的元素的具体说明如下：

- 第 0 个元素，是与表达式相匹配的文本。
- 第 1 个元素，是与正则表达式的第 1 个子表达式相匹配的文本（如果存在）。

❑ 第 2 个元素，是与正则表达式的第 2 个子表达式相匹配的文本，依此类推。

返回数组还包含几个属性，具体说明如下：

❑ length，该属性声明的是数组中的元素个数。

❑ index，该属性声明的是匹配文本的第一个字符的位置。

❑ input，该属性包含的是整个字符串。

当调用非全局模式的正则表达式对象的 exec 方法时，返回的数组与调用字符串对象的 match 方法返回的数组是完全相同的。

当执行全局匹配模式时，exec 的行为就略有变化。这时它会定义 lastIndex 属性，以指定下一次执行匹配时开始检索字符串的位置。在找到了与表达式相匹配的文本之后，exec 方法将把正则表达式的 lastIndex 属性设置为下一次匹配执行的第一个字符的位置。也就是说，可以通过反复地调用 exec 方法来遍历字符串中的所有匹配文本。当 exec 再也找不到匹配的文本时，将返回 null，并且把属性 lastIndex 重置为 0。

在下面的这个示例中，定义正则表达式直接量，用来匹配字符串 s 中每个字符。在循环结构的条件表达式中反复执行匹配模式，并将返回结果的值是否为 null 作为循环条件。当返回值为 null 时，说明字符串检测完毕。然后，读取返回数组 a 中包含的匹配子字符串，并调用该数组的属性 index 和 lastIndex，其中 index 显示当前匹配子字符串的起始位置，而 lastIndex 属性显示下一次匹配操作的起始位置。例如：

```
var s = "javascript";                    // 测试使用的字符串直接量
var r = /\w/g;                           // 匹配模式
while((a = r.exec(s)) != null){          // 循环执行匹配操作
    alert(a[0] + "\n" + a.index  + "\n" +  r.lastIndex); /* 显示每次匹配操作时返回的结
                                                            果数组信息 */
}
```

实际上通过循环结构反复调用 exec 方法是唯一获得全局模式的完整模式匹配信息的方法。

无论正则表达式是否为全局模式，exec 方法都会将完整的细节添加到返回数组中。字符串对象的 match 方法就不同，它在全局模式下返回的数组中不会包含这么多的细节信息。

建议 44：正确使用原子组

正则表达式引擎支持一种称做原子组的属性。原子组写作 (?>…)，也称为“贪婪”子表达式，省略号表示任意正则表达式模板、非捕获组和一个特殊的扭曲。存在于原子组中的正则表达式组中的任何回溯点都将被丢弃。这就为 HTML 正则表达式的回溯问题提供了一个更好的解决办法：如果将 [\s\S]*? 序列和它后面的 HTML 标记一起放在一个原子组中，所需的 HTML 标签被发现一次，这次匹配基本上就被锁定了。如果正则表达式的后续部分匹配失败，原子组中的量词没有记录回溯点，那么 [\s\S]*? 序列就不能扩展到已匹配的范围之外。

但是，JavaScript 不支持原子组，也不提供其他方法消除不必要的回溯。不过，可以利

用前瞻过程中一项鲜为人知的行为来模拟原子组：前瞻也是原子组。不同的是，前瞻在整个匹配过程中不消耗字符，前瞻只是检查自己包含的模板是否能在当前位置匹配。然而，可以避开这点，在捕获组中包装一个前瞻模板，在前瞻之外向它添加一个后向引用。

```
(?=(pattern to make atomic))\1
```

在任何使用原子组的模式中这个结构都是可重用的。只要记住，需要使用适当的后向引用次数，如果正则表达式包含多个捕获组。HTML 正则表达式在使用此技术后的修改如下：

```
/<html>(?=([\s\S]*?<head>))\1(?=([\s\S]*?<title>))\2(?=([\s\S]*?
<\/title>))\3(?=([\s\S]*?<\/head>))\4(?=([\s\S]*?<body>))\5
(?=([\s\S]*?<\/body>))\6[\s\S]*?<\/html>/
```

如果没有尾随的 </html>，那么最后一个 [\s\S]*? 将扩展至字符串结束，正则表达式将立刻失败，因为没有回溯点可以返回。正则表达式每次找到一个中间标签就退出一个前瞻，它在前瞻过程中丢弃所有回溯位置。下一个后向引用简单地重新匹配前瞻过程中发现的字符，将它们作为实际匹配的一部分。

建议 45：警惕嵌套量词和回溯失控

嵌套量词总是需要额外的关注和小心，以确保没有掩盖回溯失控问题。嵌套量词出现在一个自身被重复量词修饰的组中。

嵌套量词本身并不会造成性能危害，只是在尝试匹配字符串过程中，很容易不小心在内部量词和外部量词之间，产生一大堆分解文本的方法。例如，要匹配 HTML 标签，使用了下面的正则表达式：

```
/<(?:[^>"']|"[^"]*"|'[^']*')*>/
```

这也许过于简单，因为它不能正确地处理所有情况的有效和无效标记，但在处理有效 HTML 片段时应该没什么问题。与更加简单的 /<[^>]*>/ 相比，它的优势在于涵盖了出现在属性值中的 > 符号。在非捕获组中它不使用第二和第三分支，仅匹配单引号和双引号包围的属性值，除特定的引号外允许所有字符出现。

虽然遇到了嵌套量词 *，但目前还没有回溯失控的危险。在分组的每次重复过程中，由于第二和第三分支选项严格匹配一个带引号的字符串，所以潜在的回溯点数目随目标字符串长度而呈线性增长。

但是，查看非捕获组的第一分支：[^>"']，每次只匹配一个字符，效率似乎有些低。在字符类后面加一个量词会更好些，这样每次组重复过程就可以匹配多于一个的字符了。正则表达式可以在目标字符串的位置上发现一个匹配。通过每次匹配多个字符，正则表达式会在成功匹配的过程中跳过许多不必要的步骤。

如果正则表达式匹配一个“<”字符，但后面没有“>”，则可以令匹配成功完成，回溯

失控就会进入“快车道”，因为内部量词和外部量词的排列组合产生了数量巨大的分支路径（跟在非捕获组之后）用以匹配“<”之后的文本。正则表达式在最终放弃匹配之前必须尝试所有的排列组合。

关于嵌套量词导致回溯失控，一个更加极端的例子是，在一大串 A 上应用正则表达式 /(A+A+)+B/。虽然这个正则表达式写成 /AA+B/ 更好，但为了讨论方便，设想一下两个 A 能够匹配同一个字符串的多少种模板。

当应用在一个由 10 个 A 组成的字符串上（“AAAAAAAAAA”）时，正则表达式首先使用第一个 A+ 匹配所有 10 个字符，然后正则表达式回溯一个字符，让第二个 A+ 匹配最后一个字符。这个分组试图重复，但没有更多的 A，而且分组中的 + 量词已经符合匹配所需的最少一次，因此正则表达式开始查找 B。虽然正则表达式没有找到 B，但是还不能放弃，因为还有许多路径没有被测试过。如果第一个 A+ 匹配 8 个字符，第二个 A+ 匹配 2 个字符会怎么样呢？或者第一个匹配 3 个，第二个匹配 2 个，分组重复两次，又会怎么样呢？如果在分组的第一遍重复中，第一个 A+ 匹配 2 个字符，第二个 A+ 匹配 3 个字符，然后在第二遍重复中，第一个匹配 1 个，第二个匹配 4 个，又怎么样呢？

正则表达式在最坏情况下的复杂性是惊人的 O(2n)，也就是 2 的 n 次方。n 表示字符串的长度。在由 10 个 A 构成的字符串中，正则表达式需要 1024 次回溯才能确定匹配失败，如果是 20 个 A，回溯的数字剧增到一百万以上。25 个 A 足以挂起 Chrome、IE、Firefox 和 Opera 浏览器至少 10 分钟（如果还没死机）用以处理超过 34 000 000 次回溯以排除正则表达式的各种排列组合。唯一的例外是最新的 Safari 浏览器，它能够检测到正则表达式陷入了循环，并快速终止匹配（Safari 浏览器还限定了回溯的次数，超出则终止匹配尝试）。

预防此类问题的关键是确保正则表达式的两个部分不能对字符串的同一部分进行匹配。这个正则表达式可重写为 /AA+B/，但复杂的正则表达式可能难以避免此类问题。虽然还有其他解决办法，但是增加一个模拟原子组往往作为最后一招使用，如果可能，尽可能保持正则表达式简单易懂。如果这么做，此正则表达式将改成 /((?=(A+A+))\2)+B/，就可以彻底消除回溯问题。

建议 46：提高正则表达式执行效率

（1）关注如何让匹配更快失败

正则表达式处理慢往往是因为匹配失败过程慢，而不是匹配成功过程慢。使用正则表达式匹配一个很大字符串的一小部分，情况更为严重，正则表达式匹配失败的位置比匹配成功的位置要多得多。一个修改使正则表达式匹配更快但失败更慢，例如，通过增加所需的回溯次数尝试所有分支的排列组合，这通常是一个失败的修改。

（2）正则表达式以简单的、必需的字元开始

最理想的情况是，一个正则表达式的起始字元应当尽可能快速地测试并排除明显不匹配

的位置。用于此目的好的起始字元通常是一个锚（^ 或 $）、特定字符（如 x 或 \u363A）、字符类（如 [a-z] 或速记符、单词边界（\b））。如果可能，避免以分组或选择字元开头，避免顶级分支，如 /one|two/，因为这样会强迫正则表达式识别多种起始字元。Firefox 浏览器对起始字元中使用的任何量词都很敏感，能够优化得更好。例如，以 \s\s* 替代 \s+ 或 \s{1,}。其他浏览器大多优化掉这些差异。

（3）编写量词模板，使它们后面的字元互相排斥

当字符与字元相邻或子表达式能够重叠匹配时，一个正则表达式尝试分解文本的路径数量将增加。为避免出现此现象，尽量具体化模板。当表达“[^"\r\n]*”时不要使用“.*?”（依赖回溯）。

（4）减少分支的数量，缩小它们的范围

当分支使用 |（竖线）时，可能要求在字符串的每一个位置上测试所有的分支选项。通常可通过使用字符类和选项组件减少对分支的需求，或者将分支在正则表达式上的位置推后（允许到达分支之前的一些匹配尝试失败）。

字符类比分支更快，因为它们使用位向量实现（或其他快速实现）而不是回溯。当分支必不可少时，在不影响正则表达式匹配的情况下，将常用分支放在最前面。分支选项从左向右依次尝试，一个选项被匹配上的机会越多，它被检测的速度就越快。

注意：由于 Chrome 和 Firefox 浏览器自动执行这些优化中的某些项目，因此较少受到手工调整的影响。

（5）使用非捕获组

捕获组花费时间和内存用于记录后向引用，并保持它们是最新的。如果不需要一个后向引用，可通过使用非捕获组避免这种开销，例如，(?:…) 替代（…）。当需要一个完全匹配的后向引用时，有些人喜欢将正则表达式包装在一个捕获组中，这是不必要的，因为可以通过其他方法引用完全匹配。例如，使用 regex.exec() 返回数组的第一个元素，或替换字符串中的 $&。用非捕获组取代捕获组在 Firefox 浏览器中影响很小，但在其他浏览器上处理长字符串时影响很大。

（6）捕获感兴趣的文字，减少后处理

如果要引用匹配的一部分，应当通过一切手段，捕获那些片断，再使用后向引用处理。例如，编写代码处理一个正则表达式所匹配的引号中的字符串内容，使用 /"([^"]*)"/ 之后再使用一次后向引用，而不是使用 /"[^"]*"/ 之后从结果中手工剥离引号。当在循环中使用时，减少这方面的工作可以节省大量时间。

（7）暴露所需的字元

为帮助正则表达式引擎在如何优化查询例程时做出明智的决策，应尽量简单地判断出那些必需的字元。当字元应用在子表达式或分支中时，正则表达式引擎很难判断它们是不是必

需的，有些引擎并不做此方面的努力。例如，正则表达式 /^(ab|cd)/ 暴露它的字符串起始锚。IE 和 Chrome 浏览器会注意到这一点，并阻止正则表达式尝试查找字符串头端之后的匹配，从而使查找瞬间完成而不管字符串长度。但是，由于等价正则表达式 /(^ab|^cd)/ 不暴露它的 ^ 锚，IE 无法应用同样的优化，最终无意义地搜索字符串并在每一个位置上匹配。

（8）使用适当的量词

正如建议 45 所讨论过的那样，“贪婪”量词和“懒惰”量词即使匹配同样的字符串，其查找匹配过程也是不同的。在确保正确等价的前提下，使用更合适的量词类型（基于预期的回溯次数）可以显著提高性能，尤其在处理长字符串时。

（9）将正则表达式赋给变量，以重用它们

将正则表达式赋给变量以避免对它们重新编译。有人使用正则表达式缓存池，以避免对给定的模板和标记组合进行多次编译。不要过分担心，正则表达式编译得很快。重要的是避免在循环体中重复编译正则表达式。换句话说，不要这样做：

```
while (/regex1/.test(str1)) {
    /regex2/.exec(str2);
    ...
}
```

替代做法如下：

```
var regex1 = /regex1/,
regex2 = /regex2/;
while (regex1.test(str1)) {
    regex2.exec(str2);
    ...
}
```

（10）将复杂的正则表达式拆分为简单的片断

尽量避免一个正则表达式做太多的工作。处理复杂的搜索问题需要将条件逻辑拆分为两个或多个正则表达式，这样更容易解决问题，通常也更高效，每个正则表达式只在最后的匹配结果中执行查找。在一个模板中完成所有工作的正则表达式很难维护，而且容易引起回溯相关的问题。

建议 47：避免使用正则表达式的场景

正则表达式匹配速度是非常快的。然而，当只搜索文字字符串时正则匹配经常会显得多余，尤其当事先知道了字符串的哪一部分将要被测试时。例如，要检查一个字符串是不是以分号结束，可以使用：

```
endsWithSemicolon = /;$/.test(str);
```

当前没有哪个浏览器“聪明”到这个程度，能够意识到这个正则表达式只能匹配字符串

的末尾。最终它们所做的将是一个一个地测试整个字符串。每当发现了一个分号，正则表达式就前进到下一个字元（$），检查它是否匹配字符串的末尾。如果不是这样，正则表达式就继续搜索匹配，直到搜索了整个字符串。字符串的长度越长（包含的分号越多），它占用的时间也就越长。

在这种情况下，更好的办法是跳过正则表达式所需的所有中间步骤，简单地检查最后一个字符是不是分号：

```
endsWithSemicolon = str.charAt(str.length - 1) == ";";
```

目标字符串很小时，这种方法只比正则表达式快一点，更重要的是，字符串的长度不再影响执行测试所需要的时间。

例如，使用 charAt 函数在特定位置上读取字符。字符串函数 slice、substr 和 substring 可用于在特定位置上提取并检查字符串的值。此外，indexOf 和 lastIndexOf 函数非常适合查找特定字符串的位置，或者判断它们是否存在。所有这些字符串操作函数速度都很快，在搜索那些不依赖正则表达式复杂特性的文本字符串时，它们有助于减小正则表达式带来的性能开销。

建议 48：慎用正则表达式修剪字符串

（1）使用两个子表达式修剪字符串

去除字符串首尾的空格是一个简单而常见的任务，但到目前为止 JavaScript 还没有实现它。正则表达式允许用很少的代码实现一个修剪函数，最好的全面解决方案可能是使用两个子表达式：一个用于去除头部空格，另一个用于去除尾部空格。这样处理简单而快速，特别是处理长字符串时。

```
if(!String.prototype.trim) {
    String.prototype.trim = function() {
        return this.replace(/^\s+/, "").replace(/\s+$/, "");
    }
}
var str = " \t\n test string ".trim();
alert(str == "test string");     // alerts "true"
```

使用 if 语句进行检测，如果已经存在 trim 原生函数，则不要覆盖 trim 原生函数，因为原生函数进行了优化后通常远远快于自定义函数。使用上面代码在 Firefox 浏览器中大约有 35% 的性能提升（或多或少依赖于目标字符串的长度和内容）。将 /\s+$/（第二个正则表达式）替换成 /\s\s*$/。虽然这两个正则表达式的功能完全相同，但是 Firefox 浏览器却为那些以非量词字元开头的正则表达式提供额外的优化。在其他浏览器上，差异不显著，或者优化完全不同。

然而，改变正则表达式，在字符串开头匹配 /^\s\s*/ 不会产生明显差异，因为 ^ 锚需要

“照顾”那些快速作废的非匹配位置（避免一个轻微的性能差异，因为在一个长字符串中可能产生上千次匹配尝试）。

（2）使用一个正则表达式修剪字符串

事实上，除这里列出的方法外还有许多其他方法，可以写一个正则表达式来修剪字符串，但在处理长字符串时，这种方法执行速度总比用两个简单的表达式要慢。

```
String.prototype.trim = function() {
    return this.replace(/^\s+|\s+$/g, "");
}
```

这可能是最通常的解决方案。它通过分支功能合并了两个简单的正则表达式，并使用 /g（全局）标记替换所有匹配，而不只是第一个匹配（当目标字符串首尾都有空格时将匹配两次）。这并不是一个“可怕”的方法，但在对长字符串操作时，它比使用两个简单的子表达式要慢，因为两个分支选项都要测试每个字符位置。

```
String.prototype.trim = function() {
    return this.replace(/^\s*([\s\S]*?)\s*$/, "$1");
}
```

这个正则表达式的工作原理是匹配整个字符串，捕获从第一个到最后一个非空格字符之间的序列，记入后向引用 1。然后使用后向引用 1 替代整个字符串，就留下了这个字符串的修剪版本。

这个方法概念简单，但捕获组中的“懒惰”量词使正则表达式进行了许多额外操作（如回溯），因此在操作长目标字符串时很慢。在进入正则表达式捕获组时，[\s\S] 类的“懒惰”量词 *? 要求捕获组尽可能地减少重复次数。因此，这个正则表达式每匹配一个字符，都要停下来尝试匹配余下的 \s*$ 模板。如果由于字符串当前位置之后存在非空格字符而导致匹配失败，正则表达式将匹配一个或多个字符，更新后向引用，然后再次尝试匹配模板的剩余部分。

```
String.prototype.trim = function() {
    return this.replace(/^\s*([\s\S]*\S)?\s*$/, "$1");
}
```

这个表达式与上一个很像，但出于性能原因以“贪婪”量词取代了“懒惰”量词。为确保捕获组只匹配到最后一个非空格字符，必须尾随一个 \S。然而，由于正则表达式必须匹配全部由空格组成的字符串，整个捕获组通过尾随一个 ? 量词而成为可选组。

在此，[\s\S]* 中的“贪婪”量词“*”表示重复方括号中的任意字符模板直至字符串结束。然后，正则表达式每次回溯一个字符，直到它能够匹配后面的 \S，或者直到回溯到第一个字符而匹配整个组（之后它跳过这个组）。

如果尾部空格不比其他字符串更多，通过一个表达修剪的方案通常比前面那些使用“懒惰”量词的方案更快。事实上，这个方案在 IE、Safari、Chrome 和 Opera 浏览器上执行速度如此之快，甚至超过使用两个子表达式的方案，是因为这些浏览器包含特殊优化，专门服务

于为字符类匹配任意字符的“贪婪”重复操作，正则表达式引擎直接跳到字符串末尾而不检查中间的字符（尽管回溯点必须被记下来），然后适当回溯。不幸的是，这种方法在 Firefox 和 Opera 9 浏览器上执行得非常慢，所以到目前为止，使用两个子表达式仍然是更好的跨浏览器方案。

```
String.prototype.trim = function() {
    return this.replace(/^\s*(\S*(\s+\S+)*)\s*$/, "$1");
}
```

这是一个相当普遍的方法，但没有很好的理由使用它，因为它在所有浏览器上都是这里所列出的所有方法中执行得最慢的一个。这类似于最后两个正则表达式，它匹配整个字符串然后用打算保留的部分替换这个字符串，因为内部组每次只匹配一个单词，正则表达式必须执行大量的离散步骤。修剪短字符串时性能冲击并不明显，但处理包含多个词的长字符串时，这个正则表达式可以成为影响性能的一个问题。

将内部组修改为一个非捕获组，例如，将 (\s+\S+) 修改为 (?:\s+\S+)，在 Opera、IE 和 Chrome 浏览器上缩减了大约 20% ～ 45% 的处理时间，在 Safari 和 Firefox 浏览器上也有轻微改善。尽管如此，一个非捕获组不能完全代换这个实现。注意，外部组不能转换为非捕获组，因为它在被替换的字符串中被引用了。

虽然正则表达式的执行速度很快，但是没有它们帮助时修剪字符串的性能还是值得考虑的。例如：

```
String.prototype.trim = function() {
    var start = 0,
    end = this.length - 1,
    ws = " \n\r\t\f\x0b\xa0\u1680\u180e\u2000\u2001\u2002\u2003
    \u2004\u2005\u2006\u2007\u2008\u2009\u200a\u200b\u2028\u2029\u202f
    \u205f\u3000\ufeff";
    while (ws.indexOf(this.charAt(start)) > -1) {
        start++;
    }
    while (end > start && ws.indexOf(this.charAt(end)) > -1) {
        end--;
    }
    return this.slice(start, end + 1);
}
```

在上面代码中，ws 变量包括在 ECMAScript v5 中定义的所有空白字符。出于效率方面的考虑，在得到修剪区的起始和终止位置之前避免复制字符串的任何部分。

当字符串末尾只有少量空格时，这种情况使正则表达式处于无序状态。原因是，尽管正则表达式很好地去除了字符串头部的空格，却不能同样快速地修剪长字符串的尾部。一个正则表达式不能跳到字符串的末尾而不考虑沿途字符。正因如此，在第二个 while 循环中从字符串末尾向前查找一个非空格字符。

虽然上面代码不受字符串总长度影响，但是它有自己的弱点——长的头尾空格，因为循

环检查字符是不是空格在效率上不如正则表达式所使用的优化过的搜索代码。

（3）正则表达式与非正则表达式结合起来修剪字符串

最后一个办法是将正则表达式与非正则表达式两者结合起来，用正则表达式修剪头部空格，用非正则表达式方法修剪尾部字符。

```
String.prototype.trim = function() {
    var str = this.replace(/^\s+/, ""),
    end = str.length - 1,
    ws = /\s/;
    while (ws.test(str.charAt(end))) {
        end--;
    }
    return str.slice(0, end + 1);
}
```

当只修剪一个空格时，此混合方法非常快，同时去除了性能上的风险，如以长空格开头的字符串，完全由空格组成的字符串（尽管它在处理尾部长空格的字符串时仍具有弱点）。

注意：此方案在循环中使用正则表达式检测字符串尾部的字符是否为空格，虽然使用正则表达式增加了一点性能负担，但是它允许根据浏览器定义空格字符列表，以保持简短和兼容性。

所有修剪方法总的趋势：在基于正则表达式的方案中，字符串总长比修剪掉的字符数量更影响性能；而非正则表达式方案从字符串末尾反向查找，不受字符串总长的影响，但明显受到修剪空格数量的影响。简单地使用两个子正则表达式在所有浏览器上处理不同内容和长度的字符串时，均表现出稳定的性能，因此可以说这种方案是最全面的解决方案。混合解决方案在处理长字符串时特别快，其代价是代码稍长，在某些浏览器上处理尾部长空格时存在弱点。

建议 49：比较数组与对象同源特性

我们常常将对象和数组作为不同的数据类型来处理，这是一种有用且合理的简化，通过这种处理可以在大多数的 JavaScript 程序设计中将对象和数组作为单独的类型来处理。要完全掌握对象和数组的行为，还必须了解数组不过是一个具有额外功能层的对象。使用 typeof 运算符时就会发现这一点，因为将其作用于一个数组的值，返回值是字符串“object”。

数组是一段线性分配的内存，它通过整数去计算偏移并访问其中的元素。数组可以是访问速度很快的数据结构。不幸的是，JavaScript 没有数组这样的数据结构。相反，JavaScript 提供了一种拥有一些类数组（array-like）特性的对象，把数组的下标转变成字符串，将其作为属性。这类对象的访问速度明显比真正数组慢，但它使用更方便。属性的检索和更新方式与对象一模一样。数组有它自己的字面量格式，还有一套非常有用的内置方法。

（1）数组字面量

数组字面量提供了一种非常方便地创建新数组的表示法。一个数组字面量是在一对方括号中包围零个或多个用逗号分隔的值的表达式。数组字面量可以出现在任何表达式可以出现的地方。数组的第一个值将获得属性名“0”，第二个值将获得属性名“1”，依此类推。

```
var empty = [];
var numbers = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven',
'eight', 'nine'];
empty[1]                    // undefined
numbers[1]                  // 'one'
empty.length                // 0'
numbers.length              //10
```

（2）对象字面量

```
var numbers_object = {
    '0' : 'zero',
    '1' : 'one',
    '2' : 'two',
    '3' : 'three',
    '4' : 'four',
    '5' : 'five',
    '6' : 'six',
    '7' : 'seven',
    '8' : 'eight',
    '9' : 'nine'
};
```

产生了一个与前面的方法相似的结果。numbers 和 numbers_object 都是包含 10 个属性的对象，并且这些属性刚好有相同的名字和值。但 numbers 和 numbers_object 也有一些显著的不同，numbers 继承自 Array.prototype，而 numbers_object 继承自 Object.prototype，所以 numbers 继承了大量有用的方法。同时，numbers 有一个 length 属性，而 numbers_object 则没有。

在大多数语言中，一个数组的所有元素都要是相同的类型，而 JavaScript 允许数组包含任意混合类型的值。

```
var misc = ['string', 98.6, true, false, null, undefined, ['nested', 'array'], {
    object : true
}, NaN, Infinity];
misc.length
```

建议 50：正确检测数组类型

由于数据和对象的数据同源性，导致在 JavaScript 编程中经常会出现：在必须使用数组时使用了对象，或者在必须使用对象时使用了数组。

选用数组或对象的规则很简单：当属性名是小而连续的整数时，应该使用数组，或者当对属性的位置和排列顺序有要求时，应该使用数组。否则，使用对象。

JavaScript 语言对数组和对象的区别是混乱的。typeof 运算符检测数组的类型是“object”，这没有什么意义，因此在正确检测数组和对象方面 JavaScript 没有提供很多的机制。这时可以通过自定义 is_ array 函数来弥补这个缺陷。

```
var is_array = function(value) {
    return value && typeof value === 'object' && value.constructor === Array;
};
```

不过，上面函数在识别从不同的窗口（window）或帧（frame）中构造的数组时会失败，要想准确地检测外部数组类型，还需要更进一步地完善该函数。

```
var is_array = function(value) {
    return value &&
    typeof value === 'object' &&
    value.constructor === Array &&
    typeof value.length === 'number' &&
    typeof value.splice === 'function' &&
    !(value.propertyIsEnumerable('length'));
};
```

在完善后的函数中，首先要判断这个值是否为真，函数不接受 null 和其他值为假的值。其次，判断对这个值的 typeof 运算的结果是否是 object。对于对象、数组和 null 来说，得到的都将是 true。接着，判断这个值是否有一个值为数字的 length 属性，对于数组将总是得到 true，而对于对象来说并非如此。接下来，判断这个值是否包含一个 splice 方法。对于所有数组来说，这又将得到 true。最后，判断 length 属性是否是可枚举的，对于所有数组来说，将得到 false，这是对数组最可靠的测试。

不过，使用下面的方法也能够很好地检测数组类型，并且这种方法显得更加简洁。

```
var is_array = function(value) {
    return Object.prototype.toString.apply(value) === '[object Array]';
};
```

建议 51：理解数组长度的有限性和无限性

每个数组都有一个 length 属性。和大多数其他语言不同，JavaScript 数组的 length 是没有上限的。如果用大于或等于当前 length 的数字作为下标来保存一个元素，那么 length 将增大以容纳新元素，不会发生数组边界错误。

length 属性的值是这个数组的最大整数属性名加上 1。它不一定等于数组中属性的个数。例如，下面数组 myArray 最后长度为 10001，但它仅包含一个元素：

```
var myArray = [];
```

```
myArray.length      // 0
myArray[10000] = true;
myArray.length      // 10001
```

[] 后缀下标运算符将它的表达式转换成一个字符串，如果该表达式中有 toString 方法，就使用该方法的值。这个字符串将用做属性名。如果这个字符串看起来像一个大于或等于这个数组当前的 length 且小于 4 294 967 295 的正整数，那么这个数组的 length 就会被重新设置为新的下标加 1。根据 ECMAScript 262 标准，数组的下标必须是大于或等于 0 且小于 $2^{32}-1$ 的整数。

我们可以直接为数组设置 length 值。当设置更大的 length 时，也不用向数组分配更多的空间，而当把 length 设置为小于数组的实际长度时，将导致所有下标大于或等于新 length 的元素被删除。

```
var numbers=['zero','one','two','three','four','five','six','seven','eight','nine'];
numbers.length = 3; // numbers = ['zero', 'one', 'two']
```

通过将下标指定为一个数组的当前 length，可以附加一个新元素到该数组的尾部。

```
numbers[numbers.length] = 'ten'
```

有时用 push 方法可以更方便地完成同样的事情。

```
numbers.push('ten');
```

建议 52：建议使用 splice 删除数组

删除数组元素的方法有多种，最简单的方法是利用 length 属性来截断数组，但这种方法比较笨拙，仅能够截断尾部元素。在 JavaScript 中，由于数组其实就是对象，因此使用 delete 运算符可以从数组中移除元素。

```
var numbers=['zero','one','two','three','four','five','six','seven','eight','nine'];
delete numbers[2];//  numbers=['zero','one',undefined,'three','four','five',
'six','seven','eight','nine'];
```

但是，使用这种方式删除指定下标位置的元素后，会在数组中遗留一个空洞。这是因为排在被删除元素之后的元素保留了它们最初的名字（下标位置），而我们通常想要的是递减后面每个元素的名字（下标）。

幸运的是，JavaScript 数组有一个 splice 方法，它可以删除一些元素并将它们替换为其他的元素。splice 方法的第一个参数是数组中的一个序号，第二个参数是要删除的元素个数。任何额外的参数会在序号那个点的位置被插入数组中。例如：

```
var numbers=['zero','one','two','three','four','five','six','seven','eight','nine'];
numbers. splice (2,1) ;      // numbers = ['zero', 'one', 'three', 'four', 'five', 'six',
'seven', 'eight', 'nine'];
```

执行以上代码之后，元素值为“three”的下标值从 4 变为 3，同理，其后所有元素的下标值都递减 1。因为必须移除被删除元素后面的每个元素，并且将一个新的键值重新插入，因此，这对于大型数组来说效率会更高。

建议 53：小心使用数组维度

在 JavaScript 中，数组在默认状态下是不会初始化的。如果使用 [] 运算符创建一个新数组，那么此数组将是空的。如果访问的是数组中不存在的元素，则得到的值将是 undefined。因此，在 JavaScript 程序设计中应该时刻考虑这个问题：在尝试读取每个元素之前，都应该预先设置它的值。但是，如果在设计中假设每个元素都从一个已知的值开始（如 0），那么就必须预定义这个数组。我们也可以为 JavaScript 自定义一个静态函数：

```
Array.dim = function(dimension, initial) {
    var a = [], i;
    for( i = 0; i < dimension; i += 1) {
        a[i] = initial;
    }
    return a;
};
```

借助这个工具函数，可以轻松地创建一个初始化数组。例如，创建一个包含 100 个 0 的数组：

```
var myArray = Array.dim(100, 0);
```

JavaScript 没有多维数组，但是它支持元素为数组的数组。

```
var matrix = [
     [0, 1, 2],
     [3, 4, 5],
     [6, 7, 8]
];
matrix[2][1]       //7
```

为了自动化创建一个二维数组或一个元素为数组的数组，我们不妨这样做：

```
for( i = 0; i < n; i += 1) {
    my_array[i] = [];
}
```

注意，Array.dim(n, []) 在这里不能工作，如果使用它，每个元素都指向同一个数组的引用，那是非常糟糕的。

一个空矩阵的每个单元将拥有一个初始值 undefined。如果希望它们有不同的初始值，必须明确地设置它们的值。因此，我们可以单独为 Array 定义一个矩阵数组定义函数。

```
Array.matrix = function(m, n, initial) {
```

```
    var a, i, j, mat = [];
    for( i = 0; i < m; i += 1) {
        a = [];
        for( j = 0; j < n; j += 1) {
            a[j] = initial;
        }
        mat[i] = a;
    }
    return mat;
};
```

下面就利用这个矩阵数组定义函数构建一个 5×5 的矩阵数组，且每个元素的初始值为 0。

```
var myMatrix = Array.matrix(5, 5, 0);
document.writeln(myMatrix[2][4]);          // 0
```

建议 54：增强数组排序的 sort 功能

sort 方法不仅按字母顺序进行排序，还可以根据其他顺序执行操作。这时就必须为方法提供一个比较函数的参数，该函数要比较两个值，然后返回一个用于说明这两个值的相对顺序的数字。比较函数应该具有两个参数 a 和 b，其返回值如下：

1）如果根据自定义评判标准，a 小于 b，在排序后的数组中 a 应该出现在 b 之前，就返回一个小于 0 的值。

2）如果 a 等于 b，就返回 0。

3）如果 a 大于 b，就返回一个大于 0 的值。

在下面的示例中，将根据比较函数来比较数组中每个元素的大小，并按从小到大的顺序执行排序：

```
function f( a, b ){
    return ( a - b )
}
var a = [3, 1, 2, 4, 5, 7, 6, 8, 0, 9];          // 定义数组
a.sort(f);
alert( a );                                       //[0,1,2 ,3,4, 5,6,7 ,8,9]
```

如果按从大到小的顺序执行排序，则让返回值取反即可，代码如下：

```
function f( a, b ){
    return  -( a - b )
}
var a = [3, 1, 2, 4, 5, 7, 6, 8, 0, 9];
a.sort(f);
alert( a );        //[9,8,7 ,6,5, 4,3,2 ,1,0]
```

（1）根据奇偶性质排列数组

sort 方法用法比较灵活，但更灵活的是对比较函数的设计。例如，要根据奇偶数顺序排

列数组，只需要判断比较函数中两个参数是否为奇偶数，并决定排列顺序，代码如下：

```
function f( a, b ){
    var a = a % 2;
    var b = b % 2;
    if( a == 0 ) return 1;
    if( b == 0 ) return -1;
}
var a = [3, 1, 2, 4, 5, 7, 6, 8, 0, 9];
a.sort( f );
alert( a );                    //[3,1,5,7,9,0,8,6,4,2]
```

sort 方法在调用比较函数时，将每个元素值传递给比较函数，如果元素值为偶数，则保留其位置不动；如果元素值为奇数，则调换参数 a 和 b 的显示顺序，从而实现对数组中所有元素执行奇偶排序。如果希望偶数排在前面，奇数排在后面，则只需要取返回值。比较函数如下：

```
function f( a, b ){
    var a = a % 2;
    var b = b % 2;
    if( a  == 0 ) return -1;
    if( b  == 0 ) return 1;
}
```

（2）不区分大小写排序字符串

在正常情况下，对字符串进行排序是区分大小写的，这是因为每个字母大写形式和小写形式在字符编码表中的顺序是不同的，大写形式排在小写形式前面，例如：

```
var a = ["aB", "Ab", "Ba", "bA"];
a.sort();
alert( a );        //["Ab","Ba","aB","bA"]
```

也就是说，同一字母大写形式总是排在左侧，而小写形式总是排在右侧。如果让小写形式总是排在前面，则可以这样设计：

```
function f( a, b ){
    return ( a < b );
}
var a = ["aB", "Ab", "Ba", "bA"];
a.sort( f );
alert( a );        //["Ab","Ba","aB","bA"]
```

在比较字母大小时，JavaScript 根据字符编码大小来决定字母的大小，当比较函数的返回值为 true 时，则返回 1；当比较函数的返回值为 false 时，则返回 –1。如果不希望区分字母大小，也就是说，大写字母和小写字母按相同顺序排列，则可以这样设计：

```
function f( a, b ){
    var a = a.toLowerCase;
    var b = b.toLowerCase;
```

```
    if( a < b ){
        return  1;
    }
    else{
        return -1;
    }
}
var a = ["aB", "Ab", "Ba", "bA"];// 定义数组
a.sort( f );
alert( a );        //["aB", "Ab", "Ba", "bA"]
```

如果要调整排序顺序，则对返回值取反即可。

（3）把浮点数和整数分开排列

经常会遇到把浮点数和整数分开排列的情况。当然，借助 sort 方法实现起来并不是很难，我们可以这样设计：

```
function f( a, b ){
    if( a > Math.floor( a ) ) return  1;
    if( b > Math.floor( b ) ) return  - 1;
}
var a = [3.55555, 1.23456, 3, 2.11111, 5, 7, 3];
a.sort( f );
alert( a );        //[3,5,7,3,2.11111,1.23456,3.55555]
```

如果要调整排序顺序，则对返回值取反即可。

sort 方法的功能是非常强大的，如果比较的元素是对象而不是值类型（如数字和字符串等）这样简单的数据时，排序就变得更加有趣了，读者可以自己动手试一试。

建议 55：不要拘泥于数字下标

数组下标默认为大于或等于 0 的整数，不过 JavaScript 允许数组下标可以为任意表达式，甚至为任意类型数据。

（1）文本下标

为数组下标指定负值：

```
var a = [];        // 定义空数组直接量
a[-1] = 1;         // 为下标为 -1 的元素赋值
```

很显然，上面的用法是非法的，因为这不符合语法规范。使用 length 属性检测，返回值为 0，说明数组并没有增加长度，这是正确的，也很正常。但是，使用下面的方法可以读取该元素的值：

```
alert(a.length);  // 0，说明数组长度没有增加
alert(a[-1]);     // 1
alert(a["-1"]);   // 1，说明这个值以对象属性的形式被存储
```

不仅如此，还可以为数组指定字符串下标，或者布尔值下标，例如：

```
var a = [];
a[true] = 1;
a[false] = 0;
alert(a.length);   //0，说明数组长度没有增加
alert(a[true]);    //1
alert(a[false]);   //0
alert(a[0]);       //undefined
alert(a[1]);       //undefined
```

虽然 true 和 false 可以被转换为 1 和 0，但是 JavaScript 并没有执行转换，而是把它们视为对象属性来看待。如果文本是数字，可以直接使用数字下标来访问，这时 JavaScript 又能够自动转换它们的类型。例如：

```
a["1"] = 1;
alert(a[1]);       //1
```

这种数据存储格式称为哈希表。哈希表的数据检索速度要快于数组迭代检索，因此，对于下面的操作：

```
var a = [["张三",1],["李四",2],["王五",3]];      // 二维数组
for(var i in a){                                  // 遍历二维数组
    if(a[i][0] == "李四") alert(a[i][1]) ;        // 检索指定元素
}
```

将数组下标改为文本下标会更为高效。

```
var a = [];                 // 定义空数组
a["张三"] = 1;              // 以文本下标来存储元素的值
a["李四"] = 2;
a["王五"] = 3;
alert(a["李四"] );          // 快速定位检索
```

（2）二维数组下标

JavaScript 不支持定义二维数组的语法，但我们可以模仿其他语言中二维数组的形式来定义数组。例如，下面的写法虽然不符合语法上的规定，但是 JavaScript 不会提示编译错误：

```
var a = [];
a[0,0] = 1;
a[0,1] = 2;
a[1,0] = 3;
a[1,1] = 4;
```

如果调用 length 属性，返回值为 2，则说明仅有两个元素，分别读取元素的值，代码如下：

```
alert(a.length);   //2，说明仅有两个元素有效
alert(a[0]);       //3
alert(a[1]);       //3
```

JavaScript 把二维数组下标视为一个逗号表达式，其运算的返回值是最后一个值。前面

两行代码赋值就被后面两行代码赋值覆盖了。因此，如果经过计算之后才确定下标值，之后再进行存取操作，则可以按如下方式进行设计：

```
var a = [], i = 1;          // 初始化变量
while( i < 10 ){            // 指定循环次数
    a[i *= 2 , i] = i;      // 指定下标为 2 的指数时才进行赋值
}
alert( a.length );          //17
alert( a );                 //[,,2,, 4,,,, 8,,,,,,,, 16]
```

（3）对象下标

对象也可以作为数组下标。JavaScript 会试图把对象转换为数值，如果不行，则把它转换为字符串，然后以文本下标的形式进行操作。例如：

```
var a = [];                 // 数组直接量
var b = function(){         // 函数直接量
    return 2;
}
a[b] = 1;                   // 把对象作为数组下标
alert( a.length );          // 长度为 0
alert( a[b] );              //1
```

可以这样读取元素值：

```
var s =b.toString();        // 获取对象的字符串
alert( a[s] );              // 利用文本下标读取元素的值
```

还可以这样设计下标：

```
a[b()] = 1;         // 在下标处调用函数，则返回值为 2
alert( a[2] );      // 因此可以使用 2 来读取该元素值
```

建议 56：使用 arguments 模拟重载

在 JavaScript 中，每个函数内部可以使用 arguments 对象，该对象包含了函数被调用时的实际参数值。arguments 对象虽然在功能上与数组有些类似，但它不是数组。 arguments 对象与数组的类似体现在它有一个 length 属性，同时实际参数的值可以通过 [] 操作符来获取，但 arguments 对象并没有数组可以使用的 push、pop、splice 等方法。其原因是 arguments 对象的 prototype 指向的是 Object.prototype，而不是 Array.prototype。

Java 和 C++ 语言都支持方法重载，即允许出现名称相同而形式参数不同的方法，但 JavaScript 并不支持这种方式的重载。这是因为 JavaScript 中的 function 对象也是以属性的形式出现的，在一个对象中增加与已有 function 同名的新 function 时，旧的 function 对象会被覆盖。不过可以通过使用 arguments 来模拟重载，其实现机制是通过判断 arguments 中实际参数的个数和类型来执行不同的逻辑。

```
function sayHello() {
    switch (arguments.length) {
        case 0:
            return "Hello";
        case 1:
            return "Hello, " + arguments[0];
        case 2:
            return (arguments[1] == "cn" ? " 你好，" : "Hello, ") + arguments[0];
    };
}
sayHello();                 //"Hello"
sayHello("Alex");           // "Hello, Alex"
sayHello("Alex", "cn");     // " 你好, Alex"
```

callee 是 arguments 对象的一个属性，其值是当前正在执行的 function 对象。它的作用是使匿名 function 可以被递归调用。下面以一段计算斐波那契序列中第 N 个数的值的过程来演示 arguments.callee 的使用。

```
function fibonacci(num) {
    return (function(num) {
        if( typeof num !== "number")
            return -1;
        num = parseInt(num);
        if(num < 1)
            return -1;
        if(num == 1 || num == 2)
            return 1;
        return arguments.callee(num - 1) + arguments.callee(num - 2);
    })(num);
}
fibonacci(100);
```

第3章 函数式编程

JavaScript是一门优美的语言，具有动态性、弱类型，具有C和LISP的双重语法。JavaScript虽然是基于对象编程，但是对象不是第一型的，而函数是第一型的。

函数式编程思想的源头可以追溯到20世纪30年代阿朗佐·丘奇进行的一项关于问题的可计算性的研究，也就是后来的lambda演算。lambda演算的本质就是一切皆函数，函数可以作为另外一个函数的输出或输入，一系列的函数使用最终会形成一个表达式链，通过这个表达式链可以最终求得一个值，而这个过程即为计算的本质。在函数式编程中，会发现代码中存在大量的连续运算。

函数式编程已经在实际应用中发挥了巨大作用，更有越来越多的语言不断地加入对诸如闭包、匿名函数等的支持，从某种程度上来讲，函数式编程正在逐步同化命令式编程。

建议57：禁用Function构造函数

定义函数的方法包括3种：function语句、Function构造函数和函数直接量。不管用哪种方法定义函数，它们都是Function对象的实例，并将继承Function对象所有默认或自定义的方法和属性。

```
// 使用function语句编写函数
function f(x){
    return x;
}
// 使用Function()构造函数克隆函数
var f = new Function("x", "return x;");
// 使用函数直接量直接生成函数
var f = function(x){
    return x;
}
```

虽然这些方法定义函数的结构体相同，函数的效果相近，但是也存在很多差异，详细比较见表 3.1。

表 3.1 函数定义方法比较

	使用 function 语句	使用 Function 构造函数	使用函数直接量
兼容	完全	JavaScript 1.1 及以上	JavaScript 1.2 及以上
形式	句子	表达式	表达式
名称	有名	匿名	匿名
主体	标准语法	字符串	标准语法
性质	静态	动态	静态
解析	以命令的形式构造一个函数对象	解析函数体，能够动态创建一个新的函数对象	以表达式的形式构造一个函数对象

（1）作用域比较

使用 Function 构造函数创建的函数具有顶级作用域，JavaScript 解释器也总是把它作为顶级函数来编译，而 function 语句和函数直接量定义的函数都有自己的作用域（即局部作用域，或称为函数作用域）。例如：

```
var n = 1;
function f(){
    var n = 2;
    function e(){
        return n;
    }
    return e;
}
alert(f()());      //2
```

在上面示例中，分别在函数体外和函数体内声明并初始化变量 n，然后在函数体内使用 function 语句定义一个函数 e，定义该函数返回变量 n 的值。最后在函数体外调用函数的返回函数。通过结果可以发现，返回值为局部变量 n 的值（即为 2），也就是说，function 语句定义的函数拥有自己的作用域。同理，如果使用函数直接量定义函数 e，当调用该返回函数时，返回值是 2，而不是 1，那么也说明函数直接量定义的函数拥有自己的作用域。

但是，如果使用 Function 构造函数定义函数 e，则调用该返回函数时，返回的值是 1，而不再是 2 了，看来 Function 构造函数定义的函数作用域需要动态确定，而不是在定义函数时确定的，代码如下：

```
var n = 1;
function f(){
    var n = 2;
    var e = new Function("return n;");
    return e;
```

```
}
alert(f()());      //1
```

（2）解析效率比较

JavaScript 解释器在解析代码时，并非一行行地分析和执行程序，而是一段段地分析执行。在同一段代码中，使用 function 语句和函数直接量定义的函数结构总会被提取出来优先执行。只有在函数被解析和执行完毕之后，才会按顺序执行其他代码行。但是使用 Function 构造函数定义的函数并非提前运行，而是在运行时动态地被执行，这也是 Function 构造函数定义的函数具有顶级作用域的根本原因。

从时间角度审视，function 语句和函数直接量定义的函数具有静态的特性，而 Function 构造函数定义的函数具有动态的特性。这种解析机制的不同，必然带来不同的执行效率，这种差异性可以通过将一个循环结构放大来比较得出。

在下面这个示例中，分别把 function 语句定义的空函数和 Function 构造函数定义的空函数放在一个循环体内，让它们空转十万次，这样就会明显感到使用 function 语句定义的空函数运行效率更高。

```
// 测试 function 语句定义的空函数执行效率
var a = new Date();
var x = a.getTime();
for(var i=0;i<100000;i++){
    function(){                    // 使用 function 语句定义的空函数
        ;
    }
}
var b = new Date();
var y = b.getTime();
alert(y-x);                        //62，不同环境和浏览器会存在差异
// 测试 Function 构造函数定义的空函数执行效率
var a = new Date();
var x = a.getTime();
for(var i=0;i<100000;i++){
    new Function();                // 使用 Function 构造函数定义的空函数
}
var b = new Date();
var y = b.getTime();
alert(y-x);                        //2390
```

JavaScript 解释器首先把 function 语句定义的函数提取出来进行编译，这样每次循环执行该函数时，就不再从头开始重新编译该函数对象了，而 Function 构造函数定义的函数每次循环时都需要动态编译一次，这样效率就非常低了。

（3）兼容性比较

从兼容角度考虑，使用 function 语句定义函数不用考虑 JavaScript 版本问题，所有版本都支持这种方法。而 Function 构造函数只能在 JavaScript 1.1 及其以上版本中使用，函数直接量仅在 JavaScript 1.2 及其以上版本中有效。当然，版本问题现在已经不是问题了。

Function 构造函数和函数直接量定义函数方法有点相似，它们都是使用表达式来创建的，而不是通过语句创建的，这样带来了很大的灵活性。对于仅使用一次的函数，非常适合使用表达式的方法来创建。

由于 Function 构造函数和函数直接量定义函数不需要额外的变量，它们直接在表达式中参与运算，所以节省了资源，克服了使用 function 语句定义函数占用内存的弊端，也就是说，这些函数运行完毕即被释放而不再占用内存空间。

对于 Function 构造函数来说，由于定义函数的主体必须以字符串的形式来表示，使用这种方法定义复杂的函数就显得有点笨拙，很容易出现语法错误。但函数直接量的主体使用标准的 JavaScript 语法，这样看来使用函数直接量是一种比较快捷的方法。

通过 function 语句定义的函数称为命名式函数、声明式函数或函数常量，而通过匿名方式定义的函数称为引用式函数或函数表达式，而把赋值给变量的匿名函数称为函数对象，把该变量称为函数引用。

建立 58：灵活使用 Arguments

JavaScript 函数的参数是不固定的，调用函数时传递给它的实参也很随意，为了有效管理参数，JavaScript 支持 Arguments 机制。

Arguments 是一个伪数组，可以通过数组下标的形式获取该集合中传递给函数的参数值。例如，在下面这个函数中，没有指定形参，但在函数体内通过 Arguments 对象可以获取传递给该函数的每个实参值。

```
function f(){
    for(var i = 0; i < arguments.length; i ++ ){
        alert(arguments[i]);
    }
}
f(3, 3, 6);
```

Arguments 对象仅能够在函数体内使用，它仅作为函数体的一个私有成员而存在，因此可以通过点号运算符来指定 Arguments 对象所属的函数。由于 Arguments 对象在函数体内是唯一的和可指向的，因此一般会省略前置路径，直接引用 Arguments 对象的调用标识符 arguments。

通过数组的形式来引用 Arguments 对象包含的实参值，如 arguments[i]，其中 arguments 表示对 Arguments 对象的实际引用，变量 i 是 Arguments 对象集合的下标值，从 0 开始，直到 arguments.length（其中 length 是 Arguments 对象的一个属性，表示 Arguments 对象包含的实参的个数）。

由于 Arguments 不是 Array 的实例，因此不能够直接调用数组的方法，但通过 call 或 apply 方法能够间接实现调用数组的部分方法。

Arguments 对象中的每个元素实际上就是一个变量，这些变量用来存储调用函数时传递的实参值。通过 arguments[] 数组和已经命名的形参可以引用这些变量的值。使用 Arguments 对象可以随时改变实参的值。例如，在下面这个示例中把循环变量的值传递给 Arguments 对象元素，以实现动态改变实参的值。

```
function f(){
    for(var i = 0; i < arguments.length; i ++ ){
        arguments[i] =i;
        alert(arguments[i]);
    }
}
f(3, 3, 6);          //提示 1、2、3，而不是 3、3、6
```

通过修改 Arguments 对象的 length 属性值，可以达到改变函数实参个数的目的。当 length 属性值增大时，增加的实参值为 undefined，当 length 属性值减小时，则会丢弃 arguments 数据集合后面对应个数的元素。

Arguments 在实际开发中具有重要的价值，使用它可以监测用户在调用函数时所传递的参数是否符合要求，增强函数的容错能力，同时还可以开发出很多功能强大的函数。

如果要定义的函数参数个数不确定，或者参数个数很多，又不想为每个参数都定义一个变量，此时定义函数可以保留参数列表为空，在函数内部使用 Arguments 对象来访问调用函数时传递的所有参数。下面这个示例就是利用 Arguments 对象来计算函数任意多个参数的平均值。

```
function avg(){
    var num = 0, l = 0;
    for(var i = 0; i < arguments.length; i ++ ){
        if(typeof arguments[i] != "number")
            continue;
        num += arguments[i];
        l ++ ;
    }
    num /= l;
    return num;
}
alert(avg(1, 2, 3, 4));          //2.5
alert(avg(1, 2, "3", 4));        //2.3333333333333335
```

表单验证是页面设计中经常要完成的任务，下面的示例验证所输入的值是否符合邮箱地址格式。

```
function isEmail(){
    if(arguments.length>1) throw new Error("只能够传递一个参数");
    var regexp = /^\w+((-\w+)|(\.\w+))*\@[A-Za-z0-9]+
((\.|-)[A-Za-z0-9]+)*\.[A-Za-z0-9]+$/;
    if (arguments[0].search(regexp)!= -1)
        return true;
    else
        return false;
```

```
}
var email = "abc@163.com";
alert(isEmail(email));    //true
```

Arguments 对象包含一个 callee 属性，它能够返回当前 Arguments 对象所属的函数引用，这相当于在函数体内调用函数自身。在匿名函数中，callee 属性比较有用，通过它在函数内部引用函数自身。

在下面这个示例中，通过 arguments.callee 获取对当前匿名函数的引用，然后通过函数的 length 属性确定它的形参个数。最后，通过实参和形参数目的比较来确定传递的参数是否合法。

```
function f(x, y, z){
    var a = arguments.length;
    var b = arguments.callee.length;
    if (a != b){
        throw new Error("传递的参数不匹配");
    }
    else{
        return x + y + z;
    }
}
alert(f(3, 4, 5));          //值为 12
```

Function 对象的 length 属性返回的是函数的形参个数，而 Arguments 对象的 length 属性返回的是函数的实参个数。如果函数不是匿名函数，则 arguments.callee 等价于函数名。

建议 59：推荐动态调用函数

调用函数更便捷的方式是使用 Function 对象的 call 和 apply 方法。apply 与 call 方法在本质上没有太大区别，只不过它们传递给函数的参数方式不同，apply 是以数组形式进行参数传递，而 call 方法可以同时传递多个值。

如果某个函数仅能够接收多个参数列表，而现在希望把一个数组的所有元素作为参数进行传递，那么使用 apply 方法就显得非常便利。

```
function max(){
    var m = Number.NEGATIVE_INFINITY;    // 声明一个负无穷大的数值
    for( var i = 0; i < arguments.length; i ++ ){
        if( arguments[i] > m )
        m = arguments[i];
    }
    return m;
}
var a = [23, 45, 2, 46, 62, 45, 56, 63];
var m = max.apply( Object, a );
alert( m );                                  //63
```

在上面的示例中，首先定义一个函数来计算所传递实参的最大值。由于该函数仅能够接收多个数值参数，所以通过 apply 方法动态调用 max() 函数，然后把它绑定为 Object 对象的一个方法，并借机把一个数组传递给它，最后返回此函数的运行值。如果没有 apply 方法，想使用 max() 函数来计算数组中最大元素值，就需要把数组的所有元素读取出来，然后再传递给函数，显然这种做法是费力不讨好的。

实际上，也可以把数组元素通过 apply 方法传递给系统对象 Math 的 max 方法来计算数组的最大元素值。

```
var a = [23, 45, 2, 46, 62, 45, 56, 63]; // 声明并初始化数组
var m = Math.max.apply( Object, a );     // 调用系统函数 max
alert( m );        //63
```

使用 call 和 apply 方法可以把一个函数转换为方法传递给某个对象。这种行为只是临时的，函数最终并没有作为对象的方法而存在，当函数被调用后，该对象方法会自动被注销。下面的示例具体地说明了这种行为。

```
function f(){}
f.call( Object );
Object.f();
```

call 和 apply 方法能够更改对象的内部指针，即改变对象的 this 指向的内容，这在面向对象的编程过程中是非常有用的。

```
var x = "o";
function a(){
    this.x = "a";
}
function b(){
    this.x = "b";
}
function c(){
    alert( x );
}
function f(){
    alert( this.x );
}
f();        // 字符 o，即全局变量 x 的值。this 此时指向 window 对象
f.call( window ); f.call( new a() );    // 字符 a，即函数 a 内部的局部变量 x 的值。this 此时指向函数 a
f.call( new b() );          // 字符 b，即函数 b 内部的局部变量 x 的值。this 此时指向函数 b
f.call( c );        /*undefined，即函数 c 内部的局部变量 x 的值，但是该函数并没有定义 x 变量，所以返回没有定义。this 此时指向函数 c*/
```

通过上面示例的比较，能够很直观地发现，函数 f 内部的 this 关键字会随着所绑定的对象不同而指向不同的对象。因此，利用 call 或 apply 方法能够改变函数内部指针指向所绑定的对象，从而实现属性或方法继承。

```
function f(){
```

```
    this.a ="a";
    this.b = function(){
        alert("b");
    }
}
function e(){
    f.call(this);
    alert(a);
}
e()                    //字符串 a
```

上面的示例说明，如果在函数体内使用 call 和 apply 方法动态调用外部函数，并把 call 和 apply 方法的第一个参数值设置为关键字 this，那么当前函数 e 将继承函数 f 的所有成员。使用 call 和 apply 方法能够复制调用函数的内部变量给当前函数体，更改了函数 f 的内部关键字 this 指向函数 e，这样函数 e 就可以引用函数 f 的内部成员。

最后，再看一个比较复杂的示例。在这个示例中将演示如何使用 apply 方法循环更改当前指针，从而实现循环更改函数的结构。

```
function r( x ){
    return ( x );
}
function f( x ){
    x[0] = x[0] + ">";
    return x;
}
function o(){
    var temp = r;
    r = function(){
        return temp.apply( this, f( arguments ) );
    }
}
function a(){
    o();
    alert( r( "=" ) );
}
for( var i = 0 ; i < 10; i ++ ){
    a();
}
```

执行上面代码后会看到，提示信息框中的提示信息不断变化。该示例的核心就在于函数 o 的设计。在这个函数中，首先使用一个临时变量存储函数 r。然后修改函数 r 的结构，在修改的函数 r 的结构中，通过调用 apply 方法修改原来函数 r 的指针指向当前对象，同时执行原函数 r，并把执行函数 f 的值传递给它，从而实现修改函数 r 的 return 语句的后半部分信息，即为返回值增加一个前缀字符“=”。这样每次调用函数 o 时，都会为其增加一个前缀字符“=”，从而形成一种动态的变化效果。

当然，call 和 apply 方法的应用是非常灵活的，在大型 JavaScript 技术框架中经常会用到它们，利用它们可以实现动态更改对象的指针，从而实现各种复杂的功能。

建议 60：比较函数调用模式

在 JavaScript 中，函数就是对象，对象是“名 / 值”对的集合，并拥有一个到原型对象的隐藏连接。对象字面量产生的对象连接到 object.prototype，函数对象连接到 Function. prototype，该原型对象本身连接到 object.prototype。每个函数在创建时都有两个附加的隐藏属性：函数的上下文和实现函数行为的代码。

每个函数对象在创建时也随之带一个 prototype 属性，它的值是一个拥有 constructor 属性且 constructor 属性值为该函数的对象。这和隐藏连接到 Function. prototype 完全不同。这个令人费解的构造过程的意义将会在后面的章节中揭示。

作为对象，函数与其他值一样可以在任何变量的位置使用。函数可以作为数组元素，可以作为函数的返回值，可以作为对象的成员值，也可以作为表达式参与运算，同时函数也可以拥有自己的方法。

调用一个函数将暂停当前函数的执行，传递控制权和参数给新函数。除了声明时定义的形式参数，每个函数接收两个附加的参数：this 和 arguments。参数 this 在面向对象编程中非常重要，它的值取决于调用的模式。在 JavaScript 中一共有 4 种调用模式：方法调用模式、函数调用模式、构造器调用模式和 apply 调用模式。这些模式在如何初始化关键参数 this 上存在差异。

调用运算符是跟在任何产生一个函数值的表达式之后的一对圆括号，圆括号内可以包含零个或多个用逗号隔开的表达式。每个表达式产生一个参数值。每个参数值被赋予函数声明时定义的形式参数名。当实际参数（arguments）的个数与形式参数（parameters）的个数不匹配时不会导致运行时错误。如果实际参数值过多，那么超出的参数值将被忽略。如果实际参数值过少，那么缺失的值将会被替换为 undefined。不会对参数值进行类型检查，任何类型的值都可以传递给参数。

（1）方法调用模式

当一个函数被保存为对象的一个属性时，将称为一个方法。当一个方法被调用时，this 被绑定到该对象。如果一个调用表达式包含一个属性存取表达式（即一个点表达式或 [subscript] 下标表达式），那么它被当做一个方法来调用。

```
var obj = {
    value : 0,
    increment : function(inc) {
        this.value += typeof inc === 'number' ? inc : 1;
    }
}
obj.increment();
document.writeln(obj.value);      // 1
obj.increment(2);
document.writeln(obj.value);      // 3
```

在上面代码中创建了 obj 对象，它有一个 value 属性和一个 increment 方法。increment 方法接受一个可选的参数，如果该参数不是数字，那么默认使用数字 1。

由于 increment 方法可以使用 this 去访问对象，所以它能从对象中取值或修改该对象。this 到对象的绑定发生在调用的时候。这个延迟绑定使函数可以对 this 高度复用。通过 this 可取得 increment 方法所属对象的上下文的方法称为公共方法。

（2）函数调用模式

当一个函数并非一个对象的属性时，它将被当做一个函数来调用：

```
var sum = add(3, 4);        //7
```

当函数以此模式调用时，this 被绑定到全局对象。这是语言设计上的一个错误。倘若语言设计正确，当内部函数被调用时，this 应该仍绑定到外部函数的 this 变量。这个设计错误的后果是方法不能利用内部函数来帮助它工作，因为内部函数的 this 被绑定了错误的值，所以不能共享该方法对对象的访问权。幸运的是，有一个很容易的解决方案：如果该方法定义一个变量并将它赋值为 this，那么内部函数就可以通过这个变量访问 this。按照约定，将这个变量命名为 that。

```
var obj = {
    value : 1,
    doub : function() {
        var that = this;
        var helper = function() {
            that.value = that.value * 2;
        };
        helper();
    }
}
obj.doub();
document.writeln(obj.value);      // 2
```

（3）构造器调用模式

JavaScript 是一门基于原型继承的语言，该语言是无类别的，对象可以直接从其他对象继承属性。当今大多数语言都是基于类的语言，虽然原型继承有着强大的表现力，但它偏离了主流用法，并不被广泛理解。JavaScript 为了能够兼容基于类语言的编写风格，提供了一套基于类似类语言的对象构建语法。

如果在一个函数前面加上 new 运算符来进行调用，那么将创建一个隐藏链接到该函数的 prototype 原型对象的新实例对象，同时 this 将会被绑定到这个新实例对象上。注意，new 前缀也会改变 return 语句的行为。

```
var F = function(string) {
    this.status = string;
};
F.prototype.get = function() {
    return this.status;
```

```
};
var f = new F("new object");
document.writeln(f.get());        //"new object"
```

上面代码创建一个名为 F 的构速器函数，此函数构建了一个带有 status 属性的对象。然后，为 F 所有实例提供一个名为 get 的公共方法。最后，创建一个实例对象，并调用 get 方法，以读取 status 属性的值。

结合 new 前缀调用的函数称为构造器函数。按照约定，构造器函数应该保存在以大写字母命名的变量中。如果调用构造器函数时没有在前面加上 new，可能会发生非常糟糕的事情，既没有编译时警告，也没有运行时警告，所以大写约定非常重要。

（4）apply 调用模式

JavaScript 是函数式的面向对象编程语言，函数可以拥有方法。apply 就是函数的一个基本方法，使用这个方法可以调用函数，并修改函数体内的 this 值。apply 方法包括两个参数：第一个参数设置绑定给 this 的值；第二个参数是包含函数参数的数组。例如：

```
var array = [5, 4];
var add = function() {
    var i, sum = 0;
    for( i = 0; i < arguments.length; i += 1) {
        sum += arguments[i];
    }
    return sum;
};
var sum = add.apply({}, array);                    // 9
```

上面代码构建一个包含两个数字的数组，然后使用 apply 方法调用 add() 函数，将数组 array 中的元素值相加。

```
var F = function(string) {
    this.status = string;
};
F.prototype.get = function() {
    return this.status;
};
var obj = {
    status: 'obj'
};
var status = F.prototype.get.apply(obj);          //''obj''
```

上面代码构建了一个构造函数 F，为该函数定义了一个原型方法 get，该方法能够读取当前对象的 status 属性的值。然后定义一个 obj 对象，该对象包含一个 status 属性，使用 apply 方法在 obj 对象上调用构造函数 F 的 get 方法，返回 obj 对象的 status 属性值。

建议 61：使用闭包跨域开发

闭包是指词法表示包括不必计算的变量的函数，闭包函数能够使用函数外定义的变量。

闭包结构有以下两个比较鲜明的特性。

（1）封闭性

外界无法访问闭包内部的数据，如果在闭包内声明变量，外界是无法访问的，除非闭包主动向外界提供访问接口。

（2）持久性

对于一般函数来说，在调用完毕之后，系统会自动注销函数，而对于闭包来说，在外部函数被调用之后，闭包结构依然保存在系统中，闭包中的数据依然存在，从而实现对数据的持久使用。例如：

```
function f( x ){
    var a = x;
    var b = function(){
        return x;
    }
    return b;
}
var c = f( 1 );
alert(c());                    //1。调用闭包函数
```

在上面示例中，首先在函数 f 结构体内定义两个变量，分别存储参数和闭包结构，而闭包结构中寄存着参数值。当调用函数 f 之后，函数结构被注销，它的局部变量也随之被注销，因此变量 a 中存储的参数值也随之丢失。但由于变量 b 存储着闭包结构，因此闭包结构内部的参数值并没有被释放，在调用函数之后，依然能够从闭包结构中读取到参数值。

从结构上分析，闭包函数与普通函数没有什么不同，主要包含以下类型的标识符：

- 函数参数（形参变量）。
- arguments 属性。
- 局部变量。
- 内部函数名。
- this（指代闭包函数自身）。

其中 this 和 arguments 是系统默认的函数标识符，不需要特别声明。这些标识符在闭包体内的优先级是（其中左侧优先级大于右侧）：this → 局部变量 → 形参 → arguments → 函数名。

下面以一个经典的闭包示例来演示上述抽象描述：

```
1 function f(x){                    // 外部函数
2     var a = x;                    // 外部函数的局部变量，并把参数值传递给它
3     var b = function(){           // 内部函数
4         return a;                 // 访问外部函数中的局部变量
5     };
6     a++;                          // 访问后，动态更新外部函数的变量
7     return b;                     // 内部函数
8 }
```

```
9 var c = f(5);    // 调用外部函数，并赋值
10 alert(c());     // 调用内部函数，返回外部函数更新后的值 6
```

演示步骤说明如下：

第 1 步，程序预编译之后，从第 9 行开始解析执行，创建上下文环境，创建调用对象，把参数、局部变量、内部的函数转换为对象属性。

第 2 步，执行函数体内代码。在第 6 行执行局部变量 a 的递加运算，并把这个值传递给对象属性 a，内部函数动态保持与局部变量 a 的联系，同时更新自己内部调用变量的值。

第 3 步，外部函数把内部函数返回给全局变量 c，实现内部函数的定义，此时 c 完全继承了内部函数的所有结构和数据。

第 4 步，外部函数返回后（即返回值后调用完毕）会自动销毁，内部的结构、标识符和数据也随之丢失。

第 5 步，执行第 10 行代码命令，调用内部函数，此时返回的是外部函数返回时（销毁之前）保存的变量 a 所存储的最新数据值，即返回 6。

如果没有闭包函数的作用，那么这种数据寄存和传递就无法得以实施。例如：

```
1 function f(x){
2     var a = x;
3     var b = a;          // 直接把局部变量的值传递给局部变量 b
4     a++
5     return b;           //局部变量 b
6 }
7 var c = f(5);
8 alert(c);               // 值为 5
```

通过上面的示例可以很直观地看到，在没有闭包函数的辅助下，第 8 行代码执行后返回值并没有与外部函数的局部变量 a 最后更新的值保持一致。

闭包在程序开发中具有重要的价值。例如，使用闭包结构能够跟踪动态环境中数据的实时变化，并即时存储。

```
function f(){
    var a = 1;
    var b = function(){
        return a;
    }
    a++;
    return b;
}
var c = f();
alert(c());        //返回 2，而不是 1
```

在上面示例中，闭包中的变量 a 存储的值并不是对上面行变量 a 的值的简单复制，而是继续引用外部函数定义的局部变量 a 中的值，直到外部函数 f 调用返回。闭包不会因为外部函数环境的注销而消失，会始终存在。例如：

```
<script language="javascript" type="text/javascript">
function f(){
    var a = 1;
    b = function(){
        alert( a );
    }
    c = function(){
        a ++ ;
    }
    d = function( x ){
        a = x;
    }
}
</script>
<button onclick="f()">按钮 1：(f(    ))()</button><br />
<button onclick="b()">按钮 2：(b = function(){alert( a );})()</button><br />
<button onclick="c()">按钮 3：(c = function(){a ++ ;})()</button><br />
<button onclick="d(100)">按钮 4：(d = function( x ){a = x; }) (100)</button><br />
```

在上面示例中，在函数 f 中定义了 3 个闭包函数，它们分别指向并寄存局部变量 a 的值，并根据不同的操作动态跟踪变量 a 的值。当在浏览器中预览时，首先应该单击“按钮 1”，调用函数 f，生成 3 个闭包，3 个闭包同时指向局部变量 a 的引用，因此，当函数 f 返回时，这 3 个闭包函数都没有被注销，变量 a 由于被闭包引用而继续存在。这时，如果直接单击“按钮 2”和“按钮 4”，那么会由于没有在系统中生成闭包结构，而弹出编译错误。单击“按钮 3”将动态递增变量 a 的值，此时如果单击“按钮 2”，则会弹出提示值 2。如果单击“按钮 4”，则向变量 a 传递值 100，将动态改变闭包中寄存的值，此时如果单击“按钮 2”，则会弹出提示值 100。

建议 62：在循环体和异步回调中慎重使用闭包

闭包在开发中具有重要的应用价值，由于闭包具有持久性，生成的闭包不会立即被销毁，因此它会持续占用系统资源。如果大量使用闭包，将会造成系统资源紧张，甚至导致内存溢出等错误。另外，闭包在回调函数中会带来负面影响，因此在使用时应该慎重。

下面的示例利用闭包来存储变量所有变化的值。

```
function f( x ){
    var a = [];
    for ( var i = 0; i < x.length; i ++ ){
        var temp = x[i];
        a.push( function(){
            alert( temp + ' ' + x[i] )
        });
    }
    return a;
}
function e(){
```

```
    var a = f( ["a", "b", "c"] );
    for ( var i = 0; i < a.length; i ++ ){
        a[i]();
    }
}
e();        // 调用函数 e
```

在这个示例中，函数 f 的功能是：把数组类型的参数中每个元素的值分别封装在闭包结构中，然后把闭包存储在一个数组中，并返回这个数组。但是，在函数 e 中调用函数 f，并向其传递一个数组（["a", "b", "c"]），然后遍历函数 f 返回数组，此时会发现，数组中每个元素的值都是“c undefined”。

原来闭包中的变量 temp 并不是固定的，它会随时根据函数运行环境中的变量 temp 的值变化而更新，这样会导致临时数组元素的值都是字符“c”，而不是“a”、“b”、“c”，同时，由于循环变量 i 递增之后，最后的值是 3，x[3] 超出了数组的长度，所以结果就是 undefined。

解决闭包存在缺陷问题的方法是：为闭包再包裹一层函数，然后运行该函数，并把外界动态值传递给它，这个函数接收这些值后传递给内部的闭包函数，从而阻断了闭包与最外层函数的实时联系。

```
function f( x ){
    var a = [];
    for ( var i = 0; i < x.length; i ++ ){
        var temp = x[i];
        a.push(
            ( function( temp, i ){
                return function(){
                    alert( temp + ' ' + x[i] )
                }
            })( temp, i )
        );
    }
    return a;
}
function e(){
    var a = f( ["a", "b", "c"] );
    for ( var i = 0; i < a.length; i ++ ){
        a[i]();
    }
}
e();
```

同一个闭包通过分别引用能够在当前环境中生成多个闭包。例如：

```
function f( x ){
    var temp = x;
    return function( x ){
        temp += x;
        alert( temp );
    }
}
```

```
var a = f( 50 )
var b = f( 100 )
a( 5 )     //55
b( 10 )    //110
```

建议 63：比较函数调用和引用本质

在被调用之前，JavaScript 函数仅是词法意思上的结构，没有实际的价值，在预编译函数时，也仅是简单地分析函数的词法、语法结构，并根据函数标识符预定一个函数占据的内存空间，其内部结构和逻辑并没有被运行。但是，一旦函数被调用执行，其上下文环境也会随之产生。可以说，上下文环境是函数运行期的一个动态环境，它是一个动态概念，与函数的静态性是截然不同的概念。每个函数都有一个独立的上下文环境（即执行环境）。先看下面这个示例中变量 a 和变量 b 是否相等。

```
function f(){
    var x = 5;
    return x;
}
var a = f;
var b = f;
alert( a === b );
```

“alert(a === b);”的返回值为 true，说明变量 a 与变量 b 完全相同。继续观察下面的示例。

```
function f(){
    var x = 5;
    return function(){
        return x;
    }
}
var a = f();
var b = f();
alert( a === b );
```

“alert(a === b);”的返回值为 false，说明变量 a 与变量 b 不完全相同。这就是函数引用和函数调用的区别，下面的示意图能够很好地说明它们的异同，分别如图 3.1 和图 3.2 所示。

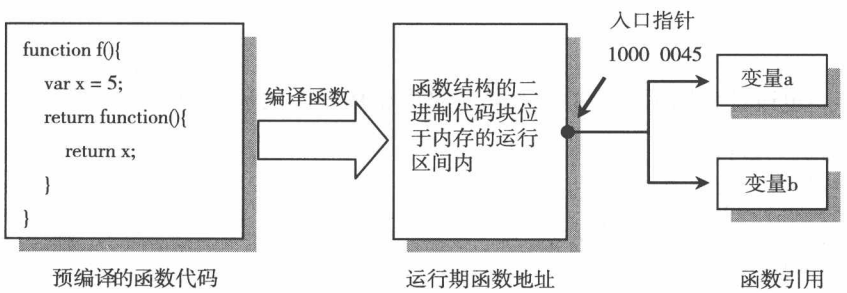


图 3.1　函数引用示意图

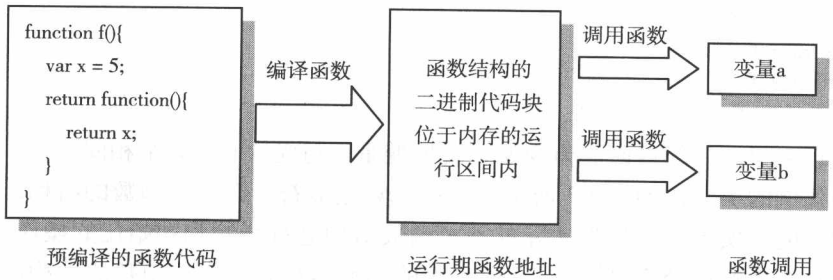


图 3.2　函数调用示意图

下面介绍函数引用与函数调用的本质区别。当引用函数时，多个变量内存储的是函数的相同入口指针。因此，对于同一个函数来说，不管有多少个变量引用该函数，这些变量的值都是相同的，都为该函数的入口指针地址。例如，针对上面第二个示例来说，如果变量 a 和 b 都引用函数 f，而不是调用，则它们是完全相同的。

相反，函数调用是执行该函数，并把返回的值传递给变量 a 和 b。也就是说，变量 a 和 b 存储的是值，而不是函数的入口指针地址。

有这么一个问题：在第一个示例中，如果变量 a 和 b 也都调用函数 f，它们却相同（如下所示），而在下面这个示例中的调用却不相同。

```
function f(){
    var x = 5;
    return x;
}
var a = f();
var b = f();
alert( a === b );          //true
```

在第一个示例中，函数调用后返回的是值类型数据（即数值 5），两个数值 5 自然是相同的。但是在第二个示例中，函数调用返回值是一个闭包函数，即引用类型的数据。虽然返回的闭包结构是完全相同的，但由于它们存储在不同变量中，即它们的地址指针是完全不同的，因此也无法相同。再看下面这个示例：

```
function F(){
    this.x = 5;
}
var a = new F();
var b = new F();
alert( a === b );
```

“alert(a === b);”的返回值为 false，说明变量 a 与变量 b 不完全相同。

```
function F(){
    this.x = function(){
        return 5;
    }
```

```
}
var a = new F();
var b = new F();
alert( a === b );
```

“alert(a === b);”的返回值为 false，说明变量 a 与变量 b 不完全相同。

函数与实例的关系，如图 3.3 所示。通过 new 运算符可以复制函数的结构，从而实现函数实例化的目的。实例化的过程实际就是对函数结构进行复制和初始化的操作过程。因此，当实例化函数 F 并赋值给变量 a 时，a 所引用的函数结构并非原来函数的结构，而是内存运行区中另一块函数结构，自然它们是完全不相同的。

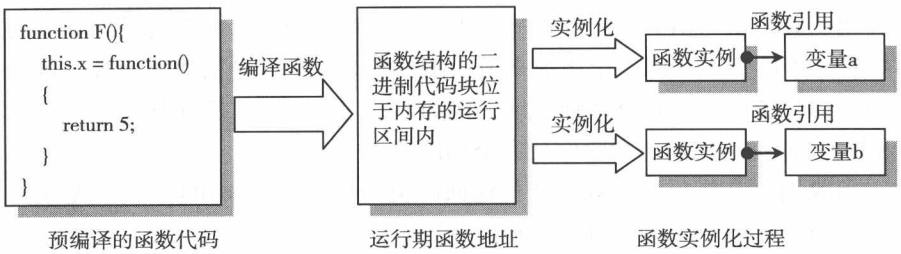


图 3.3 函数实例化过程示意图

建议 64：建议通过 Function 扩展类型

JavaScript 允许为语言的基本数据类型定义方法。通过为 Object.prototype 添加原型方法，该方法可被所有的对象使用。这样的方式对函数、数组、字符串、数字、正则表达式和布尔值都适用。例如，通过给 Function. prototype 增加方法，使该方法对所有函数可用。

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

为 Function.prototype 增加一个 method 方法后，就不必使用 prototype 这个属性了，然后调用 method 方法直接为各种基本类型添加方法。

JavaScript 并没有单独的整数类型，因此有时候只提取数字中的整数部分是必要的。JavaScript 本身提供的取整方法有些不太好，下面通过为 Number. prototype 添加一个 integer 方法来改善它。

```
Number.method('integer', function() {
    return Math[this < 0 ? 'ceil' : 'floor'](this);
});
document.writeln((-10 / 3).integer());   //-3
```

Number.method 方法能够根据数字的正负来判断是使用 Math.ceiling 还是 Math.floors，

这样就避免了每次都编写上面的代码。

```
String.method('trim', function() {
    return this.replace(/^\s+|\s+$/g, '');
});
document.writeln('"' + " abc ".trim() + '"');    // 'abc''
```

trim 方法使用了一个正则表达式，把字符串中左右两侧的空格符清除掉。

通过为基本类型扩展方法，可以大大提高语言的表现力。由于 JavaScript 原型继承的本质，因此，所有原型方法立刻被赋予到所有的实例，即使该实例在原型方法创建之前就创建好了。

注意：基本类型的原型是公共结构，在扩展基类时务必小心，避免覆盖掉基类的原生方法。一个稳妥的做法就是在确定没有该方法时才添加它。

```
Function.prototype.method = function(name, func) {
    if(!this.prototype[name]) {
        this.prototype[name] = func;
        return this;
    }
};
```

另外，for in 语句用在原型上时表现很糟糕。可以使用 hasOwnProperty 方法筛选出继承而来的属性，或者查找特定的类型。

建议 65：比较函数的惰性求值与非惰性求值

在 JavaScript 中，使用函数式风格编程时，应该对于表达式有着深刻的理解，并能够主动使用表达式的连续运算来组织代码。

1）在运算元中，除了 JavaScript 默认的数据类型外，函数也作为一个重要的运算元参与运算。

2）在运算符中，除了 JavaScript 的大量预定义运算符外，函数还作为一个重要的运算符进行计算和组织代码。

函数作为运算符参与运算，具有非惰性求值特性。非惰性求值行为自然会对整个程序产生一定的负面影响。先看下面这个示例：

```
var a = 2;
function f(x){
    return x;
}
alert(f(a,a=a*a));          //2
alert(f(a));                //4
```

在上面的示例中，两次调用同一个函数并传递同一个变量，所返回的值却不一样。在第一次调用函数时，向其传递了两个参数，第二个参数是一个表达式，该表达式对变量 a 进行重新计算和赋值。也就是说，当调用函数时，第二个参数虽然不使用，但是也被计算了。这就是 JavaScript 的非惰性求值特性，也就是说，不管表达式是否被利用，只要在执行代码行中都会被计算。

如果在一个函数参数中无意添加了几个表达式，虽然这样不会对函数的运算结果产生影响，但是由于表达式被执行，就会对整个程序产生潜在的负面影响。

在惰性求值语言中，如果参数不被调用，那么无论参数是直接量还是某个表达式，都不会占用系统资源。但是，由于 JavaScript 支持非惰性求值，问题就变得很特殊了。

```
function f(){}
f( function(){while(true);}())
```

在上面的示例中，虽然函数 f 没有参数，但是在调用时将会执行传递给它的参数表达式，该表达式是一个死循环结构的函数值，最终将导致系统崩溃。

惰性函数模式是一种将对函数或请求的处理延迟到真正需要结果时进行的通用概念，很多应用程序都采用了这种概念。从惰性编程的角度来思考问题，可以帮助消除代码中不必要的计算。例如，在 Scheme 语言中，delay 特殊表单接收一个代码块，它不会立即执行这个代码块，而是将代码和参数作为一个 promise 存储起来。如果需要 promise 产生一个值，就会运行这段代码。promise 随后会保存结果，这样将来再请求这个值时，该值就可以立即返回，而不用再次执行代码。这种设计模式在 JavaScript 中大有用处，尤其是在编写跨浏览器的、高效运行的库时非常有用。例如，下面是一个时间对象实例化的函数。

```
var t;
function f(){
    t = t ? t : new Date();
    return t;
}
f();        // 调用函数
```

上面的示例使用全局变量 t 来存储时间对象，这样在每次调用函数时都必须进行重新求值，代码的效率没有得到优化，同时全局变量 t 很容易被所有代码访问和操作，存在安全隐患。当然，可以使用闭包隐藏全局变量 t，只允许在函数 f 内访问。

```
var f =(function(){
    var t;
    return function(){
        t = t ? t : new Date();
        return t;
    }
})();
f();
```

这仍然没有提高调用时的效率，因为每次调用 f 依然需要求值：

```
var f = function() {
    var t = new Date();
    f = function() {
        return t;
    }
    return f();
};
f();
```

在上面的示例中，函数 f 的首次调用将实例化一个新的 Date 对象并重置 f 到一个新的函数上，f 在其闭包内包含 Date 对象。在首次调用结束之前，f 的新函数值也已被调用并提供返回值。

函数 f 的调用都只会简单地返回 t 保留在其闭包内的值，这样执行起来非常高效。弄清这种模式的另一种途径是，外部函数 f 的首次调用是一个保证（promise），它保证了首次调用会重定义 f 为一个非常有用的函数，保证来自于 Scheme 的惰性求值机制。

建议 66：使用函数实现历史记录

函数可以利用对象去记住先前操作的结果，从而能避免无谓的运算，这种优化称为记忆。JavaScript 的对象和数组要实现这种优化是非常方便的。

例如，使用递归函数计算 fibonacci 数列。一个 fibonacci 数字是之前两个 fibonacci 数字之和。最前面的两个数字是 0 和 1。

```
var fibonacci = function(n) {
    return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);
};
for(var i = 0; i <= 10; i += 1) {
    document.writeln('<br>' + i + ': ' + fibonacci(i));
}
```

返回下列值：

```
0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
7: 13
8: 21
9: 34
10: 55
```

在上面代码中，fibonacci 函数被调用了 453 次，其中循环调用了 11 次，它自身调用了 442 次，去计算可能已刚计算过的值。如果使该函数具备记忆功能，就可以显著减少它的运

算次数。

先使用一个临时数组保存存储结果，存储结果可以隐藏在闭包中。当函数被调用时，先看是否已经知道存储结果，如果已经知道，就立即返回这个存储结果。

```
var fibonacci = ( function() {
    var memo = [0, 1];
    var fib = function(n) {
        var result = memo[n];
        if( typeof result !== 'number') {
            result = fib(n - 1) + fib(n - 2);
            memo[n] = result;
        }
        return result;
    };
    return fib;
}());
for(var i = 0; i <= 10; i += 1) {
    document.writeln('<br>' + i + ':' + fibonacci(i));
}
```

这个函数返回同样的结果，但它只被调用了 29 次，其中循环调用了 11 次，它自身调用了 18 次，去取得之前存储的结果。当然，可以把这种函数形式抽象化，以构造带记忆功能的函数。memoizer 函数将取得一个初始的 memo 数组和 fundamental 函数。memoizer 函数返回一个管理 memo 存储和在需要时调用 fundamental 函数的 shell 函数。memoizer 函数传递这个 shell 函数和该函数的参数给 fundamental 函数。

```
var memoizer = function(memo, formula) {
    var recur = function(n) {
        var result = memo[n];
        if( typeof result !== 'number') {
            result = formula(recur, n);
            memo[n] = result;
        }
        return result;
    };
    return recur;
};
```

现在，就可以使用 memoizer 来定义 fundamental 函数，提供初始的 memo 数组和 fundamental 函数。

```
var fibonacci = memoizer([0, 1], function(recur, n) {
    return recur(n - 1) + recur(n - 2);
});
```

通过设计能产生其他函数的函数，可以极大地减少一些不必要的工作。例如，要产生一个可记忆的阶乘函数，只需提供基本的阶乘公式即可。

```
var factorial = memoizer([1, 1], function(recur, n) {
```

```
    return n * recur(n - 1);
});
```

建议 67：套用函数

套用是 JavaScript 函数一个很有趣的应用。所谓套用就是将函数与传递给它的参数相结合，产生一个新的函数。在函数式编程中，函数本身也是一个值，这种特性允许用户以有趣的方式去操作函数值。例如，在下面代码中定义一个 add() 函数，该函数能够返回一个新的函数，并把参数值传递给这个新函数，从而实现连加操作。

```
var add = function(n){
    return function(m){
        return n+m;
    }
}
document.writeln(add(2)(3));      //5
```

当然，也可以为 JavaScript 扩展一个 curry 方法，实现函数的套用应用。

```
Function.prototype.method = function(name, func) {
    if(!this.prototype[name]) {
        this.prototype[name] = func;
        return this;
    }
};
Function.method('curry', function() {
    var slice = Array.prototype.slice;
    var args = slice.apply(arguments), that = this;
    return function() {
        return that.apply(null, args.concat(slice.apply(arguments)));
    };
});
```

curry 方法通过创建一个保存原始函数和被套用函数的参数的闭包来工作。该方法返回另一个函数，该函数被调用时会返回调用原始函数的结果，并传递调用 curry 时的参数加上当前调用的参数的所有参数。curry 使用 Array 的 concat 方法连接两个参数数组。但由于 arguments 数组并非一个真正的数组，所以它并没有 concat 方法，要避开这个问题，必须在两个 arguments 数组上都应用数组的 slice 方法，这样才会产生出拥有 concat 方法的常规数组。

下面就来应用 curry 方法，通过 curry 方法调用 add 函数，会返回一个新的函数 add1，在这个新的返回函数中保存了调用 add 函数时传递的值，当调用 add1 函数时，将新旧函数的参数进行相加，返回 7。

```
var add = function() {
    var i, sum = 0;
    for( i = 0; i < arguments.length; i += 1) {
        sum += arguments[i];
```

```
    }
    return sum;
};
var add1 = add.curry(2);
document.writeln(add1(3));        // 7
```

建议 68：推荐使用链式语法

使用过 jQuery 框架的读者，都会对 jQuery 简洁的语法、灵巧的用法赞叹不已，其中一个最大亮点就是 jQuery 的链式语法。在 JavaScript 中，很多方法没有返回值，一些设置或修改对象的某个状态却不返回任何值的方法就是典型的例子。如果让这些方法返回 this，而不是 undefined，那么就要启用级联功能，即所谓的链式语法。在一个级联中，单独一条语句可以连续调用同一个对象的很多方法。

```
getElement('box').
    move(350, 150).
    width(100).height(100).
    color('red').
    border('10px outset').
    padding('4px').
    appendText(" 使用链式语法 ")
```

在上面代码中，getElement 函数获取 id='box' 的 DOM 元素，然后通过链式语法分别调用 DOM 元素的扩展方法来移动元素、修改尺寸和样式，以及添加行为。由于每一个扩展方法都返回参数对象，所以调用返回的结果可以为下一次调用所用。链式语法可以产生具备很强表现力的接口，以产生出试图一次做很多事情的接口的趋势。

在下面示例中，分别为 String 扩展了 3 个方法：trim、writeln 和 alert，其中 writeln 和 alert 方法返回值都为 this，而 trim 方法返回值为修剪后的字符串。这样用户就可以利用链式语法在一行语句中快速调用这 3 个方法。

```
Function.prototype.method = function(name, func) {
    if(!this.prototype[name]) {
        this.prototype[name] = func;
        return this;
    }
};
String.method('trim', function() {
    return this.replace(/^\s+|\s+$/g, '');
});
String.method('writeln', function() {
    document.writeln(this);
    return this;
});
String.method('alert', function() {
    window.alert(this);
    return this;
```

```
});
var str = " abc  ";
str.trim().writeln().alert();
```

建议 69：使用模块化规避缺陷

使用函数和闭包可以构建模块。所谓模块，就是一个提供接口却隐藏状态与实现的函数或对象。通过使用函数构建模块，可以完全摒弃全局变量的使用，从而规避 JavaScript 语言缺陷。全局变量是 JavaScript 最为糟糕的特性之一，在一个大中型 Web 应用中，全局变量可能会带来不可预料的后果。

例如，要为 String 扩展一个 deentityify 方法，其设计任务是寻找字符串中的 HTML 字符实体并将其替换为对应的字符。在一个对象中保存字符实体的名字及与之对应的字符是有意义的。

可以把 deentityify 放到一个全局变量中，但全局变量存在很多潜在危害。可以把 deentityify 定义在该函数本身中，但会带来运行时的损耗，因为在该函数每次被执行时，这个方法都会被求值一次。理想的方式是将 deentityify 放入一个闭包，而且也许还能提供一个增加更多字符实体的扩展方法。

```
String.method('deentityify', function() {
    var entity = {
        quot : '"',
        lt : '<',
        gt : '>'
    };
    return function() {
        return this.replace(/&([^&;]+);/g, function(a, b) {
            var r = entity[b];
            return typeof r === 'string' ? r : a;
        });
    };
}());
```

在上面代码中，为 String 类型扩展了一个 deentityify 方法，它调用字符串的 replace 方法来查找以“&”开头和以“;”结束的子字符串。如果这些字符可以在字符实体表 entity 中找到，那么就将该字符实体替换为映射表中的值。deentityify 方法用到了一个正则表达式：

```
return this.replace(/&([^&;]+);/g, function(a, b) {
    var r = entity[b];
    return typeof r === 'string' ? r : a;
});
```

在最后一行使用 () 运算符立刻调用刚刚构造出来的函数。这个调用所创建并返回的函数才是 deentityify 方法。

```
document.writeln('&lt;&quot;&gt;'.deentityify());        // <">
```

模块利用了函数作用域和闭包来创建绑定对象与私有成员的关联。在这个示例中，只有 deentityify 方法才有权访问字符实体表 entity 这个数据对象。模块开发的一般形式是：一个定义了私有变量和函数的函数，利用闭包创建可以访问到的私有变量和函数的特权函数，最后返回这个特权函数，或者把它们保存到可访问的地方。

使用模块可以避免全局变量的滥用，从而保护信息的安全性，实现优秀的设计实践。使用这种模式也可以实现应用程序的封装，或者构建其他实例对象。

模块模式通常结合实例模式使用。JavaScript 的实例就是用对象字面量表示法创建的，对象的属性值可以是数值或函数，并且属性值在该对象的生命周期中不会发生变化。模块通常作为工具为程序其他部分提供功能支持。通过这种方式能够构建比较安全的对象。

下面代码构造一个用来产生序列号的对象。serial_maker() 函数将返回一个用来产生唯一字符串的对象，这个字符串由两部分组成：字符前缀 + 序列号。这两部分可以分别使用 set_prefix 和 set_seq 方法进行设置，然后调用实例对象的 gensym 方法读取这个字符串。当执行该方法时，都会自动产生唯一一个字符串。

```
var serial_maker = function() {
    var prefix = '';
    var seq = 0;
    return {
        set_prefix : function(p) {
            prefix = String(p);
        },
        set_seq : function(s) {
            seq = s;
        },
        gensym : function() {
            var result = prefix + seq;
            seq += 1;
            return result;
        }
    };
};
var seqer = serial_maker();
seqer.set_prefix('Q');
seqer.set_seq(1000);
var unique = seqer.gensym();      //"Q1000"
var unique = seqer.gensym();      //"Q1001"
```

seqer 包含的方法都没有用到 this 或 that，因此没有办法“损害”seger，除非调用对应的方法，否则无法改变 prefix 或 seq 的值。由于 seqer 对象是可变的，所以它的方法可能会被替换掉，但替换后的方法依然不能访问私有成员。seqer 就是一组函数的集合，而且这些函数被授予特权，拥有使用或修改私有状态的能力。如果把 seqer.gensym 作为一个值传递给第三方函数，这个函数就能通过它产生唯一字符串，却不能通过它来改变 prefix 或 seq 的值。

建议 70：惰性实例化

惰性实例化要解决的问题是：避免了在页面中 JavaScript 初始化执行的时候就实例化类，如果在页面中没有使用这个实例化的对象，就会造成一定的内存浪费和性能消耗。如果将一些类的实例化推迟到需要使用它的时候才去做，就可以避免资源过早损耗，做到“按需供应”。

```
var myNamespace = function() {
    var Configure = function() {
        var privateName = "someone's name";
        var privateReturnName = function() {
            return privateName;
        }
        var privateSetName = function(name) {
            privateName = name;
        }
        //返回单例对象
        return {
            setName : function(name) {
                privateSetName(name);
            },
            getName : function() {
                return privateReturnName();
            }
        }
    }
    // 存储 configure 实例
    var instance;
    return {
        getInstance : function() {
            if(!instance) {
                instance = Configure();
            }
            return instance;
        }
    }
}();
// 使用方法上需要 getInstance 这个函数作为中间量
myNamespace.getInstance().getName();
```

上面就是简单的惰性实例化的示例，其中有一个缺点就是需要使用中间量来调用内部的 Configure 函数所返回的对象的方法，当然也可以使用变量来存储 myNamespace.getInstance() 返回的实例对象。将上面的代码稍微修改一下，就可以用比较直观的方法来使用内部的方法和属性。

```
// 惰性实例化的变体
var myNamespace2 = function() {
    var Configure = function() {
        var privateName = "someone's name";
        var privateReturnName = function() {
```

```
            return privateName;
        }
        var privateSetName = function(name) {
            privateName = name;
        }
        //返回单例对象
        return {
            setName : function(name) {
                privateSetName(name);
            },
            getName : function() {
                return privateReturnName();
            }
        }
    }
    //存储 configure 实例
    var instance;
    return {
        init : function() {
            //如果不存在实例，就创建单例实例
            if(!instance) {
                instance = Configure();
            }
            //创建 Configure 单例
            for(var key in instance) {
                if(instance.hasOwnProperty(key)) {
                    this[key] = instance[key];
                }
            }
            this.init = null;
            return this;
        }
    }
}();
//使用方式
myNamespace2.init();
myNamespace2.getName();
```

在上面代码中修改了自执行函数返回的对象的代码，在获取 Configure 函数返回的对象时，将该对象的方法赋给 myNamespace2，这样调用方式就发生了一点改变。

建议 71：推荐分支函数

分支函数解决的一个问题是浏览器之间兼容性的重复判断。解决浏览器之间的兼容性的一般方式是使用 if 逻辑来进行特性检测或能力检测，根据浏览器不同的实现来实现功能上的兼容，这样做的问题是，每执行一次代码，可能都需要进行一次浏览器兼容性方面的检测，这是没有必要的。能否在代码初始化执行的时候就检测浏览器的兼容性，在之后的代码执行过程中，就无须再进行检测了呢？

答案是：能。分支技术就可以解决这个问题，下面以声明一个 XMLHttpRequest 实例对象为例进行介绍。

```
var XHR = function() {
    var standard = {
        createXHR : function() {
            return new XMLHttpRequest();
        }
    }
    var newActionXObject = {
        createXHR : function() {
            return new ActionXObject("Msxml2.XMLHTTP");
        }
    }
    var oldActionXObject = {
        createXHR : function() {
            return new ActionXObject("Microsoft.XMLHTTP");
        }
    }
    if(standard.createXHR()) {
        return standard;
    } else {
        try {
            newActionXObject.createXHR();
            return newActionXObject;
        } catch(o) {
            oldActionXObject.createXHR();
            return oldActionXObject;
        }
    }
}();
```

从上面的例子可以看出，分支的原理就是：声明几个不同名称的对象，但为这些对象声明一个名称相同的方法（这是关键）。这些不同的对象，却拥有相同的方法，根据不同的浏览器设计各自的实现，接着开始进行一次浏览器检测，并且由浏览器检测的结果来决定返回哪一个对象，这样不论返回的是哪一个对象，最后名称相同的方法都作为对外一致的接口。

这是在 JavaScript 运行期间进行的动态检测，将检测的结果返回赋值给其他的对象，并且提供相同的接口，这样存储的对象就可以使用名称相同的接口了。其实，惰性载入函数跟分支函数在原理上是非常相近的，只是在代码实现方面有差异。

建议 72：惰性载入函数

惰性载入函数主要解决的问题也是兼容性，原理跟分支函数类似，下面是简单的示例。

```
var addEvent = function(el, type, handle) {
    addEvent = el.addEventListener ? function(el, type, handle) {
        el.addEventListener(type, handle, false);
```

```
        } : function(el, type, handle) {
            el.attachEvent("on" + type, handle);
        };
        // 在第一次执行 addEvent 函数时，修改了 addEvent 函数之后，必须执行一次
        addEvent(el, type, handle);
    }
```

从代码上看，惰性载入函数也是在函数内部改变自身的一种方式，这样在重复执行的时候就不会再进行兼容性方面的检测了。

惰性载入表示函数执行的分支仅会发生一次，即第一次调用的时候。在第一次调用的过程中，该函数会被覆盖为另一个按合适方式执行的函数，这样任何对原函数的调用都不用再经过执行的分支了。其优点如下：

- 要执行的适当代码只有在实际调用函数时才执行。
- 尽管第一次调用该函数会因额外的第二个函数调用而稍微慢点，但后续的调用都会很快，因为避免了多重条件。

由于浏览器之间的行为差异，多数 JavaScript 代码包含了大量的 if 语句，将执行引导到正确的代码中。

在下面惰性载入的 createXHR() 中，if 语句的每个分支都会为 createXHR() 变量赋值，有效覆盖了原有的函数，最后一步便是调用新赋函数。下次调用 createXHR() 的时候，就会直接调用被分配的函数，这样就不用再次执行 if 语句。

```
    function createXHR() {
        if( typeof XMLHttpRequest != 'undefined') {
            return new XMLHttpRequest();
        } else if( typeof ActiveXObject != 'undefined') {
            if( typeof arguments.callee.activeXString != 'string') { ver
                    versions = ["MSXML2.XMLHttp", "MSXML2.XMLHttp.3.0", "MSXML2.
XMLHttp.6.0"];
                for(var i = 0, len = versions.length; i < len; i++) {
                    try {
                        var xhr = new ActiveXObject(versions[i]);
                        arguments.callee.activeXString = versions[i];
                        return xhr;
                    } catch(ex) {
                        // 跳过
                    }
                }
            }
            return new ActiveXObject(arguments.callee.activeXString);
        } else {
            throw new Error("No XHR object available.");
        }
    }
```

每一次调用 createXHR() 时都要对浏览器所支持的功能仔细检查，这样每次调用 createXHR() 时都要进行相同的测试就变得没有必要了。减少 if 语句使其不必每一次都执行，

代码就会执行得快些。解决方案就是惰性载入的技巧。

```
function createXHR() {
    if( typeof XMLHttpRequest != 'undefined') {
        createXHR = function() {
            return new XMLHttpRequest();
        };
    } else if( typeof ActiveXObject != 'undefined') {
        createXHR = function() {
            if( typeof arguments.callee.activeXString != 'string') { ver
                versions=["MSXML2.XMLHttp","MSXML2.XMLHttp.3.0","MSXML2.XMLHttp.6.0"];
                for(var i = 0, len = versions.length; i < len; i++) {
                    try {
                        var xhr = new ActiveXObject(versions[i]);
                        arguments.callee.activeXString = versions[i];
                        return xhr;
                    } catch(ex) {
                        //跳过
                    }
                }
            }
            return new ActiveXObject(arguments.callee.activeXString);
        };
    } else {
        createXHR = function() {
            throw new Error("No XHR object available.");
        };
    }
    return createXHR();
}
```

如前面所述，if 语句的每一个分支都会为 createXHR 变量赋值，有效覆盖了原有函数。最后一步便是调用新赋函数，下次调用 creatXHR() 的时候就会直接调用被分配的函数，这样就不用再次执行 if 语句。

建议 73：函数绑定有价值

函数绑定就是为了纠正函数的执行上下文，特别是当函数中带有 this 关键字的时候，这一点尤其重要，稍微不小心，就会使函数的执行上下文发生跟预期不同的改变，导致代码执行上的错误。函数绑定具有 3 个特征：

- 函数绑定要创建一个函数，可以在特定环境中以指定参数调用另一个函数。
- 一个简单的 bind() 函数接收一个函数和一个环境，返回一个在给定环境中调用给定函数的函数，并且将所有参数原封不动地传递过去。
- 被绑定函数与普通函数相比有更多的开销，它们需要更多内存，同时也因为多重函数调用而稍微慢一点，最好只在必要时使用。

第一个特征常常和回调函数及事件处理函数一起使用。

```
var handler = {
    message : 'Event handled',
    handleClick : function(event) {
        alert(this.message);
    }
};
var btn = document.getElementById('my-btn');
EventUtil.addHandler(btn, 'click', handler.handleClick);     //undefined
```

出现上述结果的原因在于没有保存 handler.handleClick() 环境（上下文环境），所以 this 对象最后指向了 DOM 按钮而非 handler。可以使用闭包修正此问题：

```
var handler = {
    message : 'Event handled',
    handleClick : function(event) {
        alert(this.message);
    }
};
var btn = document.getElementById('my-btn');
EventUtil.addHandler(btn, "click", function(event) {
    handler.handleClick(event);
});
```

这是特定于这段代码的解决方案。创建多个闭包可能会令代码变得难于理解和调试，因此，很多 JavaScript 库实现了一个可以将函数绑定到指定环境的函数 bind()。

bind() 函数的功能是提供一个可选的执行上下文传递给函数，并且在 bind() 函数内部返回一个函数，以纠正在函数调用上出现的执行上下文发生的变化。最容易出现的错误就是回调函数和事件处理程序一起使用。

```
function bind(fn, context) {
    return function() {
        return fn.apply(context, arguments);
    };
}
```

在 bind() 中创建一个闭包，该闭包使用 apply 调用传入的参数，并为 apply 传递 context 对象和参数。

注意：这里使用的 arguments 对象是内部函数的，而非 bind() 的。在调用返回的函数时，会在给定的环境中执行被传入的函数并给出所有参数。

```
var handler = {
    message : 'Event handled',
    handleClick : function(event) {
        alert(this.message);
    }
```

```
};
var btn = document.getElementById('my-btn');
EventUtil.addHandler(btn, "click", bind(handler.handlerClick, handler));
```

建议 74：使用高阶函数

高阶函数作为函数式编程众多风格中的一项显著特征，经常被使用。实际上，高阶函数即对函数的进一步抽象。高阶函数至少满足下列条件之一：

- 接受函数作为输入。
- 输出一个函数。

在函数式语言中，函数不但是一种特殊的对象，还是一种类型，因此函数本身是一个可以传来传去的值。也就是说，某个函数在刚开始执行的时候，总可以送入一个函数的参数。传入的参数本身就是一个函数。当然，这个输入的函数相当于某个函数的另外一个函数。当函数执行完毕之后，又可以返回另外一个新的函数，这个返回函数取决于 return fn(){...}。上述过程出现 3 个不同的函数，分别有不同的角色。要达到这样的应用目的，需要把函数作为一个值来看待。

JavaScript 不但是一门灵活的语言，而且是一门精巧的函数式语言。下面看一个函数作为参数的示例。

```
document.write([2,3,1,4].sort());          //"1,2,3,4"
```

这是最简单的数组排序语句。实际上 Array.prototype.sort() 还能够支持一个可选的参数“比较函数”，其形式如 sort(fn)。fn 是一个函数类型的值，说明这里应用到高阶函数。再如，下面这个对日期类型排序的 sort()。

```
// 声明 3 个对象，每个对象都有属性 id 和 date
var a  = new Object();
var b  = new Object();
var c  = new Object();
a.id   = 1;
b.id   = 2;
c.id   = 3;
a.date = new Date(2012,3,12);
b.date = new Date(2012,1,15);
c.date = new Date(2012,2,10);
// 存放在 arr 数组中
var arr = [a, b, c];
// 开始调试，留意 id 的排列是按 1、2、3 这样的顺序的
arr.sort(
  function (x,y) {
    return x.date-y.date;
  }
);
// 已经对 arr 排序了，发现元素顺序发生变化，id 也发生变化。排序是按照日期进行的
```

在数组排序的时候就会执行“function (x,y) {return x.date-y.date; }”这个传入的函数。当没有传入任何排序参数时，默认当 x 大于 y 时返回 1，当 x 等于 y 时返回 0，当 x 小于 y 时返回 −1。

除了了解函数作为参数使用外，下面再看看函数返回值作为函数的情况。定义一个 wrap 函数，该函数的主要用途是产生一个包裹函数。

```
function wrap(tag) {
    var stag = '<' + tag + '>';
    var etag = '</' + tag.replace(/s.*/, '') + '>';
    return function(x) {
        return stag + x + etag;
    }
}
var B = wrap('B');
document.write(B('粗体字'));
document.write('<br>');
document.write(wrap('B')('粗体字'));
```

“var B = wrap('B');”这一语句已经决定了这是一个“加粗体”的特别函数，执行该 B() 函数就会产生 …内容… 的效果。若是 wrap('div')，就会产生 …内容… 的效果，若是 wrap('li')，就会产生 …内容…
…的效果，依此类推。wrap('B') 返回到变量 B 的是一个函数。若不使用变量，wrap('B') 也是合法的 JavaScript 语句，只要最后一个括号 () 前面的是函数类型的值即可。为什么 stag + x + etag 中的 stag/etag 没有输入也会在 wrap() 内部定义？因为 warp 作用域中就有 stag、etag 两个变量。如果从理论上描述这一特性，应该属于闭包方面的内容。

实际上，map() 函数即为一种高阶函数，在很多的函数式编程语言中均有此函数。map(array, func) 的表达式已经表明，将 func 函数作用于 array 中的每一个元素，最终返回一个新的 array。应该注意的是，map 对 array 和 func 的实现是没有任何预先的假设的，因此称为“高阶”函数。

```
function map(array, func) {
    var res = [];
    for(var i = 0, len = array.length; i < len; i++) {
        res.push(func(array[i]));
    }
    return res;
}
var mapped = map([1, 3, 5, 7, 8], function(n) {
    return n = n + 1;
});
print(mapped);              //2,4,6,8,9
var mapped2 = map(["one", "two", "three", "four"], function(item) {
    return "(" + item + ")";
});
print(mapped2);(one), //(two),(three),(four), 为数组中的每个字符串加上括号
```

mapped 和 mapped2 均调用了 map，但得到了截然不同的结果。因为 map 的参数本身已经进行了一次抽象，map 函数做的是第二次抽象，所以高阶的“阶”可以理解为抽象的层次。

建议 75：函数柯里化

柯里化是把接受多个参数的函数变换成接受一个单一参数的函数，并且返回一个新函数，这个新函数能够接受原函数的参数。下面可以通过例子来帮助理解。

```
function adder(num) {
    return function(x) {
        return num + x;
    }
}
var add5 = adder(5);
var add6 = adder(6);
print(add5(1));            // 6
print(add6(1));            //7
```

函数 adder 接受一个参数，并返回一个函数，这个返回的函数可以像预期那样被调用。变量 add5 保存着 adder(5) 返回的函数，这个函数可以接受一个参数，并返回参数与 5 的和。柯里化在 DOM 的回调中非常有用。

函数柯里化的主要功能是提供了强大的动态函数创建方法，通过调用另一个函数并为它传入要柯里化（currying）的函数和必要的参数而得到。通俗点说就是利用已有的函数，再创建一个动态的函数，该动态函数内部还是通过已有的函数来发生作用，只是传入更多的参数来简化函数的参数方面的调用。

```
function curry(fn) {
    var args = [].slice.call(arguments, 1);
    return function() {
        return fn.apply(null, args.concat([].slice.call(arguments, 0)));
    }
}
function add(num1, num2) {
    return num1 + num2;
}
var newAdd = curry(add, 5);
alert(newAdd(6));          //11
```

在 curry 函数的内部，私有变量 args 相当于一个存储器，用来暂时存储在调用 curry 函数时所传递的参数值，这样再跟后面动态创建函数调用时的参数合并并执行，就会得到一样的效果。

函数柯里化的基本方法和函数绑定是一样的：使用一个闭包返回一个函数。两者的区别在于，当函数被调用时，返回函数还需要设置一些传入的参数。

```
function bind(fn, context) {
    var args = Array.prototype.slice.call(arguments, 2);
    return function() {
        var innerArgs = Array.prototype.slice.call(arguments);
        var finalArgs = args.concat(innerArgs);
        return fn.apply(context, finalArgs);
    };
}
```

创建柯里化函数的通用方式是：

```
function curry(fn) {
    var args = Array.prototype.slice.call(arguments, 1);
    return function() {
        var innerArgs = Array.prototype.slice.call(arguments);
        var finalArgs = args.concat(innerArgs); retrun
        fn.apply(null, finalArgs);
    };
}
```

curry 函数的主要功能就是将被返回的函数的参数进行排序。为了获取第一个参数后的所有参数，在 arguments 对象上调用 slice() 方法，并传入参数 1，表示被返回的数组的第一个元素应该是第二个参数。

建议 76：要重视函数节流

比起非 DOM 交互，DOM 操作需要更多内存和 CPU 时间。连续尝试进行过多的 DOM 相关操作可能会导致浏览器变慢甚至崩溃。函数节流的设计思想就是让某些代码可以在间断情况下连续重复执行，实现的方法是使用定时器对函数进行节流。

例如，在第一次调用函数时，创建一个定时器，在指定的时间间隔后执行代码。当第二次调用时，清除前一次的定时器并设置另一个，实际上就是前一个定时器演示执行，将其替换成一个新的定时器。

```
var processor = {
    timeoutId : null,
    // 实际进行处理的方法
    performProcessing : function() {
        // 实际执行的方法
    },
    // 初始处理调用的方法
    process : function() {
        clearTimeout(this.timeoutId);
        var that = this;
        this.timeoutId = setTimeout(function() {
            that.performProcessing();
        }, 100);
    }
```

```
};
// 尝试开始执行
Processor.process();
```

简化模式：

```
function throttle(method,context){
    clearTimeout(mehtod.tId);
        mehtod.tId = setTimeout(function(){
        method.call(context);
    },100);
}
```

函数节流解决的问题是一些代码（特别是事件）的无间断执行，这个问题严重影响了浏览器的性能，可能会造成浏览器反应速度变慢或直接崩溃，如 resize、mousemove、mouseover、mouseout 等事件的无间断执行。这时加入定时器功能，将事件进行“节流”，即在事件触发的时候设定一个定时器来执行事件处理程序，可以在很大程度上减轻浏览器的负担。类似应用如支付宝中的“导购场景”导航，以及当当网首页左边的导航栏等，这些都是为了解决 mouseover 和 mouseout 移动过快给浏览器处理带来的负担，特别是减轻涉及 Ajax 调用给服务器造成的极大负担。例如：

```
oTrigger.onmouseover = function(e) {// 如果上一个定时器还没有执行，则先清除定时器
    oContainer.autoTimeoutId && clearTimeout(oContainer.autoTimeoutId);
    e = e || window.event;
    var target = e.target || e.srcElement;
    if((/li$/i).test(target.nodeName)) {
        oContainer.timeoutId = setTimeout(function() {
            addTweenForContainer(oContainer, oTrigger, target);
        }, 300);
    }
}
```

建议 77：推荐作用域安全的构造函数

构造函数其实是一个使用 new 运算符的函数。当使用 new 调用时，构造函数的内部用到的 this 对象会指向新创建的实例。

```
function Person(name, age, job) {
    this.name = name;
    this.age = age;
    this.job = job;
}
var person = new Person("Nicholas", 34, 'software Engineer');
```

在没有使用 new 运算符来调用构造函数的情况下，由于该 this 对象是在运行时绑定的，因此直接调用 Person() 会将该对象绑定到全局对象 window 上，这将导致错误属性意外增加到全局作用域上。这是由于 this 的晚绑定造成的，在这里 this 被解析成了 window 对象。

解决这个问题的方案是创建一个作用域安全的构造函数。首先确认 this 对象是否为正确的类型实例，如果不是，则创建新的实例并返回。

```
function Person(name, age, job) {
    //检测 this 对象是否是 Person 的实例
    if(this instanceof Person) {
        this.name = name;
        this.age = age;
        this.job = job;
    } else {
        return new Person(name, age, job);
    }
}
```

如果使用的构造函数获取继承且不使用原型链，那么这个继承可能就被破坏。

```
function Polygon(sides) {
    if(this instanceof Polygon) {
        this.sides = sides;
        this.getArea = function() {
            return 0;
        }
    } else {
        return new Polygon(sides);
    }
}
function Rectangle(width, height) {
    Polygon.call(this, 2);
    this.width = width;
    this.height = height;
    this.getArea = function() {
        return this.width * this.height;
    };
}
var rect = new Rectangle(5, 10);
alert(rect.sides);        //undefined
```

Rectangle 构造函数的作用域是不安全的。在新创建一个 Rectangle 实例后，这个实例通过 Polygon.call 继承了 sides 属性，但由于 Polygon 构造函数的作用域是安全的，this 对象并非是 Polygon 的实例，因此会创建并返回一个新的 Polygon 对象。

由于 Rectangle 构造函数中的 this 对象并没有得到增长，同时 Polygon.call 返回的值没有被用到，所以 Rectangle 实例中不会有 sides 属性。构造函数配合使用原型链可以解决这个问题。

```
function Polygon(sides) {
    if(this instanceof Polygon) {
        this.sides = sides;
        this.getArea = function() {
            return 0;
        }
    } else {
```

```
            return new Polygon(sides);
        }
    }
    function Rectangle(width, height) {
        Polygon.call(this, 2);
        this.width = width;
        this.height = height;
        this.getArea = function() {
            return this.width * this.height;
        };
    }
    // 使用原型链
    Rectangle.prototype = new Polygon();
    var rect = new Rectangle(5, 10);
    alert(rect.sides);    //2
```

这时构造函数的作用域就很有用了。

建议 78：正确理解执行上下文和作用域链

执行上下文（execution context）是 ECMAScript 规范中用来描述 JavaScript 代码执行的抽象概念。所有的 JavaScript 代码都是在某个执行上下文中运行的。在当前执行上下文中调用 function 会进入一个新的执行上下文。该 function 调用结束后会返回到原来的执行上下文中。如果 function 在调用过程中抛出异常，并且没有将其捕获，有可能从多个执行上下文中退出。在 function 调用过程中，也可能调用其他的 function，从而进入新的执行上下文，由此形成一个执行上下文栈。

每个执行上下文都与一个作用域链（scope chain）关联起来。该作用域链用来在 function 执行时求出标识符（identifier）的值。该链中包含多个对象，在对标识符进行求值的过程中，会从链首的对象开始，然后依次查找后面的对象，直到在某个对象中找到与标识符名称相同的属性。在每个对象中进行属性查找时，会使用该对象的 prototype 链。在一个执行上下文中，与其关联的作用域链只会被 with 语句和 catch 子句影响。

在进入一个新的执行上下文时，会按顺序执行下面的操作：

（1）创建激活（activation）对象

激活对象是在进入新的执行上下文时创建出来的，并且与新的执行上下文关联起来。在初始化构造函数时，该对象包含一个名为 arguments 的属性。激活对象在变量初始化时也会被用到。JavaScript 代码不能直接访问该对象，但可以访问该对象的成员（如 arguments）。

（2）创建作用域链

接下来的操作是创建作用域链。每个 function 都有一个内部属性 [[scope]]，它的值是一个包含多个对象的链。该属性的具体值与 function 的创建方式和在代码中的位置有很大关系

（见本建议后面介绍的“function 对象的创建方式”内容）。此时的主要操作是将上一步创建的激活对象添加到 function 的 [[scope]] 属性对应的链的前面。

（3）变量初始化

这一步对 function 中需要使用的变量进行初始化。初始化时使用的对象是创建激活对象过程中所创建的激活对象，不过此时称做变量对象。会被初始化的变量包括 function 调用时的实际参数、内部 function 和局部变量。在这一步中，对于局部变量，只是在变量对象中创建了同名的属性，其属性值为 undefined，只有在 function 执行过程中才会被真正赋值。全局 JavaScript 代码是在全局执行上下文中运行的，该上下文的作用域链只包含一个全局对象。

函数总是在自己的上下文环境中运行，如读 / 写局部变量、函数参数，以及运行内部逻辑结构等。在创建上下文环境的过程中，JavaScript 会遵循一定的运行规则，并按照代码顺序完成一系列操作。这个操作过程如下：

第 1 步，根据调用时传递的参数创建调用对象。

第 2 步，创建参数对象，存储参数变量。

第 3 步，创建对象属性，存储函数定义的局部变量。

第 4 步，把调用对象放在作用域链的头部，以便检索。

第 5 步，执行函数结构体内语句。

第 6 步，返回函数返回值。

针对上面的操作过程，下面进行详细描述。

首先，在函数上下文环境中创建一个调用对象。调用对象与上下文环境是两个不同的概念，也是另一种运行机制。对象可以定义和访问自己的属性或方法，不过这里的对象不是完整意义上的对象，它没有原型，并且不能够被引用，这与 Arguments 对象的 arguments[] 数组不是真正意义上的数组一样。

调用对象会根据传递的参数创建自己的 Arguments 对象，这是一个结构类似数组的对象，该对象内部存储着调用函数时所传递的参数。接着，创建名为 arguments 的属性，该属性引用刚创建的 Arguments 对象。

然后，为上下文环境分配作用域。作用域由对象列表或对象链组成。每个函数对象都有一个内部属性（scope），这个属性值也是由对象列表或对象链组成的。 scope 属性值构成了函数调用上下文环境的作用域，同时，调用对象被添加到作用域链的头部，即该对象列表的顶部（作用域链的前端）。

实际上，这个头部是针对该函数的作用域链而言的，把调用对象添加到作用域的头部就是把调用对象排在函数作用域链的最上面。例如，在下面这个示例中，当调用函数 e() 时，将创建函数 e() 的调用对象和函数 e() 的作用域，但在调用函数 e() 之前，会先调用函数 g()，并且生成调用函数 g() 的对象。而调用函数 e() 的对象会在函数 e() 的作用域范围内处于头部位置，即排在最前面。代码如下：

```
function f(){
```

```
    return e();
    function e(){
        return g();
        function g(){
            return 1;
        }
    }
}
alert(f());        // 1
```

接着，正式执行函数体内代码，此时 JavaScript 会对函数体内创建的变量执行变量实例化操作（即转换为调用对象的属性）。下面进行具体说明。

将函数的形参也创建为调用对象的命名属性，如果调用函数时传递的参数与形参一致，则将相应参数的值赋给这些命名属性，否则会将命名属性赋值为 undefined。

对于内部定义函数（注意其与嵌套函数的区分，两者语义不完全重合），会以其声明时所用名称为调用对象创建同名属性，对应的函数则被创建为函数对象，并将其赋值给该属性。

将在函数内部声明的所有局部变量创建为调用对象的命名属性。注意，在执行函数体内的代码并计算相应的赋值表达式之前不会对局部变量进行真正的实例化。

由于 arguments 属性与函数局部变量对应的命名属性都属于同一个调用对象，因此可以将 arguments 作为函数的局部变量来看待。

最后，创建 this 对象并对其进行赋值。如果赋值为一个对象，则 this 将指向该对象引用。如果赋值为 null，则 this 就指向全局对象。

创建全局上下文环境的过程与上面的描述稍微不同，因为全局上下文环境没有参数，所以不需要通过定义调用对象来引用这些参数。全局上下文环境会有一个作用域，即全局作用域，它的作用域链实际上只由一个对象组成，即全局对象（window）。全局上下文环境也会有变量实例化的过程，它的内部函数就是涉及大部分 JavaScript 代码的、常规的顶级函数声明。全局上下文环境也会使用 this 对象来引用全局对象。

JavaScript 作用域可以细分为词法作用域和动态作用域。词法作用域又称为定义作用域，这是从静态角度来说的。在函数没有被调用之前，根据函数结构的嵌套关系来确定函数的作用域。因此词法作用域取决于源代码，通常编译器可以进行静态分析来确定每个标识符实际的引用。

动态作用域也称为执行作用域，这是从动态角度来说的。当函数被调用之后，其作用域会因为调用而发生变化，此时作用域链也会随之调整。

定义作用域就是用来说明函数在定义时存在的嵌套关系。当函数被执行时，作用域可能会发生变化。JavaScript 函数运行在它们被定义的作用域中，而不是它们被执行的作用域中。

在 JavaScript 中，function 对象的创建方式有 3 种：function 声明、function 表达式和使用 Function 构造器。

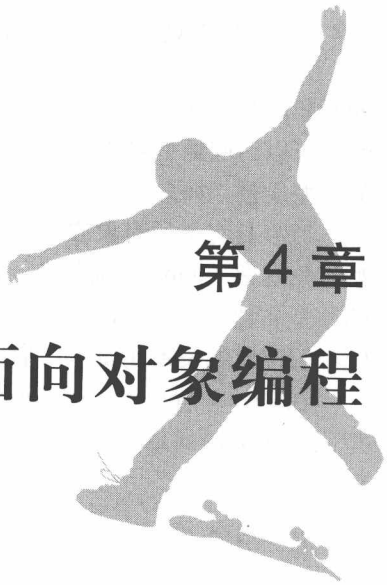
```
function a() {}
```

```
var a = function() {}
var a = new Function()
```

通过这3种方法创建出来的function对象的scope属性的值有所不同，从而影响function执行过程中的作用域链，具体说明如下：

- 使用function语句声明的function对象是在进入执行上下文时的变量初始化过程中创建的。该对象的scope属性的值是它被创建时的执行上下文对应的作用域链。
- 使用function表达式的function对象是在该表达式被执行的时候创建的。该对象的scope属性的值与使用function声明创建的对象一样。
- 使用Function构造器声明一个function通常有两种方式，常用格式是var funcName = new Function(p1, p2,..., pn, body)，其中p1，p2，…，pn表示的是该function的形式参数，body是function的内容，使用该方式的function对象是在构造器被调用的时候创建的。该对象的scope属性的值总是一个只包含全局对象的作用域链。

function对象的length属性可以用来获取声明function时指定的形式参数的个数，而function对象被调用时的实际参数是通过arguments来获取的。



第4章 面向对象编程

JavaScript的许多特性都借鉴自其他语言，如语法借鉴自Java，函数借鉴自Scheme，原型继承借鉴自Self，而正则表达式特性借鉴自Perl。

在面向对象的语言（如C#、Java等）中，类是面向对象的基础，并且具有明显的层次概念和继承关系，每个类都有一个超类，从超类中继承属性和方法，类还可以进一步被扩展（扩展类称为子类），这样就构建了一个多层的、复杂的对象继承关系。但由于JavaScript是基于对象的弱类型语言，它是以对象为基础，以函数为模型，以原型为继承机制的开发模式，因此对于习惯于面向对象开发的用户来说，需要适应JavaScript语言的灵活性和特殊性。

在大多数编程语言中，继承都是重要的主题之一。在那些基于类的语言（如Java）中，继承拥有两个优势。类继承第一个优势是，它是代码重用的一种形式，如果一个新的类与一个已存在的类大部分相似，那么只需要具体说明其不同点即可。代码重用的模式极为重要，因为它们可以显著地减少软件开发的成本。类继承的另一个优势是，它包括了一套类型系统的规范。由于程序员无须编写显式类型转换的代码，所以大大减少了工作量。

建议79：参照Object构造体系分析prototype机制

原型是JavaScript核心特性之一，它通过prototype这个属性表现出来。在JavaScript中，对象（Object）是没有原型的，只有构造函数拥有原型，而构造类的实例对象都能够通过prototype属性访问原型对象。

prototype不仅是JavaScript实现和管理继承的一种机制，更是一种OO（面向对象）的设计思想。从语义角度分析，prototype表示类的原型，就是构造类拥有的原始成员。构造函数的prototype属性存储着一个引用对象的指针，该指针指向一个原型对象，它是一个特殊

的对象，相当于一个数据集合，内部存储着构造函数的原始属性和方法。借助 prototype 属性，可以访问原型对象内部成员。当构造函数实例化后，所有实例对象都可以访问构造函数的原型成员。如果在原型对象中声明一个成员，则所有实例对象都可以共享它。

原型具有普通对象的结构，可以将任何普通对象实例设置为原型对象，在默认状态下原型对象继承于 Object 抽象类，由 JavaScript 原生并依附于每个构造函器上，从而实现构造类的原型属性和原型方法能够被所有实例对象继承。原型对象、原型属性在 JavaScript 对象系统中的位置和关系如图 4.1 所示。

在 JavaScript 中，对象应该是类（class）和实例（instance）的关系演化。类是对象的模型化，而实例则是类的特征具体化。类包含很多概念类型，如元类、超类、泛类和类型等。例如：

```
function Class(type){     // 构造函数
    this.type = type;
}
var instance1 = new Class("instance1");  // 实例对象 1
var instance2 = new Class("instance2");  // 实例对象 2
```

使用 instanceof 运算符可以验证它们的关系：

```
alert(instance1 instanceof Class); //true，说明 instance1 对象是 Class 构造函数的实例
alert(instance2 instanceof Class); //true，说明 instance2 对象是 Class 构造函数的实例
```

instance1 和 instance2 都是对象，但 Class 构造函数不是它们唯一的类型，Object 也是它们的类型：

```
alert(instance1 instanceof Object); //true，说明 instance1 对象也是 Object 构造函数的实例
alert(instance2 instanceof Object); //true，说明 instance2 对象也是 Object 构造函数的实例
```

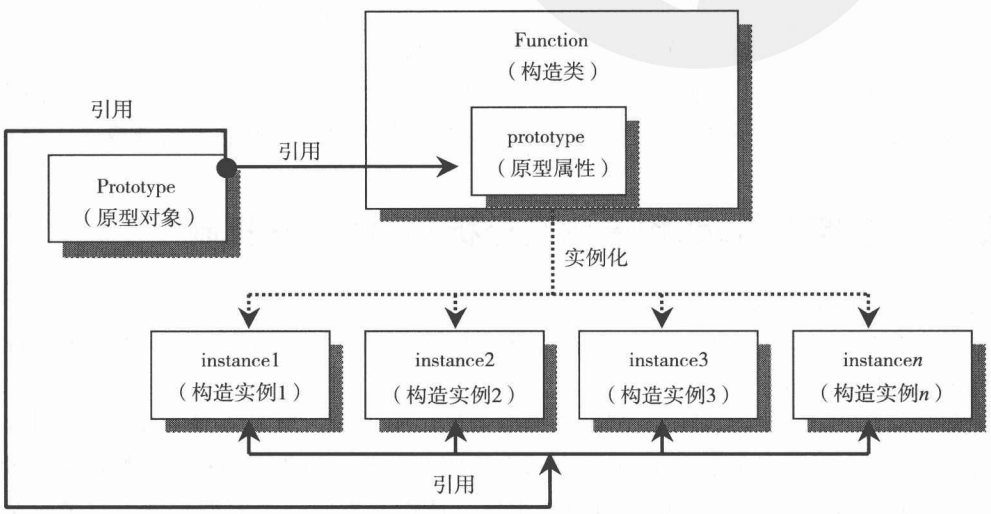


图 4.1　原型对象、原型属性在 JavaScript 对象系统中的位置和关系

Object 比 Class 类型更加抽象，它们之间应该属于一种继承关系。

```
alert(Class instanceof Object);  //true，说明 Class 类是 Object 对象的实例（或子类）
```

但 instance1 和 instance2 对象却不是 Function 构造函数的实例，这说明它们之间没有直接关系。

```
alert(instance1 instanceof Function);    //false，说明它们不是类型与实例的关系
alert(instance2 instanceof Function);    //false，说明它们不是类型与实例的关系
```

而 Object 与 Function 之间的关系就非常微妙，它们都是高度抽象的类型，互为对方的实例。

```
alert(Object instanceof Function) ; //true，说明 Object 对象是 Function 函数的实例（即子类）
alert(Function instanceof Object) ; //true，说明 Function 函数是 Object 对象的实例（即子类）
```

Object 与 Function 同时也是两个不同类型的构造器。下面的代码能够很好地显示它们的差异。

```
var f = new Function();   // 实例化 Function 对象
var o = new Object();     // 实例化 Object 对象
alert(f instanceof Function);     //true，说明 f 是 Function 对象的实例
alert(f instanceof Object);       //true，说明 f 是 Object 对象的实例
alert(o instanceof Function);     //false，说明 o 不是 Function 对象的实例
alert(o instanceof Object);       //true，说明 o 是 Object 对象的实例
```

instance（对象实例）、Class（类型）、Object（抽象类）和 Function（构造类）之间的关系如图 4.2 所示。

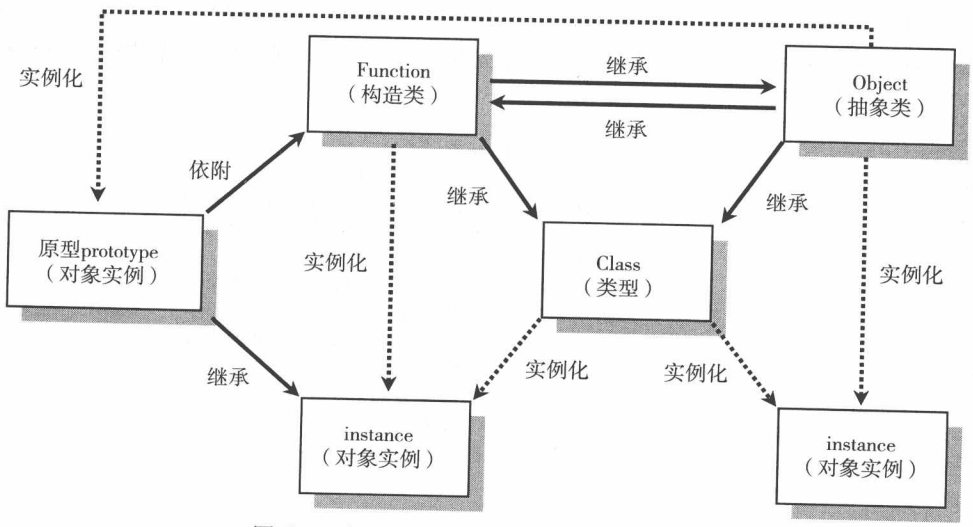


图 4.2　类型、原型和对象实例之间的关系

prototype 是属于 Function 的成员，而 prototype 对象又是 Object 的一个实例，构造函数通过点语法访问 prototype，再通过 prototype 访问原型对象成员。原型属性与本地特性之间的关系如图 4.3 所示。

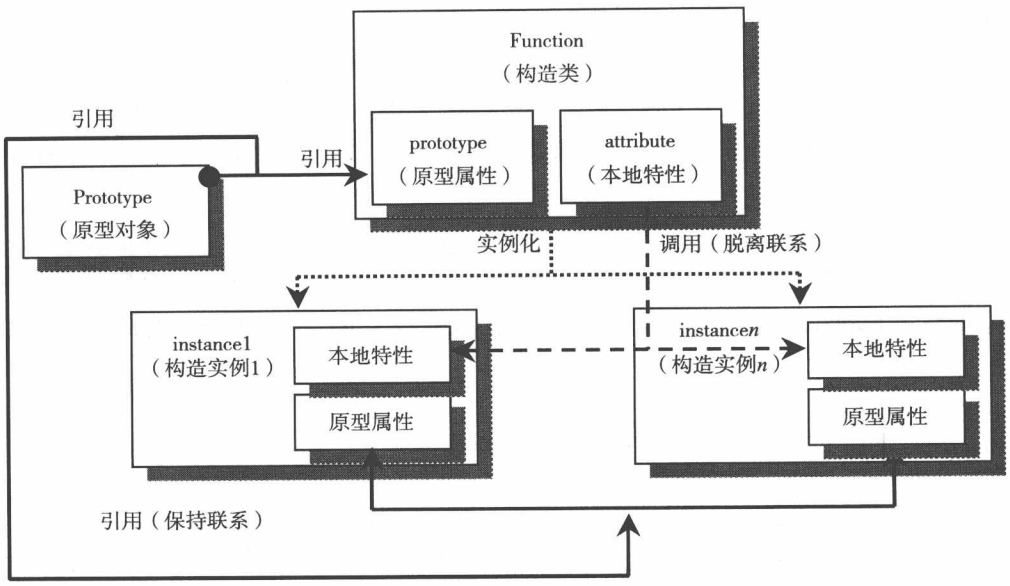


图 4.3　原型属性与本地特性之间的关系

Object 和 Function 都可以定义原型，Object 被视为 Function 的子类。下面的示例能够很好地说明 Object 原型和 Function 原型的异同，这些原型及其属性之间的关系如图 4.4 所示。

```
Object.prototype.a = 1;    // 声明 Object 的原型属性 a 的值为 1
Function.prototype.a= 2;   // 声明 Function 的原型属性 a 的值为 2
alert(Object.a);           //2，说明属性 a 指向 Function 构造函数的原型
alert(Function.a);         //2，说明属性 a 指向 Function 构造函数的原型
var o = {}                 // 空的对象直接量
alert(o.a);                //1，说明属性 a 指向 Object 构造函数的原型
var f = Object;            // 引用 Object 构造函数
alert(f.a);                //2，说明属性 a 指向 Function 构造函数的原型
var f1 = new Function();   // 实例化 Function 对象
alert(f1.a);               //2，说明属性 a 指向 Function 构造函数的原型
var o1 = new Object();     // 实例化 Object 对象
alert(o1.a);               //1，说明属性 a 指向 Object 构造函数的原型
```

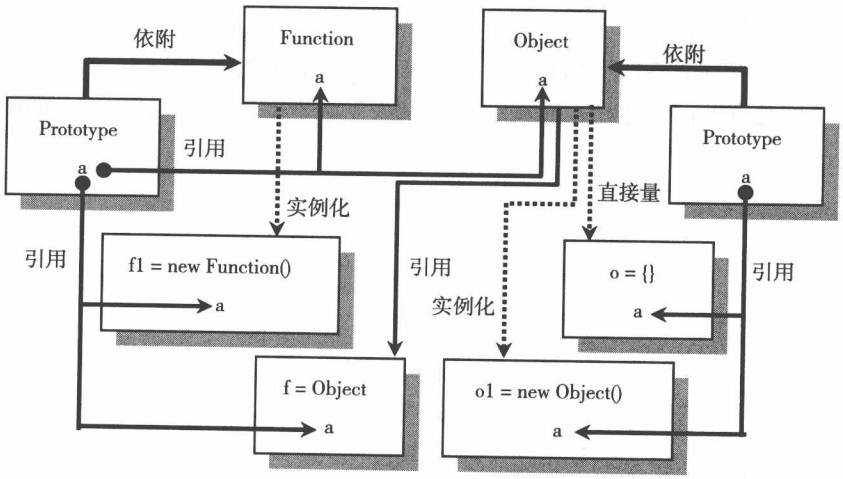


图 4.4　Function、Object、Prototype 及其属性间的关系

建议 80：合理使用原型

1. 使用原型设置默认值

JavaScript 是一种动态语言，它的对象系统也是动态的。在程序中，可以根据需要设置原型值，从而影响所有实例对象。如果为构造函数定义了与原型属性同名的本地属性，则本地属性会覆盖原型属性，例如：

```
function p(x){                    // 构造函数
    this.x = x;
}
p.prototype.x = 1;
var p1 = new p(10);
alert(p1.x);                      //10，本地属性覆盖原型属性
```

但原型属性并没有被删除，它依然存在，仅是被同名的本地属性覆盖了，如果使用 delete 运算符删除本地属性，那么原型属性依然会显示出来。因此，当原型属性与本地属性同时存在时，它们之间可以出现交流现象。利用这种现象为对象初始化默认值，例如：

```
function p(x){                    // 构造函数
    if(x)                         // 如果参数存在，则使用该参数设置属性，此条件是关键
        this.x = x;
}
p.prototype.x = 0;                // 利用原型属性，设置本地属性 x 的默认值
var p1 = new p();
alert(p1.x);                      //0，显示本地属性的默认值
var p2 = new p(1);
alert(p2.x);                      //1，显示本地属性的初始化值
```

2. 使用原型实现数据备份

把本地对象的数据完全赋值给原型对象，相当于为该对象定义一个副本，这就是备份对象。当对象属性被修改时，可以通过原型对象来恢复本地对象的初始值。下面示例演示了如何使用原型备份数据。

```
function p(x){     // 构造函数
    this.x = x;
}
p.prototype.backup = function(){          // 原型方法，备份本地对象的数据到原型对象中
    for(var i in this){
        p.prototype[i] = this[i];
    }
}
var p1 = new p(1);
p1.backup();       // 备份实例对象中的数据
p1.x =10;          // 改写本地对象的属性值
alert(p1.x) ;      //10，说明属性值已经被改写
p1 = p.prototype; ;        // 恢复备份
alert(p1.x) ;      //1，说明对象的属性值已经被恢复到原始值
```

3. 使用原型设置只读属性

利用原型还可以为对象属性设置“只读”特性，这样可以避免对象内部数据被任意篡改。这里的“只读”只是一个表象，并不是真正禁止对象属性进行修改，而是借助闭包体存储属性值，这样就可以避免属性值被动态修改。

下面示例演示了如何根据平面上两点坐标来计算它们之间的距离。构造函数 p 用来设置定位点坐标，如果传递了两个参数值，会返回以参数为坐标值的点；如果省略参数，则默认点为原点（0,0）。而在构造函数 l 中，通过传递的两点坐标对象来计算它们的距离。

如果无意间修改了构造函数的方法 b() 或 e() 的值，则构造函数中 length() 方法的计算值也随之发生变化。这种动态效果对于动态跟踪两点坐标变化来说，是非常必要的。但是，我们并不需要在初始化实例之后随意地改动坐标值，毕竟方法 b() 和 f() 与参数 a 和 b 是没有多大联系的，但它们的参数值却同时指向同一个对象的引用指针。

```
function p(x,y){  // 求坐标点构造函数
    if(x) this.x =x;       // 初始 x 轴值
    if(y) this.y = y;      // 初始 y 轴值
    p.prototype.x =0;      // 默认 x 轴值
    p.prototype.y = 0;     // 默认 y 轴值
}
function l(a,b){  // 求两点间距离的构造函数
    var a = a;
    var b = b;
    var w = function(){   // 计算 x 轴距离，返回对函数引用
        return Math.abs(a.x - b.x);
    }
    var h = function(){   // 计算 y 轴距离，返回对函数引用
```

```
        return Math.abs(a.y - b.y);
    }
    this.length = function(){// 计算两点间距离，使用小括号调用私有方法 w() 和 h()
        return Math.sqrt(w()*w() + h()*h());
    }
    this.b = function(){  // 获取起点坐标对象
        return a;
    }
    this.e = function(){  // 获取终点坐标对象
        return b;
    }
}
var p1 = new p(1,2);        // 声明一个点
var p2 = new p(10,20);      // 声明另一个点
var l1 = new l(p1,p2);      // 实例化构造函数，传递两点对象
alert(l1.length());         //20.12461179749811，调用 length() 方法计算两点间距离
l1.b().x = 50;              // 不经意地改动方法 b() 的一个属性为 50
alert(l1.length());         //43.86342439892262，说明上面改动影响到两点间距离
```

为了避免因为改动方法 b() 的属性 x 的值而影响两点间距离，可以在方法 b() 和 e() 中新建一个临时性的构造类，设置该类的原型为 a，然后实例化构造类并返回，这样就阻断了方法 b() 与私有变量 a 的直接联系，它们之间仅是值的传递，而不是对对象 a 的引用，从而避免因方法 b() 的属性值变化而影响私有对象 a 的属性值。

```
this.b = function(){       // 方法 b()
    function temp(){};     // 临时构造类
    temp.prototype = a;    // 把私有对象传递给临时构造类的原型对象
    return new temp();     // 实例化后的对象，阻断直接返回 a 所出现的引用关系
}
this.e = function(){       // 方法 f()
    function temp(){};     // 临时构造类
    temp.prototype = a;    // 把私有对象传递给临时构造类的原型对象
    return new temp();     // 实例化后的对象，阻断直接返回 a 所出现的引用关系
}
```

还有一种方法，这种方法是在为私有变量 w 和 h 赋值时，不是向函数赋值，而是函数调用表达式，这样私有变量 w 和 h 存储的是值类型数据，而不是对函数结构的引用，从而不再受后期相关属性值的影响。

```
function l(a,b){  // 求两点间距离的构造函数
    var a = a;
    var b = b;
    var w = function(){   // 计算 x 轴距离，返回函数表达式的计算值
        return Math.abs(a.x - b.x);
    }()
    var h = function(){   // 计算 y 轴距离，返回函数表达式的计算值
        return Math.abs(a.y - b.y);
    }()
    this.length = function(){     // 计算两点间距离，直接使用私有变量 w 和 h 来计算
        return Math.sqrt(w()*w() + h()*h());
```

```
    }
    this.b = function(){  // 获取起点坐标对象
        return a;
    }
    this.e = function(){  // 获取终点坐标对象
        return b;
    }
}
```

4. 使用原型进行批量复制

先看下面一个示例：

```
function f( x ){  // 构造函数
    this.x = x;
}
var a = [];
for( var i = 0; i < 100; i ++ ){ // 使用 for 循环结构批量复制构造类 f 的同一个实例
    a[i] = new f( 10 );   // 把实例分别存入数组
}
```

上面的代码演示了如何复制同一个实例对象 100 次。如果后期需要修改数组中每个实例对象，就会非常麻烦。现在可以尝试使用原型来进行批量复制操作，如下所示。

```
function f( x ){  // 构造函数
    this.x = x;
}
var a = [];
function temp(){};          // 定义一个临时的空构造类 temp
temp.prototype = new f( 10 );     // 把构造类 f 实例化，并把该实例传递给构造类 temp 的原型对象
for( var i = 0; i < 100; i ++ ){ ;  // 使用 for 循环结构批量复制临时构造类 temp 的同一个实例
    a[i] = new temp();     // 把实例分别存入数组
}
```

把构造类 f 的实例存储在临时构造类的原型对象中，然后通过临时构造类 temp 实例来传递复制的值。因此，要想修改数组的值，只需要修改类 f 的原型即可，从而避免逐一修改数组中每个元素。

建议 81：原型域链不是作用域链

prototype 是一种模拟面向对象的机制，它通过原型实现类与实例之间的继承关系并进行管理。以 prototype 机制模拟继承机制是一种原型继承，它也是 JavaScript 的核心功能之一。但是，prototype 的真正价值在于它能够以对象结构为载体，创建大量的实例，这些实例能够在构造类下实现共享。也正因为如此，很多人利用 prototype 的这个特性模拟对象的继承机制。

protoype 原型域可以允许原型属性引用任何类型的对象。因此，如果在 protoype 原型域中没有找到指定的属性，那么 JavaScript 将会根据引用关系，继续向外查找 protoype 原型域

所指向对象的 protoype 原型域，直到对象的 prototype 域为它自己，或者出现循环为止。下面这个示例演示了对象属性查找的 prototype 规律。

```
function a(x){                  // 构造函数 a
    this.x = x;
}
a.prototype.x = 0;
function b(x){                  // 构造函数 b
    this.x = x;
}
b.prototype = new a(1);         // 原型对象为构造函数 a 的实例
function c(x){                  // 构造函数 c
    this.x = x;
}
c.prototype = new b(2);         // 原型对象为构造函数 b 的实例
var d = new c(3);
alert(d.x);                     //3
delete d.x;
alert(d.x);                     //2
delete c.prototype.x;
alert(d.x);                     //1
delete b.prototype.x;
alert(d.x);                     //0
delete a.prototype.x;
alert(d.x);                     //undefined
```

原型链可以帮助我们更清楚地认识 JavaScript 面向对象的继承关系。每个对象实例都可以访问它的构造器的原型，把这种层层指向父原型的关系称为原型链（prototype chain），如图 4.5 所示。

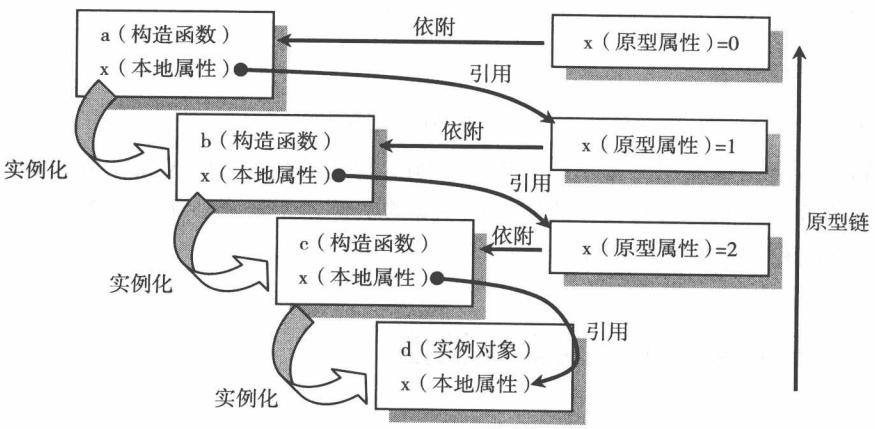


图 4.5　原型链检索示意图

在 JavaScript 中，一切都是对象，函数是第一型。Function 和 Object 都是函数的实例。构造函数的父原型指向 Function 的原型，Function.prototype 的父原型是 Object 的原型，Object 的父原型也指向 Function 的原型，Object.prototype 是所有父原型的顶层。

```
Function.prototype.a = function(){          // Function 原型方法
    alert( "Function" );
}
Object.prototype.a = function(){            // Object 原型方法
    alert( "Object" );
}
function f(){                               // 构造函数 f
    this.a = "a";
}
f.prototype = {                             // 构造函数 f 的原型方法
    w : function(){
        alert( "w" );
    }
}
alert( f instanceof Function );             //true，说明 f 是 Function 的实例
alert( f.prototype instanceof Object );     //true，说明 f 的原型也是对象
alert( Function instanceof Object );        //true，说明 Function 是 Object 的实例
alert( Function.prototype instanceof Object ); //true，说明 Function 的原型是 Object 的实例
alert( Object instanceof Function );        //true，说明 Object 是 Function 的实例
alert( Object.prototype instanceof Function );      //false，说明 Object.prototype 是所有父
原型的顶层
```

原型的引用关系也容易带来副作用。改变某个原型上的引用类型的属性值，将会影响该原型作用的所有实例对象。很多时候这种动态影响会给程序带来安全隐患。

```
function a(){       // 构造函数 a
    this.x = [];
}
function b(){       // 构造函数 b
}
b.prototype = new a();     // 构造函数 b 的原型指向 a 的实例
var f1 = new b();
var f2 = new b();
f1.x.push(1);
alert(f2.x);        //1，说明实例 f2 的 x 属性值也受到影响
```

上面的示例演示了原型对于不同实例的影响。构造函数 a 中的 x 属性值为数组，该数组是引用类型的数据，而构造函数 b 的原型引用 a 的实例，于是直接引用了数组的指针，从而导致了实例之间相互影响的这种现象。

建议 82：不要直接检索对象属性值

检索对象中包含的值有两种形式。

- 当属性名为任意字符形式，或者无法确定属性名时，采用在 [] 后缀中括住一个字符串表达式的方式。采用这种方式比较安全，例如：

```
obj["first-name"]
```

- 如果字符串表达式是一个常数，并是一个合法的 JavaScript 标识符而并非保留字，那么也可以用点（.）表示法代替。优先考虑使用点表示法，因为它更紧凑且可读性更好，例如：

```
obj. obj_1. Obj_2
```

注意，如果尝试检索一个并不存在的成员元素的值，将返回一个 undefined 值，例如：

```
obj ["middle-name"] ;      // undefined
obj. middle_name ;         // undefined
obj ["FIRST-NAME"] ;       // undefined
```

读者可以使用 || 运算符为对象属性设置默认值，当对象属性值未定义时，以默认值进行传递，例如：

```
var middle = obj ["middle-name"] || "(none)";
var status = obj.status || "unknown";
```

由于 JavaScript 在检索一个 undefined 值时将会导致 TypeError 异常，所以建议读者不要直接检索对象属性值。为了稳妥起见，不妨考虑通过 && 运算符来避免此类错误，例如：

```
obj.obj_1;                          // undefined
obj.obj_1.model;                    // 抛出 "TypeError" 异常
obj.obj_1 && obj.obj_1.model;       // undefined
```

在上面代码中，先检索 obj.obj_1 是否存在，如果存在，则进一步检索 obj.obj_1 包含的属性值。如果直接去读取 obj.obj_1.model 属性值，那么有可能会发生异常。

建议 83：使用原型委托

任何一个对象都连接到一个原型对象，并且可以从中继承属性。通过对象字面量创建的对象都可以连接到 Object.prototype 这个基本原型对象，例如：

```
var  obj = {};
Object.prototype.name = "prototype";
obj.name;                                   // "prototype"
```

当然，在创建一个新对象时也可以选择某个对象作为它的原型。JavaScript 提供的实现机制比较烦琐，不妨设计一个中间件，以简化这种操作：为 Object 对象增加一个 create 方法，由 create 方法创建一个使用参数对象作为其原型的新对象。

```
if( typeof Object.create !== 'function') {
    Object.create = function(o) {
        var F = function() {};
        F.prototype = o;
        return new F();
    };
};
```

在上面代码中先检测 Object.create 是否为 Object 对象的方法，如果不是，则添加该方法，在该方法内创建一个构造器，把参数对象作为原型传递给它，然后实例化构造器，最后将这个实例返回。

现在就可以定义两个对象 obj1 和 obj2，借助 Object.create 这个中间件，把 obj1 作为 prototype 绑定到 obj2 上，此时就可以在 obj2 中继承 obj1 对象中的属性了。

```
var obj1 = {
    name : "obj1"
}
var obj2 = Object.create(obj1);
obj2.name;                          // "obj1"
```

原型连接在更新时是不起作用的。当改变某个对象属性值时，不会触及该对象的原型，例如：

```
obj2 ['first-name'] = 'first';
obj2 ['middle-name'] = 'middle';
obj2.name = 'More';
```

原型连接只有在检索值时才会用到。如果尝试去获取对象的某个属性值，并且该对象没有此属性名，那么 JavaScript 会试着从原型对象中获取属性值。如果原型对象也没有该属性，那么再从原型对象的原型中寻找，依此类推，直到该过程最后到达终点 Object.prototype。如果想要的属性完全不存在于原型链中，那么最后结果将返回 undefined 值。这个检索的过程称为原型委托。

原型关系是一种动态的关系。如果添加一个新的属性到原型中，那么该属性会立即被所有基于该原型创建的对象继承。

建议 84：防止原型反射

检查对象并确定对象有什么属性是很容易的事情，只要试着去检索对象属性并验证取得的值即可。同时，使用 typeof 操作符可以判断属性的类型。

```
typeof obj.number          // 'number'
typeof obj.status          // 'string'
typeof obj.name            // 'undefined'
```

但是，根据原型继承的关系，原型链中的所有属性也会产生反射，这种反射称为原型反射。

```
typeof obj.toString        // 'function'
typeof obj.constructor     // 'function'
```

为了避免原型继承的干扰，有两种方法可用于处理这些不需要的属性：

- 编写程序进行检查并剔除函数值。一般来说，对象反射的目标是读取对象包含的数据，因此当对象成员值为函数时，一般都不是我们需要的数据。

❑ 使用 hasOwnProperty 方法。如果对象拥有独有的属性，那么调用该方法将返回 true，也就是说，hasOwnProperty 方法不会检查原型链。

```
obj.hasOwnProperty('number')      // true
obj.hasOwnProperty('constructor')        // false
```

要快速枚举对象的所有属性，可以使用 for in 语句。该语句可遍历对象中的所有属性名，枚举过程将会列出所有的属性，包括方法和原型属性。因此，在枚举过程中，必须过滤掉那些不想要的值，具体方法如下：

❑ 使用 hasOwnProperty 方法过滤原型属性。

❑ 使用 typeof 运算符排除方法函数。

例如：

```
var name;
for(name in obj) {
    if( typeof obj[name] !== 'function') {
        document.writeln(name + ': ' + obj[name]);
    }
}
```

当使用 for in 语句枚举对象时，由于属性名出现的顺序是不确定的，因此要对任何可能出现的顺序有所准备。要确保属性以特定的顺序出现，最好的办法就是避免使用 for in 语句，创建一个数组，在其中以正确的顺序包含属性名。

```
var i;
var properties = ['one', 'two', 'three', 'four'];
for( i = 0; i < properties.length; i += 1) {
    document.writeln(properties[i] + ': ' + obj[properties[i]]);
}
```

通过使用 for 而不是 for in 可以枚举需要的属性，从而不用担心可能反射到原型链中的属性，并且可以按正确的顺序取得它们的值。

建议 85：谨慎处理对象的 Scope

在面向对象的 JavaScript 编程中，包装继承了大量的对象，同时对象之间还有很多复杂的引用关系，这就使得很多在 function 被调用时的 Scope 不是想要的 Scope。产生这些问题的原因都源于没有深入了解 JavaScript 的机制。

在 JavaScript 中，function 是作用于词法范围而不是动态运行范围的，也就是说，function 的作用范围是它声明的范围，而不是它在执行时的范围。简单地说，一个 function 在执行时的上下文环境 Context 在其定义时就固定下来了，就是它定义时的作用范围。有一点需要注意，很多时候动态地将某个方法注入到一个对象内部，然而在运行时总是得不到想要的上下

文环境，这是因为没有正确理解 JavaScript 的 Scope。

当一个 function 被 JavaScript 引擎调用执行时，这个 function 的 Scope 才起作用，并且被加到 Scope 链中。然后，将一个名为 Call Object 的调用对象或运行对象加到 Scope 的最前面。这个调用对象在初始化时会加入一个 arguments 属性，用来引用 function 调用时的参数。如果这个 function 显式地定义了调用参数，那么这些参数也会被加入到这个对象中，之后在这个函数运行过程中所有的局部变量也都将包含在这个对象中。这也就是在 function 体内部既可以通过 arguments 数组，又可以直接通过显式定义参数名来引用调用时传入参数的原因。

调用对象和通过 this 关键字引用的对象是两个概念，调用对象是 function 在运行时的 Scope，其中包含了 function 在运行时的全部参数和局部变量。通过 this 关键字引用对象，是当 function 作为一个对象的方法运行时对这个对象进行引用。如果这个 function 没有被定义在一个对象中，传给 this 的对象是全局对象，那么在这个 function 内部通过 this 取到的变量就是全局定义的变量。例如，下面代码运行结果应该弹出“Hello,I am Qin Jian”。

```
var hello = "Hello,I am Shao Yu!";
function sayHello() {
    var hello = "Hello, I am Qin Jian."
    function anotherFun() {
        alert(hello);
    }
    anotherFun();
}
sayHello();
```

在 JavaScript 中，允许定义匿名 function 和在 function 中嵌套定义另一个 function。由于 Scope 链的作用，function 的内部总是可以访问外部的变量而外部却不能访问内部的变量。另外，由于 Call Object 的机制，可以使用 function 嵌套定义和调用来做很多事情。在 JavaScript 中，这样的调用称为 Closure 闭包。例如，下面代码使用闭包生成唯一 ID。

```
uniqueID = (function() {
    var id = 0;
    return function() {
        return id++;
    };
})();
alert(uniqueID());          //0
alert(uniqueID());          //1
alert(uniqueID());          //2
```

上面这段代码很清楚地说明了闭包做了什么事情。当外层的 function 被执行时，它的 Scope 被加入到 Scope chain 上，然后为它创建 Call Object 并加入到 Scope 中，之后又创建了局部变量 id 并将它保存在该 function 的 Call Object 中。如果没有“return function () {return id++; };”这条语句，那么外层的 function 将会运行结束并退出，同时它的 Call Object 会被释放，Scope 会从 Scope chain 上移除。由于“return function () {return id++; };”这条语句创建了一个内部的 function，并且将其引用返回给一个变量，因此内部 function 的 Scope 会

被添加到之前外部 function 的 Scope 之下，使得在外部 function 运行结束后它的 Scope 不能被撤销和释放。这样就是用外部 function 的 Call Object 保存了变量 id，并且除了内部的 function 以外没有别的程序能访问到这个变量。虽然看起来有些复杂，但是闭包确实是一项非常有用的功能，经常用来保存变量、控制访问域等。例如，在下面示例中利用 Call Object 和闭包保存数据。

```
function makefunc(x) {
    return function() {
        return x;
    }
}
var a = [makefunc("I am Qin Jian"), makefunc("I am shao yu"), makefunc("I am xu ming")];
alert(a[0]());     //I am Qin Jian
alert(a[1]());     //I am shao yu
alert(a[2]());     //I am xu ming
```

JavaScript 中提供了两个非常有意思的方法：call 和 apply，使用它们可以将一个 function 作为另一个 Object 的对象方法来调用，也就是说，可以在选择 function 调用时，将其传入给 this 关键字的对象。这两个方法第一个参数是相同的，都是想要传入给 this 关键字的对象。不同之处是，call 方法直接将 function 参数列在后面，而 apply 方法是将所有 function 参数以一个数组的形式传入。例如：

```
var fun = function(arg1, arg2) {
    //...
}
fun.call(object, arg1, arg2);
fun.apply(object, [arg1, arg2]);
```

这两个方法在面向对象的 JavaScript 编程中是非常有用的，因为有时希望给某个对象添加一个事件监听，然而回调方法的 context 却不一定是需要的，这时就需要使用 call 或 apply 方法了。

eval 方法用于执行某个字符串中的 JavaScript 脚本，并返回执行结果。它允许动态生成的变量、表达式、函数调用、语句段得到执行，使得 JavaScript 程序的设计过程更为灵活，如通过 Ajax 方式从服务器端获得代表 JavaScript 脚本的字符串，然后就可以用 eval 方法在浏览器端执行这段脚本。传统的 Ajax 通信设计更多的是在服务器端与浏览器端交换数据，但是在 eval 方法的支持下可以在两者之间交换逻辑，这是一种很有趣的事情。

Eval 方法很灵活也比较简单，在调用此方法时将要执行的 JavaScript 脚本字符串作为参数。比较常用的就是将服务器端发送过来的 JSON 字符串在浏览器端转化为 JSON 对象。例如，使用 eval 方法将 JSON 字符串转换为对象。

```
var JSONString = "{'name':{'qinjian':'I am qinjian','shaoyu':'I am shaoyu'}}";
var JSONObject = eval("(" + JSONString + ")");
alert(JSONObject.name.qinjian)
```

在上面的代码中，在 JSON 字符串中又加上了一对括号，这样做可以迫使 eval 方法在评估 JavaScript 代码时强制将原最外层括号内的内容作为表达式来执行从而得到 JSON 对象，而不是作为语句来执行。因此，只有用新的小括号将原来的 JSON 字符串包含起来才能够转换出所需的 JSON 对象。

对于执行一般的包含在字符串中的 JavaScript 语句，自然就不需要像上面那样再次添加括号了，例如：

```
function testStatement() {
      eval("var m = 1");
      alert(m);
 }
testStatement();
```

上面的函数会在弹出的提示对话框中输出变量 m 的值。虽然 eval 函数中的语句只是一个简单的赋值，但是有一个问题值得注意，那就是 eval 中的语句是在什么样的 Scope 中执行，为了更好地说明这个问题，执行下面的代码：

```
function testStatement() {
      eval("var m = 1");
      alert(m);
 }
testStatement();
alert(typeof m);
```

可以得到，变量 m 所在的 Scope 是在 testStatement 函数内，从 eval 调用的位置来看，这个执行结果是合理的。接下来用 window.eval 替换上面代码中的 eval 后，在 Firefox 浏览器（注意是 Firefox）上执行代码，从执行后得到的结果中可以发现，这时的 m 的作用域变成了 window，也就是说，m 变成了一个全局变量。那么在 IE 上执行会如何呢？经测试发现，使用 window.eval 和 eval 会在 IE 上得到相同的结果，即 m 的作用域在 testStatement 函数中。但 IE 提供了另外一个方法 execScript，它会将输入的 JavaScript 脚本字符串放到全局作用域下执行。总结一下：eval 方法是将输入的代码在局部作用域下执行，若要使 JavaScript 字符串中包含的代码具有全局作用域执行效果，要么把 eval 放到全局作用域下调用，要么在 Firefox 中使用 window.eval，在 IE 中使用 execScript。最后需要提醒的是，因为 eval 的执行效率较低，所以在程序中最好不要频繁使用。为了避免 Scope 问题，应该注意下列几方面内容。

（1）巧用闭包

了解了闭包利用 Call Object 的产生原理后，就可以很容易利用闭包，如利用 namespace 隔离和保存局部数据等。同时，闭包也很容易使 this 实例不是我们想要的 this 实例，这时就可以利用内层能够访问外层变量的特点将外层 this 实例赋给一个变量，内层可以通过这个变量顺利访问外层 this 实例。

```
switchAds : function (index) {
    var _this = this;
```

```
        dojo.fadeOut({
            node : _this.adBack,
            duration : 500,
            onEnd : function() {
                dojo.fadeIn({
                    node : _this.adBack,
                    duration : 500,
                    onEnd : function() {
                        _this.currentAd = index;
                    }
                }).play();
            }
        }).play();
    }
```

（2）使用 call 和 apply 指定 function 调用时的 Scope

我们经常会提供一种类似回调函数的机制，在设计某个接口的时候并不提供函数的实现而只负责调用该接口。例如，对外提供一个 onclick 事件接口，由于 JavaScript 的 Scope 机制属于词法范围而不是动态运行范围，因此回调函数运行的 Scope 往往不是我们想要的。对于这种情况，可以在注册回调函数时将 Scope 一起传进来，在需要调用该回调函数时使用 call 和 apply 方法来调用这个函数，同时将需要的 Scope 传递给这个函数。例如，dojo 的事件机制就是这样处理的，在注册回调函数的同时可以指定函数调用的 Scope。

```
dojo.connect(node, "onclick", this , this._collapse);
```

（3）慎用 eval

eval 可以动态地从字符串中执行代码，它使 JavaScript 的功能更加强大。通常，eval 有全局或局部两种运行 Scope 方式。当其运行在一个局部的 function 中时，需要注意，这时 eval 运行在这个 function 的 Call Object 中，它可以通过 this 关键字访问到这个 function 的 Scope。另外，eval 方法运行效率非常低，并且运行的脚本是未经验证的，因此在使用 eval 方法时要十分慎重。

建议 86：使用面向对象模拟继承

JavaScript 是一种弱类型解释运行的脚本语言，就语言本身来讲，它不是一门面向对象语言，但我们可以利用一些语言特性来模拟面向对象编程和继承机制，这一切都需要从 JavaScript 中的 function 讲起。

函数是一个定义一次但调用或执行无数次的 JavaScript 代码片段。函数可以有零个或多个输入参数和一个返回值，它通常用于完成一些计算或事务处理等任务。通常可以这样定义一个 function：

```
function distance(x1, y1, x2, y2) {
```

```
    var dx = x2 - x1;
    var dy = y2 - y1;
    return Math.sqrt(dx*dx + dy*dy);
}
```

在 JavaScript 中，function 不仅是一种语法结构，还可以作为一种数据。这意味着它可以被赋值给变量，在对象或数组中作为元素的属性存储，或者作为函数参数传递等。例如，把 function 作为数据使用：

```
var d = distance;
d(1,1,2,2);
```

当在一个 object 中定义和调用一个 function 时，这个 function 被称做该 object 的一个方法。需要注意的是，当这个 function 被调用时，这个 object 会以隐含参数的形式传入到 function 中，function 内部可以通过 this 关键字来引用这个 object 的属性。例如，在下面这个例子的运行结果中，calculator.result 的值为 2。

```
var calculator = {
        operand1: 1,
        operand2: 1,
    compute: function () {
        this.result = this.operand1 + this.operand2;
        }
    };
 calculator.compute();
 alert(calculator.result);
```

在 JavaScript 中，对象的创建通常是通过 new 运算符来完成的，new 关键字后面必须是一个可执行的 function。例如：

```
var array = new Array(10);
var today = new Date( );
```

当上面这条创建语句被执行时，首先会创建一个空的对象赋给前面的变量，然后调用后面紧跟的 function，并且将这个空的对象作为隐含参数传入到 function 内部。这样在 function 内部就可以通过 this 关键字引用该对象，做一些对象初始化工作。这样的 function 通常被称为构造方法或简单构造方法。例如：

```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}
var rect1 = new Rectangle(2, 4);
```

到这里已经有了对象、函数、作为数据的函数、对象方法和构造函数，于是可以使用这些 JavaScript 的特性模拟写出类似于 Java 风格的面向对象的代码。例如，利用下面代码就可以模拟面向对象。

```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
    this.area = function (){
        return this.width*this.height;
    };
}
var rect1 = new Rectangle(2, 4);
rect1.area();
```

上面的代码看起来非常不错，但似乎还缺少一些东西。面向对象的精髓是对事物的抽象和继承，以达到代码重用的目的，在 JavaScript 中如何做到这一点呢？

在 JavaScript 中，每一个 object 都有一个 prototype 属性，这个属性引用了另一个 object 对象。当需要读取一个 object 中的某个属性 p 时，JavaScript 引擎先查找这个 object 中是否存在属性 p，如果存在则返回该属性值，如果不存在则在这个 object 的 prototype 对象中查找是否存在属性 p。这样做会有两点好处：第一，在每个类中可以抽象出来一些共有的属性和方法定义在 prototype 对象中，当使用构造函数创建多个类的实例时，多个实例使用同一 prototype 对象大大减小了对内存的占用量。第二，新创建的对象的属性是在创建对象时由 prototype 对象带来的，这样就可以实现属性和方法的继承。

在前面对 function 的介绍中曾提到，当 new 操作符被调用时会创建一个空的对象赋给变量并调用后面的构造函数。实际上在利用 new 操作符创建一个空对象后，还会为这个对象设置它的 prototype 属性，这个属性值等于它的构造函数的 prototype 属性值。所有的函数在其定义时就已经自动创建和初始化好了 prototype 属性，这个初始化好的 prototype 属性指向一个只包含一个 constructor 属性的对象，并且这个 constructor 属性指向这个 function 自身。这就解释了为什么每一个 object 都会有一个 constructor 属性。运行下面的代码，将打印出一个名为 Object 的简版的函数体。

```
var obj = new Object();
alert(obj.constructor);
```

了解了 JavaScript 模拟面向对象编程基础知识，就可以通过将一些共有的属性和方法定义在构造方法中来实现属性的继承。

```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}
Rectangle.prototype.area = function() {
    return this.width * this.height;
};
```

另外，要继承另一个类并重写和添加一些方法，可能需要复制这个类的构造方法的 prototype 对象中的属性和方法，然后再根据需要重写和添加一些方法和属性。

建议 87：分辨 this 和 function 调用关系

JavaScript 的 this 一直是容易让人误用的关键词，尤其对于熟悉 Java 的程序员来说，因为 JavaScript 中的 this 与 Java 中的 this 是有很大不同的。在一个 function 的执行过程中，如果在变量的前面加上了 this 作为前缀，如 this.myVal，对此变量的求值就从 this 所表示的对象开始。

this 的值取决于 function 被调用的方式，一共有 5 种：

- 如果一个 function 是一个对象的属性，那么在 funtion 被调用时，this 的值是这个对象。
- 如果 function 调用的表达式包含句点（.）或 []，那么 this 的值是句点（.）或 [] 之前的对象。如在 myObj.func 和 myObj["func"] 中，func 被调用时的 this 是 myObj。
- 如果一个 function 不是作为一个对象的属性，那么在 function 被调用时，this 的值是全局对象。当一个 function 中包含内部 function 时，不理解 this 的正确含义就很容易造成错误，因为内部 function 的 this 值与它外部的 function 的 this 值是不一样的。

```
var myObj = {
    myVal : "Hello World",
    func : function() {
        alert( typeof this.myVal);       // string
        var self = this;
        function inner() {
            alert( typeof this.myVal);   //undefined
            alert( typeof self.myVal);   // string
        }
        inner();
    }
};
myObj.func();
```

在 myObj 函数体内有个内部名为 inner 的函数，在 inner 被调用时，this 的值是全局对象，因此找不到名为 myVal 的变量。这时通常的解决办法是将外部 function 的 this 值保存在一个变量中（此处为 self），在内部函数中使用它来查找变量。

- 如果在一个 function 之前使用 new，则会创建一个新的对象，该 funtion 也会被调用，而 this 的值是新创建的那个对象。

```
function User(name) {
    this.name = name
};
var user1 = new User("Alex");
```

调用 new User("Alex") 会创建一个新的对象，以 user1 来引用，User 这个 function 也会被调用，会在 user1 这个对象中设置名为 name 的属性，其值是 Alex。

- 如果通过 function 的 apply 和 call 方法来指定它被调用时的 this 的值，那么 apply 和 call 的第一个参数都是要指定的 this 值，两者不同的是调用的实际参数在 apply 中是

以数组的形式作为第二个参数传入的，而在 call 中除了第一个参数外其他参数都是调用的实际参数。

JavaScript 中并没有 Java 或 C++ 中的类的概念，而是采用构造器的方式来创建对象。在 new 表达式中使用构造器就可以创建新的对象。由构造器创建出来的对象有一个隐含的引用指向该构造器的 prototype。

所有的构造器都是对象，但并不是所有的对象都能成为构造器。能作为构造器的对象必须实现隐含的 Construct 方法。如果 new 操作符后面的对象并不是构造器，会抛出 TypeError 异常。

new 操作符会影响 function 调用中 return 语句的行为。当调用 function 时有 new 作为前缀，如果返回的结果不是一个对象，那么新创建的对象将会被返回。

```
function user(name) {
    this.name = name;
}
function anotherUser(name) {
    this.name = name;
    return {
        "badName" : name
    };
}
var u1 = new User("Alex");
alert( typeof u1.name);          // string
var u2 = new anotherUser("Alex");
alert( typeof u2.name);          // undefined
alert( typeof u2.badName);       // string
```

在上面代码中，函数 anotherUser 通过 return 语句返回了一个对象，因此 u2 引用的是返回的那个对象；而函数 user 并没有使用 return 语句，因此 u1 引用的是新创建的 User 对象。

建议 88：this 是动态指针，不是静态引用

this 是一个动态指针。在 JavaScript 中，类似指针特性的标识符还有以下 3 个。

- callee：函数的参数集合包含的一个静态指针，它始终指向参数集合所属的函数。
- prototype：函数包含的一个半静态指针，在默认状态下它始终指向函数附带的原型对象，不过可以改变这个指针，使它指向其他对象。
- constructor：对象包含的一个指针，它始终指向创建该对象的构造函数。

this 所指向的对象是由 this 所在的执行域决定的，而不是由 this 所在的定义域决定（如图 4.6 所示）。this 是 JavaScript 执行作用域的一个属性，它的指针始终指向当前调用对象。

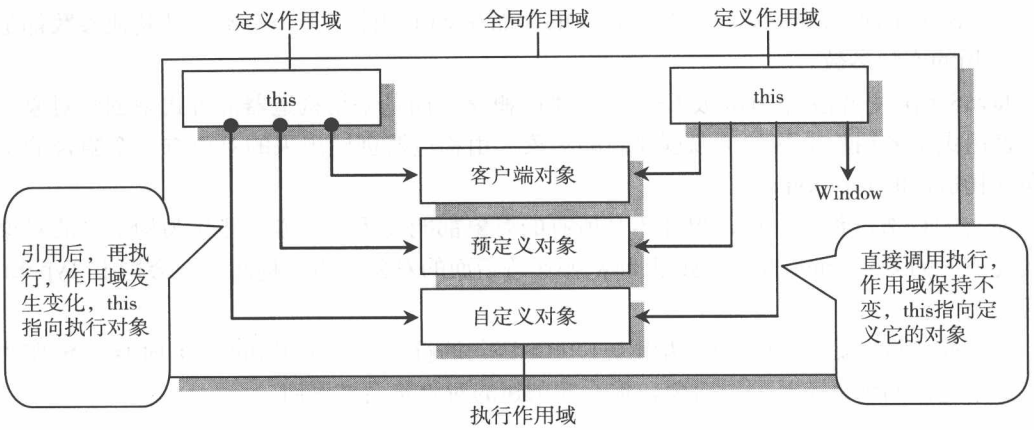


图 4.6 this 指针的变化示意图

JavaScript 中的函数可以在多个地方被引用，而且这种引用是在执行时才确定的。因此，JavaScript 方法的灵活性注定了 this 指针只能在运行环境中动态地确定。因此，只有当 this 被最后执行时才能够准确确定它所指代的对象。

在 JavaScript 中，闭包扮演着非常重要的角色，它能够改变 this 的指向。例如，在下面这个示例中，在一个对象中（如 o1）引用或调用另一个对象（如 o）的方法时，该方法中的 this 指针是变化的。当引用对象 o 的方法 f() 时，它的 this 就会根据执行对象而定，但在调用对象 o 的方法 f() 时，它的 this 还是指向原定义的对象 o。

```
var name = "this = window";
var o ={
    name : "this = o",
    f : function(){
        return this;
    }
};
var o1 = {
    name : "this = o1",
    f : o.f;        // 引用对象 o 中的方法 f()
}
var a = o1.f();
alert(a.name);     // this 实际指向对象为 o1
```

下面尝试把对象 o 的方法 f() 封装在闭包中，然后再进行引用：

```
var o1 = {
    name : "this = o1",
    f : function(){        // this 使用闭包封装方法的引用
        return o.f;
    }
}
var a = o1.f()();
alert(a.name);             // this 实际指向对象为 Window
```

方法 f() 中的 this 既不指向对象 o，也不指向对象 o1，而是指向对象 Window。因为在执行对象 o1 的方法 f() 时，仅指定了闭包运行的上下文环境，即执行闭包的对象为 o1，但是闭包内“包裹”的方法 f() 并没有被执行，当再次执行闭包内的方法时，执行环境已经发生了变化，执行对象从 o1 变为全局对象 Window，所以 this 就指向了 Window。

call 和 apply 方法能够强制改变函数的执行作用域，它们会破坏函数引用和调用的一般规律，因此使用这两个方法可以强制地指定 this 的指代对象。异步调用也会破坏 this 指针应用的一般规律，这是因为函数在被传递给定时器或事件处理函数时才被调用，这破坏了函数的上下文运行环境。下面结合代码具体进行说明：

（1）指向当前 DOM 对象

下面这个按钮定义了一个单击事件属性，其中包含了 this 关键字。

```
<input type="button" value=" 主人是谁？ " onclick = "this.value =' 是我呀，我就是主人 '" />
```

其中，onclick 事件属性中包含的 this 就代表当前 DOM 对象 input。

（2）指向构造函数的实例对象

定义一个构造函数，在其中使用 this 关键字作为临时代表，然后使用 new 运算符实例化构造函数。

```
function F(){
    this.name = " 我就是主人 ";
}
var f = new F();
alert(f.name);
```

这里的 this 就代表当前实例对象 f。

（3）指向当前对象直接量

下面是一个对象直接量，它包含了两个属性，其中方法 me() 返回关键字 this。

```
var o = {
    name : " 我是对象 o",
    me : function(){
        return this;
    }
}
var who = o.me();
alert(who.name);  // 读取 this 所代表对象的属性 name，返回字符串 " 我是对象 o"
```

在调用对象直接量 o 的方法 me() 后，变量 who 的值就是 this 的值，它代表当前对象直接量 o，然后读取对象 o 的属性 name，返回字符串 " 我是对象 o"。

（4）指向全局对象

在函数 f() 中调用 this 关键字，并且为 this 定义并初始化一个属性 name，在调用函数之后，可以直接读取属性 name。

```
function f(){
    this.name = "我是谁？";
}
f();        // 调用方法 f()
alert(name);        // 直接读取属性 name，返回值为 "我是谁？"
```

在函数 f() 中，this 实际上就是代表 window。实际上，上面代码可以写成如下形式：

```
window.f = function(){
    this.name = "我是谁？"
}
window.f();
alert(window.name);
```

（5）指向当前作用域对象

this 关键字并不总是代表当前对象，下面这个示例能够很好地说明这个问题。

```
<input type="button" value="主人是谁？" onclick =
"this.value ='是我呀，我就是主人'" />
<input type="button" value="主人是谁？" onclick = "f()" />
<script language="javascript" type="text/javascript">
function f(){
    this.value = "我在哪儿？";
}
</script>
```

上面示例中的第一个按钮的事件属性中包含的 this 就指向当前对象。但由于在第二个按钮的鼠标单击事件属性中以调用的方式调用全局作用域中的函数 f()，所以其中的 this 就代表 Window 对象，而不是当前按钮（第二个按钮）。

要是改变函数的用法，先在脚本中获取第二个按钮对象的引用，然后把函数 f() 作为一个值传递给对象的 onclick 事件属性，代码如下：

```
<input type="button" value="主人是谁？" />
<script language="javascript" type="text/javascript">
var btn2 = document.getElementsByTagName("input")[1];
btn2.onclick = f;
function f(){
    this.value = "我回来了";
}
</script>
```

在上面的示例中，因为 this 关键字就表示按钮对象本身，所以单击按钮之后就能够改变按钮的值。这也说明在将函数赋值给按钮对象之后，虽然函数的定义作用域没有发生变化，但它的执行作用域已经从全局作用域变为对象作用域，因此，this 关键字所代表的对象也会随之发生变化。

如果以同样的方式把该函数作为事件处理函数赋值给不同的按钮对象，那么 this 会分别代表不同的按钮对象。这也说明如果改变函数的执行作用域，那么函数所包含的 this 关键字也会指向不同的对象。

下面这个示例可以更好地说明作用域对于 this 关键字的影响。

```
function f(){
    return this;
}
var o ={
    name : "对象o",
    me : f,
    o1 : {
        name : "对象o1",
        me : f,
        o2 : {
            name : "对象o2",
            me : f
        }
    }
}
var who = o.o1.o2.me();
alert(who.name);          //字符串"对象o2"，说明当前this代表对象o2
var who = o.o1.me();
alert(who.name);          //字符串"对象o1"，说明当前this代表对象o1
var who = o.me();
alert(who.name);          //字符串"对象o"，说明当前this代表对象o
```

首先，定义函数 f()，设置其返回值为 this 关键字，然后把该函数作为值传递给不同作用域中的属性 me，最后分别调用它们，这时会发现 this 所代表的对象是不同的。

但是，如果不是以值的方式传递函数 f()，而是直接调用，则会发现其包含的 this 都代表 Window 对象。也就是说，this 的执行作用域并没有发生变化，仍然根据定义它的作用域来确定，即全局作用域。

建议 89：正确应用 this

this 总是指向当前作用域对象，如果当前定义对象的作用域没有发生变化，则 this 会指向当前对象。但是，this 关键字的用法比较灵活，这也在一定程度上干扰了用户对于它所代表对象的准确判定。另外，this 关键字可以存在于任何位置，并不局限于对象的方法内，还可以被应用在全局域内、函数内，以及其他特殊上下文环境中。

（1）函数的引用和调用

函数的引用和调用分别表示不同的概念，虽然它们都无法改变函数的定义作用域，但是引用函数能够改变函数的执行作用域，而调用函数是不会改变函数的执行作用域的。继续上面的示例进行说明：

```
var o = {
    name : "对象o",
    f : function(){
        return this;
```

```
    }
}
o.o1 = {
    name : "对象o1",
    me : o.f       // 引用对象o的方法f
}
```

可以看到，函数中的 this 所代表的是当前执行域对象 o1：

```
var who = o.o1.me();
alert(who.name);  // 字符串"对象o1"，说明当前this代表对象o1
```

如果把对象 o1 的 me 属性值改为函数调用：

```
o.o1 = {
    name : "对象o1",
    me : o.f()     // 调用对象o的方法f
}
```

则会看到，函数中的 this 所代表的是定义函数时所在的作用域对象 o：

```
var who = o.o1.me;
alert(who.name);  // 字符串"对象o"，说明当前this代表对象o
```

（2）call 和 apply

call 和 apply 方法可以直接改变被执行函数的作用域，使其作用域指向所传递的参数对象。因此，函数中包含的 this 关键字也指向参数对象。

```
function f(){
    // 如果当前执行域对象的构造函数等于当前函数，则表示this为实例对象
    if(this.constructor == arguments.callee) alert("this = 实例对象");
    // 如果当前执行域对象等于window，则表示this为Window对象
    else if (this == window) alert("this = window对象");
    // 如果当前执行域对象为其他对象，则表示this为其他对象
    else alert("this == 其他对象 \n this.constructor = " + this.constructor );
}
f();       // this指向Window对象
new f();   // this指向实例对象
f.call(1);         // this指向数值实例对象
```

在这个示例中，当直接调用函数 f() 时，因为函数的执行作用域为全局域，所以 this 代表 window。当使用 new 运算符调用函数时，将创建一个新的实例对象，函数的执行作用域为实例对象所在的上下文，因此，this 就指向这个新创建的实例对象。

而在使用 call() 方法执行函数 f() 时，call 会把函数 f() 的作用域强制修改为参数对象所在的上下文。由于 call() 方法的参数值为数字 1，因此 JavaScript 解释器会把数字 1 强制封装为数值对象，此时 this 就会指向这个数值对象。

再看一个很有趣的用法。在下面这个示例中，call() 方法把函数 f() 强制转换为对象 o 的一个方法并执行，这样函数 f() 中的 this 就指代对象 o，this.x 的值就等于 1，而 this.y 的值就

等于 2，结果就返回 3。

```
function f(){      // 函数 f()
    alert(this.x + this.y);
}
var o = {          // 对象直接量
    x : 1,
    y : 2
}
f.call(o);         // 执行函数 f()，返回值为 3
```

（3）原型继承

JavaScript 通过原型模式实现类的延续和继承，如果父类的成员中包含了 this 关键字，当子类继承了父类的这些成员时，this 的指向就会很迷惑人。

在一般情况下，子类继承父类的方法后，this 可能指向子类的实例对象，也可能指向子类的原型对象，而不是子类的实例对象。例如：

```
function Base(){           // 基类
    this.m = function(){   // 基类的方法 m()
        return "Base";
    };
    this.a = this.m();     // 基类的属性 a，调用当前作用域中的 m() 方法
    this.b = this.m();     // 基类的方法 b()，引用当前作用域中的 m() 方法
    this.c = function(){   // 基类的方法 c()，以闭包结构调用当前作用域中的 m() 方法
        return this.m();
    }
}
function F(){      // 子类
    this.m = function(){
        return "F"
    }
}
F.prototype = new Base();          // 继承基类
var f = new F();   // 实例化子类
alert(f.a);        // 字符串 "Base"，说明 this.m() 中 this 指向 F 的原型对象
alert(f.b());      // 字符串 "Base"
alert(f.c());      // 字符串 "F"，说明 this.m() 中 this 指向 F 的实例对象
```

在上面的示例中，基类 Base 包含 4 个成员，其中成员 b 和 c 以不同方式引用当前作用域内的方法 m()，而成员 a 存储着当前作用域内的方法 m() 的调用值。在将这些成员继承给子类 F 后，其中 m、b 和 c 成为原型对象的方法，而 a 成为原型对象的属性。但 c 的值为一个闭包体，当在子类的实例中调用它时，实际上它的返回值已经成为实例对象的成员，也就是说，闭包体在哪里被调用，其中包含的 this 就会指向哪里。所以，可以看到 f.c() 中的 this 指向实例对象，而不是 F 类的原型对象。

为了避免因继承关系而影响父类中 this 所代表的对象，除了通过上面介绍的方法把函数的引用传递给父类的成员外，还可以为父类定义私有函数，然后再把它的引用传递给其他父类成员，这样就避免了由于函数闭包的原因而改变 this 的值。例如：

```
function Base(){
    var _m = function(){          // 定义基类的私有函数 _m()
        return "Base";
    };
    this.a = _m;
    this.b = _m();
}
```

这样基类的私有函数 _m() 就具有完全隐私性，外界其他任何对象都无法直接访问基类的私有函数 _m()。因此，在一般情况下，在定义方法时，对于相互依赖的方法，可以把它定义为私有函数，并且以引用函数的方式对外公开，这样就避免了外界对于依赖方法的影响。

（4）异步调用

异步调用就是通过事件机制或计时器来延迟或调整函数的调用时间和时机。因为调用函数的执行作用域不再是原来的定义作用域，所以函数中的 this 总是指向引发该事件的对象。例如：

```
<input type="button" value="Button" />
<script language="javascript" type="text/javascript">
var button = document.getElementsByTagName("input")[0];
var o ={};
o.f = function(){
    if(this == o) alert("this = o");
    if(this == window) alert("this = window");
    if(this == button) alert("this = button");
}
button.onclick = o.f;
</script>
```

根据上面的讲解可知，这里的方法 f() 所包含的 this 不再指向对象 o，而是指向按钮 button，因为它是被传递给按钮的事件处理函数之后再被调用的。由于函数的执行作用域发生了变化，所以不再指向定义方法时所指定的对象。

如果使用 DOM 2 级标准为按钮注册事件处理函数：

```
if(window.attachEvent){   // 兼容 IE
    button.attachEvent("onclick", o.f);
}
else{       // 兼容符合 DOM 标准的浏览器
    button.addEventListener("click", o.f, true);
}
```

则会看到，在 IE 中，this 指向 window 和 button，而在符合 DOM 标准的浏览器中仅指向 button。在 IE 中，attachEvent() 是 Window 对象的方法，在调用该方法时，执行作用域为全局作用域，所以 this 会指向 window。同时由于该方法被注册到按钮对象上，因此它的真正执行作用域应该为 button 对象所在的上下文。这一点可以在符合 DOM 标准的浏览器中看到。这种解释可能很勉强，但在 IE 中 this 同时指向 Window 和 Button 对象本身。

为了解决这个问题，可以借助 call() 或 apply() 方法强制在对象 o 上执行 f() 方法，也就

是说，强制改变 f() 方法的执行作用域，避免因为环境的不同而影响函数作用域的变化。代码如下：

```
if(window.attachEvent){
    button.attachEvent("onclick", function(){// 以闭包的形式封装 call() 方法来强制执行 f()
        o.f.call(o);
    });
}
else{
    button.addEventListener("click", function(){
        o.f.call(o);
    }, true);
}
```

这样，当再次预览时，其中包含的 this 关键字始终指向对象 o，也就是说，f() 方法的执行作用域始终与它的定义作用域保持一致。

（5）定时器

异步调用的另一种形式就是使用定时器来调用函数。定时器就是通过调用 Window 对象的 setTimeout() 或 setInterval() 方法来延期调用函数。例如，可以这样来设计延期调用方法 o.f()。

```
var o ={};
o.f = function(){
    if(this == o) alert("this = o");
    if(this == window) alert("this = window");
    if(this == button) alert("this = button");
}
setTimeout(o.f, 100);
```

经测试发现，在 IE 中，this 指向 Window 和 Button 对象，具体原因与上面讲解的 attachEvent() 方法相同。但是，在符合 DOM 标准的浏览器中，this 指向 Window 对象，而不是 Button 对象，因为 setTimeout() 方法是在全局作用域中被执行的，所以 this 自然指向 Window 对象。要解决这个问题，仍然可以使用 call() 或 apply() 方法来实现：

```
setTimeout(function(){
    o.f.call(o);
}, 100);
```

建议 90：预防 this 误用的策略

this 虽然用法灵活，但容易出错。好的使用习惯可以很好地预防 this 误用。事实上，this 的复杂性在很大程度上取决于用户的使用方式。由于 this 指代灵活，如果把它放在复杂的应用环境中，它也会变得很不确定，因此必须牢记：确保在同一作用域中操作 this，避免把包含有 this 的全局函数动态用于局部作用域中，同时应避免在不同作用域的对象之间相互引用

包含 this 的方法。

如果把 this 作为参数值来调用函数，那么可以避免了 this 多变的问题，因为 this 始终与当前对象保持一致。例如，下面的做法是错误的，因为 this 在这里始终指向 Window 对象，而不是所期望的当前按钮对象：

```
<input type="button" value="按钮 1" onclick="f()" />
<input type="button" value="按钮 2" onclick="f()" />
<input type="button" value="按钮 3" onclick="f()" />
<script language="javascript" type="text/javascript">
function f(){
    alert(this.value);
}
</script>
```

但是，如果把 this 作为参数值进行传递，那么它就会代表当前对象：

```
<input type="button" value="按钮 1" onclick="f(this)" />
<input type="button" value="按钮 2" onclick="f(this)" />
<input type="button" value="按钮 3" onclick="f(this)" />
<script language="javascript" type="text/javascript">
function f(o){
    alert(o.value);
}
</script>
```

如果要确保构造函数的方法在初始化之后其中所包含的 this 指针不再发生变化，一个很简单的方法就是：在构造函数中把 this 指针存储在私有变量中，然后在方法中使用私有变量来引用 this 指针，这样所引用的对象始终都是初始化的实例对象，而不会在类型继承中发生变化。例如：

```
function Base(){            // 基类
    var _this = this;       // 存储初始化时对象的引用指针
    this.m = function(){
        return _this;       //初始化时对象的引用指针
    };
    this.name = "Base";
}
function F(){       // 子类
    this.name = "F";
}
F.prototype = new Base();           // 继承基类
var f = new F();
var n = f.m();
alert(n.name);      // this 始终指向原型对象，而不再是子类的实例对象
```

对于对象直接量来说，如果希望使用 this 代表当前对象直接量，则可以直接调用对象直接量的名称，而不用 this 关键字。

当然，作为一个动态指针，this 也是可以被转换为静态指针的，实现的方法是主要利用

Function 对象的 call() 或 apply() 方法。这两个方法都可以强制指定 this 的指代对象。例如，为 Function 对象扩展一个原型方法 pointTo()，具体代码如下：

```
// 把this转换为静态指针，参数o表示预设置this所指代的对象，返回一个闭包函数
Function.prototype.pointTo = function(o){
    var _this = this;      // 存储当前函数对象
    return function(){     // 一个闭包函数
        return  _this.apply(o, arguments); /* 执行当前函数并把当前函数的作用域强制设置为
                                              指定对象 */
    }
}
```

这个方法将调用当前函数，并在由参数指定的对象上执行，从而把 this 绑定到该对象上。

然后应用这个函数扩展方法，以实现强制指定对象 o 的方法 b() 中的 this 始终指向定义对象 o。

```
var o ={
    name : "this = o"
}
o.b = (function(){
    return this;
}).pointTo(o);     // 把this绑定到对象o身上
var o1 ={
    name : "this = o1",
    b : o.b
}
var a = o1.b();
alert(a.name);     // 字符串 "this=o"，说明this的值没有发生变化
```

还可以扩展 new 运算符的替代方法，从而间接使用自定义函数实例化类。

```
// 把构造函数转换为实例对象，参数f表示构造函数，返回构造函数f()的实例对象
function instanceFrom(f){
    var a = [].slice.call(arguments, 1); // 获取构造函数的参数
    f.prototype.constructor = f;          // 手工设置构造函数的原型构造器
    f.apply(f.prototype, a);      // 在原型对象上强制执行构造函数
    return f.prototype;   // 返回原型对象
}
```

例如，下面的示例演示了如何使用这个自定义的实例化类方法把一个简单的构造函数转换为具体的实例对象。

```
function F(){
    this.name = "F";
}
var f = instanceFrom(F);
alert(f.name);
```

通过这个示例也进一步说明，call() 和 apply() 方法具有强大的功能，它不仅能够执行普通函数，也能够实例化构造函数，具备 new 运算符的运算功能。

建议 91：推荐使用构造函数原型模式定义类

JavaScript 中定义类型的方式有多种，这也形成了不同的类型模式。

（1）工厂模式

工厂模式是指通过函数把一个类型实例包装起来，这样可以通过调用函数来实现类型的实例化。

```
function wrap(title,pages){
    var book = new Object();
    book.title = title;
    book.pages = pages;
    book.what = function(){
        alert(this.title +this.pages)
    }
    return book;  // 初始化后的对象
}
```

也可以对该模式进行优化，进而消除重复创建相同函数的弊端，节约大量资源。

```
what = function(){
    alert(this.title +this.pages)
}
function wrap(title,pages){
    var book = new Object();
    book.title = title;
    book.pages = pages;
    book.what = what;
    return book;
}
```

工厂模式只是一种伪装的构造函数，不推荐使用。

（2）构造函数模式

在 JavaScript 中，构造函数具有如下特性：

- 构造函数使用 new 运算符进行调用。
- 在构造函数内部，this 关键字指代当前实例对象。
- 在构造函数内部，必须通过点运算符来声明和引用成员。构造函数的结构体内可以包含一般函数的执行语句。

例如，对于下面这个构造函数 Box()，传递给 Box() 的是一个新创建的空对象，该对象是高度抽象但未知的，通过 this 关键字来代称，this 的值就是这个新创建的空对象引用。当使用 new 运算符实例化构造函数时，可以通过传递参数来初始化这个对象的属性值。

```
function Box(w,h){          // 构造函数
    this.w = w;
    this.h = h;
}
var box1 = new Box(4,5);  // 实例并初始化构造函数
```

由于每一个构造函数代表一种类型，为了与普通函数进行区分，函数名应该很直观，并且首字母要大写（非强制的）。如果构造函数返回对象，那么被返回的对象将覆盖 this 的值。例如：

```
function Box(w,h){          // 构造函数
    this.w = w;
    this.h = h;
    return this;
}
```

（3）原型模式

先声明一个构造函数，并且利用构造函数的 prototype 属性为该构造函数定义原型属性 title 和 pages，以及原型方法 what()。构造函数的原型成员将会被所有实例对象继承，这样当使用 new 运算符实例化对象时，所有对象都拥有原型属性中定义的成员。

```
function Book(){  // 空类
}
Book.prototype.title = "Javascript 设计方法 ";
Book.prototype.pages = 200;
Book.prototype.what = function(){
    alert(this.title +this.pages);
};
```

从语义的角度分析，通过原型继承的方式，实现了在前面两种模式中将对象与其方法分离的设计思想。使用 instanceof 运算符能够方便地检测对象实例的类型。原型模式存在以下两个问题：

- 由于构造函数已被事先声明，而原型属性在类结构声明之后才被定义，因此无法通过构造函数参数向原型属性动态传递值。这样所带来的后果：由该类实例化的所有对象都是一个“模样”，没有“个性”。如果改变原型属性值，那么所有实例都受到干扰。这是非常严重的问题，如果无法解决，该项研究将无果而终。
- 当原型属性的值为引用类型数据时，如果在一个对象实例中修改该属性值，将会影响所有的实例。由于原型属性 x 的值为一个引用类型数据，因此所有对象实例的属性 x 的值都是指向该对象的引用指针。一旦某个对象的属性值被改动，其他实例对象的属性值也会随着发生变化。

（4）构造函数原型模式

构造函数原型模式是建立在原型模式基础上的一种混合设计模式，将构造函数模式与原型模式混合使用。对于可能会相互影响且希望动态传递参数的属性，将其拆分出来使用构造函数模式进行设计。而对于不需要个性、希望共享，并且又不会相互影响的方法或属性，单独使用原型模式来设计。

```
function Book(title,pages){       // 构造函数模式设计
    this.title = title;
```

```
    this.pages = pages;
}
Book.prototype.what = function(){          // 原型模式设计
    alert(this.title +this.pages);
};
```

在混合使用构造函数与原型模式时，可以不使用构造函数来定义对象的所有非函数属性（即对象属性），而使用原型模式来定义对象的函数属性（即对象方法）。这样所有方法都只创建一次，而每个对象都能够根据需要自定义属性值。这种混合型模式成为 ECMAScript 定义类的推荐标准，这也是使用最广的一种设计模式，它具有前面 3 种设计模式的所有优点，而且去除了它们的副作用。

遵循面向对象的设计原则，类的所有成员都应该封装在类结构体内，因此可以进一步优化构造函数原型模式，从而产生动态原型模式。优化的思路：使用条件结构封装该原型方法，判断原型方法是否存在，如果存在，则不再创建该方法，否则就创建该方法。

```
function Book(title,pages){
    this.title = title;
    this.pages = pages;
    if(typeof Book.isLock == "undefined"){ // 创建原型方法的锁，如果不存在该方法则创建
        Book.prototype.what = function(){
            alert(this.title +this.pages);
        };
        Book.isLock = true;        // 创建原型方法后，把锁锁上，避免重复创建
    }
}
```

“typeof Book.isLock”表达式能够检测该属性值的类型，如果返回 undefined 字符串，则表示不存在该属性值，说明还没有创建原型方法，允许进入分支结构创建原型方法，并设置该属性的值为 true，这样它的类型返回值就是 boolean 字符串。

动态原型模式与构造函数原型模式在性能上是等价的，不过目前使用最广泛的是构造函数原型混合模式。

建议 92：不建议使用原型继承

原型继承是一种简化的继承机制，也是目前最流行的一种 JavaScript 继承方式。实际上 JavaScript 就是一种基于原型的语言。在原型继承中，类和实例概念被淡化了，一切都从对象的角度来考虑。

但是，原型继承就不再需要使用类来定义对象的结构，直接定义对象，该对象被其他对象引用，这样就形成了一种继承关系，其中引用对象称为原型对象（prototype object）。JavaScript 能够根据原型链来查找对象之间的这种继承关系。下面就是一个简单的原型继承示例：

```
function A(x){     // A类
    this.x1= x;
    this.get1 = function(){
        return this.x1;
    }
}
function B(x){     // B类
    this.x2 = x;
    this.get2 = function(){
        return this.x2 + this.x2;
    };
}
B.prototype = new A(1);   // 用原型对象继承A的实例
function C(x){     // C类
    this.x3 = x;
    this.get3 = function(){
        return this.x2 * this.x2;
    };
}
C.prototype = new B(2);   // 用原型对象继承B的实例
```

在上面示例中，分别定义了 3 个函数，然后通过原型继承方法把它们连在一起，这样 C 能够继承 B 和 A 的成员，B 能够继承 A 的成员。prototype 的最大特点就是能够允许对象实例共享原型对象的成员。因此，如果把某个对象作为一个类型的原型，那么就可以说这个类型的实例以这个对象为原型。实际上此时这个对象的类型也可以作为那些以这个对象为原型的实例的类型。此时，可以在 C 的实例中调用 B 和 A 的成员：

```
var b = new B(2);
var c = new C(3);
alert(b.x1);       // 在实例对象b中调用A的属性x1，返回1
alert(c.x1);       // 在实例对象c中调用A的属性x1，返回1
alert(c.get3());   // 在实例对象c中调用C的方法get3()，返回9
alert(c.get2());   // 在实例对象c中调用B的方法get2()，返回4
```

基于原型的编程是面向对象编程的一种特定形式。在这种编程模型中，不需要声明静态类，而可以通过复制已经存在的原型对象来实现继承关系。因此，基于原型的模型没有类的概念，原型继承中的类仅是一种模拟。原型继承非常简单，其优点是结构简单，不需要每次构造函数都调用父类的构造函数，并且不需要通过复制属性的方式就能快速地实现继承。但它也存在以下几个缺点：

- 由于每个类型只有一个原型，因此它不直接支持多重继承。
- 不能很好地支持多参数或动态参数的父类。在原型继承阶段，用户还不能决定以什么参数来实例化构造函数。
- 使用不够灵活。用户需要在原型声明阶段实例化父类对象，并且把它作为当前类型的原型，这会限制父类实例化的灵活性，很多时候无法确定父类对象实例化的时机和场合。

建议 93：推荐使用类继承

类继承也称为构造函数继承，还称为对象模拟法。其表现形式：在子类中执行父类的构造函数。其实现本质：构造函数也是函数，与普通函数相比，它只不过是一种特殊结构的函数而已。可以将一个构造函数（如 A）的方法赋值为另一个构造函数（如 B），然后调用该方法，使构造函数 A 在构造函数 B 内部被执行，这时构造函数 B 就拥有在构造函数 A 中定义的属性和方法，这就是所谓 B 类继承 A 类。下面看一个示例：

```
function A(x){                  // 构造函数 A
    this.x = x;
    this.say = function(){
        alert(this.x);
    }
}
function B(x,y){                // 构造函数 B
    this.m = A;                 // 把构造函数 A 作为一个普通函数引用给临时方法 m()
    this.m(x);                  // 把当前构造函数参数 x 作为值传递给构造函数 A，并执行
    delete this.m;              // 清除临时方法
    this.y = y;
    this.call = function(){
        alert(this.y);
    }
}
var a = new A(1);
var b = new B(2,3);
a.say();                        // 调用实例化 A 的方法 say()，返回 1
b.say();                        // 在 B 类中调用 A 类的方法 say()，返回 2，说明继承成功
b.call();                       // 调用实例化 B 的方法 call()，返回 3
```

构造函数能够使用 this 关键字为所有属性和方法赋值。在默认情况下，关键字 this 引用的是构造函数当前创建的对象。不过在这个方法中，this 不是指向当前正在使用的实例对象，而是调用构造函数的方法所属的对象，即构造函数 B。此时，构造函数 A 已经不是构造函数了，而被视为一个普通可执行函数。

上面的示例演示了类继承的实现基础。实际上，在复杂的编程中，是不会使用上面方法来定义类继承的，因为它的设计模式太随意，缺乏严密性。严谨的设计模式应该考虑到各种可能存在的情况和类继承关系中的相互耦合性。为了更直观地说明，先看一个示例：

```
function A(x){      // 构造函数 A
    this.x = x;
}
A.prototype.getx = function(){
    return this.x;
}
```

在上面的代码中，先创建一个构造函数，它相当于一个类，类名是构造函数的名称 A。在结构体内使用 this 关键字创建本地属性 x。方法 getx() 被放在类的原型对象中成为公共方

法。然后，借助 new 运算符调用构造函数，返回的是新创建的对象实例：

```
var a1 = new A(1);
```

最后，对象 a1 就可以继承类 A 的本地属性 x，也有人称之为实例属性，当然还可以访问类 A 的原型方法 getx()：

```
alert(a1.x);        // 继承类 A 的属性 x
alert(a1.getx()); // 引用类 A 的方法 getx()
```

上面的代码是一个简单的类的演示。现在，创建一个类 B，让其继承类 A，实现的代码如下：

```
function B(x,y){            // 构造函数 B
    this.y = y;
    A.call(this,x);         // 在构造函数 B 中调用超类 A，实现绑定
}
B.prototype = new A();      // 设置原型链，建立继承关系
B.prototype.constructor = B;        // 恢复 B 的原型对象的构造函数为 B
B.prototype.gety = function(){
    return this.y;
}
```

在构造函数 B 的结构体内，使用函数 call() 调用构造函数 A，把 B 的参数 x 传递给调用函数。让 B 能够继承 A 的所有属性和方法，即执行“A.call(this,x);”语句行。在构造函数 A 和 B 之间建立原型链，即执行“B.prototype = new A();”语句行。恢复 B 的原型对象的构造函数，即执行“B.prototype.constructor = B;”语句行。当定义构造函数时，其原型对象（prototype 属性值）默认是一个 Object 类型的一个实例，其构造器（constructor 属性值）会被默认设置为该构造函数本身。如果改动 prototype 属性值，使其指向另一个对象，那么新对象就不会拥有原来的 constructor 属性值，所以必须重新设置 constructor 属性值。

此时，就可以在子类 B 的实例对象中调用超类 A 的属性和方法了。

```
var f2 = new B(10,20);
alert(f2.getx());           //10
alert(f2.gety());           //20
```

最后，看一个更复杂的多重继承的实例。

```
// 基类 A
function A( x ){
    this.getl = function(){
        return x;
    }
}
A.prototype.has = function(){
    return ! ( this.getl() == 0 );
}
// 超类 B
```

```
function B(){
    var a = [];
    a = Array.apply(a, arguments);
    A.call(this, a.length );
    this.add = function(){
        return a.push.apply(a, arguments);
    }
    this.geta = function(){
        return a;
    }
}
B.prototype = new A();              //建立原型链
B.prototype.constructor = B;        // 恢复构造器
B.prototype.str = function(){
    return this.geta().toString();
}
// 子类C
function C(){
    B.apply(this, arguments);       // 在当前对象中调用B类
    this.sort = function(){
        var a = this.geta();
        a.sort.apply(a, arguments);
    }
}
C.prototype = new B();              //建立原型链
C.prototype.constructor = C;        // 恢复C类原型对象的构造器
// 超类B的实例继承类A的成员
var b= new B(1, 2, 3, 4);
alert(b.getl());                    //4
alert(b.has());                     //true
// 子类C的实例继承类B和类A的成员
var c = new C(30, 10, 20, 40);
c.add(6, 5);
alert(c.geta())                     // 数组30,10,20,40,6,5
c.sort()                            // 排序数组
alert(c.geta())                     // 数组10,20,30,40,5,6
alert(c.getl())                     //4
alert(c.has());                     //true
alert(c.str());                     //10,20,30,40,5,6
```

在上面的示例代码中，设计类C继承类B，而类B又继承了类A。A、B、C三个类之间的继承关系是通过在子类中调用父类的构造函数来维护的。例如，在C类中添加“B.apply(this, arguments);”语句，该行语句能够在B类中调用A类，并且把B的参数传递给A，从而使B类拥有A类的所有成员。同理，在B类中添加“A.call(this, a.length);”语句，该行语句把B类的参数长度作为值传递给A类，并进行调用，从而实现B类拥有A类的所有成员。

从继承关系上看，B类继承了A类的本地方法getl()。为了确保B类还能够继承A类的原型方法，还需要为它们建立原型链，从而实现原型对象的继承关系，方法是添加语句行“B.prototype = new A();”。同理，在C类中添加语句行“C.prototype = new B();”，这样就可

以把 A、B 和 C 三个类通过原型链连在一起，从而实现子类能够继承超类成员，甚至还可以继承基类成员。这里的成员主要指类的原型对象包含的成员，当然它们之间也可以通过相互调用来实现对本地成员的继承关系。

注意原型继承中的先后顺序，在为 B 类的原型指定 A 类的实例前，不能再为其定义任何原型属性或方法，否则就会被覆盖。如果要扩展原型方法，就只有在进行原型绑定之后，再定义扩展方法。

建议 94：建议使用封装类继承

在面向对象编程中，语言自身都有一套严格的封装机制，开发人员只是按惯性思维去开发具体的项目，很少关心封装问题，因为语言会自动完成基本的功能封装，当然具体应用的功能还需要程序人员自己去封装。但是，JavaScript 语言没有提供良好的封装机制，只能够依靠开发人员的方法来实现部分功能封装。类继承是在 JavaScript 程序开发中应用得比较广泛的继承模式，为了更方便地使用，建议读者对这种模式进行规范和封装，以便提高代码利用率。

首先，定义一个封装函数。设计入口为子类和超类对象，函数功能是子类能够继承超类的所有原型成员，不设计出口：

```
function extend(Sub,Sup){          // 类继承封装函数
    // 参数 Sub 表示子类，Sup 表示超类
}
```

在函数体内，首先定义一个空函数 F，用来实现功能中转。设计 F 的原型为超类的原型，然后把空函数的实例传递给子类的原型，这样就避免了直接实例化超类可能带来的系统负荷。在实际开发中，超类的规模可能会很大，进行实例化会占用大量内存。

恢复子类原型的构造器子类，同时，检测超类的原型构造器是否与 Object 的原型构造器发生耦合，如果是，则恢复它的构造器为超类自身。

```
function extend(Sub,Sup){          // 类继承封装函数
    var F = function(){};          // 定义一个空函数
    F.prototype = Sup.prototype; // 设置空函数的原型为超类的原型
    Sub.prototype = new F();       // 实例化空函数，并把超类原型引用传递给子类
    Sub.prototype.constructor = Sub;      // 恢复子类原型的构造器为子类自身
    Sub.sup = Sup.prototype;       // 在子类中存储超类原型，避免子类和超类耦合
    if(Sup.prototype.constructor == Object.prototype.constructor){ // 检测超类原型构
造器是否为自身
        Sup.prototype.constructor =Sup   // 类继承封装函数
    }
}
```

一个简单的功能封装函数就这样实现了。下面定义两个类，尝试把它们绑定为继承关系。

```
function A(x){     // 构造函数A
    this.x = x;
    this.get = function(){
        return this.x;
    }
}
A.prototype.add = function(){
    return this.x + this.x;
}
A.prototype.mul = function(){
    return this.x * this.x;
}
function B(x){     // 构造函数B
    A.call(this,x);        // 在函数体内调用构造函数A，实现内部数据绑定
}
extend(B,A);       // 调用类继承封装函数，把A和B的原型捆绑在一起
var f = new B(5);
alert(f.get())     //5
alert(f.add())     //10
alert(f.mul())     //25
```

在类继承封装函数中，有这样的语句“Sub.sup = Sup.prototype;”，在上面的代码中没有体现，为了理解它的价值，先看下面的代码：

```
extend(B,A);
B.prototype.add = function(){     // 为B类定义一个原型方法
    return this.x + "" + this.x
}
```

上面的代码是在调用封装函数之后再为 B 类定义了一个原型方法，该方法名与 A 类的原型方法 add 同名，但功能不同。如果此时测试程序，那么会发现子类 B 定义的原型方法 add() 将会覆盖超类 A 的原型方法 add()，如下：

```
alert(f.add())     // 字符串55，而不是数值10
```

在 B 类的原型方法 add() 中调用超类的原型方法 add()，从而避免代码耦合的现象发生：

```
B.prototype.add = function(){
    return B.sup.add.call(this);          // 在函数内部调用超类的方法add()
}
```

建议 95：慎重使用实例继承

对类进行实例化操作就会产生一个新的对象，这个实例对象将继承类的所有特性和成员，而实例继承正是对于这种实例化过程的一种概括。类继承和原型继承在客户端是无法继承 DOM 对象的，同时它们也不支持继承系统静态对象、静态方法等。为了更直观地说明此问题，不妨先看下面两个示例。

（1）使用类继承法继承 Date 对象

```
function D(){                          // 自定义构造函数
    Date.apply(this,arguments);        // 调用 Date 对象，对其进行引用，实现继承的目的
}
var d = new D();                       // 实例化自定义构造函数
alert(d.toLocaleString());             //[object Object]
```

上面的示例说明，使用类继承无法实现对静态对象的继承，这是因为系统对象的结构比较特殊，它不是简单的函数体结构，对声明、赋值和初始化等操作都进行了独立的封装，所以无法实现在自定义构造函数中的那种操作。

（2）使用原型继承法继承 Date 对象

```
function D(){                          // 自定义空构造函数
}
D.prototype = new Date();              // 把 Date 对象的实例赋值给 D 的原型对象
var d = new D();                       // 实例化 D
alert(d.toLocaleString());             // 错误提示
```

上面的示例说明了使用原型继承也无法实现对静态对象的继承。不过，使用实例继承法能够实现对所有 JavaScript 核心对象的继承。例如，在下面的示例中，把 Date 对象的实例化过程和方法调用封装在一个函数中，然后返回实例对象，这样就可以解决核心静态对象无法继承的问题。

```
function D(){                          // 封装函数
    var d = new Date();                // 实例化 Date 对象
    d.get = function(){                // 定义本地方法，间接调用 Date 对象的 toLocaleString() 方法
        alert(d.toLocaleString());
    }
    return d;                          // 实例对象
}
var d = new D();                       // 实例化封装函数
d.get();                               // 调用本地方法，返回当前本地的日期和时间
```

构造函数是一种特殊结构的函数，它没有返回值，通过 this 关键字来初始化实例对象。当然，在构造函数中可以增加 return 语句，为其设置一个返回值，这时返回值就是 new 运算符执行表达式的值。因此，在构造函数中完成对类的实例化操作，然后返回实例对象，这就是实例继承的由来。

- 使用实例继承法能够实现对所有对象的继承，包括自定义类、核心对象和 DOM 对象等。不过也应该清楚实例继承的缺点，它不是真正的继承机制，仅是一种模拟方法。
- 实例继承法无法传递动态参数。类的实例化操作是在封闭的函数体内实现的，不能够通过 call() 或 apply() 方法来传递动态参数。如果继承需要传递动态参数，那么这种继承就会带来很多不便。
- 实例继承只能够返回一个对象，与原型继承一样，不支持多重继承。
- 由于通过封装的方法把对象实例化和初始化操作都封装在一个函数体内，因此最后通

过对封装函数执行实例化操作来获取继承的对象。但这种做法无法真正实现继承对象是封装类的实例，它仍然保持与原对象的实例关系。

建议 96：避免使用复制继承

复制继承是最原始的方法，其设计思路：利用 for in 语句遍历对象成员，然后逐一将其复制给另一个对象，通过这种“蚂蚁搬家”的方式来实现继承关系。例如，在下面的示例中，先定义一个 F 类，它包含 4 个成员，然后将其实例化并把它的所有属性和方法都复制给一个空对象 o，这样对象 o 就拥有了 F 类的所有属性和方法。

```
function F(x,y){   // 构造函数 F
    this.x = x;
    this.y = y;
    this.add = function(){
        return this.x + this.y;
    }
}
F.prototype.mul = function(){
    return this.x * this.y;
}
var f = new F(2,3)
var o = {}
for(var i in f){   // 遍历构造函数的实例，把它的所有成员都赋值给对象 o
    o[i] = f[i];
}
alert(o.x);        //2
alert(o.y);        //3
alert(o.add());    //5
alert(o.mul());    //6
```

对于该复制继承法，可以将其封装，使其具有较大的灵活性。

```
Function.prototype.extend = function(o){ // 为 Function 扩展复制继承的方法
    for(var i in o){
        this.constructor.prototype[i] = o[i]; // 把参数对象成员复制给当前对象的构造函数原型对象
    }
}
```

上面的封装函数通过原型对象为 Function 核心对象扩展一个方法，该方法能够把指定的参数对象完全复制给当前对象的构造函数的原型对象。this 关键字指向的是当前实例对象，而不是构造函数本身，所以要为其扩展原型成员，就必须使用 constructor 属性来指向它的构造器，然后通过 prototype 属性指向构造函数的原型对象。

接下来，新建一个空的构造函数，并为其调用 extend() 方法，把传递进来的 F 类的实例对象完全复制为原型对象成员。注意，此时就不能够定义对象直接量，因为 extend() 方法只能为构造函数复制继承：

```
var o = function(){};
o.extend(new F(2,3));
```

复制继承法也不是真正的继承，它是通过反射机制复制类对象的所有可枚举属性和方法来模拟继承。这种方法能够实现模拟多继承。不过，它的缺点也很明显：

- 由于是反射机制，复制继承法不能继承非枚举类型的方法，系统核心对象的只读方法和属性也是无法继承的。
- 通过反射机制来复制对象成员的执行效率会非常差。对象结构越庞大，这种低效就表现得越明显。
- 如果当前类型包含同名成员，那么这些成员可能会被父类的动态复制所覆盖。
- 在多重继承的情况下，复制继承不能够清晰地描述父类与子类的相关性。
- 只有在类被实例化后，才能够实现遍历成员和复制成员，因此它不能够灵活支持动态参数。
- 由于复制继承法仅是简单地引用赋值，如果父类的成员值包含引用类型，那么用复制继承法继承后，与原型继承法一样副作用很多。

还可以对复制继承法进行适当优化，通过对象克隆方式来实现，这样就可以避免一个个复制对象成员所带来的低效率，具体方法如下：

首先，为 Function 对象扩展一个方法，该方法能够把参数对象赋值给一个空构造函数的原型对象，然后实例化构造函数并返回实例对象，这样该对象就拥有构造函数包含的所有成员，例如：

```
Function.prototype.clone = function(o){  // 对象克隆方法
    function Temp(){};                   // 新建空构造函数
    Temp.prototype = o;                  // 把参数对象赋值给该构造函数的原型对象
    return new Temp();                   // 实例化后的对象
}
```

然后，调用该方法来克隆对象。克隆方法返回的是一个空对象，不过它存储了指向给定对象的原型对象指针，这样就可以利用原型链来访问这些变量，从而在不同对象之间实现继承关系。例如：

```
var o = Function.clone(new F(2,3));      // 调用 Function 对象的克隆方法
alert(o.x);                              //2
alert(o.y);                              //3
alert(o.add());                          //5
alert(o.mul());                          //6
```

建议 97：推荐使用混合继承

混合继承是指把多种继承方法结合在一起使用，从而发挥各自优势，扬长避短，以实现各种复杂的应用。其中最常见的形式是把类继承与原型继承混合使用，以解决类继承中存在的问

题。下面比较一下类继承、原型继承、实例继承、复制继承和克隆继承之间的不同，见表 4.1。

表 4.1 继承方法综合比较

项目	类继承	原型继承	实例继承	复制继承	克隆继承
原型属性	不继承	继承	继承	继承	继承
本地成员	继承	不继承	继承	继承	继承
多重继承	支持	不支持	不支持	支持	不支持
多参数	支持	不支持	不支持	不支持	不支持
执行效率	高	高	中	低	低
instanceof	false	true	false	false	false

类继承与原型继承是两种截然不同的继承模式，它们生成的对象的行为方式也是不同的。面向对象的开发人员对于类继承比较熟悉，几乎所有使用面向对象的 JavaScript 应用都用到了这种继承模式，但是，因为 JavaScript 中的类继承仅仅是对真正基于类继承的一种模仿，所以深入理解 JavaScript 的开发人员应该懂得原型继承的工作机制。

原型继承更能节约内存。原型链读取成员的方式使得所有克隆出来的对象都共享一个实例，只有在直接设置了某个克隆出来的对象的属性和方法时，情况才会有所变化。而在类继承方式中，创建的每一个对象在内存中都有自己的一套属性和方法副本。原型继承比类继承显得更为简单。

```
function A(x,y){   // 构造函数A
    this.x = x;
    this.y = y;
}
A.prototype.add = function(){
    return this.x + this.y;
}
function B(x,y){
    A.call(this,x,y);      // 类继承实现
}
B.prototype = new A();     // 原型继承实现
var b = new  B(10,20);
alert(b.x);          //10
alert(b.y);          //20
alert(b.add());      //30
```

上面的示例把原型继承和类继承混用在一起，从而实现了一种比较完善的继承机制。

建议 98：比较使用 JavaScript 多态、重载和覆盖

在 JavaScript 中，加号是一个多态运算符，它能够根据传入值的类型进行不同的计算。

从某种意义上来说，多态是面向对象中重要的一部分，也是实施继承的主要目的。一个实例可以拥有多种类型，既可以是这种类型，也可以是那种类型，这种多类型称为类的多态。

多态表现为两个方面：类型的模糊和类型的识别。JavaScript 是一种弱类型语言，通过 typeof 运算符来判断值的类型，但通过 typeof 无法确定对象的类型，所有类型的实例对象对于 typeof 运算符来说都是基本的 object，因此 JavaScript 的类型是比较模糊的。由于没有严格的类型检测，因此可以为任何对象调用任何方法，无须考虑它是否被设计为拥有该方法。使用 JavaScript 的原型可以设计类的多态特性。

```
function A(){      // 超类 A
    this.get = function(){
        alert("A");
    }
}
function B(){      // 子类 B
    this.get = function(){
        alert("B");
    }
}
B.prototype = new A();    // 设置 B 类继承 A 类
function C(){      // 子类 C
    this.get = function(){
        alert("C");
    }
}
C.prototype = new A();    // 设置 C 类继承 A 类
function F(x){     // 多态类 F
    this.x =x;
}
F.prototype.get = function(){
    if(this.x instanceof A)        // 判断是否为超类的实例，然后调用不同类的方法
        this.x.get()
}
var b = new B();
var c = new C();
var f1 =new F(b);
var f2 =new F(c);
f1.get();  //B，此时该方法指向的是 B 类中的方法 get()
f2.get();  //C，此时该方法指向的是 C 类中的方法 get()
```

重载和覆盖是两个不同的类型概念，重载（overload）就是指同名方法有多个实现，依靠参数的类型或参数的个数来区分和识别它们。在 JavaScript 中，函数的参数是没有类型的，并且参数个数也是任意的。看下面的示例：

```
function f(x,y){
    return x+y;
}
```

示例中的函数 f() 虽然指定了两个形参，但是仍然可以在调用时传递任意多个实参，参

数的类型也可以是任意的。由于 JavaScript 语言是弱类型语言，不会根据传递的参数个数和类型来决定要执行的行为，因此，要定义重载方法，只能够通过 arguments 来实现。

```
function f(){
    var sum = 0;
    for( var i = 0; i < arguments.length; i ++ ){
        if(typeof arguments[i] == "number")
            sum += arguments[i];
    }
    return sum;
}
```

上面的函数实现了重载对任意多个参数求和的函数。不管函数 f() 中包含多少个参数，也不管参数类型如何，该函数将会自动把其中的数值类型的参数相加并返回总数。

```
alert(f( 3, 4, 6, 7, 8, 9 ));      // 重载函数 f()，返回 37
alert(f( 3, 4 ));                  // 重载函数 f()，返回 7
```

结合 instanceof 运算符和 constructor 属性来判断参数类型，并且根据参数个数和类型执行不同的操作，可以实现复杂的方法重载。

覆盖（overrid）是指在子类中定义的方法与超类中的方法同名，并且参数类型和个数也相同，当子类被实例化后，从超类中继承的同名方法将被隐藏。下面是一个简单的示例。

```
function A(){      // 超类 A
    this.m = function(){
        alert("A");
    }
}
function B(){      // 子类 B
    this.m = function(){
        alert("B");
    };
}
B.prototype = new A();     // 类 B 继承类 A
B.prototype.constructor = B;       // 恢复 B 类的原型对象的构造器
var b = new B();b.m();     // 字符 B，说明子类 B 的方法 m() 将覆盖类 A 的方法 m()
```

在强类型语言中，在覆盖的方法中可以调用被覆盖的方法（超类的方法），不过可以通过临时私有变量先保存超类的同名方法，然后在子类同名方法中调用即可。实现的代码如下：

```
function A(){
    this.m = function(){
        alert("A");
    }
}
function B(){
    var m = this.m;                // 先使用私有变量保存超类继承的同名方法
    this.m = function(){
        m.call(this);
        alert("B");
```

```
    };
}
B.prototype = new A();                    // 类 B 继承类 A
B.prototype.constructor = B;
var b = new B();
b.m();                                    // 字符 B，说明子类 B 的方法 m() 将覆盖类 A 的方法 m()
```

在覆盖方法中调用超类的同名方法时，需要使用 call 或 apply 方法来改变执行上下文为 this，如果直接调用该方法，执行上下文就会变成全局对象，在特殊语境中可能会发生歧义。

建议 99：建议主动封装类

封装（encapsulation）就是把对象内部数据和操作细节进行隐藏。很多面向对象语言都支持封装，JavaScript 不支持该特性，但使用闭包可以实现类型的封装功能。

很多 JavaScript 程序喜欢类的被动封装。所谓被动封装，就是对对象内部数据进行适当约定，这种约定具有很强的主观性，没有强制性保证，这主要针对公共对象而言。一般来说，JavaScript 类对象所包含的数据都是公开的，没有隐私可言，任何人都可以访问其中的信息。

为了数据安全，在代码中适当增加了一些条件限制，避免非法侵入，当然可以增加更完善的监测方法，以保护输入数据的完整性。

```
var Card = function(name, sex, work, detail){    // 较安全的公共类
    if(! checkName(name)) throw new Error("name 值非法 "){
        this.name = name;
    }
    this.sex = checkSex(sex);
    this.work = checkWork(work);
    this.detail = checkDetail(detail);
}
Card.prototype = {          // 类内部数据检测方法
    checkName : function(name){ // 检测 name，参数为 name，返回布尔值，检测是否合法
    }
    checkSex : function(sex){ // 检测 sex，参数为 sex，返回 sex，检测是否符合约定
    }
    checkWork : function(work){ // 检测 work，参数为 work，返回 work，检测是否符合约定
    }
    checkDetail : function(detail){ // 检测 detail，参数为 detail，返回 detail，检测是否
符合约定
    }
}
```

上面代码仅列出了各种方法的框架。当然，从更安全和更扩展的角度来讲，凡是类都应该定义接口，这样才能确保数据存取更加安全，同时也方便与其他开发人员和用户进行交流。

内部私有方法监测和接口措施能够在一定程序上保护对象内部数据，但它们也存在一个致命的漏洞，即这些属性和方法可以被公开重置。面对公开覆盖属性和方法值，任何人都无

法阻止。不管操作是有意还是无意的，属性都可能会被设置为无效值。同时内部检测和接口在一定程度上占用了系统开销，这个问题也是需要必须认真考虑的。

很多开发人员习惯使用命名规范来区分公共成员与私有成员，即在一些方法和属性的名称前后加下画线以示其私有特性。由于下画线在 JavaScript 中可以用做标识符的第一个字符，因此它们仍然是有效的变量名。

下画线命名法是一种约定俗成的命名规范，它表明一个属性和方法仅供对象内部使用，直接访问此属性可能会导致意想不到的后果。虽然它不是强制性规定，但是有助于防止开发人员无意识的误用。

上述数据保护的方法和措施都是被动性防御，带有很强的主观性。因为它们只是一种约定，只有在得到遵守时才有效果，而且并没有什么强制性手段可以保证实施，所以它们不是真正可以用来隐藏对象内部数据的解决方案，主要适用于非敏感性的内部方法和属性。

在 JavaScript 中，只有函数具有作用域。在函数内部声明的变量，在函数外部是无权访问的。从本质上分析，所谓私有属性和私有方法，就是在对象外部无法访问，因此要真正实现类的封装设计要求，使用函数作用域是最佳选择。

可以根据函数的这一特性，把上面示例中的私有数据用函数作用域和闭包进行封装。实现方法：在函数结构体内部定义变量，这些变量可以被定义该作用域中的所有函数访问。

```
var Card = function(name, sex, work, detail){              // 安全的类
    var _name = name, _sex = sex, _work = work, _detail = detail;      // 私有属性
    function _checkName(_name){  // 私有方法
    }
    function _checkSex(_sex){    // 私有方法
    }
    function _checkWork(_work){  // 私有方法
    }
    function _checkDetail(_detail){      // 私有方法
    }
    if(! _checkName(_name)) throw new Error("name 值非法 "){
        this.name = _name;
    }
    this.sex = _checkSex(_sex);
    this.work = _checkWork(_work);
    this.detail = _checkDetail(_detail);
}
Card.prototype = {
    // 公共方法
}
```

要使外界可以访问某些私有方法，可以采用如下方法来实现：

```
var Card = function(name, sex, work, detail){
    var _name = name, _sex = sex, _work = work, _detail = detail;
    function _checkName(_name){
    }
    this.checkName = function(){          //私有方法，实现外部调用
```

```
            return _checkName
        }
    }
```

函数作用域内部的方法无权被外界访问，但在函数作用域内的其他公共方法可以访问内部方法，于是将公共方法作为中转平台，可以巧妙地把内部私有方法公开化。因此，这些公共方法也称为特权方法，即在方法的前面加上关键字 this。因为这些方法被定义于函数作用域中，所以它们能够访问到私有属性，对于不需要直接访问私有属性和方法的方法，建议将它们放在类的原型对象中进行声明。

使用这种方式创建的对象具有真正的封装特性，但它也有缺点：生成的每一个新实例对象都会为每一个私有方法和特权方法生成一个新的副本，这会占用大量的系统资源，不适宜大量使用，仅在必要时适当使用。同时，这种方法不利于类的继承，因为所有派生的子类都不能访问超类的任何私有属性和方法。例如：

```
var Card = function(){                    // 超类
    var _name = 1;
    function _checkName(){
        return _name;
    }
    this.checkName = function(){
        return _name;
    }
}
function F(){                             // 子类
    Card.call(this);                      // 继承类 Card
    this.name =  _name;
}
var a = new F();
alert(a.name);                            // 访问无效
alert(a._checkName());                    // 无法访问，抛出解析错误
```

不过，读者可以通过特权方法来访问超类中的私有属性和方法：

```
alert(a.checkName());                     // 访问超类公共方法，间接访问私有属性和方法
```

建议 100：谨慎使用类的静态成员

在面向对象的编程中，类是不能够直接访问的，必须实例化后才可以访问，但静态属性和方法与类本身直接联系，可以直接通过类来访问。例如，JavaScript 核心对象中的 Math 和 Global 都是静态对象，不需要实例化就可以直接访问。

类的静态成员包括私有和公共两种类型，不管是公共成员还是私有成员，它们在系统中只有一份副本，不会被分成多份传递给不同的对象，而是通过函数指针进行引用，这与闭包截然不同。下面示例为类型定义一个私有的静态成员。

```
var F = (function(){
    var _a = 1;    //私有变量
    this.a = _a;   //公共属性
    this.get1 =function(){          //公共方法
        return _a;
    };
    this.set1 = function(x){        //公共方法
        _a = x;
    };
    return function(){              //构造函数类
       this.get2 =function(){       //提供访问私有变量的接口
           return _a;
       };
       this.set2 = function(x){     //提供修改私有变量的接口
           _a = x;
        };
    }
})();
// 定义类的静态公共方法和属性
F.get3 =function(){
    return get1();
};
F.set3 = function(x){
     set1(x);
}
```

与一般类的创建方法一样，这里的私有成员和特权成员仍然被声明在构造器中，并借助 var 和 this 关键字来实现。这里的构造器却由原来的普通函数变成了一个内嵌函数，并且作为外层函数的返回值赋值给了变量 F，这就创建了一个闭包。在这个闭包中，还可以声明静态私有成员，例如：

```
var F = (function(){
    function set5(x){     // 静态私有方法
        _a = x;
    }
    function get5(){      // 静态私有方法
        return _a;
    }
})();
```

这些静态私有成员可以在构造器内部访问，这意味着所有私有函数和特权函数都能访问它们。与其他方法相比，静态方法有一个优点，那就是在内存中仅存放一份。那些被声明在构造器之外的公共静态方法，以及下文中将要提到的 F 类原型属性都不能访问在构造器中定义的任何私有属性，因此它们不是特权成员。

定义在构造器中的私有方法能够调用其中的静态私有方法，反之则不然。要判断一个私有方法是否应该被设计为静态方法，可以看它是否需要访问任何实例数据。如果它不需要，那么将其设计为静态方法会更有效率，因为它只被创建一份。

定义类的静态公共方法和属性一般在类的外面进行，这种外挂定义的方式在前面的示例中也曾经介绍过。这种外挂的静态方法和属性可以直接访问，这实际上相当于把构造器作为命名空间来使用。同时，由于它们仍然属于构造器结构的一部分，因此在这些静态方法和属性中可以访问闭包中的私有成员。

```
alert(F.get3());  // 直接访问类 F 的静态方法 get3()，返回 1
F.set3(2);        // 修改私有变量的值
alert(F.get3());  //2，说明修改成功
```

注意，类 F 是返回的内层函数，该值是一个构造函数，它无法访问外层函数的公共方法 get1() 和 set1()，但能够访问返回构造函数体内的公共方法 get2() 和 set2()。例如：

```
var a = new F()   // 实例化类 F
alert(a.get2());  // 调用类 F 的公共方法 get2()，返回 1
a.set2(2);        // 调用类 F 的公共方法 set2()，修改私有变量 _a
alert(a.get2());  // 调用类 F 的公共方法 get2()，返回 2
```

但下面的用法都是错误的，因为级别比较低的 F 类（F 是返回的匿名构造函数）无权访问闭包体内的变量、属性和方法（不管是私有的还是公共的）。

```
var a = new F()
alert(a.get1());
a.set1(2);
alert(a.get1());
```

闭包体内的所有对象都可以访问闭包体内的私有或公共变量、属性和方法。由于类 F 是闭包体内返回的构造函数，因此根据作用域链，它们可以向上访问闭包所有成员。

```
F.prototype = {              // 类 F 的原型对象
    get4 : function(){       // 原型方法 get4()
        return get1();       // 访问闭包内数据
    },
    set4 : function(x){      // 原型方法 set4()
        set1(x);             // 访问闭包内数据
    }
};
var a = new F();             // 实例化类 F
alert(a.get4());             //1
```

建议 101：比较类的构造和析构特性

构造和析构是创建和销毁对象的过程，它们是对象生命周期中的起点和终点，是最重要的环节。当一个对象诞生时，构造函数负责创建并初始化对象的内部环境，包括分配内存、创建内部对象和打开相关的外部资源等。析构函数负责关闭资源、释放内部的对象和已分配的内存。

在面向对象的编程中，构造和析构是类的两个重要特性。构造函数将在对象产生时调

用，析构函数将在对象销毁时调用。调用的过程和实现方法由编译器完成，我们只需要记住它们调用的时间，因为它们的调用是自动完成的，不需要人工控制。

（1）构造函数

在 JavaScript 中，被 new 运算符调用的函数就是构造函数。构造函数被 new 运算符计算后，将返回实例对象，也就是所谓的对象初始化，即对象的诞生。调用构造函数的过程也是类实例化的过程。

如果构造函数有返回值，并且返回值是引用类型的，那么经过 new 运算符计算后，返回的不再是构造函数自身对应的实例对象，而是构造函数包含的返回值（即引用类型值）。

```
function F(x,y){
    this.x = x;
    this.y = y;
    return [];
}
var f = new F(1,2);
alert(f.constructor == F);          //false，说明 F 不再是 f 的构造函数
```

在上面的示例中，返回值是一个空的数组，而不再是实例对象。原来构造函数的返回值覆盖了 new 运算符的运算结果，此时如果调用 f 的 constructor 属性，那么返回值是：

```
function Array () {          // 被封闭的 Array 核心结构
    [ native code ]
}
```

上面示例说明返回值是 Array 的实例，使用下面的代码可以检测出来：

```
alert(f.constructor == Array);    //true，说明 Array 是 f 的构造函数
```

利用 call 和 apply 方法可以实现动态构造。例如，在下面这个示例中，构造函数 A、B 和 C 相互之间通过 call 方法关联在一起，当构造对象 c 时，将调用构造函数 C，而在执行构造函数 C 时，会先调用构造函数 B。在调用构造函数 B 之前，会自动调用构造函数 C，从而实现动态构造对象的效果。这种多个构造函数相互关联在一起的情况称为多重构造。

```
function A(x){
    this.x = x || 0;
}
function B(x){
    A.call(this,x);
    this.a = [x];
}
function C(x){
    B.call(this,x);
    this.y = function(){
        return this.x;
    }
}
var c = new C(3);
```

```
alert(c.y());      //3
```

根据动态构造特性可以设计类的多态处理：

```
function F(x, y){          // 多态类型
    function A(x, y){
        this.add = function(){
            return x + "" + y;
        }
    }
    function B(x, y){
        this.add = function(){
            return x + y;
        }
    }
    if(typeof x == "string" || typeof y == "string"){
        A.call(this, x, y);
    }
    else{
        B.call(this, x, y);
    }
}
var f1 = new F(3,4);
alert(f1.add());           // 调用对象方法 add()，返回数值 7
var f2 = new F("3","4");   // 实例化类 F，传递字符串
alert(f2.add());           // 调用对象方法 add()，返回字符串 34
```

（2）析构函数

析构是销毁对象的过程。由于 JavaScript 能够自动回收垃圾，不需要人工清除，所以当对象使用完毕时，JavaScript 会调用对象回收程序来销毁内存中的对象，这个回收程序相当于一个析构函数。从文法角度来分析，JavaScript 是不支持析构语法的。当然，我们也可以主动定义析构函数对对象进行清理。例如，先定义一个析构函数，该函数中包含一个析构方法，把该方法继承给任意对象，就可以调用它清除对象内部所有成员了。

```
function D(){      // 析构函数
}
D.prototype = {
    d : function(){        // 析构方法
        for(var i in this){
            if(this[i] instanceof D){
                this[i].d();
            }
            this[i] = null;        // 清除成员
        }
    }
}
function F(){
    this.x = 1;
    this.y = function(){
        alert( 2 );
```

```
    }
}
F.prototype =new D();       // 绑定析构函数，继承析构方法
var f = new F();            // 实例化试验函数
f.d();                      // 调用析构方法
alert(f.x);                 //null，说明属性已经被注销
f.y()                       // 编译错误，说明方法已不存在
```

构造和析构有一个顺序问题。在其他强类型语言中，构造是从基类开始按继承的层次顺序进行的，析构的时候顺序正好相反。这样处理是因为子类可能在构造函数中使用父类的成员变量，如果父类还没有创建，那么就会有问题。而在析构的时候，如果父类先析构，也会出现这样的问题。JavaScript 对此没有严格的要求，但它遵循从下到上的顺序进行构造，而析构则没有这方面的要求，只要对象没有成员引用或对象引用即可进行析构。

建议 102：使用享元类

享元类就是类的类型，即创建类型的类。享元类与类的关系正如类与对象的关系一样，是一种创建型的泛化关系。享元类能够接受类作为参数，即享元类操作的对象是类，而不是具体的数据。一般享元类返回的是类，而不是具体的数据。下面的示例就是一个简单的享元类，它包含了一个返回的类。

```
function O(x){     // 享元类
    return function(){     // 类
        this.x = x;
        this.get = function(){
            alert(this.x);
        }
    }
}
var o = new O(1);
var f = new o();
f.get();   // 调用返回类的方法 get()，返回 1
```

从上面的示例可以看到，享元类与普通函数没有什么两样，不过它的返回值是类，而不是具体数值。实际上，JavaScript 核心对象 Function 就是一个享元类，虽然说它没有返回值，但是我们可以通过字符串的形式创建返回类。例如：

```
var O = new Function("this.x=1;this.y=2")          // 实例化之后返回的是类
var o = new O();  // 实例化返回类
alert(o.x);       // 调用实例的属性值，返回 1
```

上面的示例演示了简单的函数中包含一个返回类结构，当然享元类并非如此简单。下面再演示一个比较复杂的示例，在这个享元类中参数值包含类类型，返回值也是类类型。

首先，定义一个普通类，作为一个参数值准备传递给享元类。

```
function F(x, y){
    this.x = x;
    this.y = y;
}
F.prototype.add = function(){
    alert(this.x + this.y);
}
```

然后，定义一个享元类，该函数类包含 3 个参数，其中第一个参数为类类型，第二个和第三个参数是值类型数据。

```
function O(o, x, y){
    this.say = function(){
        alert( "享元类" );
    }
    return function(){
        this.say = function(){
            alert( "返回类" );
        }
        var a = new o(x, y);              // 实例化参数类
        for(var i in a ){                 // 通过实例继承法，继承参数类给返回类
            this[i] = a[i];               // 此时 this 关键字只返回类的当前对象
        }
    }
}
```

最后，使用 new 运算符调用享元类，第一个参数值为上文定义的类 F，第二个和第三个参数为普通数值，返回的类赋值给变量 A，则 A 就变成了一个类结构。此时不能够通过 A 来读取享元类的本地方法 say()。

```
var A = new O(F, 1, 2);
var B = new A();
A.say();   // 如果直接调用享元类的本地方法，将提示编译错误
```

但可以通过实例化后的 B 对象来访问参数类 F 中的成员，以及返回类内部定义的本地属性。

```
B.say();   // 字符串 "返回类"
B.add();   // 数值 3
alert(B.x);        // 数值 1
alert(B.y);        // 数值 2
```

注意，当一个类有返回值时，如果是值类型数据，则可以访问类的成员，也可以获取返回值。

```
function F(){
    this.x = 1;
    return 2;
}
var f = new F();
alert(f.x);        //1
alert(F());        //2
```

如果类返回的是引用类型或函数体，则类的成员将不可访问，它们将成为闭包结构内的私有数据，不再对外开放。

```
function F(){
    this.x = 1;
    return function(){
        return this.x;
    };
}
F.prototype.y = function(){
    alert(3);
}
var f = new F();
alert(f.x);          // 访问本地属性 x 失败，返回 undefined
alert(F()());        // 调用返回的函数，返回 1，说明它可以访问本地属性 x
alert(f.y());        // 提示编译错误，没有这个成员
```

建议 103：使用掺元类

掺元类就是共享通用的方法和属性，将差异比较大的类中相同功能的方法集中到一个类中声明，这样需要这些方法的类就可以直接从掺元类中进行扩展，通用的方法只需要声明一遍就行了。从代码的大小、质量方面来说，这还是有一定效益的。如图 4.7 所示，掺元类就是让一个类被多个类继承，这种继承关系被形象地称为多亲继承，它是一种比较特殊的类形式。

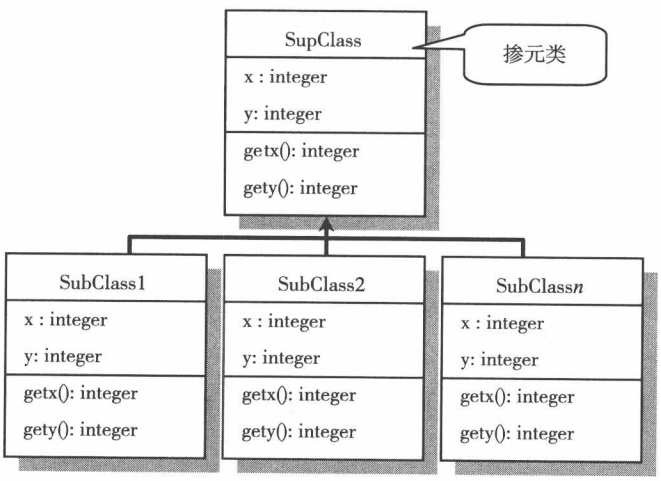


图 4.7　多亲继承示意图

如果希望某个函数被多个类调用，那么可以通过扩充的方式让这些类共享该函数。具体的设计思路：先创建包含通用函数的超类，然后利用这个超类扩充子类，这种包含通用方法的类可以称为掺元类。例如，先设计一个掺元类 F，设想两个子类 A 和 B 能够继承掺元类 F

的通用方法 getx() 和 gety()。代码如下：

```
var F = function(x,y){     // 构造函数 F，掺元类
    this.x =x;
    this.y = y;
}
F.prototype = {
    getx : function(){
        return this.x;
    },
    gety : function(y){
        return this.y;
    }
}
```

然后，定义两个子类 A 和 B，利用类继承方法先继承掺元类中的本地属性，以方便继承的方法正确获取值。实际应用中不使用类继承来继承掺元类的本地属性和方法。

```
A = function(x,y){          // 子类 A
    F.call(this,x,y) ;      // 继承掺元类 F
};
B = function(x,y){          // 子类 B
    F.call(this,x,y) ;      // 继承掺元类 F
};
```

要让 A 类和 B 类都继承 F 类，可以使用原型继承方法来实现，但原型继承需要实例化 F 类。我们可以模仿复制继承方法设计一个专门函数来实现这种继承关系，具体代码如下：

```
// 掺元类继承封装函数，其中参数 Sub 表示子类，参数 Sup 表示掺元类
function extend(Sub,Sup){
    for(m in Sup.prototype){        // 遍历掺元类的原型对象
        if(!Sub.prototype[m]){      // 如果子类不存在同名成员，则复制掺元类原型成员给子类原型对象
            Sub.prototype[m] = Sup.prototype[m];
        }
    }
}
```

该函数很简单，使用 for in 循环遍历掺元类的原型对象中的每一个成员，并将其添加到子类的原型对象中。如果子类中已存在同名成员，则跳过该成员，转而处理下一个，这样能够确保子类原型对象中的成员不会被改写。有了这个封装函数，就可以直接调用它来快速生成多个相同的子类。传递子类参数必须事先声明，并且应通过类继承方法来继承 F 的本地属性和方法。

```
extend(A,F);        // 继承 F 的子类 A
extend(B,F);        // 继承 F 的子类 B
```

最后，实例化 A 类和 B 类，这样就可以调用 F 定义的通用方法了。

```
var a = new A(1,2);
var b = new B(10,20);
```

```
alert(a.getx());  //1
alert(a.gety());  //2
alert(b.getx());  //10
alert(b.gety());  //20
```

也可以把多个子类合并到一个类中来实现多重继承。例如，下面的示例定义了两个类 A 和 B，并分别为它们定义两个原型方法。

```
var A = function(){}        // 类 A
A.prototype = {
    x : function(){
        return "x";
    }
}
var B = function(){}        // 类 B
B.prototype = {
    y : function(){
        return "y";
    }
}
C = function(){};           // 空类 C
extend(C,A);                // 把类 A 继承给类 C
extend(C,B);                // 把类 B 继承给类 C
var c = new C();            // 实例化类 C
alert(c.x())                // 字符 x
alert(c.y())                // 字符 y
```

面向对象中并不是所有的事物泛型都是使用继承关系来描述的，继承关系只是泛型关系的一种，除此之外，创建关系、原型关系、聚合关系、组合关系等，都是泛型的一种类型。泛型概念很宽泛，通常使用继承、聚合和组合来描述事物的名词特性，而使用原型、元类等其他概念来描述事物的形容词概念。

建议 104：谨慎使用伪类

JavaScript 的原型存在诸多矛盾，某些看起来有点像基于类的语言的复杂语法问题遮蔽了它的原型机制。原型不但不让对象直接从其他对象继承，反而插入了一个多余的间接层，从而使构造器函数产生对象。当一个函数对象被创建时，Function 构造器产生的函数对象会运行类似这样的一些代码：

```
this.prototype = {constructor: this};
```

新函数对象被赋予一个 prototype 属性，其值包含一个 constructor 属性且属性值为该新函数对象。该 prototype 对象是存放继承特征的地方。因为 JavaScript 语言没有提供一种方法来确定哪个函数是用做构造器的，所以每个函数都会得到一个 prototype 对象，constructor 属性没什么太大作用，重要的是 prototype 对象。

定义一个构造器并扩展它的原型：

```
var Mammal = function(name) {
    this.name = name;
};
Mammal.prototype.get_name = function() {
    return this.name;
};
Mammal.prototype.says = function() {
    return this.saying || '';
};
```

构造实例：

```
var myMammal = new Mammal('mammal');
var name = myMammal.get_name();          // ' mammal'
```

构造另一个伪类来继承 Mammal，这是通过定义它的 constructor 函数并替换它的 prototype 为一个 Mammal 的实例来实现的。

```
var Cat = function(name) {
    this.name = name;
    this.saying = 'meow';
};
Cat.prototype = new Mammal();
Cat.prototype.purr = function(n) {
    var i, s = '';
    for( i = 0; i < n; i += 1) {
        if(s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
};
Cat.prototype.get_name = function() {
    return this.says() + ' ' + this.name + ' ' + this.says();
};
var myCat = new Cat('cat');
var says = myCat.says();          // 'meow'
var purr = myCat.purr(5);         // 'r-r-r-r-r'
var name = myCat.get_name();      // 'meow cat meow'
```

伪类模式的本意是想向面向对象靠拢，但它看起来与面向对象格格不入。我们可以隐藏一些“丑陋”的细节，这是通过使用 method 方法定义一个 inherits 方法来实现的。

```
Function.method('inherits', function (Parent) {
    this.prototype = new Parent();
    return this;
});
```

inherits 和 method 方法都返回 this，这样就可以通过链式语法只用一行语句构造 Cat。

```
var Cat = function(name) {
    this.name = name;
    this.saying = 'meow';
}.inherits(Mammal).method('purr', function(n) {
    var i, s = '';
    for( i = 0; i < n; i += 1) {
        if(s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
}).method('get_name', function() {
    return this.says() + ' ' + this.name + ' ' + this.says();
});
```

隐藏了 prototype 操作细节，现在看起来就没那么怪异了。现在有了行为像“类”的构造器函数，但它们没有私有环境，所有的属性都是公开的，因此无法访问父类 super 的方法。同时，使用构造器函数存在一个严重的隐患。如果在调用构造器函数时忘记了在前面加上 new 前级，那么 this 将不会被绑定到一个新对象上，而是被绑定到全局对象上，这样不但没有扩充新对象，反而会破坏全局变量。发生这种情况时，既没有编译时警告，也没有运行时警告。

这是一个严重的语言设计错误。为了降低这个问题带来的风险，所有的构造器函数都约定命名成首字母大写的形式，并且不以首字母大写的形式拼写任何其他的东西。这样至少可以通过人工检查去发现是否缺少了 new 前缀。当然，一个更好的备选方案就是根本不使用 new。

伪类形式可以给不熟悉 JavaScript 的程序员提供便利，但它也隐藏了该语言的真实本质。借鉴类的表示法可能误导程序员去编写过于深入和复杂的层次结构。许多复杂的类层次结构的产生就是源于静态类型检查的约束。JavaScript 完全摆脱了这些约束。在基于类的语言中，类的继承是代码重用的唯一方式。JavaScript 有着更多且更好的选择。

建议 105：比较单例的两种模式

在 JavaScript 中，单例模式（即实例结构模式）是最基本、最有用的模式之一。这种模式提供了一种将代码组织为一个逻辑单元的手段，这个逻辑单元中的代码可以通过单一的变量进行访问。确保单例对象只有一份实例，就可以确信自己的所有代码使用的都是同样的全局资源。单例类在 JavaScript 中用途广泛：

- ❑ 可以用来划分命名空间，以减少全局变量的数量。
- ❑ 可以在一种名为分支的技术中用来封装浏览器之间的差异。
- ❑ 可以借助于单例模式，将代码组织得更为一致，从而使代码更容易阅读和维护。

1. 第一种模式：对象直接量

最基本的单例实际上是一个对象字面量，它将一批有一定关联的方法和属性组织在一起。

```
var Singleton = {
    attribute1: true;
    attribute2: 10
    method1: function() { },
    method2: function() { }
};
```

这些成员可以通过 Singleton 加圆点运算符来访问，例如：

```
Singleton.attribute1 = false;
var total = Singleton. attribute2 + 5;
var result = Singleton.method1();
```

对象字面量只是用于创建单例的方法之一，并非所有对象字面值都是单体，如果它只是用来模仿关联数组或容纳数据的话，那显然不是单例。但如果它用来组织一批相关方法和属性，那就可能是单例，其区别主要在于设计者的意图。

在单例对象内创建类的私有成员的最简单、最直接的方法是使用下画线表示法。在 JavaScript 业界，如果变量和方法使用下画线，则表示该变量和方法是私有方法，只允许内部调用，第三方不应该去调用。

```
GiantCorp.DataParser = {
    // 私有方法
    _stripWhitespace: function(str) {
        return str.replace(/\s+/, '');
    },
    // 公用方法
    stringToArray: function(str, delimiter, stripWS) {
        if (stripWS) {
            str = this._stripWhitespace(str);
        }
        var outputArray = this._stringSplit(str, delimiter);
        return outputArray;
    }
};
```

在 stringToArray 方法中使用 this 访问单体中的其他方法，这是访问单体中其他成员或方法最简便的方式。这样做有一点风险，因为 this 并不一定指向 GiantCorp.DataParser。例如，如果把某个方法用做事件监听器，那么其中的 this 指向的是 window 对象，因此大多数 JavaScript 库都会为事件关联进行作用域校正，如在这里使用 GiantCorp.DataParser 比使用 this 更为安全。

2．第二种模式：使用闭包

在单例对象中创建私有成员的第二种方法是借助闭包。因为单例只会被实例化一次，所以不必担心自己在构造函数中声明了多少成员。由于每个方法和属性都只会被创建一次，所以可以把它们声明在构造函数内部。使用闭包创建拥有私有成员的单例类的示例如下：

```
MyNamespace.Singleton = (function() {
     // 私有成员
     var privateAttribute1 = false;
     function privateMethod1() {}
     return {
        // 公有成员
        publicAttribute1: true;
        publicMethod1: function() { },
    };
})();
```

这种单例模式又称为模块模式，指的是它可以把一批相关方法和属性组织为模块并起到划分命名空间的作用。将私有成员放在闭包中可以确保其不会在单例对象之外被使用，因此开发人员可以自由地改变对象的实现细节，而不会殃及别人的代码。还可以使用这种办法对数据进行保护和封装。

在 JavaScript 中，使用单例模式的主要优点如下：

- 对代码的组织作用。单例模式将相关方法和属性组织在一个不会被多次实例化的单例中，可以使代码的调试和维护变得更轻松。描述性的命名空间还可以增强代码的自我说明性。将方法包裹在单例中，可以防止它们被其他程序员误改。
- 单例模式的一些高级变体可以在开发周期的后期用于对脚本进行优化。

在 JavaScript 中，使用单例模式的主要缺点如下：

- 因为提供的是一种单点访问，所以它有可能导致模块间的强耦合。单体最好是留给定义命名空间和实现分支型方法使用，在这些情况下耦合不是什么问题。
- 有某些更高级的模式会更符合任务的需要。虚拟代理能给予开发人员对实例化方式更多的控制权。也可以用一个真正的对象工厂来取代分支型单例。

第5章 DOM编程

DOM（文档对象模型）是一个独立于JavaScript语言的，使用XML和HTML文档操作的应用程序接口（API）。在浏览器中，DOM主要与HTML文档打交道，因此在网页应用中检索XML文档也很常见。虽然DOM是与语言无关的API，但是浏览器中的接口却是以JavaScript实现的。客户端大多数脚本程序与文档打交道，因此DOM就成为JavaScript代码日常行为中重要的组成部分。浏览器通常要求DOM实现和JavaScript实现保持相互独立。例如，在IE中，被称为JScript的JavaScript实现位于库文件jscript.dll中，而DOM实现位于另一个库文件mshtml.dll中。

这种分离技术为其他技术和语言提供了准入接口。Safari使用WebKit的WebCore处理DOM和渲染，具有一个分离的JavascriptCore引擎（最新版本的绰号是SquirrelFish）。Google Chrome也使用WebKit的WebCore库渲染页面，却实现了自己的JavaScript引擎V8。在Firefox中，JavaScript实现采用Spider-Monkey（最新版是TraceMonkey），与其Gecko渲染引擎相分离。

对DOM操作代价很高，在富网页应用中通常是一个性能瓶颈。两个独立的部分以功能接口连接就会带来性能损耗。我们可以把DOM看成一个岛屿，把JavaScript（ECMAScript）看成另一个岛屿，两者之间通过一座桥连接。每次ECMAScript需要过桥访问DOM时，都会带来一定的性能损耗。操作DOM次数越多，损耗越大。一般的建议是尽量减少这种过桥操作次数，努力停留在ECMAScript岛上。

建议106：建议先检测浏览器对DOM支持程度

W3C制定的DOM规范包括多个版本，不同版本（或称级别）又包含不同的子规范和模块，不同浏览器对DOM支持是千差万别的。另外，DOM不同版本之间可能会存在个别不兼

容的规定，这一点需要特别留意。

（1）DOM 0 级

在 W3C 推出 DOM 标准之前，市场已经流行了几个版本不太一致的 DHTML 规范，主要包括 IE 和 Netscape 两个不同版本。这组 DHTML 规范规定了一套文档对象、集合、方法和属性，虽然不同版本 DHTML 规范的特性存在很大差异，但是一些基本思路和用法还是有章可循的，如事件处理函数、脚本化样式、文档基本结构对象等。

（2）DOM 1 级

1998 年 10 月，W3C 推出了 DOM 1.0 版本规范（http://www.w3.org/DOM/DOMTR. html#dom1），主要包括两个子规范。

- DOM Core（核心部分）：把 XML 文档设计为树形节点结构，并为这种结构的运行机制制定了一套规范化标准，同时定义了创建、编辑、操纵这些文档结构的基本属性和方法。
- DOM HTML：针对 HTML 文档、标签集合，以及与个别 HTML 标签相关的元素定义了对象、属性和方法。

（3）DOM 2 级

2000 年 11 月，W3C 正式发布了更新后的 DOM 核心部分，并在这次发布中添加了一些新规范，这次发布的 DOM 称为 2 级规范。

2003 年 1 月，W3C 又正式发布了对 DOM HTML 子规范的修订，添加了针对 HTML 4.01 和 XHTML 1.0 版本文档中的很多对象、属性和方法。W3C 把新修订的 DOM 规范统称为 DOM 2.0 推荐版本（http://www.w3.org/DOM/DOMTR.html#dom2），该版本主要包括 6 个推荐子规范。

- DOM2 Core：继承于 DOM Core 子规范，系统规定了 DOM 文档结构模型，添加了更多的特性，如针对命名空间的方法等（参阅 http://www.w3.org/TR/ 2000/REC-DOM-Level-2-Core-20001113/changes.html）。
- DOM2 HTML：继承于 DOM HTML，系统规定了针对 HTML 的文档结构模型，并添加了一些属性（参阅 http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML- 20030109/changes.html）。
- DOM2 Events：规定了与鼠标相关的事件（包括目标、捕获、冒泡和取消）的控制机制，但不包含与键盘相关事件的处理部分。
- DOM2 Style（或 DOM2 CSS）：提供了访问和操纵所有与 CSS 相关的样式及规则的能力。
- DOM2 Traversal 和 DOM2 Range：这两个规范允许开发人员通过迭代方式访问 DOM，以便根据需要对文档进行遍历或其他操作。
- DOM2 Views：提供了访问和更新文档表现的能力。

DOM 2 级规范已经成为目前各大浏览器支持的主流标准，但 IE 对于该规范的支持不尽

完善，特别是在对 DOM2 Traversal 和 DOM2 Range 的支持上。

（4）DOM 3 级

2004 年 4 月，W3C 发布了 DOM 3.0 版本（http://www.w3.org/DOM/DOMTR.html#dom3）。DOM 3 级版本主要包括以下 3 个推荐子规范。

- DOM3 Core：继承于 DOM2 Core，并添加了更多的新方法和新属性，同时也修改了已有的一些方法（参阅 http://www.w3.org/TR/2004/REC-DOM-Level-3-Core- 20040407/changes.html）。
- DOM3 Load and Save：提供将 XML 文档的内容加载到 DOM 文档中和将 DOM 文档序列化为 XML 文档的能力。
- DOM3 Validation：提供了确保动态生成的文档的有效性的能力，即如何符合文档类型声明。

为了能够检测不同类型浏览器及各个版本对DOM规范的支持程度，可以调用DOMImplementation对象，该对象被DOM核心规范规定，通过document对象的implementation属性来调用。如果浏览器支持DOM某个特性，那么可以通过implementation对象的hasFeature()方法来检测。检测代码如下：

```
var dom = "HTML";                    // 指定 DOM 模块
var ver = "1.0";                     // 指定 DOM 级别
if(document.implementation){         // 如果浏览器支持 implementation 对象，则进行测试
    if(document.implementation.hasFeature(dom, ver)){  // 检测对指定 DOM 模块及版本的支持情况
        alert(" 支持 :\n" + dom + " " + ver);
    }
    else{
        alert(" 不支持 :\n" + dom + " " + ver);
    }
}
else{
    alert(" 不支持 :\n DOMImplementation 对象 ");
}
```

如果浏览器中不存在document.implementation对象，那么基本可以确定它不支持DOM，不过也可能部分支持，如IE 6.0支持HTML，而不支持Core，但为了支持HTML，自然会支持DOM核心的某些部分，因为HTML需要核心方法。

hasFeature()方法包含两个参数，第一个参数用来指定DOM模块的名称，第二个参数指定模块的级别，包括1、2、3，所有参数都以字符串的形式进行传递。

访问http://www.w3.org/2003/02/06-dom-support.html，页面会自动显示当前浏览器版本所支持的模块。其中“N/A”表示某个模块在某个级别下是不适用的，即还没有开发或推荐标准。如果显示白色背景，提示信息为“supported”，则表示支持这个级别的模块。如果显示红色背景，提示这个级别版本的发布时间，则说明当前浏览器版本不支持这个模块。

访问http://www.w3.org/DOM/Test/，对DOM规范中每一个模块的个别部分进行更具体的测试。这种测试的时间稍长，但会检测每个对象的每个方法，因而可以检测出哪些方法符

合规范。

建议 107：应理清 HTML DOM 加载流程

HTML DOM 文档加载是按顺序执行的，这与浏览器的渲染方式有关系，一般浏览器渲染操作的顺序如下：

第 1 步，解析 HTML 结构。

第 2 步，加载外部脚本和样式表文件。

第 3 步，解析并执行脚本代码。

第 4 步，构造 HTML DOM 模型。

第 5 步，加载图片等外部文件。

第 6 步，页面加载完毕。

例如，下面这个简单的 DOM 文档。

```
<html>
<head>
<title> 网页标题 </title>
<style type="text/css">
body { font-size:12px;}
</style>
<link href="style.css" rel="stylesheet" type="text/css" media="all">
<script src="js.js" type="text/javascript"></script>
</head>
<body>
    <div>
        <script type="text/javascript">
            function f1(){}
        </script>
        <img src="1.gif" />
    </div>
    <script type="text/javascript">
        function f2(){}
    </script>
</body>
</html>
```

这个文档的加载和构造顺序如下，所谓构造就是把对应的标签元素添加到 DOM 文档对象模型中。

html → head → title → #text（网页标题）→ style → 加载样式 → 解析样式 → link → 加载外部样式表文件 → 解析外部样式 → script → 加载外部脚本文件 → 解析外部脚本 → 执行外部脚本 → body → div → script → 加载脚本 → 解析脚本 → 执行脚本 → img → script → 加载脚本 → 解析脚本 → 执行脚本 → 加载外部图像文件 → 页面初始化完毕。

通过上面的 HTML DOM 加载顺序，可以看到网页头部的脚本（由外部文件加载）会在

构造 HTML DOM 文档结构之前执行，这就会导致执行脚本无法访问文档结构模型。所以，一般可执行脚本都放在页面初始化事件处理函数中，这样能够确保完全加载完文档之后再执行脚本。

但是，如果页面中包含很多外部文件，如大量图片、视频、音频、动画等文件，可能会延迟脚本的执行时间。为了避免 JavaScript 脚本处于较长时间的等待，可以把需要执行的脚本分块放在 HTML 文档结构中间，这样只要在构造 DOM 后执行到脚本所在结构位置，就会执行脚本。

这种方法虽然能够提前执行脚本，但是不能够保证脚本可以访问该位置后面的文档结构，因为这些文档结构还没有被构造。不过，如果在页面最后一个元素之前嵌入脚本，就可以最早执行脚本，并能够确保脚本可以访问 HTML 文档结构模型中所有元素。例如：

```
<html>
<head>
<title></title>
</head>
<body>
< ! -- 文档结构 -->
<script language="javascript" type="text/javascript">
// JavaScript 执行脚本 </script>
</body>
</html>
```

上述方法容易破坏文档的结构，使整个文档看起来很混乱，不利于管理。可以利用一种间接的方法来实现文档结构的有序显示，当加载完 DOM 文档后，也意味着 Document 对象的属性加载完毕，这样可以判断 Document 对象的几个重要方法，如果存在，则说明 DOM 已经加载完毕，否则说明 DOM 还在加载中。通过这种方法既不影响文档结构，又可以快速捕捉到 DOM 加载的过程，实现的代码如下：

```
function f(){
    if(document && document.getElementsByTagName && document.
getElementById && document.body ){
        clearInterval( timer );
        // JavaScript 执行脚本
    }
}
var timer = setInterval(f, 10);
```

在函数 f() 中，首先判断 Document 对象的几个重要方法是否已经加载完毕，如果加载完毕，则说明 DOM 结构已经完成加载，执行预定的 JavaScript 脚本。为了能够实时跟踪加载过程，这里设计了一个定时器，不断调用函数 f()，以便快速、准确地判断 DOM 加载状态。如果 DOM 加载完毕，则清除定时器，并开始执行脚本。

建议 108：谨慎访问 DOM

访问一个 DOM 元素的代价是较高的，修改元素的代价更高，因为它经常导致浏览器重新计算页面的几何变化。当然，访问或修改元素最坏的情况是使用循环执行此操作，特别是在 HTML 集合中使用循环。下面看一个简单的示例：

```
function innerHTMLLoop() {
    for (var count = 0; count < 15000; count++) {
        document.getElementById('here').innerHTML += 'a';
    }
}
```

上面函数在循环中更新页面内容。这段代码的问题是在每次循环中都对 DOM 元素访问两次：一次是读取 innerHTML 属性能容，另一次是写入它。更有效率的写法是使用局部变量存储更新后的内容，在循环结束时一次性写入，例如：

```
function innerHTMLLoop2() {
    var content = '';
    for (var count = 0; count < 15000; count++) {
        content += 'a';
    }
    document.getElementById('here').innerHTML += content;
}
```

在所有浏览器中，上面代码的运行速度都要快得多。对 DOM 的访问越多，代码的执行速度就越慢。因此，建议读者尽量减少对 DOM 的访问，并保持在 ECMAScript 范围内缓存 DOM 引用。

建议 109：比较 innerHTML 与标准 DOM 方法

在更新页面时，是使用不标准的 innerHTML 属性，还是使用纯 DOM 方法（如 document.createElement()）？如果不考虑标准问题，它们的性能如何？

两者性能基本相近，不过在几乎所有浏览器中，innerHTML 速度更快一些，但最新的基于 WebKit 的浏览器（Chrome 和 Safari）除外。

例如，分别使用两种方法来创建一个 1000 行的表格。构造一个 HTML 字符串，然后更新 DOM 的 innerHTML 属性。使用 innerHTML 创建表格的代码如下：

```
function tableInnerHTML() {
    var i, h = ['<table border="1" width="100%">'];
    h.push('<thead>');
    h.push('<tr><th>id<\/th><th>yes?<\/th><th>name<\/th><th>url<\/th><th>action<\/
th><\/tr>');
    h.push('<\/thead>');
    h.push('<tbody>');
```

```
    for( i = 1; i <= 1000; i++) {
        h.push('<tr><td>');
        h.push(i);
        h.push('<\/td><td>');
        h.push('And the answer is... ' + (i % 2 ? 'yes' : 'no'));
        h.push('<\/td><td>');
        h.push('my name is #' + i);
        h.push('<\/td><td>');
        h.push('<a href="http://example.org/'+i+'.html">http://example.org/'+i+ '.html<\/a>');
        h.push('<\/td><td>');
        h.push('<ul>');
        h.push(' <li><a href="edit.php?id=' + i + '">edit<\/a><\/li>');
        h.push(' <li><a href="delete.php?id="' + i + '-id001">delete<\/a><\/li>');
        h.push('<\/ul>');
        h.push('<\/td>');
        h.push('<\/tr>');
    }
    h.push('<\/tbody>');
    h.push('<\/table>');
    document.getElementById('here').innerHTML = h.join('');
};
```

下面通过 DOM 标准方法 document.createElement () 和 document.createTextNode() 创建同样的表，代码有些较长。

```
function tableDOM() {
    var i, table, thead, tbody, tr, th, td, a, ul, li;
    tbody = document.createElement('tbody');
    for( i = 1; i <= 1000; i++) {
        tr = document.createElement('tr');
        td = document.createElement('td');
        td.appendChild(document.createTextNode((i % 2) ? 'yes' : 'no'));
        tr.appendChild(td);
        td = document.createElement('td');
        td.appendChild(document.createTextNode(i));
        tr.appendChild(td);
        td = document.createElement('td');
        td.appendChild(document.createTextNode('my name is #' + i));
        tr.appendChild(td);
        a = document.createElement('a');
        a.setAttribute('href', 'http://example.org/' + i + '.html');
        a.appendChild(document.createTextNode('http://example.org/' + i + '.html'));
        td = document.createElement('td');
        td.appendChild(a);
        tr.appendChild(td);
        ul = document.createElement('ul');
        a = document.createElement('a');
        a.setAttribute('href', 'edit.php?id=' + i);
        a.appendChild(document.createTextNode('edit'));
        li = document.createElement('li');
        li.appendChild(a);
        ul.appendChild(li);
```

```
            a = document.createElement('a');
            a.setAttribute('href', 'delete.php?id=' + i);
            a.appendChild(document.createTextNode('delete'));
            li = document.createElement('li');
            li.appendChild(a);
            ul.appendChild(li);
            td = document.createElement('td');
            td.appendChild(ul);
            tr.appendChild(td);
            tbody.appendChild(tr);
        }
        tr = document.createElement('tr');
        th = document.createElement('th');
        th.appendChild(document.createTextNode('yes?'));
        tr.appendChild(th);
        th = document.createElement('th');
        th.appendChild(document.createTextNode('id'));
        tr.appendChild(th);
        th = document.createElement('th');
        th.appendChild(document.createTextNode('name'));
        tr.appendChild(th);
        th = document.createElement('th');
        th.appendChild(document.createTextNode('url'));
        tr.appendChild(th);
        th = document.createElement('th');
        th.appendChild(document.createTextNode('action'));
        tr.appendChild(th);
        thead = document.createElement('thead');
        thead.appendChild(tr);
        table = document.createElement('table');
        table.setAttribute('border', 1);
        table.setAttribute('width', '100%');
        table.appendChild(thead);
        table.appendChild(tbody);
        document.getElementById('here').appendChild(table);
    };
```

使用 innerHTML 的好处在早期的浏览器上是显而易见的（在 IE 6 中使用 innerHTML 要比使用 DOM 快三四倍），但在最新版本的浏览器上就不那么明显了。而在最新的基于 WebKit 的浏览器上结果正好相反，使用 DOM 方法更快。因此，采用哪种方法将取决于用户经常使用的浏览器，以及个人的编码偏好。

要在一个性能苛刻的操作中更新一大块 HTML 页面，innerHTML 在大多数浏览器中执行得更快。但对于大多数日常操作而言，使用 innerHTML 和使用 DOM 的差异并不大，应当根据代码可读性、可维护性、团队习惯，以及代码风格来综合决定采用哪种方法。

建议 110：警惕文档遍历中的空格 Bug

在遍历 DOM 文档元素时，空格的存在很容易造成误解，因为 DOM 把空格也作为一个

节点进行解析（包括换行符）。通过下面的代码可以检测，DOM 把元素之间的空格也视为一个文本节点。

```
var e = document.documentElement.lastChild.firstChild.nodeName;
```

为了解决这个问题，可以定义一个功能函数，设计在文档结构加载完毕之后调用该函数清除所有仅包含空格的文本节点。这样当执行文档遍历时，就不存在元素之间的空格影响了。详细代码如下：

```
// 清除指定元素及其所有子元素之间的空格
// 参数：指定要清除空格的起始节点
function clean(e){
    var e = e || document;
    var f = e.firstChild;
    while(f != null){
        if(f.nodeType == 3 && /\s/.test(f.nodeValue)){ /
            e.removeChild(f);
        }
        else if(f.nodeType == 1){
            arguments.callee(f);
        }
        f = f.nextSibling;
    }
}
```

在准备遍历 DOM 文档时，可以先调用上面这个功能函数清除所有元素之间的空格，然后再遍历文档。这样就可以实现在不同浏览器中进行兼容，并准确定位节点的位置。通过这种方式先清除空格，再实施遍历，不仅不会对 HTML 渲染产生副作用，还会让定位 DOM 变得更容易。但需要注意的是，该函数的功能仅是临时性地清除元素之间的空格，需要在 HTML 文档的每一次加载时都重新执行一遍清除操作。当然，使用这种方法虽然比较高效，但是在每次遍历文档之前，都需要先执行一次遍历操作，如果文档的结构比较庞杂，那么这个操作所消耗的系统资源是不可小视的。一般应该尝试使用其他途径来定位节点，不可轻易地使用这种比较原始的方法。

建议 111：克隆节点比创建节点更好

使用 DOM 方法更新页面内容的另一个途径是克隆已有 DOM 元素，而不是创建新的元素，也就是使用 element.cloneNode()（element 是一个已存在的节点）代替 document.createElement()。

在大多数浏览器上，克隆节点更有效率，但提高得不太多。用克隆节点的办法创建 1000 行表格，只创建一次单元格，然后重复执行复制操作，这样会快一些。使用 element.cloneNode() 创建表格的代码如下：

```
    function tableClonedDOM() {
        var i, table, thead, tbody, tr, th, td, a, ul, li, oth = document.createElement('th'),
otd = document.createElement('td'), otr = document.createElement('tr'), oa = document.
createElement('a'), oli = document.createElement('li'), oul = document.createElement('ul');
        tbody = document.createElement('tbody');
        for( i = 1; i <= 1000; i++) {
            tr = otr.cloneNode(false);
            td = otd.cloneNode(false);
            td.appendChild(document.createTextNode((i % 2) ? 'yes' : 'no'));
            tr.appendChild(td);
            td = otd.cloneNode(false);
            td.appendChild(document.createTextNode(i));
            tr.appendChild(td);
            td = otd.cloneNode(false);
            td.appendChild(document.createTextNode('my name is #' + i));
            tr.appendChild(td);
            // ....
        }
        // ...
    }
```

建议 112：谨慎使用 HTML 集合

HTML 集合是用于存放 DOM 节点引用的类数组对象。下列方法的返回值都是一个集合：

- document.getElementsByName()
- document.getElementsByClassName()
- document.getElementsByTagName_r()

下列属性也属于 HTML 集合：

- document.images：页面中所有的 <img> 元素。
- document.links：所有的 <a> 元素。
- document.forms：所有表单。
- document.forms[0].elements：页面中第一个表单的所有字段。

这些方法和属性返回 HTMLCollection 对象，是一种类似数组的列表。它不是数组，因为它没有数组的方法，比如 push()、slice() 等，但它提供了一个 length 属性，与数组一样可以使用索引访问列表中的元素。例如，document.images[1] 返回集合中的第二个元素。正如 DOM 标准中所定义的那样，HTML 集合是一个虚拟存在，意味着当底层文档更新时它将自动更新。

HTML 集合实际上在查询文档，当更新信息时，每次都要重复执行这种查询操作。例如，读取集合中元素的数目，也就是集合的 length。这正是执行低效率的原因。

```
var alldivs = document.getElementsByTagName_r('div');
for(var i = 0; i < alldivs.length; i++) {
    document.body.appendChild(document.createElement('div'))
}
```

上面这段代码看上去只是简单地增加了页面中 div 元素的数量：遍历现有 div，每次创建一个新的 div 并附加到 body 上面。实际上这是个死循环，因为循环终止条件 alldivs.length 在每次迭代中都会增加，它反映出底层文档的当前状态。

像这样遍历 HTML 集合会导致逻辑错误，而且也很慢，因为每次迭代都需要进行查询，所以不建议用数组的 length 属性做循环判断条件。访问集合的 length 比数组的 length 还要慢，因为这意味着每次都要重新运行查询过程。在下面的例子中，将一个集合 coll 复制到数组 arr 中，然后比较每次迭代所用的时间。

```
function toArray(coll) {
    for(var i = 0, a = [], len = coll.length; i < len; i++) {
        a[i] = coll[i];
    }
    return a;
}
```

设置一个集合，并把它复制到一个数组：

```
var coll = document.getElementsByTagName_r('div');
var ar = toArray(coll);
```

比较下列两个函数：

```
// 比较慢
function loopCollection() {
    for (var count = 0; count < coll.length; count++) {
    }
}
// 比较快
function loopCopiedArray() {
    for (var count = 0; count < arr.length; count++) {
    }
}
```

当每次迭代过程访问集合的 length 属性时，将会导致集合器更新，在所有浏览器上都会产生明显的性能损失。优化的办法很简单，只要将集合的 length 属性缓存到一个变量中，然后在循环判断条件中使用这个变量。

```
function loopCacheLengthCollection() {
    var coll = document.getElementsByTagName_r('div'),
    len = coll.length;
    for (var count = 0; count < len; count++) {
    }
}
```

上面函数的运行速度与 loopCopiedArray() 一样快。

遍历数组比遍历集合快，如果先将集合元素复制到数组，访问它们的属性将更快。记住这需要一个额外的步骤——遍历集合，因此，应当评估在特定条件下使用这样一个数组副本是否有益。

建议 113：用局部变量访问集合元素

一般来说，访问任何类型的 DOM，当同一个 DOM 属性或方法被访问一次以上时，最好使用一个局部变量缓存该 DOM 成员。当遍历一个集合时，第一个要优化的是将集合引用存储于局部变量，并在循环之外缓存 length 属性。然后，如果在循环体中多次访问同一个集合元素，那么使用局部变量缓存它。

在下面示例中，循环访问每个元素的 3 个属性。执行最慢的方法是每次都要访问全局变量 document，优化后的代码缓存了一个指向集合的引用，执行最快的方法是将集合的当前元素存入局部变量。

```
// 较慢方法
function collectionGlobal() {
    var coll = document.getElementsByTagName_r('div'), len = coll.length, name = '';
    for(var count = 0; count < len; count++) {
        name = document.getElementsByTagName_r('div')[count].nodeName;
        name = document.getElementsByTagName_r('div')[count].nodeType;
        name = document.getElementsByTagName_r('div')[count].tagName;
    }
    return name;
};
// 较快方法
function collectionLocal() {
    var coll = document.getElementsByTagName_r('div'), len = coll.length, name = '';
    for(var count = 0; count < len; count++) {
        name = coll[count].nodeName;
        name = coll[count].nodeType;
        name = coll[count].tagName;
    }
    return name;
};
// 最快方法
function collectionNodesLocal() {
    var coll = document.getElementsByTagName_r('div'), len = coll.length, name = '',
el = null;
    for(var count = 0; count < len; count++) {
        el = coll[count];
        name = el.nodeName;
        name = el.nodeType;
        name = el.tagName;
    }
    return name;
};
```

建议 114：使用 nextSibling 抓取 DOM

DOM 提供了多种途径访问整个文档结构的特定部分。当在多种可行方法之间进行选择时，最好针对特定操作选择最有效的方法。

我们经常需要从一个 DOM 元素开始，操作周围的元素，或者递归迭代所有的子节点。这时可以使用 childNodes 集合或使用 nextSibling 获得每个元素的兄弟节点。

```
function testNextSibling() {
    var el = document.getElementById('mydiv'), ch = el.firstChild, name = '';
    do {
        name = ch.nodeName;
    } while (ch = ch.nextSibling);
    return name;
};
function testChildNodes() {
    var el = document.getElementById('mydiv'), ch = el.childNodes, len = ch.length, name = '';
    for(var count = 0; count < len; count++) {
        name = ch[count].nodeName;
    }
    return name;
};
```

比较上面两个功能相同的函数，它们都采用非递归方式遍历一个元素的子节点。childNodes 是一个集合，要小心处理，在循环中缓存 length 属性，避免在每次迭代中更新 length 的值。在不同浏览器上，这两种函数的运行时间基本相等，但在 IE 中，nextSibling 表现得比 childNodes 好。在 IE 6 中，nextSibling 比 childNodes 快 16 倍，而在 IE 7 中快 105 倍。鉴于这些结果，在旧版本 IE 中性能严苛的使用条件下，用 nextSibling 抓取 DOM 是首选，在其他情况下，主要依个人和团队的使用习惯而定。

建议 115：实现 DOM 原型继承机制

符合 DOM 标准的浏览器都支持 HTMLElement 类，DOM 文档中所有元素都继承于这个类，而 HTMLElement 对象又继承于 Element 类（Element 类继承于 Node 类），这样通过在 HTMLElement 类的原型对象上定义方法，为 HTML DOM 文档中所有元素绑定函数和数据。例如：

```
HTMLElement.prototype.pre = function(){  // 扩展原型方法
    var e = this.previousSibling;
    while (e && e.nodeType != 1){
        e = e.previousSibling;
    }
    return e;
}
```

为 HTMLElement 类的原型对象定义方法，这样每个 HTML DOM 元素都会继承这个方法。注意，在函数体内，应该通过关键字 this 来指向当前元素对象，而不用从参数变量中获取当前元素。在应用这个原型方法时，可以直接把它绑定到元素后面，例如：

```
window.onload = function(){
    var e = document.getElementsByTagName("div")[0];
    e = e.pre();
    alert(e.nodeName);
}
```

这样通过 HTMLElement 类的原型对象 prototype 就可以很方便地扩展每个 HTML 元素的方法或属性。但 IE 隐藏了这个类，禁止通过 JavaScript 脚本来访问它。为了能够兼容 IE，可以通过扩展方法解决这个问题。扩展方法如下：

```
var DOMElement ={
    extend : function(name, fn){
        // 添加名称为 name 的方法 fn
        if( ! document.all)
        // IE 之外的浏览器都能够访问到 HTMLElement 这个类
            eval("HTMLElement.prototype." + name + " = fn");
        else{
            // 在 IE 中不能访问 HTMLElement 这个类
            // 为了达到同样的目的，必须重写下面几个函数
            // document.createElement
            // document.getElementById
            // document.getElementsByTagName
            // 这几个函数都是获得 HTML 元素的方法
            // 修改这些方法，使通过这些方法获得的每个元素拥有名称为 name 的方法 fn
            var _createElement = document.createElement;
            document.createElement = function(tag){
                var _elem = _createElement(tag);
                eval("_elem." + name + " = fn");     // _elem[name] = fn; 也可以达到同样的目的
                return _elem;
            }
            var _getElementById = document.getElementById;
            document.getElementById = function(id){
                var _elem = _getElementById(id);
                eval("_elem." + name + " = fn");
                return _elem;
            }
            var _getElementsByTagName = document.getElementsByTagName;
            document.getElementsByTagName = function(tag){
                var _arr = _getElementsByTagName(tag);
                for(var _elem = 0; _elem < _arr.length; _elem ++ )
                eval("_arr[_elem]." + name + " = fn");
                return _arr;
            }
        }
    }
};
```

上面扩展函数的设计思路比较灵巧，实现方法也很简单，下面进行详细分析。

首先，利用 Document 对象的 All 对象来判断浏览器的类型，因为只有 IE 支持 All 对象集合。

对于非 IE 浏览器来说，由于它们一般都支持 DOM 标准模型，因此可以直接使用 HTMLElement.prototype 来设计原型方法，实现被所有文档元素继承。

对于 IE 来说，它不能够对原型对象进行设计，用户只能够通过 Document 对象的 getElementById() 或 getElementsByTagName() 方法来获取文档中的元素，或者通过 Document 对象的 createElement() 方法创建一个新的元素。那么，只要把元素将要继承的方法绑定到这些方法内部就可以间接实现所有元素都拥有这个方法。因为，获取文档中的元素只能够通过这 3 种方法中的一种才可以实现。把住这 3 种方法的关口，即可实现为当前元素捎带一个继承方法的目的。

由于 Document 对象允许用户重写这些方法，那么可以在重写过程中顺便加入用户指定的方法。为了避免在重写方法过程中破坏原方法的引用，可以先借助变量存储原方法的引用：

```
var _createElement = document.createElement;
```

然后重写方法，在重写过程中先执行原方法的代码：

```
var _elem = _createElement(tag);
```

接着，通过动态形式，为当前元素加入一个用户的方法：

```
eval("_elem." + name + " = fn");
```

最后，再返回原方法执行的值，这样既不会破坏原方法的功能，又为当前元素扩展了一个方法。具体应用的方法如下：

```
DOMElement.extend("pre",function(){
    var e = this.previousSibling;
    while (e && e.nodeType != 1){
        e = e.previousSibling;
    }
    return e;
})
```

在应用 DOMElement.extend() 方法时，应确保在获取或创建元素之前执行它，即在调用 Document 对象的 getElementById()、getElementsByTagName() 和 createElement() 方法之前调用 DOMElement.extend() 方法。在页面初始化后为当前元素调用扩展的方法，实际上这个方法与上面示例中 HTMLElement 类原型方法是完全相同的。

```
window.onload = function(){
    var e = document.getElementsByTagName("div")[0];
    e = e.pre();
```

```
        alert(e.nodeName);
    }
```

建议 116：推荐使用 CSS 选择器

childNodes、firstChild 和 nextSibling 属性是不区分元素节点和其他类型节点的，如注释节点和文本节点（这两个标签之间往往只是一些空格）。在许多情况下，只有元素节点会被访问，所以在循环中，似乎应当对节点返回类型进行检查，过滤出非元素节点。事实上，这些检查和过滤都是不必要的 DOM 操作。

目前许多浏览器提供了只返回元素节点的 API 函数，如果可用最好利用起来，因为写这些函数比在 JavaScript 中写过滤函数要快。

遍历 children 比遍历 childNodes 更快，因为集合项更少。HTML 源码中的空格实际上是文本节点，它们不包括在 children 集合中。在所有浏览器中，children 比 childNodes 更快，差别不是太大，通常只快 1.5 ～ 3 倍。特别值得注意的是，在 IE 中，遍历 children 明显快于遍历 childNodes，在 IE 6 中快 24 倍，在 IE 7 中快 124 倍。

选择 DOM 元素经常需要更精细的控制，而不只是采用 getElementById() 和 getElementsByTagName_r() 之类的函数。有时结合这些函数调用并迭代操作它们返回的节点，以获取所需要的元素，这一精细的过程可能使效率降低。

另外，使用 CSS 选择器是一个便捷的确定节点的方法，这是因为大家已经对 CSS 很熟悉了。许多 JavaScript 库为此提供了 API，而且最新的浏览器提供了一个名为 querySelectorAll() 的原生浏览器 DOM 函数。显然这种方法比使用 JavaScript 和 DOM 迭代并缩小元素列表的方法要快。

```
var elements = document.querySelectorAll('#menu a');
```

elements 的值将包含一个引用列表，指向那些具有 id="menu" 属性的元素。函数 querySelectorAll() 接收一个 CSS 选择器字符串参数并返回一个 NodeList（由符合条件的节点构成的类数组对象）。此函数不返回 HTML 集合，这就避免了 HTML 集合所固有的性能问题，以及潜在的逻辑问题。如果不使用 querySelectorAll()，达到同样目标的代码会冗长一些。

```
var elements = document.getElementById('menu').getElementsByTagName_r('a');
```

在这种情况下，elements 将是一个 HTML 集合，要想得到与 querySelectorAll() 同样的返回值类型，还需要将它复制到一个数组中。

当需要联合查询时，使用 querySelectorAll() 更加便利。例如，在页面中，一些 div 元素的 class 名称是“warning”，另一些 class 名称是“notice”，可以用 querySelectorAll() 一次性获得这两类节点。

```
var errs = document.querySelectorAll('div.warning, div.notice');
```

如果不使用 querySelectorAll()，那么获得同样列表需要更多工作。一个办法是选择所有的 div 元素，然后通过迭代操作过滤出那些不需要的单元。

```
var errs = [], divs = document.getElementsByTagName_r('div'), classname = '';
for(var i = 0, len = divs.length; i < len; i++) {
    classname = divs[i].className;
    if(classname === 'notice' || classname === 'warning') {
        errs.push(divs[i]);
    }
}
```

比较上面两种不同的用法，使用选择器 querySelectorAll() 比使用 getElementsByTagName_r() 的性能要好很多。因此，如果浏览器支持 document.querySelectorAll()，那么最好使用它。如果使用 JavaScript 库所提供的选择器 API，那么确认一下该库是否确实使用了原生方法。如果不是，则需要将库升级到新版本。

还可以使用另一个函数 querySelector() 获取节点，它可以返回符合查询条件的第一个节点。由于 querySelectorAll() 和 querySelector() 这两个函数都是 DOM 节点的属性，所以可以使用 document.querySelector('.myclass') 来查询整个文档中的节点，或者使用 elref.querySelector('.myclass') 在子树中进行查询，其中 elref 是一个 DOM 元素的引用。

建议 117：减少 DOM 重绘和重排版次数

浏览器在完成所有页面 HTML 标记、JavaScript、CSS、图片下载后，将解析文件并创建两个内部数据结构。

- 一棵 DOM 树：表示页面结构。
- 一棵渲染树：表示 DOM 节点如何显示。

在渲染树中为每个需要显示的 DOM 树节点存放至少一个节点（隐藏的 DOM 元素在渲染树中没有对应节点）。将渲染树上的节点称为“框”或者“盒”，符合 CSS 模型的定义，将页面元素看做一个具有填充、边距、边框和位置的盒。一旦 DOM 树和渲染树构造完毕，浏览器就可以显示（绘制）页面上的元素了。

当 DOM 改变影响到元素的几何属性（宽和高）时，如改变边框宽度或在段落中添加文字将发生一系列后续动作，浏览器需要重新计算元素的几何属性，而且其他元素的几何属性和位置也会因此改变并受到影响。浏览器使渲染树上受到影响的部分失效，然后重构渲染树，这个过程称做重排版。当重排版完成时，浏览器会在一个重绘进程中重新绘制屏幕上受影响的部分。

不是所有的 DOM 改变都会影响几何属性。例如，改变一个元素的背景颜色不会影响它的宽度或高度。在这种情况下，只需要重绘（不需要重排版），因为元素的布局没有改变。

重绘和重排版是负担很重的操作，可能导致网页应用的用户界面失去响应。因此，应

尽可能减少这类事情的发生。当布局和几何发生改变时需要重排版。在下述情况中会发生重排版：

- 添加或删除可见的 DOM 元素。
- 元素位置改变。
- 元素尺寸改变（因为边距、填充、边框宽度、宽度和高度等属性改变）。
- 内容改变，如文本改变或图片被另一个不同尺寸的图片所替代。
- 最初的页面渲染。
- 浏览器窗口改变尺寸。

根据改变的性质，渲染树上或大或小的一部分需要重新计算。某些改变可能导致重排版整个页面，如当一个滚动条出现时。

因为计算量与每次重排版有关，因此大多数浏览器都通过队列化修改和批量显示来优化重排版过程。然而，可能经常不由自主地强迫队列进行刷新并要求立刻应用所有计划改变的部分。获取布局信息的操作将导致刷新队列动作，这意味着使用了下面这些方法：

- offsetTop、offsetLeft、offsetWidth、offsetHeight
- scrollTop、scrollLeft、scrollWidth、scrollHeight
- clientTop、clientLeft、clientWidth、clientHeight
- getComputedStyle()（在 IE 中此函数称为 currentStyle）

布局信息是由这些方法返回最新的数据，浏览器不得不运行渲染队列中待改变的项目并重新排版以返回正确的值。

在改变样式的过程中，最好不要使用前面列出的那些属性。任何一个访问都将刷新渲染队列，即使正在获取那些最近未发生改变的或与最新的改变无关的布局信息。例如，下面示例改变同一个风格属性 3 次。

```
var computed, tmp = '', bodystyle = document.body.style;
if(document.body.currentStyle) { // IE, Opera
    computed = document.body.currentStyle;
} else {   // W3C
    computed = document.defaultView.getComputedStyle(document.body, '');
}
bodystyle.color = 'red';
tmp = computed.backgroundColor;
bodystyle.color = 'white';
tmp = computed.backgroundImage;
bodystyle.color = 'green';
tmp = computed.backgroundAttachment;
```

在上面代码中，body 元素的前景色被改变了 3 次，在每次改变之后都导入了 computed 的风格。导入的属性 backgroundColor、backgroundImage 和 backgroundAttachment 与颜色改变无关。然而，浏览器需要刷新渲染队列并重排版，因为 computed 的风格是被查询而引发的。

更好的方法是不要在布局信息改变时查询 computed 风格。如果将查询 computed 风格的代码移到末尾，那么在所有浏览器上都会执行得更快。

```
bodystyle.color = 'red';
bodystyle.color = 'white';
bodystyle.color = 'green';
tmp = computed.backgroundColor;
tmp = computed.backgroundImage;
tmp = computed.backgroundAttachment;
```

由于重排版和重绘代价较高，因此，提高程序响应速度的一个好策略是减少此类操作发生的机会。为减少发生次数，应该将多个 DOM 和风格改变后合并到一个批次中一次性执行。

```
var el = document.getElementById('mydiv');
el.style.borderLeft = '1px';
el.style.borderRight = '2px';
el.style.padding = '5px';
```

上面代码中改变了 3 个样式属性，每次改变都影响到元素的几何属性，导致浏览器重排版了 3 次。目前大多数浏览器都优化了这种情况，只进行一次重排版，但在旧版本浏览器中，效率将十分低下。如果其他代码在这段代码运行时查询布局信息，将导致 3 次重布局发生。而且，此代码访问 DOM 4 次，可以被优化。

实现相同效果但效率更高的方法：将所有改变合并在一起执行，只修改 DOM 一次。具体可通过使用 cssText 属性实现：

```
var el = document.getElementById('mydiv');
el.style.cssText = 'border-left: 1px; border-right: 2px; padding: 5px;';
```

在这个示例中，修改 cssText 属性，覆盖已存在的风格信息。如果打算保持当前的风格，那么可以将它附加在 cssText 字符串的后面。

```
el.style.cssText += '; border-left: 1px;';
```

另一个方法是修改 CSS 的类名称，而不是修改内联风格代码。这种方法适用于那些风格不依赖于运行逻辑且不需要计算的情况。改变后的 CSS 类名称更清晰，更易于维护，虽然它可能带来轻微的性能冲击，但是有助于保持脚本免除显示代码。

```
var el = document.getElementById('mydiv');
el.className = 'active';
```

当需要对 DOM 元素进行多次修改时，可以通过以下步骤减少重绘和重排版的次数。

第 1 步，从文档流中摘除该元素。

第 2 步，对其应用多重改变。

第 3 步，将元素带回文档中。

此过程引发两次重排版：第 1 步引发一次，第 3 步引发一次。如果忽略了这两个步骤，

那么第 2 步中每次改变都将引发一次重排版。

经历以下 3 步后可以将 DOM 从文档中摘除：

- 隐藏元素，进行修改，然后再显示它。
- 使用一个文档片断在已存 DOM 之外创建一个子树，然后将它复制到文档中。
- 将原始元素复制到一个脱离文档的节点中，修改副本，然后覆盖原始元素。

下面示例中有一个链接列表，它必须被更多的信息所更新。

```
<ul id="mylist">
    <li><a href="#">链接 1</a></li>
    <li><a href="#">链接 2</a></li>
</ul>
```

假设附加数据已经存储在一个对象中了，需要将其插入到这个列表中。这些数据定义如下：

```
var data = [{
    "name" : "链接 3",
    "url" : "#"
}, {
    "name" : "链接 4",
    "url" : "#"
}];
```

下面是一个通用的函数，用于将新数据更新到指定节点中：

```
function appendDataToElement(appendToElement, data) {
    var a, li;
    for(var i = 0, max = data.length; i < max; i++) {
        a = document.createElement('a');
        a.href = data[i].url;
        a.appendChild(document.createTextNode(data[i].name));
        li = document.createElement('li');
        li.appendChild(a);
        appendToElement.appendChild(li);
    }
};
```

将数据更新到列表而不管重排版问题，最显著的方法如下：

```
var ul = document.getElementById('mylist');
appendDataToElement(ul, data);
```

然而，将 data 队列上的每个新条目追加到 DOM 树都会导致重排版。第一种减少重排版的方法：改变 display 属性，临时从文档上移除 <ul> 元素然后再恢复它。

```
var ul = document.getElementById('mylist');
ul.style.display = 'none';
appendDataToElement(ul, data);
ul.style.display = 'block';
```

第二种减少重排版的方法：在文档之外创建并更新一个文档片断，然后将它附加在原始

列表上。文档片断是一个轻量级的 document 对象，它被设计用于更新、移动节点之类的任务。文档片断一个便利的语法特性：在向节点附加一个片断时，实际添加的是文档片断的子节点群，而不是文档片断自己。下面的例子减少一行代码，只引发一次重排版。

```
var fragment = document.createDocumentFragment();
appendDataToElement(fragment, data);
document.getElementById('mylist').appendChild(fragment);
```

第三种减少重排版的方法：首先创建要更新节点的副本，然后在副本上操作，最后用新节点覆盖老节点。

```
var old = document.getElementById('mylist');
var clone = old.cloneNode(true);
appendDataToElement(clone, data);
old.parentNode.replaceChild(clone, old);
```

尽可能使用文档片断（第二种方法）来减少重排版，因为它涉及最少数量的 DOM 操作和重排版。唯一潜在的隐患：当前文档片断还没有得到充分利用。

浏览器通过队列化修改和批量运行的方法，尽量减少重排版次数。当查询布局信息如偏移量、滚动条位置或风格属性时，浏览器刷新队列并执行所有修改操作，以返回最新的数值。应尽量减少对布局信息的查询，查询时将查询次数赋给局部变量，并通过局部变量参与计算。

例如，将元素 myElement 向右下方向平移，每次一个像素，起始于 100 像素 ×100 像素位置，结束于 500 像素 ×500 像素位置，在 timeout 循环体中可以使用。

```
myElement.style.left = 1 + myElement.offsetLeft + 'px';
myElement.style.top = 1 + myElement.offsetTop + 'px';
if (myElement.offsetLeft >= 500) {
    stopAnimation();
}
```

这样做很没有效率，因为每次元素移动，代码查询偏移量，就会导致浏览器刷新渲染队列，并不会从优化中获益。还有一个办法，只需要获得起始位置值一次，将它存入局部变量中（var current = myElement.offsetLeft;），然后在动画循环中使用 current 变量而不再查询偏移量。

```
current++
myElement.style.left = current + 'px';
myElement.style.top = current + 'px';
if (current >= 500) {
    stopAnimation();
}
```

重排版有时只影响渲染树的一小部分，但也可能影响很大一部分，甚至整个渲染树。浏览器需要重排版的部分越小，应用程序的响应速度就越快，因此，当一个页面顶部的动画推

移了差不多整个页面时，将引发巨大的重排版动作，使用户感到动画不流畅。渲染树的大多数节点需要重新计算，这使情况变得更糟糕。

使用以下步骤可以避免对大部分页面进行重排版：

- 使用绝对坐标定位页面动画的元素，使它位于页面布局流之外。
- 启动元素动画，当它扩大时，将会临时覆盖部分页面。这是一个重绘过程，但只影响页面的一小部分，避免重排版及重绘一大块页面。
- 当动画结束时，重新定位。

建议 118：使用 DOM 树结构托管事件

当页面中存在大量元素，并且每个元素有一个或多个事件句柄连接（如 onclick）时，可能会影响性能。连接每个句柄都是有代价的，这代价可能是加重了页面负担（更多的页面标记和 JavaScript 代码），也可能表现在运行期的运行时间上。要访问和修改更多的 DOM 节点，程序就会更慢，特别是事件连接过程都发生在 onload（或 DOMContentReady）事件中时，对任何一个富交互网页来说这都是一个繁忙的时间段。连接事件占用了处理时间，另外，浏览器需要保存每个句柄的记录，也会占用更多内存。当这些工作结束时，由于这些事件句柄中的相当一部分根本不需要（因为并不是百分之百的按钮或链接都会被用户单击到），所以很多工作都是不必要的。

一个简单而优雅的处理 DOM 事件的技术是事件托管。它基于这样一个事实：事件逐层冒泡总能被父元素捕获。采用事件托管技术后，只需要在一个包装元素上连接一个句柄，用于处理子元素发生的所有事件。根据 DOM 标准，每个事件有 3 个阶段：

- 捕获
- 到达目标
- 冒泡

IE 不支持捕获，只要实现托管技术使用冒泡就足够了。

```
<div>
    <ul id="menu">
        <li><a urn="#1"></a></li>
    </ul>
</div>
```

例如，对于上面的结构，当用户单击“menu #1”链接时，单击事件首先被 <a> 元素收到，然后它沿着 DOM 树冒泡，被 <li> 元素收到，之后被 <ul> 元素收到，接着是 等，一直到达文档的顶层，甚至 window 对象。这使得可以只在父元素上连接一个事件句柄，以接收所有子元素产生的事件通知。

假设要为上面文档结构提供一个逐步增强的 Ajax 体验，用户关闭了 JavaScript，菜单中的链接仍然可以正常地重载页面。当已经打开 JavaScript 且用户代理有足够能力时，如果希

望截获所有单击，阻止默认行为（转入链接），发送一个 Ajax 请求获取内容，然后不刷新页面就能够更新部分页面，那么使用事件托管实现此功能，可以在 menu 单元连接一个单击监听器，它封装所有链接并监听所有 click 事件。

```
document.getElementById('menu').onclick = function(e) {
    e = e || window.event;
    var target = e.target || e.srcElement;
    var pageid, hrefparts;
    if(target.nodeName !== 'A') {
        return;
    }
    hrefparts = target.href.split('/');
    pageid = hrefparts[hrefparts.length - 1];
    pageid = pageid.replace('.html', '');
    ajaxRequest('xhr.php?page=' + id, updatePageContents);
    if( typeof e.preventDefault === 'function') {
        e.preventDefault();
        e.stopPropagation();
    } else {
        e.returnValue = false;
        e.cancelBubble = true;
    }
};
```

事件托管技术并不复杂，只需要通过监听事件侦测事件是不是从目标元素中发出的。这里有一些冗余的跨浏览器代码，如果将它们移入一个可重用的库中，代码就变得相当干净。跨浏览器部分包括：

- 访问事件对象，并判断事件源（目标）。
- 结束文档树上的冒泡（可选）。
- 阻止默认动作（可选，在本示例中是必须的，因为任务是捕获这些链接而不转入这些链接）。

建议 119：使用定时器优化 UI 队列

在 JavaScript 中使用 setTimeout() 或 setInterval() 创建定时器时，这两个函数都接收一样的参数：一个是要执行的函数，另一个是执行这个函数之前的等待时间（单位毫秒）。setTimeout() 函数创建一个只运行一次的定时器，而 setInterval() 函数创建一个周期性重复运行的定时器。

定时器与 UI 线程交互的方式有助于分解长运行脚本为较短的片断。调用 setTimeout() 或 setInterval() 告诉 JavaScript 引擎等待一定时间，然后将 JavaScript 任务添加到 UI 队列中。例如：

```
function greeting(){
```

```
    alert("Hello world!");
}
setTimeout(greeting, 250);
```

在上面代码中，在 250 ms 之后向 UI 队列插入一个 JavaScript 任务来运行 greeting() 函数。在此时间点之前，所有其他 UI 更新和 JavaScript 任务都在运行。记住，第二个参数指出什么时候应当将任务添加到 UI 队列之中，并不是说那时代码将被执行，这个任务必须等到队列中的其他任务都执行之后才能被执行。例如：

```
var button = document.getElementById("my-button");
button.onclick = function(){
    oneMethod();
    setTimeout(function(){
        document.getElementById("notice").style.color = "red";
    }, 250);
};
```

在上面示例中，当按钮被单击时，将调用一个方法设置一个定时器。用于修改 notice 元素颜色的代码被包含在一个定时器设备中，它将在 250 ms 之后被添加到队列中。250 ms 是从调用 setTimeout() 时开始计算的，而不是从整个函数运行结束时开始计算的。如果 setTimeout() 在时间点 n 上被调用，那么运行定时器代码的 JavaScript 任务将在 n+250 的时刻加入 UI 队列。

定时器代码只有等创建它的函数运行完成之后才有可能被执行。假设在前面的代码中定时器延时变得更小，在创建定时器之后又调用了另一个函数，那么定时器代码有可能在 onclick 事件处理完成之前加入队列。

```
var button = document.getElementById("my-button");
button.onclick = function(){
    oneMethod();
    setTimeout(function(){
        document.getElementById("notice").style.color = "red";
    }, 50);
    anotherMethod();
};
```

如果 anotherMethod() 执行时间超过 50 ms，那么定时器代码将在 onclick 处理完成之前加入到队列中。其结果是等 onclick 处理运行完毕，定时器代码立即执行，察觉不出其间的延迟。

在任何一种情况下，创建一个定时器会造成 UI 线程暂停，如同定时器会从一个任务切换到下一个任务。因此，定时器代码复位所有相关的浏览器限制，包括长运行脚本时间。此外，调用栈也在定时器代码中复位为零。这一特性使定时器成为长运行 JavaScript 代码理想的跨浏览器解决方案。

JavaScript 定时器延时往往不准确，快慢大约几毫秒。指定定时器延时 250 ms，并不意味任务将在调用 setTimeout() 之后精确的 250 ms 后加入队列。所有浏览器试图尽可能准确，

但通常会发生几毫秒的滑移，或快或慢。正因为这个原因，定时器不可用于测量实际时间。

在 Windows 系统上定时器的分辨率为 15 ms，也就是说，一个值为 15 的定时器延时将根据最后一次系统时间的刷新而转换为 0 或 15。由于设置定时器延时小于 15 将在 IE 中导致浏览器锁定，所以建议最小值为 25 ms（实际时间是 15 ms 或 30 ms），以确保至少 15 ms 的延迟。

最小定时器延时也有助于避免其他浏览器和操作系统上产生的定时器分辨率问题。大多数浏览器在定时器延时小于 10 ms 时表现出差异性。

一个常见的长运行脚本就是循环占用了太长的运行时间。如果尝试循环优化之后还不能缩减足够的运行时间，那么定时器就是下一个优化步骤。基本方法是将循环工作分解到定时器序列中。典型的循环模式如下：

```
for (var i=0, len=items.length; i < len; i++){
    process(items[i]);
}
```

导致循环结构运行时间过长的因素有两个：process() 的复杂度和 items 的大小。这两个因素有可能同时存在。可用定时器取代循环的两个决定性因素如下：

- 处理过程不需要同步处理。
- 数据不需要按顺序处理。

一种基本异步代码模式如下：

```
var todo = items.concat();
setTimeout(function() {
    process(todo.shift());
    if(todo.length > 0) {
        setTimeout(arguments.callee, 25);
    } else {
        callback(items);
    }
}, 25);
```

这个模式的基本思想是创建一个原始数组的副本，将它作为处理对象。第一次调用 setTimeout() 创建一个定时器处理队列中的第一个项。调用 todo.shift() 返回它的第一个项，然后将它从数组中删除。第一项的值作为参数传给 process()。接着检查是否还有更多项需要处理。如果 todo 队列中还有内容，那么就再启动一个定时器。因为下个定时器需要运行相同的代码，所以将第一个参数传入 arguments.callee，此值指向当前正在运行的匿名函数。如果不再有内容需要处理，那么将调用 callback() 函数。此模式与循环相比需要更多代码，可将此功能封装起来，例如：

```
function processArray(items, process, callback) {
    var todo = items.concat();
    setTimeout(function() {
        process(todo.shift());
```

```
        if(todo.length > 0) {
            setTimeout(arguments.callee, 25);
        } else {
            callback(items);
        }
    }, 25);
}
```

processArray() 函数以一种可重用的方式实现了先前的模板，并且接收 3 个参数：待处理数组、对每个项调用的处理函数、处理结束时执行的回调函数。该函数用法如下：

```
var items = [123, 789, 323, 778, 232, 654, 219, 543, 321, 160];
function outputValue(value) {
    console.log(value);
}
processArray(items, outputValue, function() {
    console.log("Done!");
});
```

此段代码使用 processArray() 方法将数组值输出到终端，当所有处理结束时再打印一条消息。通过将代码封装在一个函数中，定时器可在多处重用而无须多次实现。

建议 120：使用定时器分解任务

通常将一个任务分解成一系列子任务。如果一个函数运行时间太长，那么可以考虑查看它是否可以分解成一系列能够短时间完成的较小的函数。可将一行代码简单地看做一个原子任务，多行代码组合在一起构成一个独立任务。某些函数可基于函数调用进行拆分，例如：

```
function saveDocument(id) {
    openDocument(id)
    writeText(id);
    closeDocument(id);
    updateUI(id);
}
```

如果函数运行时间太长，那么可以将它拆分成一系列更小的步骤，把独立方法放在定时器中调用。可以将每个函数都放入一个数组，然后使用前一节中提到的数组处理模式。

```
function saveDocument(id) {
    var tasks = [openDocument, writeText, closeDocument, updateUI];
    setTimeout(function() {
        var task = tasks.shift();
        task(id);
        if(tasks.length > 0) {
            setTimeout(arguments.callee, 25);
        }
    }, 25);
}
```

上面代码将每个方法放入任务数组，然后在每个定时器中调用一个方法。从根本上说，现在以上方法成为数组处理模式，只有一点不同：处理函数就包含在数组项中。该模式也可封装起来重用。

```
function multistep(steps, args, callback) {
    var tasks = steps.concat();
    setTimeout(function() {
        var task = tasks.shift();
        task.apply(null, args || []);
        if(tasks.length > 0) {
            setTimeout(arguments.callee, 25);
        } else {
            callback();
        }
    }, 25);
}
```

multistep() 函数接收 3 个参数：用于执行的函数数组、为每个函数提供参数的参数数组、当处理结束时调用的回调函数。函数用法如下：

```
function saveDocument(id) {
    var tasks = [openDocument, writeText, closeDocument, updateUI];
    multistep(tasks, [id], function() {
        alert("Save completed!");
    });
}
```

注意，传给 multistep() 的第二个参数必须是数组，它在创建时只包含一个 id。与数组处理一样，使用此函数的前提条件：任务可以异步处理而不影响用户体验或导致依赖代码出错。

建议 121：使用定时器限时运行代码

有时每次只执行一个任务，这样效率不高。考虑这样一种情况：处理一个拥有 1000 项的数组，每处理一项需要 1 ms。如果在每个定时器中处理一项，在两次处理之间间隔 25 ms，那么处理此数组的总时间是 (25 + 1) × 1000 = 26 000 ms，也就是 26 s。如果每批处理 50 个，每批之间间隔 25 ms，那么结果会怎么样呢？

整个处理过程变成 (1000 / 50) × 25 + 1000 = 1500 ms，也就是 1.5 s，而且用户也不会察觉界面阻塞，因为最长的脚本运行只持续了 50 ms。通常批量处理比每次处理一项速度更快。

如果记住 JavaScript 可连续运行的最大时间是 100 ms，那么可以优化先前的模式。建议将这个数字削减一半，不要让任何 JavaScript 代码持续运行超过 50 ms，这只是为了确保代码永远不会影响用户体验。可通过原生的 Date 对象跟踪代码的运行时间，这是大多数 JavaScript 分析工具所采用的工作方式，例如：

```
var start = +new Date(), stop;
```

```
someLongProcess();
stop = +new Date();
if(stop - start < 50) {
    alert("Just about right.");
} else {
    alert("Taking too long.");
}
```

由于每个新创建的 Data 对象都以当前系统时间初始化，因此可以周期性地创建新 Data 对象并比较它们的值，以获取代码运行时间。通过加号（+）将 Data 对象转换为一个数字，这样在后续的数学运算中就不必再转换了。这一技术也可用于优化以前的定时器模板。timedProcessArray() 方法通过一个时间检测机制可在每个定时器中执行多次处理，例如：

```
function timedProcessArray(items, process, callback) {
    var todo = items.concat();
    setTimeout(function() {
        var start = +new Date();
        do {
            process(todo.shift());
        } while (todo.length > 0 && (+new Date() - start < 50));
        if(todo.length > 0) {
            setTimeout(arguments.callee, 25);
        } else {
            callback(items);
        }
    }, 25);
}
```

此函数中添加了一个 do-while 循环，它在处理完每个数组项后检测时间。在定时器函数运行时，因为数组中存放了至少一个项，所以后对循环进行测试比先测试更合理。在 Firefox 3 中，如果 process() 是一个空函数，处理一个 1000 项的数组需要 34 ~38 ms，那么原始的 timedProcessArray() 函数处理同一个数组需要超过 25 000 ms。这就是定时任务的作用，避免将任务分解成过于琐碎的片断。

建议 122：推荐网页工人线程

自 JavaScript 诞生以来，还没有办法在浏览器 UI 线程之外运行代码。网页工人线程 API 改变了这种状况，它引入一个接口，使代码运行而不占用浏览器 UI 线程的时间。作为最初的 HTML 5 的一部分，网页工人线程 API 已经分离出去成为独立的规范（http://www.w3.org/TR/workers/）。网页工人线程已经被 Firefox 3.5、Chrome 3 和 Safari 4 原生实现。

网页工人线程对网页应用来说是一个潜在的巨大性能提升，因为新的工人线程在自己的线程中运行 JavaScript。这意味着，工人线程中运行的代码不仅不会影响浏览器 UI 线程，而且也不会影响其他工人线程中运行的代码。

由于网页工人线程不绑定浏览器 UI 线程，这也意味着它们将不能访问许多浏览器资源。JavaScript 和 UI 更新共享同一个进程的部分原因是它们之间互访频繁，如果互访失控将导致糟糕的用户体验。网页工人线程修改 DOM 将导致用户界面出错，因为每个网页工人线程都有自己的全局运行环境，只有 JavaScript 特性的一个子集可用。工人线程的运行环境由下列部分组成：

- 一个浏览器对象，只包含 4 个属性：appName、appVersion、userAgent 和 platform。
- 一个 Location 对象（和 Window 对象的一样，只是所有属性都是只读的）。
- 一个 Self 对象指向全局工人线程对象。
- 一个 importScripts() 方法，使工人线程可以加载外部 JavaScript 文件。
- 所有 ECMAScript 对象，如 Object、Array、Data 等。
- XMLHttpRequest 构造器。
- setTimeout() 和 setInterval() 方法。
- close() 方法可立即停止工人线程。

因为网页工人线程有不同的全局运行环境，所以不能在 JavaScript 代码中创建网页工人线程。事实上，需要创建一个完全独立的 JavaScript 文件，以包含那些在工人线程中运行的代码。要创建网页工人线程，必须传入这个 JavaScript 文件的 URL：

```
var worker = new Worker("code.js");
```

此代码一旦执行，将为指定文件创建一个新线程和一个新的工人线程运行环境。此 JavaScript 文件被异步下载，直到下载并运行完此文件之后才启动工人线程。

工人线程和网页代码通过事件接口进行交互。网页代码可通过 postMessage() 方法向工人线程传递数据，它接收单个参数，即传递给工人线程的数据。此外，在工人线程中还有 onmessage 事件句柄用于接收信息。例如：

```
var worker = new Worker("code.js");
worker.onmessage = function(event){
    alert(event.data);
};
worker.postMessage("Nicholas");
```

网页工人线程从 message 事件中接收数据。这里定义了一个 onmessage 事件句柄，事件对象具有一个 data 属性用于存放传入的数据。网页工人线程可通过它自己的 postMessage() 方法将信息返回给页面。

```
self.onmessage = function(event){
    self.postMessage("Hello, " + event.data + "!");
};
```

最终的字符串结束于网页工人线程的 onmessage 事件句柄。消息系统是页面和网页工人线程之间唯一的交互途径。只有某些类型的数据可以使用 postMessage() 传递，这些数据可

以是原始值（string、number、boolean、null 和 undefined），也可以是 Object 和 Array 的实例，其他类型的数据就不允许传递了。有效数据被序列化，然后传入或传出工人线程，最后反序列化。当工人线程通过 importScripts() 方法加载外部 JavaScript 文件时，它接收一个或多个 URL 参数来指出要加载的 JavaScript 文件网址。工人线程以阻塞方式调用 importScripts()，直到所有文件加载完成并执行之后，脚本才继续运行。由于网页工人线程在 UI 线程之外运行，因此这种阻塞不会影响 UI 响应。例如：

```
importScripts("file1.js", "file2.js");
self.onmessage = function(event){
    self.postMessage("Hello, " + event.data + "!");
};
```

此代码第一行包含两个 JavaScript 文件，它们将在网页工人线程中使用。

网页工人线程适合于那些纯数据的或与浏览器 UI 没关系的长运行脚本。这种线程看起来用处不大，不过在网页应用程序中通常有一些数据处理功能将受益于网页工人线程，而不是定时器。

例如，解析一个很大的 JSON 字符串（JSON 解析将在第 7 章讨论）。假设数据足够大，至少需要 500 ms 才能完成解析任务。很显然，时间太长会导致不允许 JavaScript 在客户端上运行网页工人线程，因为它会干扰用户体验。由于此任务难以分解成用于定时器的小段任务，所以工人线程成为理想的解决方案。下面的代码说明了网页工人线程在网页上的应用。

```
var worker = new Worker("jsonparser.js");
worker.onmessage = function(event){
    var jsonData = event.data;
    evaluateData(jsonData);
};
worker.postMessage(jsonText);
```

工人线程的代码负责 JSON 解析，例如：

```
self.onmessage = function(event){
    var jsonText = event.data;
    var jsonData = JSON.parse(jsonText);
    self.postMessage(jsonData);
};
```

注意，即使 JSON.parse() 可能需要 500 ms 或更多时间，也没有必要添加更多代码来分解处理过程。由于此处理过程发生在一个独立的线程中，因此可以让它一直运行完解析过程而不会干扰用户体验。

页面使用 postMessage() 将一个 JSON 字符串传给工人线程。工人线程在它的 onmessage 事件句柄中收到这个字符串也就是 event.data，然后开始解析它。完成解析时所产生的 JSON 对象通过工人线程的 postMessage() 方法传回页面，此后此对象便成为页面 onmessage 事件句柄的 event.data。记住，此工程只能在 Firefox 3.5 及其更高版本中运行，而在 Safari 4 和

Chrome 3 中，页面和网页工人线程之间只允许传递字符串。解析一个大字符串只是许多受益于网页工人线程的任务之一。其他可能受益的任务如下：

- 编 / 解码一个大字符串。
- 复杂数学运算（包括图像或视频处理）。
- 给一个大数组排序。

在进行任何超过 100 ms 的处理时，都应当考虑工人线程方案是不是比基于定时器的方案更合适，当然，还要考虑浏览器是否支持工人线程。

第 6 章
客户端编程

HTML、JavaScript 和 CSS 构成了客户端开发的三块基石，缺一不可。JavaScript 通过事件处理模型实现与 HTML 的交互，而 CSS 可以直接作用于 HTML 结构。在 JavaScript 开发中，CSS 的作用不容忽视，很多交互效果都需要 CSS 的配合才能够实现。使用 CSS 和 JavaScript 可以创造出各种奇幻的视觉效果，利用脚本化 CSS 样式的能力，可以动态改变 HTML 的颜色、字体等。更重要的是，可以用 JavaScript 设置和改变元素的位置，甚至隐藏或显示元素，这意味着可以使用 CSS 设计动画效果。

用户交互的效果是通过操作与响应来实现的，执行操作的可以是人（即用户），也可以是物（即浏览器），做出响应的是页面上的各种对象，或者浏览器自身。在操作与响应之间，需要事件作为桥梁。鼠标单击是一个事件，单击按钮就触发了页面响应，也许会弹出一个提示框，也许会提交表单信息，也许会关闭窗口等。所有这些行为都可以由开发人员自己去设计。

建议 123：比较 IE 和 W3C 事件流

IE 解决事件流的方案称为冒泡（dubbed bubbling）技术。冒泡型事件流的基本思路：事件流按照从最特定的事件目标到最不特定的事件目标（document 对象）的顺序触发。简单概括为事件从下向上传递，这个传递过程犹如水冒泡一样不断上升到顶端。例如：

```
<script language="javascript" type="text/javascript">
function f(a){
    alert(a);
}
</script>
<body onclick="f('BODY')">
    <p onclick="f('P')">冒泡型事件</p>
</body>
```

在这个示例中，单击段落文本会触发事件流。首先响应的是 p 元素身上绑定的鼠标单击事件，即弹出提示信息为“P”。接着响应的是 body 元素身上绑定的鼠标单击事件，即弹出提示信息为“BODY”。整个事件流动的顺序是从下（p 元素）到上（body 元素）的，如图 6.1 所示。

图 6.1　IE 6.0 以下版本浏览器的冒泡过程

对于不同类型或版本的浏览器来说，冒泡型事件流的具体约定也略有不同。

- IE 5.5 及其以下版本：p → body → document。
- IE 6.0 及其以上版本：p → body → html → document。
- Firefox：p → body → html → document → window。

从 IE 6.0 版本浏览器开始，html 元素也可以接收到事件流，事件流能够影响到 html 元素，而这在 IE 5.5 及其以下版本中是不允许的。例如：

```
<script language="javascript" type="text/javascript">
function f(a){
    alert(a);
}
</script>
<html onclick="f('HTML')">
<body onclick="f('BODY')">
    <p onclick="f('P')"> 冒泡型事件 </p>
</body>
</html>
```

在这个示例中，单击段落文本会触发事件流。首先响应的是 p 元素身上绑定的鼠标单击事件，接着响应的是 body 元素身上绑定的鼠标单击事件，最后是 html 元素身上绑定的鼠标单击事件，如图 6.2 所示。Firefox 类型浏览器支持冒泡型事件流，但它能够影响到全局作用域，也就是说，事件流最后会影响到 window 对象，如图 6.3 所示。

捕获型事件流与冒泡型事件流正好相反，事件总是从最不精确的对象（document 对象）开始触发，最后到最精确的对象，即事件流从上到下按顺序响应。该解决方案从 Netscape 4.0 版本浏览器开始得到支持。例如，上面的示例在 Netscape 4.0 及其以上版本浏览器中运行，单击段落文本会首先触发 html 元素，接着是 body 元素，最后是 p 元素，如图 6.4 所示。

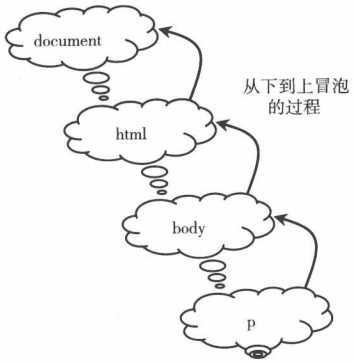


图 6.2　IE 6.0 及其以上版本浏览器的冒泡过程

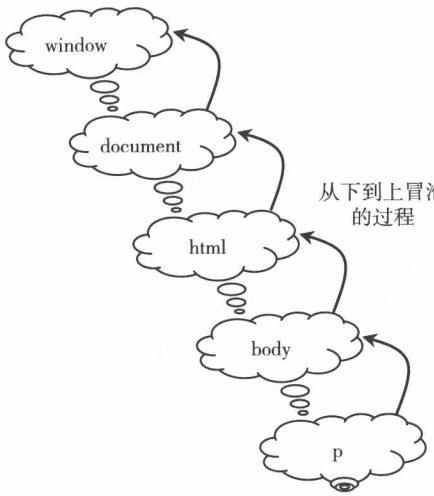


图 6.3　Firefox 浏览器的冒泡过程

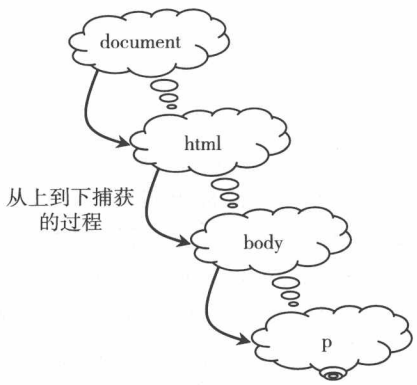


图 6.4　Netscape 4.0 及其以上版本浏览器的捕获过程

DOM 2.0 对事件流进行了标准化，同时支持冒泡型事件流和捕获型事件流。DOM 2.0 标准规定捕获型事件流先进行响应，然后才响应冒泡型事件流。这两种事件流会触及 DOM 中的所有对象，从 document 对象开始，最后在 document 对象结束。

不过大部分浏览器在支持 DOM 标准事件流时，会影响 window 对象。例如，针对上面的示例，在兼容 DOM 标准的浏览器中单击段落文本，事件流会按如图 6.5 所示的过程进行传导。

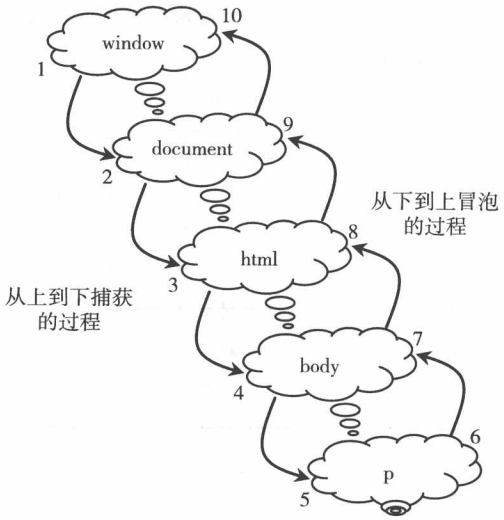


图 6.5　DOM 2.0 标准的事件流过程

在默认情况下，事件使用冒泡型事件流，开发者可以显式设置并使用捕获型事件流，方法是在注册事件时传入 useCapture 参数，将这个参数设为 true。除了元素能够响应事件外，DOM 标准还规定文本节点也可以响应事件，但 IE 并不支持响应事件。

建议 124：设计鼠标拖放方案

mousemove 是一个实时响应的事件类型，当鼠标指针的位置发生变化时（至少移动 1 个像素）就会触发 mousemove 事件。该事件响应的灵敏度主要参考鼠标指针移动速度，以及浏览器跟踪更新的速度。要对鼠标拖放操作进行设计，需要理清和解决以下几个问题。

- 定义拖放元素为绝对定位，以及设计事件的响应过程，这个比较容易实现。
- 清楚几个坐标概念：按下鼠标按键时的指针坐标，移动中当前鼠标的指针坐标，松开鼠标按键时的指针坐标，拖放元素的原始坐标，拖动中的元素坐标。
- 算法设计：按下鼠标按键时，获取被拖放元素和鼠标指针的位置，在移动中实时计算鼠标偏移的距离，利用该偏移距离加上被拖放元素的原坐标位置来获得拖放元素的实时坐标。

如图 6.6 所示，其中变量 ox 和 oy 分别记录按下鼠标时被拖放元素的横坐标值和纵坐标值，它们可以通过事件对象的 offsetLeft 和 offsetTop 属性获取。变量 mx 和 my 表示按下鼠标按键时鼠标指针的坐标位置。而 event.mx 和 event.my 是事件对象的自定义属性，用它们来存储当鼠标移动时鼠标指针的实时位置。

在获取了上面提到的 3 对坐标值之后，就可以动态计算处在拖动状态中的元素的实时坐标位置，即 x 轴方向的值为 ox + event.mx – mx，y 轴方向的值为 oy + event.my – my。当释放鼠标按键时，可以释放事件类型，并记下松开鼠标指针时被拖动元素的坐标值及鼠标指针的位置，留待下一次拖放操作时调用。

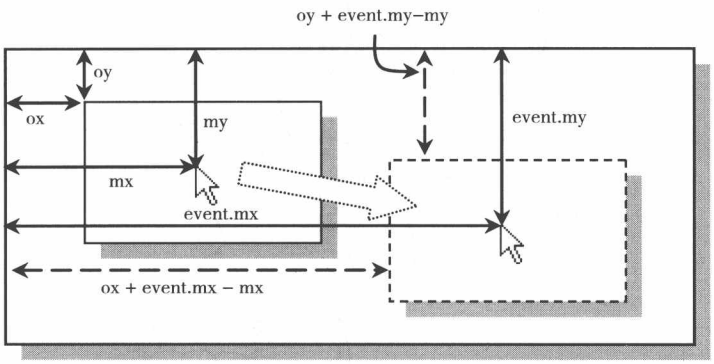


图 6.6　拖放操作设计示意图

鼠标拖放操作的设计方案如下：

```
<div id="box" ></div>
<script language="javascript" type="text/javascript">
// 初始化拖放对象
var box = document.getElementById("box");          // 获取页面中被拖放元素的引用指针
box.style.position = "absolute";
box.style.width = "160px";
box.style.height = "120px";
box.style.backgroundColor = "red";
// 初始化变量，标准化事件对象
var mx, my, ox, oy;
function e(event){ // 定义事件对象标准化函数
   if( ! event){
      event = window.event;
      event.target = event.srcElement;
      event.layerX = event.offsetX;
      event.layerY = event.offsetY;
   }
   event.mx = event.pageX || event.clientX + document.body.scrollLeft; // 计算鼠标指针的 x 轴距离
   event.my = event.pageY || event.clientY + document.body.scrollTop; // 计算鼠标指针的 y 轴距离
   return event;
}
// 定义鼠标事件处理函数
document.onmousedown = function(event){ // 按下鼠标按键时初始化处理
```

```
    event = e(event);
    o = event.target;
    ox = parseInt(o.offsetLeft);
    oy = parseInt(o.offsetTop);
    mx = event.mx;
    my = event.my;
    document.onmousemove = move;
    document.onmouseup = stop;
}
function move(event){ // 鼠标移动处理函数
    event = e(event);
    o.style.left = ox + event.mx - mx  + "px";
    o.style.top = oy + event.my - my + "px";
}
function stop(event){ // 松开鼠标按键时的处理函数
    event = e(event);
    ox = parseInt(o.offsetLeft);
    oy = parseInt(o.offsetTop);
    mx = event.mx ;
    my = event.my ;
    o = document.onmousemove = document.onmouseup = null;
}
</script>
```

建议 125：设计鼠标指针定位方案

当事件发生时，获取鼠标指针的位置是很重要的操作。不同浏览器分别在各自事件对象中定义了不同的属性，见表 6.1。这些属性都以像素值定义了鼠标指针的坐标，但它们参照的坐标系不同，从而使准确计算鼠标的位置（能够兼容不同浏览器）成为一件很麻烦的事。

表 6.1　鼠标定位事件属性

运算符	执行的运算	兼容性
clientX	以浏览器窗口左上顶角为原点，定位 x 轴坐标	所有浏览器，不兼容 Safari
clientY	以浏览器窗口左上顶角为原点，定位 y 轴坐标	所有浏览器，不兼容 Safari
offsetX	以当前事件的目标对象左上顶角为原点，定位 x 轴坐标	所有浏览器，不兼容 Firefox
offsetY	以当前事件的目标对象左上顶角为原点，定位 y 轴坐标	所有浏览器，不兼容 Firefox
pageX	以 document 对象（即文档窗口）左上顶角为原点，定位 x 轴坐标	所有浏览器，不兼容 IE
pageY	以 document 对象（即文档窗口）左上顶角为原点，定位 y 轴坐标	所有浏览器，不兼容 IE
screenX	以计算机屏幕左上顶角为原点，定位 x 轴坐标	所有浏览器
screenY	以计算机屏幕左上顶角为原点，定位 y 轴坐标	所有浏览器
layerX	以最近的绝对定位的父元素（如果没有，则为 document 对象）左上顶角为原点，定位 x 轴坐标	Firefox 和 Safari
layerY	以最近的绝对定位的父元素（如果没有，则为 document 对象）左上顶角为原点，定位 y 轴坐标	Firefox 和 Safari

首先，screenX 和 screenY 这两个属性获得了所有浏览器的支持，应该说是最优选用属性。它们的坐标系是计算机屏幕，也就是说，以计算机屏幕左上角为定位原点。这对于以浏览器窗口为活动空间的网页来说，没有任何价值，因为不同的屏幕分辨率，不同的浏览器窗口大小和位置都使在网页中定位鼠标成为一件很困难的事情。

如果以 document 对象为坐标系，则可以考虑选用 pageX 和 pageY 属性在浏览器窗口中进行定位。这对设计鼠标跟随来说是一个好主意，因为跟随元素一般都以绝对定位的方式在浏览器窗口中移动，所以只要在 mousemove 事件处理函数中把 pageX 和 pageY 属性值传递给绝对定位元素的 top 和 left 样式属性即可。

由于 clientX 和 clientY 属性是以 window 对象为坐标系，并且 IE 支持它们，因此可以选用它们。不过考虑 window 等对象可能出现的滚动条偏移量，还应加上相对于 window 对象的页面滚动的偏移量。因此可以这样来设计：

```
var posX = 0, posY = 0;
var event = event || window.event;
if(event.pageX || event.pageY){
    posX = event.pageX;
    posY = event.pageY;
}
else if(event.clientX || event.clientY){
    posX = event.clientX + document.documentElement.scrollLeft +
document.body.scrollLeft;
    posY = event.clientY + document.documentElement.scrollTop +
document.body.scrollTop;
}
```

在上面的代码中，先检测 pageX 和 pageY 属性是否存在，如果存在，则获取它们的值；如果不存在，则检测并获取 clientX 和 clientY 属性值，然后加上 document.documentElement 和 document.body 对象的 scrollLeft 和 scrollTop 属性值，这样就可以在不同浏览器中获得相同的坐标值。

然后定义一个封装函数，设计函数传入参数为对象引用指针、相对鼠标指针的偏移距离，以及事件对象。接着封装函数能够根据事件对象获取鼠标的坐标值，并设置该对象为绝对定位，绝对定位的值为鼠标指针当前的坐标值。具体封装代码如下：

```
var pos = function(o, x, y,event){
    var posX = 0, posY = 0;
    var e = event || window.event;
    if(e.pageX || e.pageY){
        posX = e.pageX;
        posY = e.pageY;
    }
    else if(e.clientX || e.clientY){
        posX = e.clientX + document.documentElement.scrollLeft +
document.body.scrollLeft;
        posY = e.clientY + document.documentElement.scrollTop +
```

```
document.body.scrollTop;
    }
    o.style.position = "absolute";
    o.style.top = (posY + y) + "px";
    o.style.left = (posX + x) + "px";
}
```

为 document 对象注册鼠标移动事件处理函数，并传入鼠标定位封装函数，传入的对象为 元素，设置其位置向鼠标指针右下方偏移（10,20）的距离。由于非 IE 浏览器是通过参数形式传递事件对象的，所以不要忘记在调用函数中还要传递事件对象。代码如下：

```
<div id="div1"> 鼠标跟随 </div>
<script language="javascript" type="text/javascript">
var div1 = document.getElementById("div1");
document.onmousemove = function(event){
    pos(div1,10,20,event);
}
</script>
```

建议 126：小心在元素内定位鼠标指针

要获取鼠标指针在元素内的坐标，目前还没有更稳妥的解决方案。使用 offsetX 和 offsetY 属性可以获取鼠标指针在元素内的坐标，但 Firefox 浏览器对此并不支持。不过可以选用 layerX 和 layerY 属性来兼容 Firefox 浏览器。代码如下：

```
var event = event || window.event;
if(event.offsetX || event.offsetY ){
    x = event.offsetX;
    y = event.offsetY;
}
else if(event.layerX || event.layerY ){
    x = event.layerX;
    y = event.layerY;
}
```

但是，layerX 和 layerY 属性以绝对定位的父元素为参照物，而不是以元素自身为参照物，因此，如果没有绝对定位的父元素，则以 document 对象为参照物。为此，我们可以通过脚本动态添加或手动添加的方式来设计在元素的外层包围一个绝对定位的父元素，从而解决浏览器兼容问题。考虑到元素之间的距离所造成的误差，可以适当减去一个或几个像素的偏移量。例如：

```
<input type="text" id ="text" />
<span style="position:absolute;">
    <div id="div1" style="width:200px;height:160px;border:solid 1px
red;"> 鼠标跟随 </div>
</span>
<script language="javascript" type="text/javascript">
```

```
var  t = document.getElementById("text");
var div1 = document.getElementById("div1");
div1.onmousemove = function(event){
    var event = event || window.event;
    if(event.offsetX || event.offsetY ){
        t.value = event.offsetX + " " + event.offsetY;
    }
    else if(event.layerX || event.layerY ){
        t.value = (event.layerX - 1) + " " + (event.layerY -1) ;
    }
}
```

这种做法能够解决在元素内部定位鼠标指针的问题，但在元素外面包裹一个绝对定位的元素会破坏整个页面的结构布局。在确保这种人为方式不会导致结构布局混乱的前提下，可以考虑选用这种方法，否则不建议使用这种方法。

由于浏览器的不兼容性，鼠标定位一直是一个难题。在 JavaScript 程序开发中，应该避免将鼠标定位作为事件触发的条件。例如，当鼠标经过某个元素时，不应该通过计算鼠标指针的坐标是否位于元素内部来触发事件，而应该通过元素的 mouseover 事件类型进行触发，或者通过焦点事件进行控制。

建议 127：妥善使用 DOMContentLoaded 事件

在传统事件模型中，load 是页面中最早被触发的事件。不过当使用 load 事件来初始化页面时可能会存在一个问题，那就是当页面中包含很大的文件时，load 事件需要等到所有图像全部载入完成之后才会被触发。这时可以考虑使用 DOMContentLoaded 事件，作为 DOM 标准事件，它是在 DOM 文档结构加载完毕的时候被触发的，要比 load 事件先被触发。目前，Firefox 和 Opera 新版本已经支持了 DOMContentLoaded 事件，而 IE 和 Safari 浏览器还不支持。

在标准 DOM 中可以这样设计：

```
<html>
<head>
<script language="javascript" type="text/javascript">
window.onload = f1;
if(document.addEventListener){
    document.addEventListener("DOMContentLoaded", f, false);
}
function f(){
    alert("我提前执行了");
}
function f1(){
    alert("页面初始化完毕");
}
</script>
</head>
<body>
```

```
<img src="Winter.jpg">
</body>
</html>
```

这样，在图片加载之前，会弹出“我提前执行了”的提示信息，而在图片加载完毕后才会弹出“页面初始化完毕”提示信息。这说明在页面 HTML 结构加载完毕之后触发 DOMContentLoaded 事件，也就是说，在文档标签加载完毕触发该事件，并调用函数 f()，然后在文档所有内容加载完毕（包括图片下载完毕）才触发 load 事件，并调用函数 f1()。

由于 IE 不支持 DOMContentLoaded 事件，为了实现兼容处理，我们需要运用一点小技巧，即在文档中写入一个新的 script 元素，不过该元素会延迟到文件最后加载。在使用 script 元素的 onreadystatechange 方法进行类似的 readyState 检查后及时调用载入事件，代码如下：

```
if(window.ActiveXObject){
    document.write("<script id=ie_onload defer src=javascript:void(0)>
<\/script>");
    document.getElementById("ie_onload").onreadystatechange=function(){
        if(this.readyState == "complete"){
            this.onreadystatechange = null;
            f();
        }
    }
}
```

在写入的 <script> 标签中包含了 defer 属性，defer 表示“延期”的意思，使用 defer 属性可以让脚本在整个页面加载完成之后再解析，而非边加载边解析。这对于只包含事件触发的脚本来说，可以提高整个页面的加载速度。与 src 属性联合使用，还可以使这些脚本在后台被下载，前台的内容则正常显示给用户。目前只有 IE 支持 defer 属性。在定义了 defer 属性后，<script> 标签中就不应该包含 document.write 命令了，因为 document.write 将产生直接输出效果，并且不包括任何立即执行脚本要使用的全局变量或函数。

由于 <script> 标签在文档结构加载完毕之后才加载，因此只要判断它的状态就可以确定当前文档结构是否已经加载完毕，并触发响应的事件。

针对 Safari 浏览器，我们可以使用 setInterval() 函数周期性地检查 document 对象的 readyState 属性，随时监控文档是否加载完毕，如果加载完成则调用回调函数，代码如下：

```
if (/WebKit/i.test(navigator.userAgent)){
    var _timer = setInterval(function(){
        if (/loaded|complete/.test(document.readyState)) {
            clearInterval(_timer);
            f();
        }
    }, 10);
}
```

把上面 3 段条件结构合并在一起即可实现兼容不同浏览器的 DOMContentLoaded 事件处理函数。

建议 128：推荐使用 beforeunload 事件

unload 表示卸载的意思，这个事件在从当前浏览器窗口内移动文档的位置时触发，也就是说，在通过超链接、前进或后退按钮等从一个页面跳转到其他页面，或者关闭浏览器窗口时触发。例如，下面函数的提示信息将在卸载页面时发生，即在离开页面或关闭窗口前执行。

```
window.onunload = f;
function f(){
    alert("888");
}
```

在 unload 事件中无法有效阻止默认行为，因为该事件结束后，页面将不复存在。由于在窗口关闭或离开页面之前只有很短的时间来执行事件处理函数，因此不建议使用该事件类型。使用该事件类型的最佳方式是取消该页面的对象引用。

beforeunload 事件与 unload 事件功能相近，不过它更人性化，如果 beforeunload 事件处理函数返回字符串信息，那么该字符串会显示在一个确认对话框中，询问用户是否离开当前页面。例如，运行下面的示例，当刷新或关闭页面时，会弹出提示信息。

```
window.onbeforeunload = function(e){
    return "你的数据还没有保存呢！";
}
```

beforeunload 事件处理函数的返回值可以为任意类型，IE 和 Safari 浏览器的 JavaScript 解释器能够调用 toString() 方法把它转换为字符串，并且将字符串信息显示在提示对话框中。而对于 Firefox 浏览器来说，则会把返回值视为空字符串显示。如果 beforeunload 事件处理函数没有返回值，则不会弹出任何提示对话框，此时 beforeunload 事件与 unload 事件响应效果相同。

虽然 beforeunload 不是标准事件类型，但是它获得绝大数主流浏览器的支持，目前仅有 Opera 浏览器不支持该事件类型。使用该事件类型可以在用户离开页面时提醒是否保存数据或完成相关操作。

建议 129：自定义事件

事件是一种称做观察者的设计模式，是一种创建松散耦合代码的技术。对象可以发布事件，以表示该对象声明周期中某个有趣的时刻到了。其他对象可以观察该对象，等待有趣的时刻到来并通过运行代码来响应。观察者模式由两类对象组成：主体和观察者。

- 主体负责发布事件，同时观察者通过订阅这些事件来观察主体。
- 主体并不知道观察者的任何事情，它可以独自存在并正常运作（即使观察者不在）。

自定义事件就是对已经存在的事件进行包装，也就是说，自定义事件在执行的时候还是需要依赖已有的键盘、鼠标、HTML 等事件来执行，或者由其他函数触发执行，这里的触发是指直接调用自定义事件中声明的某个接口方法，以便不断执行添加到自定义事件中的函数。

自定义事件内部有一个事件存储器，它根据添加的事件的类型不同来存储各个类的事件执行函数，当再次触发这类事件时，就轮询执行添加到该类型下的函数。自定义事件隐含的作用是创建一个管理事件的对象，让其他对象监听那些事件。基于自定义事件的原理，可以想象自定义事件很多时候用于实现订阅—发布—接收性质的功能。

❑ 基本模式：

```
function EventTarget() {
    this.handlers = {};
}
EventTarget.prototype = {
    constructor : EventTarget,
    addHandler : function(type, handler) {
        if( typeof this.handlers[type] == "undefined") {
            this.handlers[type] = [];
        }
        this.handlers[type].push(handler);
    },
    fire : function(event) {
        if(!event.target) {
            event.target = this;
        }
        if(this.handlers[event.type] instanceof Array) {
            var handlers = this.handlers[event.type];
            for(var i = 0, len = handlers.length; i < len; i++) {
                handlers[i](event);
            }
        }
    },
    removeHandler : function(type, handler) {
        if(this.handlers[type] instanceof Array) {
            var handlers = this.handlers[type];
            for(var i = 0, len = handlers.length; i < len; i++) {
                if(handlers[i] === handler) {
                    break;
                }
            }
            Handlers.splice(i,1);
        }
    }
};
```

❑ 使用 EventTarget 类型的自定义事件的方法如下：

```
function handleMessage(event) {
    alert("message received:" + event.message);
}
```

```
// 创建一个新对象
var target = new EventTarget();
// 添加一个事件处理程序
target.addHandler("message", handleMessage);
// 触发事件
target.fire({
    type : "message",
    message : "hello world!"
});
// 删除事件处理程序
target.removeHandler("message", handleMessage);
```

❑ 使用实例：

```
function Person(name, age) {
    eventTarget.call(this);
    this.name = name;
    this.age = age;
}
inheritPrototype(Person, EventTarget);
Person.prototype.say = function(message) {
    this.fire({
        type : "message",
        message : message
    });
};
function handleMessage(event) {
    alert(event.target.name + "says: " + event.message);
}
// 创建新 person
var person = new Person("Nicholas", 29);
// 添加一个事件处理程序
Person.addHandler("message", handleMessage);
// 在该对象上调用一个方法，它触发消息事件
person.say("Hi there");
```

建议 130：从 CSS 样式表中抽取元素尺寸

每个元素的显示属性都存储在 CSS 样式表中，如果能够从中读取元素的 width 和 height 属性，就可以精确地获得它的大小。在 JavaScript 中访问和设置元素的 CSS 属性，可以通过元素的 style 属性进行。style 是一个集合对象，它内部包含很多 CSS 脚本属性。例如，使用 style 属性设置元素的显示宽度，并读取该宽度值：

```
var div = document.getElementsByTagName("div")[0];
div.style.width = "100px";
var w = div.style.width;          // 返回字符串 "100px"
```

但是，在 JavaScript 中设置或读取 CSS 属性值时，都必须包含单位，并且传递或返回的值都是字符串，同时通过这种方式获得的信息往往是不准确的，因为 style 属性中并不包

含元素的样式属性的默认值。例如，在样式表或行内样式中未显式定义 div 元素的宽度，根据它的默认值（即 auto 值），实际宽度显示为 100%。此时，如果使用元素的 style 属性读取 width 值，则返回空字符串。

```
<div id="div" style="border:solid;"></div>
<script language="javascript" type="text/javascript">
var div = document.getElementsByTagName("div")[0];
alert(div.style.width);                                    // 返回空字符串
</script>
```

由于不同浏览器之间不兼容，获取元素最终样式的属性还需要针对不同的浏览器分别设计，开发者应该考虑 IE 与支持 DOM 标准的浏览器存在不同的处理方法。

首先，自定义一个扩展函数来兼容 IE 和 DOM 的实现方法。扩展函数的参数为当前元素（即 e）和它的属性名（即 n），函数返回值为该元素的样式的属性值。注意，这里的属性名是遵循驼峰命名法定义的 CSS 脚本属性名。代码如下：

```
// 获取指定元素的样式属性值
// 参数：e 表示具体的元素，n 表示要获取元素的脚本样式的属性名，如 "width"、"borderColor"
// 返回值：返回该元素 e 的样式属性 n 的值
function getStyle(e,n){
    if(e.style[n]){
        return e.style[n];
    }
    else if(e.currentStyle){
        return e.currentStyle[n];
    }
    else if(document.defaultView && document.defaultView.getComputedStyle){
        n = n.replace(/([A-Z])/g,"-$1");
        n = n.toLowerCase();
        var s = document.defaultView.getComputedStyle(e,null);
        if(s)
            return s.getPropertyValue(n);
    }
    else
        return null;
}
```

DOM 标准在读取 CSS 属性值时的规定比较特殊，它遵循 CSS 语法规则中的约定来命名属性名，即在复合属性名中使用连字符来连接多个单词，而不是遵循驼峰命名法，利用首字母大写的方式来区分不同的单词。例如，属性 borderColor 在传递给 DOM 时就需要转换为 border-color，否则就会错判。因此，传递的参数名还需要进行转换，不过利用正则表达式可以轻松实现。下面调用这个扩展函数来获取指定元素的实际宽度：

```
<div id="div"></div>
<script language="javascript" type="text/javascript">
var div = document.getElementsByTagName("div")[0];
var w = getStyle(div,"width");              // 调用扩展函数，返回字符串 "auto"
```

```
</script>
```

如果为 div 元素显式定义 200 像素的宽度：

```
<div id="div" style="width:200px;border-style:solid;"></div>
```

则调用扩展函数 getStryle() 后会返回字符串“200px”：

```
var w = getStyle(div,"width");              // 调用扩展函数，返回字符串 "200px"
```

虽然，我们知道 auto 值等于父元素的宽度，但是这只有通过人工计算才能够获取。例如，下面的示例中嵌套的结构就比较复杂，中间包含多层元素，并且宽度取值都是百分比，只需简单的口算过程就可以知道，最内层元素的宽度的实际值为 25 像素。

```
<div style="width:200px;">
    <div style="width:50%;">
        <div style="width:50%;">
            <div style="width:50%;">
                <div id="div" style="border-style:solid;"></div>
            </div>
        </div>
    </div>
</div>
```

设计一个简单的迭代计算，使用 getStyle() 扩展函数抽取每层元素的宽度值，然后把百分比转换为数值，之后相乘即可。例如：

```
var div1 = document.getElementsByTagName("div")[0];
var w1 = parseInt(getStyle(div1, "width"));
var div2 = document.getElementsByTagName("div")[1];
var w2 = parseInt(getStyle(div2, "width"))/100;
var div3 = document.getElementsByTagName("div")[2];
var w3 = parseInt(getStyle(div3, "width"))/100;
var div4 = document.getElementsByTagName("div")[3];
var w4 = parseInt(getStyle(div4, "width"))/100;
var w = w1*w2*w3*w4;        // 返回数值 25
```

上面的方法虽然很直接，但是比较简陋，缺乏灵活性。下面设计一个扩展函数 fromStyle()，该函数对 getStyle() 扩展函数的功能进行补充。设计 fromStyle () 函数的参数为要获取尺寸的元素，以及利用 getStyle() 函数所得到的值，然后返回这个元素的具体尺寸值（即为具体的数字）。代码如下：

```
// 把 fromStyle () 函数返回值转换为实际的值
// 参数：e 表示具体的元素，w 表示元素的样式属性值，通过 getStyle() 函数获取，p 表示当前元素百分比
   转换为小数的值，以便在上级元素中计算当前元素的尺寸
// 返回值：返回具体的数字值
function fromStyle(e, w, p){
    var p = arguments[2];
    if(!p) p = 1;
    if(/px/.test(w)&& parseInt(w)) return parseInt(parseInt(w)*p);
```

```
    else if(/\%/.test(w)&& parseInt(w)){          // 如果元素宽度值为百分比值
        var b = parseInt(w)/100;
        if((p!=1) && p)b*= p;
        e = e.parentNode;
        if(e.tagName == "BODY") throw new Error(" 整个文档结构都没有定义固定尺寸，没法计
            算了，请使用其他方法获取尺寸 .");
        w = getStyle(e, "width");
        return arguments.callee(e,w,b);
    }
    else if(/auto/.test(w)){
        var b = 1;
        if((p!=1) && p) b *= p;
        e = e.parentNode;
        if(e.tagName == "BODY") throw new Error(" 整个文档结构都没有定义固定尺寸，没法计
            算了，请使用其他方法获取尺寸 .");
        w = getStyle(e, "width");
        return arguments.callee(e,w,b);
    }
    else
        throw new Error(" 元素或其父元素的尺寸定义了特殊的单位 .");
}
```

最后，针对上面的嵌套结构，调用该函数就可以直接计算出元素的实际值：

```
var div = document.getElementById("div");
var w = getStyle(div, "width");
w = fromStyle(div, w);                    // 返回数值 25
```

要获取元素的高度值，在 getStyle() 函数中修改第二个参数值为字符串“height”即可。

建议 131：慎重使用 offsetWidth 和 offsetHeight

可以使用 offsetWidth 和 offsetHeight 属性来获取元素的尺寸，其中 offsetWidth 表示元素在页面中所占据的总宽度，offsetHeight 表示元素在页面中所占据的总高度。例如：

```
<div style="height:200px;width:200px;">
    <div style="height:50%;width:50%;">
        <div style="height:50%;width:50%;">
            <div style="height:50%;width:50%;">
                <div id="div" style="height:50%;width:50%;border-style:solid;"></div>
            </div>
        </div>
    </div>
</div>
<script language="javascript" type="text/javascript">
var div = document.getElementById("div");
var w = div.offsetWidth;                          // 返回元素的总宽度
var h = div.offsetHeight;                         // 返回元素的总高度
</script>
```

上面的示例在 IE 的“怪异”模式下和支持 DOM 模型的浏览器中解析结果差异很大，其中在 IE“怪异”模式下解析返回宽度为 13 像素，高度为 26 像素，而在支持 DOM 模型的浏览器中返回高度和宽度都为 18 像素（但是 FF 返回 19 像素，因为小数取舍方法不同）。

IE“怪异”模式是一种非标准的解析方法，与标准模式相对应，主要是 IE 为了兼容大量传统布局的网页而采用的。“怪异”模式在 IE 6.0 以下版本中存在，但在 IE 6.0 及其以上版本中，如果将页面明确设置为“怪异”模式显示，或者 HTML 文档的 DOCTYPE（文档类型）没有明确定义，也会按“怪异”模式进行解析。

根据示例中内行样式定义的值，可以算出最内层元素的宽和高都为 12.5 像素，实际取值为 12 像素。但对于 IE“怪异”解析模式来说，样式属性 width 和 height 的值就是元素的总宽度和总高度。由于 IE 是根据四舍五入法处理小数部分的，因此该元素的总高度和总宽度都是 13 像素。同时，由于 IE 模型定义每个元素都有一个默认行高，即使元素内不包含任何文本，因此实际高度就显示为 26 像素。

而对于支持 DOM 模型的浏览器来说，它们认为元素样式属性中的宽度和高度仅是元素内部包含的内容区域的尺寸，而元素的总高度和总宽度应该加上补白和边框，由于元素默认边框值为 3 像素，因此最后计算的总高度和总宽度都是 18 像素。至于，Firefox 返回值为 19 像素，是因为它在处理小数部分时，并没有完全舍去，而是根据条件和环境的不同增加了 1 个像素值。

也许 offsetWidth 和 offsetHeight 属性是获取元素尺寸的最好的方法，但在实践中会发现：当为元素定义隐藏属性，即设置样式属性 display 的值为 none 时，元素的尺寸总为 0，代码如下：

```
<div id="div" style="height:200px;width:200px;
border-style:solid;display:none;"></div>
<script language="javascript" type="text/javascript">
var div = document.getElementById("div");
var w = div.offsetWidth;          // 返回 0
var h = div.offsetHeight;         // 返回 0
</script>
```

这种情况还会发生在父级元素的 display 样式属性为 none 时，即使当前元素没有设置隐藏显示，根据继承关系也会将其隐藏显示，此时 offsetWidth 和 offsetHeight 属性值都是 0。总之，对于隐藏元素来说，不管它的实际高度和宽度是多少，最终读取的 offsetWidth 和 offsetHeight 属性值都是 0。

要解决这个问题，还需要自定义函数专门弥补 offsetWidth 和 offsetHeight 属性的缺陷。具体设计思路：先判断元素的样式属性 display 的值是否为 none，如果不是，则直接调用 offsetWidth 和 offsetHeight 属性读取元素的宽和高即可。如果元素的样式属性 display 的值为 none，则可以暂时显示元素，然后读取它的尺寸，读取完之后再把它恢复为隐藏样式。为此，不妨先设计两个小的功能函数，通过它们可以分别重设和恢复元素的样式属性值。代码

如下：

```
// 重设元素的样式属性值
// 参数：e 表示重设样式的元素，o 表示要设置的值，它是一个对象，可以包含多个名 - 值对
// 返回值：重设样式的原属性值，以对象形式返回
function setCSS(e, o){
    var a = {};
    for(var i in o){
        a[i] = e.style[i];
        e.style[i] = o[i];
    }
    return a;
}
// 恢复元素的样式属性值
// 参数：e 表示重设样式的元素，o 表示要恢复的值，它是一个对象，可以包含多个名 - 值对
// 返回值：无
function resetCSS(e,o){
    for(var i in o){
        e.style[i] = o[i];
    }
}
```

有了这两个小的功能函数后，再自定义函数 getW() 和 getH()，不管元素是否被隐藏显示，这两个函数能够获取元素的宽度和高度。具体实现代码如下：

```
// 获取元素的存在宽度
// 参数：e 表示元素
// 返回值：存在宽度
function getW(e){
    if(getStyle(e,"display") != "none") return e.offsetWidth ||
fromStyle(getStyle(e,"width"));
    var r = setCSS( e, {
        display:"",
        position:"absolute",
        visibility:"hidden"
    });
    var w = e.offsetWidth || fromStyle(getStyle(e,"width"));
    resetCSS(e,r);
    return w;       // 返回存在宽度
}
// 获取元素的存在高度
// 参数：e 表示元素
// 返回值：存在高度
function getH(e){
    if(getStyle(e,"display") != "none") return e.offsetHeight ||
fromStyle(getStyle(e,"height"));
    var r = setCSS( e, {
        display:"",
        position:"absolute",
        visibility:"hidden"
    });
    var h = e.offsetHeight || fromStyle(getStyle(e,"height"));
```

```
    resetCSS(e,r);
    return h;
}
```

最后，调用 getW() 和 getH() 扩展函数来测试它的性能：

```
<div id="div" style="height:200px;width:200px;
border-style:solid;display:none;"></div>
<script language="javascript" type="text/javascript">
var div = document.getElementById("div");
var w = div.offsetWidth;  // 返回 0
var h = div.offsetHeight; // 返回 0
var w1 = getW(div);       // 返回 200
var h1 = getH(div);       // 返回 200
</script>
```

建议 132：正确计算区域大小

不同浏览器对 offsetWidth 和 offsetHeight 属性的解析标准是不同的，这种不同会在动画的精确控制中产生一定影响。更重要的是，元素显示环境的复杂性导致了元素在不同场合下所呈现的效果不同。在某些情况下，需要精确计算元素的尺寸，这时候可以选用一些 HTML 元素特有的属性，见表 6.2。这些属性虽然还不是 DOM 标准的一部分，但是由于它们得到了所有浏览器的支持，因此在 JavaScript 开发中还是被普遍应用。

表 6.2 与元素尺寸相关的属性

属　性	说　明
clientWidth	获取元素可视部分的宽度，即 CSS 的 width 和 padding 属性值之和，元素边框和滚动条不包括在内，也不包含任何可能的滚动区域
clientHeight	获取元素可视部分的高度，即 CSS 的 height 和 padding 属性值之和，元素边框和滚动条不包括在内，也不包含任何可能的滚动区域
offsetWidth	元素在页面中占据的宽度总和，包括 width、padding、border，以及滚动条的宽度
offsetHeight	元素在页面中占据的高度总和，包括 height、padding、border，以及滚动条的高度
scrollWidth	当元素设置了 overflow:visible 样式属性时，元素的总宽度。也有人把它解释为元素的滚动宽度。在默认状态下，如果该属性值大于 clientWidth 属性值，则元素会显示滚动条，以便能够翻阅被隐藏的区域
scrollHeight	当元素设置了 overflow:visible 样式属性时，元素的总高度。也有人把它解释为元素的滚动高度。在默认状态下，如果该属性值大于 clientHeight 属性值，则元素会显示滚动条，以便能够翻阅被隐藏的区域

为了更直观地比较这些属性的异同，现在设计一个简单的盒模型，盒模型的 height 值为 200 像素，width 值为 200 像素，边框显示为 50 像素，补白区域定义为 50 像素。盒模型内部包含信息框，其宽度为 400 像素，高度也为 400 像素，换句话说就是盒模型的内容区域为（400px，400px）。设置结构和样式的代码如下：

```
<div id="div" style="height:200px;width:200px;border:solid 50px
red;overflow:auto;padding:50px;">
    <div id="info" style="height:400px;width:400px;
border:solid 1px blue;"></div>
</div>
```

然后，利用 JavaScript 脚本在信息框中插入一些行列号，以方便观察。

```
var info = document.getElementById("info");
var m = 0, n = 1, s = "";
while(m ++ < 19){
    s += m + "  ";
}
s += "<br />";
while(n ++ < 21){
    s += n + "<br />";
}
info.innerHTML = s;
```

盒模型呈现效果如图 6.7 所示。

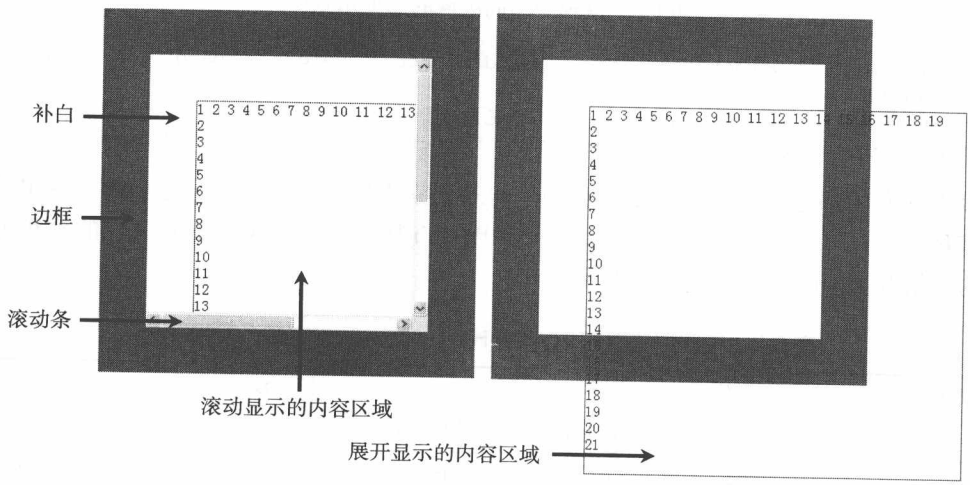


图 6.7　盒模型及其相关构成区域

现在分别调用 offsetHeight、scrollHeight、clientHeight 属性，以及自定义函数 getH()，可以看到获取了不同区域的高度（如图 6.8 所示）。

```
var div = document.getElementById("div");
var ho = div.offsetHeight;                  // 返回 400
var hs = div.scrollHeight;                  // 返回 502
var hc = div.clientHeight;                  // 返回 283
var hg = getH(div);                         // 返回 400
```

通过对图 6.8 的比较，可以很直观地看出 offsetHeight、scrollHeight、clientHeight 这 3 个属性与自定义函数 getH() 的值，具体说明如下：

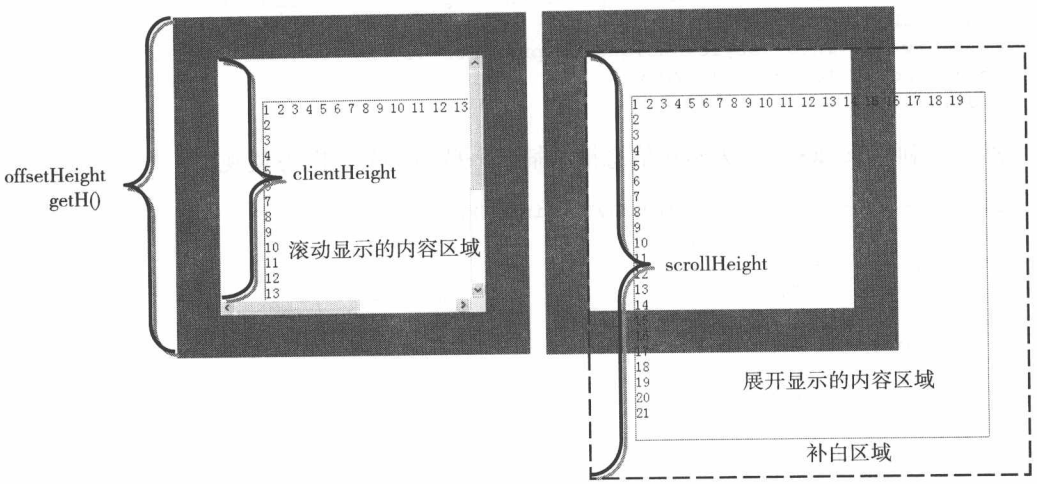


图 6.8　盒模型不同区域的高度示意图

```
    offsetHeight = border-top-width + padding-top + height + padding-bottom + borde
r-bottom-width
    scrollHeight = padding-top + 包含内容的完全高度 + padding-bottom
    clientHeight = padding-top + height +  border-bottom-width - 滚动条的宽度
```

以上的介绍都是围绕元素高度进行的，针对元素宽度的计算方式也是如此，这里就不再重复解释了。注意，针对 scrollHeight 和 scrollWidth 属性，不同浏览器对它们的解析方式是不同的。结合上面的示例，具体说明见表 6.3。

表 6.3　不同浏览器解析 scrollHeight 和 scrollWidth 属性比较

浏览器	返回值	计算公式
IE	502	padding-top + 包含内容的完全高度 + padding-bottom
Firefox	452	padding-top + 包含内容的完全高度
Opera	419	包含内容的完全高度 + 底部滚动条的宽度
Safari	452	padding-top + 包含内容的完全高度

如果设置盒模型的 overflow 属性为 visible，就会发现 clientHeight 的值为 300，即：

```
clientHeight = padding-top + height +  border-bottom-width
```

也就是说，如果隐藏滚动条显示，则 clientHeight 属性值不用减去滚动条的宽度，即滚动条的区域被转化为可视内容区域。同时，不同浏览器对于 clientHeight 和 clientWidth 的解析也不同，再结合上面示例，具体说明见表 6.4。

表 6.4　不同浏览器解析 clientHeight 和 clientWidth 属性比较

浏览器	返回值	计算公式
IE 7	502	padding-top + 包含内容的完全高度 + padding-bottom
Firefox 3	400	border-top-width + padding-top + height + padding-bottom + border-bottom-width
Opera 9	502	padding-top + 包含内容的完全高度 + padding-bottom
Safari 4	502	padding-top + 包含内容的完全高度 + padding-bottom

建议 133：谨慎计算滚动区域大小

scrollLeft 和 scrollTop 属性比较特殊，见表 6.5。利用它们可以定义当拖动滚动条时移出可视区域外的宽度和高度。实际上，可以利用这两个属性设定滚动条的位置，也可以使用它们获取当前滚动区域内容。

表 6.5　scrollLeft 和 scrollTop 属性说明

属性	说明
scrollLeft	元素左侧已经滚动的距离（像素值）。更通俗地说，就是设置或获取位于元素左边界与元素中当前可见内容的最左端之间的距离
scrollTop	元素顶部已经滚动的距离（像素值）。更通俗地说，就是设置或获取位于元素顶部边界与元素中当前可见内容的最顶端之间的距离

下面这个示例演示了如何设置和更直观地获取滚出区域的尺寸。

```
<textarea id="text" rows="5" cols="25" style="float:right;"></textarea>
<div id="div" style="height:200px;width:200px;border:solid 50px red;
padding:50px;overflow:auto;">
    <div id="info" style="height:400px;width:400px;
border:solid 1px blue;"></div>
</div>
var div = document.getElementById("div");
div.scrollLeft = 200;
div.scrollTop = 200;
var text = document.getElementById("text");
div.onscroll = function(){
    text.value ="scrollLeft = " + div.scrollLeft + "\n" +
                "scrollTop = " + div.scrollTop + "\n" +
                "scrollWidth = " + div.scrollWidth + "\n" +
                "scrollHeight = " + div.scrollHeight ;
}
```

上面代码运行后呈现的效果如图 6.9 所示。

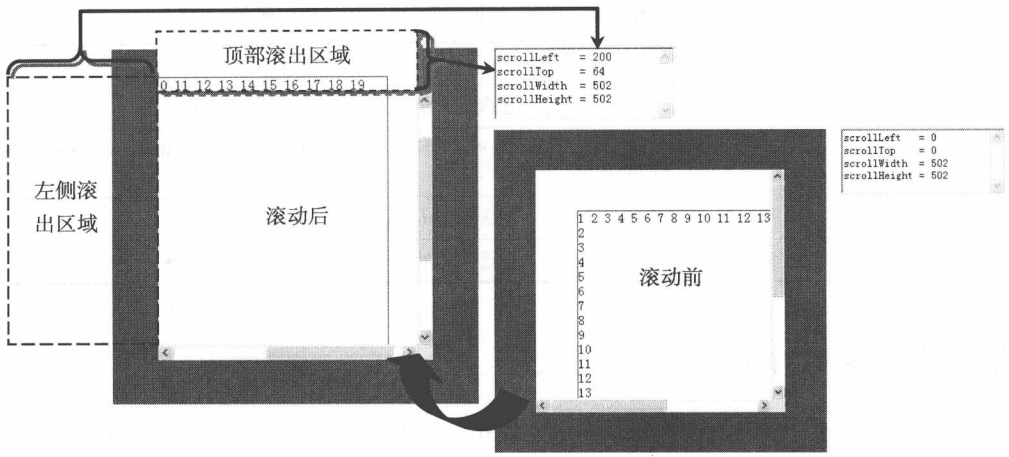


图 6.9 scrollLeft 和 scrollTop 属性指示区域示意图

建议 134：避免计算窗口大小

对于浏览器窗口来说，通过获取 <html> 标签的 clientWidth 和 clientHeight 属性就可以得到浏览器窗口的可视宽度和高度，由于 <html> 标签在脚本中表示为 document.documentElement，因此可以这样设计：

```
var w = document.documentElement.clientWidth;
var h = document.documentElement.clientHeight;
```

不过在 IE 6.0 以下版本的浏览器中（即在 IE 浏览器的“怪异”解析模式下），body 是最顶层的可视元素，而 html 元素保持隐藏，因此只有通过 <body> 标签的 clientWidth 和 clientHeight 属性才可以得到浏览器窗口的可视宽度和高度，而 <body> 标签在脚本中表示为 document.body，要兼容 IE“怪异”解析模式可以这样设计：

```
var w = document.body.clientWidth;
var h = document.body.clientHeight;
```

然而，支持 DOM 解析模式的浏览器都把 body 视为一个普通的块级元素，而 <html> 标签包含整个浏览器窗口。因此，考虑浏览器的兼容性，可以这样设计，如果浏览器支持 DOM 标准，那么使用 documentElement 对象读取 body 元素，若该对象不存在，则使用 body 对象读取 body 元素：

```
var w = document.documentElement.clientWidth || document.body.clientWidth;
var h = document.documentElement.clientHeight || document.body.clientHeight;
```

如果窗口包含内容超出了窗口可视区域，那么应该使用 scrollWidth 和 scrollHeight 属性

来获取窗口的实际宽度和高度。不过对于 document.documentElement 和 document.body 来说，不同浏览器对于它们的支持略有差异。例如：

```
<body style="border:solid 2px blue;margin:0;padding:0">
    <div style="width:2000px;height:1000px;border:solid 1px red;">
</div>
</body>
<script language="javascript" type="text/javascript">
var wb = document.body.scrollWidth;
var hb = document.body.scrollHeight;
var wh = document.documentElement.scrollWidth;
var hh = document.documentElement.scrollHeight;
</script>
```

不同浏览器的 scrollHeight 和 scrollWidth 返回值比较见表 6.6。

表 6.6　不同浏览器的 scrollHeight 和 scrollWidth 返回值

浏览器	body.scrollWidth	body.scrollHeight	documentElement.scrollWidth	documentElement.scrollHeight
IE	2002	1002	2004	1006
Firefox	1007	1006	2004	1006
Opera	2004	1006	2004	1006
Safari	2004	1006	2004	1006

通过上面的返回值比较可以看出，不同浏览器使用 documentElement 对象获取浏览器窗口的实际尺寸是一致的，但使用 Body 对象获取对应尺寸就会存在很大差异，特别是 Firefox 浏览器，它把 scrollWidth 与 clientWidth 属性值视为相等，显然这是错误的。

建议 135：正确获取绝对位置

任何元素都拥有 offsetLeft 和 offsetTop 属性，它们用于描述元素的偏移位置。不过不同浏览器定义元素的偏移参照对象不同，例如，IE 会以父元素为参照对象进行偏移，而支持 DOM 标准的浏览器会以最近非静态定位元素为参照对象进行偏移。

下面的示例是一个三层嵌套的结构，其中最外层 div 元素被定义为相对定位显示，在 JavaScript 脚本中使用“alert(box.offsetLeft);”语句获取最内层 div 元素的偏移位置，这样 IE 返回值为 50 像素，而其他支持 DOM 标准的浏览器会返回 101 像素。注意，Opera 返回值为 121 像素，因为它以 ID 为 wrap 元素的边框外壁为起点进行计算，而其他支持 DOM 标准的浏览器以 ID 为 wrap 元素的边框内壁为起点进行计算，代码如下：

```
<style type="text/css">
div {
    width:200px; height:100px; border:solid 1px red; padding:50px;
```

```
}
#wrap {
    position:relative;
    border-width:20px;
}
</style>
<div id="wrap">
    <div id="sub">
        <div id="box"></div>
    </div>
</div>
```

获取元素的位置的呈现效果如图 6.10 所示。

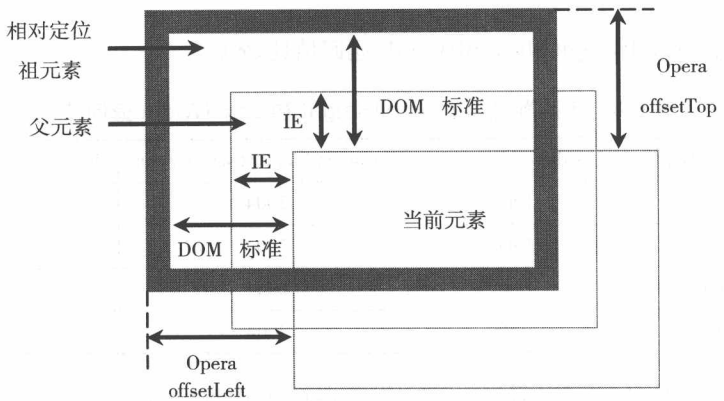


图 6.10 获取元素的位置示意图

所有浏览器都支持 offsetParent 属性，该属性总能够指向定位参考的元素，并得到所有浏览器的认可，因此针对上面的嵌套结构，有如下几种情况。

- 对于 IE 来说，当前定位元素（即 ID 为 box 的 div 元素）的 offsetParent 属性将指向 ID 为 sub 的 div 元素。对于 sub 元素来说，它的 offsetParent 属性将指向 ID 为 wrap 的 div 元素。
- 对于支持 DOM 的浏览器来说，当前定位元素的 offsetParent 属性将指向 ID 为 wrap 的 div 元素。

可以根据以上情况设计一个能够兼容不同浏览器的等式：

- IE：(#box).offsetLeft + (#sub).offsetLeft =(#box).offsetLeft + (#box).offsetParent.offsetLeft
- DOM：(#box).offsetLeft

对于任何浏览器来说，offsetParent 属性总能够自动识别当前元素偏移的参照对象，不用担心 offsetParent 在不同浏览器中具体指代什么元素。这样就能够通过迭代来计算当前元素距离窗口左上顶角的坐标值，具体演示如图 6.11 所示。

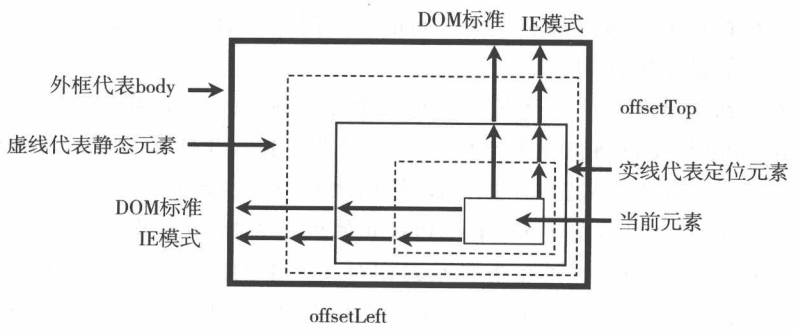


图 6.11　能够兼容不同浏览器的元素偏移位置计算演示图

虽然不同浏览器的 offsetParent 属性指代的元素不同，但是通过迭代计算，当前元素距离浏览器窗口的坐标距离都是相同的。因此，根据这个规律，可以设计一个扩展函数：

```
// 获取指定元素距离窗口左上角的偏移坐标
// 参数：e 表示获取位置的元素
// 返回值：返回对象直接量，其中属性 x 表示 x 轴偏移距离，属性 y 表示 y 轴偏移距离
function getW(e){
    var x = y = 0;
    while(e.offsetParent){
        x += e.offsetLeft;
        y += e.offsetTop;
        e = e.offsetParent;
    }
    return {
        "x" : x,
        "y" : y
    };
}
```

由于 body 和 html 元素没有 offsetParent 属性，因此，当迭代到 body 元素时，会自动停止并计算出当前元素距离窗口左上角的坐标距离。

调用该扩展函数应注意：不要为包含元素定义边框，因为不同浏览器对边框的处理方式不同。例如，IE 会忽略所有包含元素的边框，因为所有元素都是参照对象，并且以参照对象的边框内壁作为边线进行计算。Firefox 和 Safari 浏览器会把静态元素的边框作为实际距离进行计算，因为对于它们来说，静态元素不作为参照对象。而对于 Opera 浏览器来说，由于它以非静态元素边框的外壁作为边线进行计算，所以该浏览器所获取的值又有所不同。如果不为所有包含元素定义边框，那么可以避免不同浏览器解析的分歧，最终实现返回相同的距离。

建议 136：正确获取相对位置

在复杂的嵌套结构中，仅仅获取元素相对于浏览器窗口的位置并没有多大利用价值，因

为定位元素是根据最近的上级非静态元素进行的。同时对于静态元素来说，它是根据父元素的位置来决定自己的显示位置的。

要获取相对父级元素的位置，不能简单地读取 CSS 样式的 left 和 top 属性，因为这两个属性值是相对最近的上级非静态元素来说的。可以调用建议 135 中定义的 getW() 扩展函数分别获取当前元素和父元素距离窗口的距离，然后求两个值的差即可。

为了提高执行效率，可以先判断 offsetParent 属性是否指向父级元素，如果是，则可以直接使用 offsetLeft 和 offsetTop 属性获取元素相对父元素的距离；否则调用 getW() 扩展函数分别获得当前元素和父元素距离窗口的坐标，然后对这两个值求差。详细代码如下：

```
// 获取指定元素距离父元素左上角的偏移坐标
// 参数：e 表示获取位置的元素
// 返回值：返回对象直接量，其中属性 x 表示 x 轴偏移距离，属性 y 表示 y 轴偏移距离
function getP(e){
    if(e.parentNode == e.offsetParent){
        var x = e.offsetLeft;
        var y = e.offsetTop ;
    }
    else{
        var o = getW(e);
        var p = getW(e.parentNode);
        var x = o.x - p.x;
        var y = o.y - p.y;
    }
    return {
        "x" : x,
        "y" : y
    };
}
```

下面就可以调用该扩展函数获取指定元素相对父元素的偏移坐标了，代码如下：

```
var box = document.getElementById("box");
var o = getP(box);
alert(o.x);
alert(o.y);
```

获取元素相对包含块的位置与获取元素相对父级元素的位置不同，所谓包含块，就是定位元素参照的对象，即定位元素所根据的元素。包含块一般为距离当前定位元素最近的上级非静态元素。获取元素相对包含块的位置可以直接读取 CSS 样式中 left 和 top 属性值，这两个值记录了定位元素的坐标值。

下面定义一个扩展函数 getB()，它调用了 getStyle() 扩展函数，该函数能够获取元素的 CSS 样式属性值（参考建议 135 的代码）。对于默认状态的定位元素或静态元素，它们的 left 和 top 属性值一般为 auto，因此，获取 left 和 top 属性值后，可以尝试使用 parseInt() 方法把它们转换为数值。如果转换失败，则说明其值为 auto，设置为 0，否则返回转换的数值。代码如下：

```
// 获取指定元素距离包含块元素左上角的偏移坐标
// 参数：e 表示获取位置的元素
// 返回值：返回对象直接量，其中属性 x 表示 x 轴偏移距离，属性 y 表示 y 轴偏移距离
function getB(e){
    return {
        "x" : (parseInt(getStyle(e, "left")) || 0) ,
        "y" : (parseInt(getStyle(e, "top")) || 0)
    };
}
```

第 7 章
数据交互和存储

Ajax 是高性能 JavaScript 的基石，是提升网站可用性的最大改进之一。利用 Ajax 可以在网站大量使用异步请求，解决需要加载太多资源的问题。Ajax 可以通过延迟下载大量资源使页面加载更快，还可以通过在客户端和服务器之间异步传送数据避免页面集体加载。Ajax 还用于在一次 HTTP 请求中获取整个页面的资源，通过选择正确的传输技术和最有效的数据格式，可以显著改善用户与网站之间的互动。

作为数据格式，纯文本和 HTML 是被高度限制的，但它们可节省客户端的 CPU 周期。XML 被广泛应用和普遍支持，但它非常冗长且解析缓慢。JSON 是轻量级的，解析迅速，交互性与 XML 相当。字符分隔的自定义格式是非常轻量级的，在大量数据集解析时速度最快，但需要编写额外的程序在服务器端构造格式，并且需要在客户端解析。

XHR（XMLHttpRequest）将所有传入数据视为一个字符串，这种用法方便控制，让使用更具灵活性，当然它也有缺点，可能会降低解析速度。另外，XHR 虽然允许跨域请求和本地运行，但是这个接口不够安全，且不能读取信息头或响应报文代码。大部分 XHR 可减少请求的数量，可在一次响应中处理不同的文件类型，尽管它不能缓存收到的响应报文。XHR 也可用 POST 方法发送大量数据。

建议 137：使用隐藏框架实现异步通信

隐藏框架只是异步交互的一个载体，它仅负责信息的传输，而交互的核心是一种信息处理机制，这种处理机制就是回调函数。所谓回调函数，就是客户端页面中的一个普通函数，不过该函数在服务器端被调用，并负责处理服务器端响应的信息。

下面示例初步展现了异步信息交互中请求和响应的完整过程，其中回调函数的处理是整个流程的焦点。

第 1 步，构建一个框架集（回调处理 .htm），如下：

```
<html>
<head>
<title> 使用隐藏框架与服务器进行异步通信 </title>
</head>
<frameset rows="*,0">
    <frame src=" 回调处理 _main.htm" name="main" />
    <frame src=" 回调处理 _black.htm" name="serve" />
</frameset>
<noframes> 你的浏览器不支持框架集，请升级浏览器版本！ </noframes>
</html>
```

这个框架集由上下两个框架组成，框架 2（下框架）高度为 0。建议尽量不要设置高为 0 像素，因为在一些老版本的浏览器中依然会显示高度为 0 的框架。这两个框架的分工如下：

- 框架 1（main），负责与用户进行信息交互。
- 框架 2（serve），负责与服务器进行信息交互。

由于老版本浏览器可能不支持框架技术，所以应使用 <noframes> 来兼容它们，使设计显得更友好。

第 2 步，在默认状态下，框架集中的框架 2 加载一个空白页面（回调处理 _black.htm），框架 1 加载与客户进行交互的页面（回调处理 _main.htm）。

框架 1 中主要包含两个函数：一个是响应用户操作的回调函数，另一个是向服务器发送请求的事件处理函数。具体代码如下：

```
<html>
<head>
<title> 与客户交互页面 </title>
<script language="javascript" type="text/javascript">
function request(){                                          // 向服务器发送请求的异步请求函数
    var user = document.getElementById("user");             // 获取输入的用户名
    var pass = document.getElementById("pass");             // 获取输入密码
    var s = "user=" + user.value + "&pass=" + pass.value;       // 构造查询字符串
    parent.frames[1].location.href = " 回调处理 _serve.htm?" + s; /* 为框架集中的框架 2
加载服务器端请求文件，并附加查询字符串，传送客户端信息，以实现异步信息的双向交互 */
}
function callback(b, n){ // 异步交互的回调函数
    if(b){ // 如果参数 b 为真，说明输入信息正确
        var e = document.getElementsByTagName("body")[0]; /* 获取框架 1 中 body 元素的
引用指针，以实现向其中插入信息 */
        e.innerHTML = "<h1>" + n + "</h1><p> 您好，欢迎登录站点 </p>";/* 在交互页面中插入
新的交互信息 */
    }
    else{// 如果参数 b 为假，说明输入信息不正确
        alert(" 你输入的用户名或密码有误，请重新输入 ");          // 提示重新输入信息
        var user = parent.frames[0].document.getElementById("user");   /* 获取框架 1
中的用户名文本框 */
        var pass = parent.frames[0].document.getElementById("pass");   /* 获取框架 1
中的密码文本框 */
```

```
            user.value = "";            // 清空用户名文本框中的值
            pass.value = "";            // 清空密码文本框中的值
        }
    }
    window.onload = function(){// 页面初始化处理函数
        var b = document.getElementById("submit");  // 获取【提交】按钮
        b.onclick = request;    // 绑定鼠标单击事件处理函数
    }
    </script>
    </head>
    <body>
    <h1> 用户登录 </h1>
    用户名：<input name="" id="user" type="text"><br />
    密  码：<input name="" id="pass"  type="password"><br />
    <input name="submit" type="button" id="submit" value=" 提交 " />
    </body>
    </html>
```

由于回调函数是在服务器端文件中被调用的，因此对象作用域的范围就发生了变化，此时应该指明它的框架集和框架名或序号，否则在页面操作中会找不到指定的元素。

第 3 步，在服务器端的文件中设计响应处理函数，该函数将分解 HTTP 传递过来的 URL 信息，获取查询字符串，并且根据查询字符串中用户名和密码来判断当前输入的信息是否正确，进而决定具体响应的信息。

```
    <html>
    <head>
    <title> 服务器端响应和处理页面 </title>
    <script language="javascript" type="text/javascript">
    window.onload = function(){       // 服务器响应处理函数，当该页面被请求加载时触发
        var query = location.search.substring(1); // 获取 HTTP 请求的 URL 中所包含的查询字符串
        var a = query.split("&");     // 将查询字符串分离成数组
        var o ={};     // 临时对象直接量
        for(var i = 0; i < a.length; i ++ ){          // 遍历查询字符串数组
            var pos = a[i].indexOf("=");      // 找到等号的下标位置
            if(pos == - 1) continue;          // 如果没有等号，则忽略
            var name = a[i].substring(0, pos);        // 获取等号前面的字符串
            var value = a[i].substring(pos + 1);      // 获取等号后面的字符串
            o[name] = unescape(value);        // 把名 - 值对传递给对象
        }
        var n, b;
        ((o["user"]) && o["user"] == "admin") ? (n = o["user"]) : (n = null );        /* 如果用户
名存在，且等于 "admin"，则记录该信息，否则设置为 null*/
        ((o["pass"]) && o["pass"] == "123456") ? (b = true ) : (b = false ) ;   /* 如果密码
存在，且等于 "123456"，则设置变量 b 为 true，否则为 false*/
        parent.frames[0].callback(b, n); /* 调用客户端框架集中框架 1 中的回调函数，并把处理的信
息传递给它 */
    }
    </script>
    </head>
    <body>
    <h1> 服务器端响应和处理页面 </h1>
```

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

在实际开发中，服务器端文件一般为动态服务器类型的文件，并且借助服务器端脚本来获取用户的信息，然后决定响应的内容，一般还会像查询数据库一样返回查询内容等。本示例以简化的形式演示这个异步通信的过程，因此没有采用服务器技术。预览框架集，在客户交互页面中输入用户的登录信息，在向服务器提交请求之后，服务器首先接收从客户端传递过来的信息并进行处理，然后调用客户端的回调函数把处理后的信息响应回去。

建议 138：使用 iframe 实现异步通信

使用框架集实施异步交互存在着很多弊端：

- 一方面，页面被深深地打上框架的烙印，不利于结构优化，浏览器的支持也受到很多限制，需要更多的文件配合使用。
- 另一方面，框架集缺乏灵活性，要完全使用脚本控制异步请求与交互就非常不方便。

iframe 与 frameset（框架集）都可以实现动态加载客户端或服务器端任何类型的网页文件，也就是说，它们的功能相同，但表现形式各异。iframe 被定义为文档结构元素，与页面中其他普通元素无异，完全与框架集无关，但 frameset 被定义为窗口元素，与 html、head 和 body 等基本文档结构元素平行使用，可以说 iframe 与 frameset 分别属于不同级别的元素。因此，浮动框架可以插入到页面中的任意位置，与页面中其他元素能够很好地融合。同时开发人员还可以在 JavaScript 脚本中动态创建 iframe 元素并进行控制，这就给异步交互开发带来新的活力。下面仍然以上面的示例为基础进行扩展。

第 1 步，在客户端交互页面中新建函数 hideIframe()，使用该函数动态创建浮动框架，借助这个浮动框架实现与服务器进行异步通信，具体代码如下：

```
// 创建浮动框架
// 参数：url 表示要请求的服务器端文件路径
// 返回值：无
function hideIframe(url){
    var hideFrame = null;
    hideFrame = document.createElement("iframe");        // 创建 iframe 元素
    hideFrame.name = "hideFrame";
    hideFrame.id = "hideFrame";
    hideFrame.style.height = "0px";
    hideFrame.style.width = "0px";
    hideFrame.style.position = "absolute";
    hideFrame.style.visibility = "hidden";
    document.body.appendChild(hideFrame);
    setTimeout(function() {
        frames["hideFrame"].location.href = url;
    }, 10)
}
```

当使用 DOM 创建 iframe 元素时，应注意设置同名的 name 和 id 属性，因为不同类型浏览器在引用框架时会分别使用 name 或 id 属性值。在创建好 iframe 元素后，大部分浏览器（如 Firefox 和 Opera）会需要一点时间（约为几毫秒）来识别新框架并将其添加到帧集合中，因此，当加载地址准备向服务器进行请求时，应该使用 setTimeout() 函数将发送请求的操作延迟 10 ms。当执行请求时，浏览器能够识别这些新的框架，避免发生错误。

如果页面中多处需要调用请求函数，则建议定义一个全局变量，专门用来存储浮动框架对象，这样就可以避免每次请求时都创建新的 iframe 对象。

第 2 步，修改客户端交互页面中 request() 函数的请求内容，直接调用 hideIframe() 函数，并传递 URL 参数信息。

```
function request(){// 异步请求函数
    var user = document.getElementById("user");
    var pass = document.getElementById("pass");
    var s = "iframe_serve.html?user=" + user.value + "&pass=" +
pass.value;
    hideIframe(s);
}
```

由于浮动框架与框架集属于不同级别的作用域，浮动框架被包含在当前窗口中，所以应该使用 parent 而不是 parent.frames[0] 来调用回调函数，或者在回调函数中读取文档中的元素，代码如下：

```
function callback(b, n){
    if(b && n){
        var e = document.getElementsByTagName("body")[0];
        e.innerHTML = "<h1>" + n + "</h1><p>您好，欢迎登录站点</p>";
    }
    else{
        alert("你输入的用户名或密码有误，请重新输入");
        var user = parent.document.getElementById("user");
        var pass = parent.document.getElementById("pass");
        user.value = "";
        pass.value = "";
    }
}
```

在服务器端响应页面中也应该修改引用客户端回调函数的路径，代码如下：

```
window.onload = function(){
    //...
    parent.callback(b, n);
}
```

这样 iframe 浮动框架只需要两个文件：客户端交互页面（iframe_main.html）和服务器端响应页面（iframe_serve.html），就可以完成异步信息交互的任务。该示例的演示效果与建议 137 的示例相同，只不过完善了部分代码，因此不再进行效果演示和代码详解。

建议 139：使用 script 实现异步通信

利用 <script> 标签能够动态加载外部 JavaScript 脚本文件。JavaScript 脚本文件不仅可以被执行，还可以传输数据，所以不妨在服务器端使用 JavaScript 文件来附加传递响应信息，当在客户端使用 script 元素加载远程脚本文件时，这些附加在 JavaScript 文件中的响应信息也一同被加载到客户端，自然也能够达到异步信息交互的目的。

例如，新建一个客户端信息交互和请求页面（script_main.htm），然后输入下面的代码：

```
<html>
<head>
<title> 异步信息交互 </title>
<script language="javascript" type="text/javascript">
function callback(info){
    alert(info);
}
</script>
<script type="text/javascript" src="script_serve.js"></script>
</head>
<body>
<h1> 客户端信息交互页面 </h1>
</body>
</html>
```

接着在服务端的 script_serve.js 脚本文件中调用回调函数 callback()，代码如下：

```
callback("Hi，大家好，我是从服务器端过来的信息使者。");
```

此时，如果运行客户端交互页面（script_main.htm），将会在客户端交互页面中弹出“Hi，大家好，我是从服务器端过来的信息使者。”的响应信息，这些信息来自服务器端。

当服务器端 JavaScript 文件被加载到客户端交互页面中时，它包含的脚本就成为交互页面作用域中的一部分，也就是说，script_serve.js 文件中的脚本实际上已经成为 script_main.htm 页面脚本的一部分，即最终的运行结果应该是这样的：

```
<html>
<head>
<title> 异步信息交互 </title>
<script language="javascript" type="text/javascript">
function callback(info){
    alert(info);
}
</script>
<script language="javascript" type="text/javascript">
// 服务器端响应页面
callback("Hi，大家好，我是从服务器端过来的信息使者 .");
</script>
</head>
<body>
<h1> 客户端信息交互页面 </h1>
```

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

当然也可以动态生成 script 元素，以实现通过 script 元素实施异步交互功能的封装。

第 1 步，在页面中定义一个异步请求的封装函数，代码如下：

```
// 创建 <script> 标签
// 参数：URL 表示要请求的服务器端文件路径
// 返回值：无
function request(url){
    if( ! document.script){
        document.script = document.createElement("script");
        document.script.setAttribute("type", "text/javascript");
        document.script.setAttribute("src", url);
        document.body.appendChild(document.script);
    }
    else{
        document.script.setAttribute("src", url);
    }
}
```

第 2 步，完善客户端交互页面的结构和脚本代码。上面这个请求函数是整个 script 异步交互的核心。下面就可以来设计客户端交互页面，代码如下：

```
<html>
<head>
<title> 异步信息交互 </title>
<script language="javascript" type="text/javascript">
function callback(info){  // 客户端回调函数
    alert(info);
}
function request(url){    // script 异步请求函数
    // 代码同上
}
window.onload = function(){// 页面初始化处理函数
    var b = document.getElementsByTagName("input")[0];
    b.onclick = function(){// 为页面按钮绑定异步请求函数
        request("script_serve.js");
    }
}
</script>
</head>
<body>
<h1> 客户端信息交互页面 </h1>
<input name="submit" type="button" id="submit" value=" 向服务器发出请求 " />
</body>
</html>
```

第 3 步，在服务器端的响应文件（script_serve.js）中输入下面的代码：

```
callback("Hi，大家好，我是从服务器端过来的信息使者。");
```

上面 script_serve.js 文件是 JavaScript 脚本文件，而不是其他类型的网页文件，这样当单击按钮时才会触发异步请求和响应行为。

当使用 script 元素作为异步通信的工具时，实现信息交换的最简单方法就是利用参数从客户端向服务器端传递信息，这种在 URL 中附加参数的方式是最快捷的方法（URL 中附加参数即查询字符串）。然后服务器端接收这些参数，并把响应信息以 JavaScript 脚本形式传回客户端。

例如，在客户端交互页面（如“script 异步通信之参数传递 _main.htm”）中以下面的形式向服务器发出请求：

```
<html>
<head>
<title> 异步信息交互 </title>
<script language="javascript" type="text/javascript" src=
"code_serve.js?id=8"></script>
<body>
<h1> 客户端信息交互页面 </h1>
</body>
</html>
```

在 JavaScript 外部文件的 URL 中附加了一个参数 id=8，这个参数是客户端传递给服务器端的，希望服务器能够接收该参数，并能够根据该参数响应相应的信息，传回响应信息。这种想法是好的，问题是该如何让服务器端获取这个参数信息（如 id=8）呢？

利用 Location 对象的 search 属性能够捕获 HTTP 的 URL 以查询字符串信息，即在服务器端的 code_serve.js 文件中输入下面的代码：

```
var  queryString = location.search.substring(1);
alert(queryString);
```

但是，当运行客户端交互页面时，提示信息为空，说明服务器端并没有接收到这个参数，是不是很奇怪？如果使用下面的代码接收 HTTP 中完整的 URL 字符串信息，则会返回客户端交互页码的 URL 字符串，而不是链接的 JavaScript 文件 URL（如“http://localhost/mysite/script 异步通信之参数传递 _main.htm”字符串）。难怪使用上面代码会接收不到任何参数呢！

```
var  queryString = location.href;
alert(queryString);
```

使用 Location 对象是不能够接收客户端交互页面中包含的外部 JavaScript 连接文件的 URL 字符串信息的。在服务器端 JavaScript 文件中，使用脚本来读取客户端交互页面中 <script> 标签的 src 属性值，代码如下：

```
/* 遍历客户端交互页面的所有 <script> 标签，找到 src 属性包含 "script 异步通信之参数传递 _serve.js"
的标签，并匹配出来该 URL 的参数，从中筛选出附带回调函数名称的参数，然后利用这个回调函数执行服务器端传递
的信息 */
```

```
    var js = "script 异步通信之参数传递 _serve.js";        // 匹配的 JavaScript 文件名称
    var r = new RegExp(js + "(\\?(.*))?$");                // 定义匹配参数的正则表达式
    var script = document.getElementsByTagName("script");/* 获取客户端交互页面中包含的所有
script 元素 */
    for (var i = 0; i < script.length; i ++ ){ // 遍历所有 script 元素
       var s = script[i];
       if(s.src && s.src.match(r)){ // 判断是否存在参数
          var oo = s.src.match(r)[2];
          if (oo && (t = oo.match(/([^&=]+)=([^=&]+)/g))) { // 匹配出所有参数
             for (var l = 0; l < t.length; l ++ ) {// 遍历所有参数
                r = t[l];
                var c = r.match(/([^&=]+)=([^=&]+)/);        // 匹配每个参数
                if (c && (c[2]=="callback")){
                   var f = eval(c[2]);        // 激活回调函数名称字符串
                   f("Hi, 大家好，我是从服务器端过来的信息使者。");
                }
             }
          }
       }
    }
```

上面的 JavaScript 文件是服务器端请求的脚本文件。接下来运行客户端交互页面（script 异步通信之参数传递 _main_js.htm），在单击其中的【请求】按钮之后，会弹出正确信息。

要使用 JavaScript 文件向服务器请求信息，并希望得到服务器的积极响应，可以把 script 元素的 src 属性设置为请求服务器端脚本文件，而不是 JavaScript 文件。例如，以 ASP 服务器技术为例，可以这样进行请求：

```
window.onload = function(){
    var b = document.getElementsByTagName("input")[0];
    b.onclick = function(){
        var url = "script 异步通信之参数传递 _serve.asp?callback=callback"; // 请求 ASP 文件
        request(url);
    }
}
```

即请求的是文件名为 ASP 类型的服务器端脚本文件，而不是 JavaScript 脚本文件。这样，就可以利用服务器技术来接收请求传递的参数了，代码如下（参见“script 异步通信之参数传递 _serve.asp”文件）：

```
<%@LANGUAGE="VBSCRIPT" CODEPAGE="65001"%>
<%
callback = Request.QueryString("callback")
Response.Write("callback('Hi, 大家好，我是从服务器端过来的信息使者。')")
%>
```

另一个需要读者注意的问题就是字符编码的一致性。当服务器向客户端响应信息时，在 HTTP 传输中所使用的字符编码默认为 UTF-8，即国际通用编码。如果服务器端脚本编码为中文简体，则应该在服务器端响应信息的头部定义信息的编码为 GB2312。例如，在 ASP 脚

本文件中可以这样设置：

```
<%@LANGUAGE="VBSCRIPT" CODEPAGE="936"%>
<%
callback = Request.QueryString("callback")
Response.AddHeader "Content-Type","text/html;charset=gb2312"
Response.Write("callback('Hi，大家好，我是从服务器端过来的信息使者。')")
%>
```

在客户端交互页面中应该设置页面编码，具体编码与服务器端请求页面的编码类似：

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

要确保在异步交互过程中不发生乱码现象，应该保证相关页面和代码的字符编码是一致的，既可以统一使用国际通用编码，也可以统一使用中文简体编码（936 或 GB2312），默认使用国际通用编码（即 65001 或 UTF-8）。虽然 <script> 标签 src 属性请求的是 ASP 文件，但是 ASP 响应的字符串是符合 JavaScript 语法规则的字符串，当这些字符串被加载到客户端的 <script> 标签内部时，就会被转换为可以执行的 JavaScript 脚本代码。

最后，为了帮助读者了解一个完整而又清晰的 script 异步通信中参数传递的过程，我们把客户端和服务器端对应的文件代码全部整理出来（遵循默认的国际通用编码）。客户端交互页面的完整代码（html 文件类型）如下：

```
<html>
<head>
<title>异步信息交互</title>
<script language="javascript" type="text/javascript">
function callback(info){ // 回调函数
    alert(info);
}
function request(url){ // 请求函数
    if( ! document.script){
        document.script = document.createElement("script");
        document.script.setAttribute("type", "text/javascript");
        document.script.setAttribute("src", url);
        document.body.appendChild(document.script);
    }else{
        document.script.setAttribute("src", url);
    }
}
window.onload = function(){// 页面初始化处理
    var b = document.getElementsByTagName("input")[0];
    b.onclick = function(){// 鼠标单击事件处理函数，传递请求的服务器端脚本 URL 和参数
        var url = "script 异步通信之参数传递 _serve.asp?callback=callback"
        request(url);
    }
}
</script>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8"></head>
```

```
<body>
<h1> 客户端信息交互页面 </h1>
<input name="submit" type="button" id="submit" value=" 向服务器发出请求 " />
</body>
</html>
```

服务器端响应页面的完整代码（ASP 文件类型）如下：

```
<%@LANGUAGE="VBSCRIPT" CODEPAGE="65001"%>
<%
callback = Request.QueryString("callback")' 接收参数
Response.Write("callback('Hi，大家好，我是从服务器端过来的信息使者。')")' 输出响应信息
%>
```

在测试上面代码时，应确保在服务器环境下运行，否则达不到预期结果。

建议 140：正确理解 JSONP 异步通信协议

浏览器安全模型规定：XMLHttpRequest 和框架（frame）等只能够在同一个域内进行异步通信。受同源策略的限制，不能够使用 Ajax 或框架实现跨域（即与外部服务器）通信。从安全角度考虑，这个规定很合理，不过也会给分布式 Web 开发带来麻烦。而 JSONP 是一种可以绕过同源策略的方法，从任何服务器端直接返回可执行的 JavaScript 函数调用（即回调函数）或 JavaScript 对象（JSON 格式数据）。

JSONP 是 JSON with Padding 的简称，它能够通过在当前文档（客户端）中生成脚本（即 <script> 标签）来调用跨域脚本（服务器端脚本文件），这是一个非官方的协议。

JSONP 允许在服务器端集成 Script Tags（脚本标记，即服务器动态生成的 JavaScript 脚本字符串）并把这些脚本标记返回到客户端，通过 JavaScript Callback（JavaScript 回调函数）的形式实现跨域访问。当然，这仅仅是 JSONP 简单的实现形式。现在的一些 JavaScript 技术框架都在使用 JSONP 实现跨域异步通信，如 dojo、JQuery、Google Social Graph API 、Digg API 、GeoNames webservice、豆瓣 API、Del.icio.us JSON API 等。下面通过一个示例介绍如何通过 JSONP 约定来实现跨域异步信息交互。

第 1 步，在客户端调用提供 JSONP 支持的 URL Service（URL 服务），获取 JSONP 格式数据。所谓 JSONP 支持的 URL Service，就是在请求的 URL 中必须附加在客户端的可以回调的函数，并且按约定正确设置回调函数参数，默认参数名为 jsonp，偶尔也会见到以“callback”为参数名传送回调函数名的情况。总之，只要服务器能够识别即可。例如，定义 URL Service 为：

```
http:// localhost/mysite/ serve.ASP?jsonp=callback&d=1
```

其中参数 jsonp 的值为约定的回调函数名。JSONP 格式的数据就是把 JSON 数据作为参数传递给回调函数并传回。如果响应的 JSON 数据为：

```
{
    "title" : "JSONP Test",
    "link" : "http:// www.jscode.cn/",
    "modified" : "2012-4-1",
    "items" : {
        "id" : 1,
        "title" : " 百度 ",
        "link" : "http:// www.baidu.com/",
        "description" : " 百度一下，你就知道 "
    }
}
```

那么真正返回到客户端的 Script Tags（脚本标记）如下，即所谓的 JSONP 格式数据。

```
callback({
    "title" : "JSONP Test",
    "link" : "http:// www.jscode.cn/",
    "modified" : "2012-4-1",
    "items" : {
        "id" : 1,
        "title" : " 百度 ",
        "link" : "http:// www.baidu.com/",
        "description" : " 百度一下，你就知道 "
    }
})
```

第 2 步，在客户端向服务器端发出 URL Service 请求且服务器接到请求之后，应该完成两件事情：一是接收并处理参数信息，特别是要获取回调函数名；二是根据参数信息生成符合客户端需要的 Script Tags（脚本标记）字符串，并且把这些字符串响应给客户端。例如，服务器端的处理脚本文件如下：

```
<%@LANGUAGE="VBSCRIPT" CODEPAGE="65001"%>
<%
callback = Request.QueryString("jsonp")  // 接收回调函数名的参数值
id = Request.QueryString("id")           // 接收响应信息的编号
Response.AddHeader "Content-Type","text/html;charset=utf-8"     // 设置响应信息的字符
编码为 uft-8
Response.Write(callback & "(")           // 输出回调函数名，开始生成 Script Tags 字符串
%>
{
    "title" : "JSONP Test",
    "link" : "http:// www.jscode.cn/",
    "modified" : "2012-4-1",
    "items" :
<%
if id = "1" then     // 如果 id 参数值为 1，则输出下面的对象信息
%>
    {
        "title" : " 百度 ",
        "link" : "http:// www.baidu.com/",
        "description" : " 百度一下，你就知道 "
```

```
    }
<%
elseif id = "2" then  //如果 id 参数值为 2，则输出下面的对象信息
%>
    {
        "title" : "搜狗",
        "link" : "http:// www.sogou.com/",
        "description" : "上网从搜狗开始"
    }
<%
else  //否则输出空信息
    Response.Write(" ")
end if     //结束条件语句
Response.Write("))")    //封闭回调函数，输出 Script Tags 字符串
%>
```

包含在“<%”和“%>”分隔符之间的代码是 ASP 处理脚本。在该分隔符之后的是输出到客户端的普通字符串。在 ASP 脚本中，使用 Response.Write() 方法输出回调函数名和运算符号。其中还用条件语句来判断从客户端传递过来的参数值，并根据参数值决定响应的具体信息。

第 3 步，在客户端设计回调函数。设计回调函数应该根据具体的应用项目，以及返回的 JSONP 数据进行处理。例如，针对上面返回的 JSONP 数据，把其中的数据列表显示出来，代码如下：

```
function callback(info){
    var temp = "";
    for(var i in info){
        if(typeof info[i] != "object"){
            temp += i + " = \"" + info[i] + "\"<br />";
        }
        else if( (typeof info[i] == "object")){
            temp += "<br />" + i + " = " + " {<br />";
            var o = info[i];
            for(var j in o ){
                temp += "    " + j + " = \"" + o[j]
+ "\"<br />";
            }
            temp += "}";
        }
    }
    var div  = document.getElementById("test");
    div.innerHTML = temp;
}
```

第 4 步，设计客户端交互页面与信息展示。这一步比较简单，可以在页面中插入一个 标签，然后把输出的信息插入到该元素中。同时为页面设计一个交互按钮，单击该按钮将触发请求函数，并且向服务器端发出 URL Service 请求。响应完毕，在 Script Tags 字符串被传回到客户端之后，将调用回调函数对响应的数据进行处理和显示。

```
<div id="test"></div>
```

建议 141：比较常用的服务器请求方法

目前，主要有 5 种常用技术用于向服务器请求数据：

- XMLHttpRequest（XHR）
- Dynamic script tag insertion（动态脚本标签插入）
- iframe
- Comet
- Multipart XHR（多部分 XHR）

在现代高性能 JavaScript 中，推荐使用的技术是 XHR、动态脚本标签插入和多部分 XHR。作为数据传输技术，往往在极限情况下使用 Comet 和 iframe。

（1）XHR

目前最常用的方法是使用 XHR 实现异步收发数据。目前所有浏览器都能够很好地支持这种方法，而且能够精细地控制发送请求和接收数据。可以向请求中添加任意的头信息和参数（包括 GET 和 POST），读取从服务器返回的头信息，以及响应文本自身。

```
var url = '/data.php';
var params = ['id=934875', 'limit=20'];
var req = new XMLHttpRequest();
req.onreadystatechange = function() {
    if(req.readyState === 4) {
        var responseHeaders = req.getAllResponseHeaders();
        var data = req.responseText;
    }
}
req.open('GET', url + '?' + params.join('&'), true);
req.setRequestHeader('X-Requested-With', 'XMLHttpRequest');
req.send(null);
```

上面代码显示了如何从 URL 请求数据、使用参数，以及如何读取响应报文和头信息。readyState 等于 4 表示整个响应报文已经接收完并可用于操作。readyState 等于 3 则表示此时正在与服务器交互，响应报文还在传输之中。这就是所谓的流，它是提高数据请求性能的强大工具。

```
req.onreadystatechange = function() {
    if(req.readyState === 3) {// 一些但不是全部数据被接收
        var dataSoFar = req.responseText;
    } else if(req.readyState === 4) {// 所有数据被接收
        var data = req.responseText;
    }
}
```

由于 XHR 提供了高级别的控制，浏览器增加了一些限制，因此用户不能使用 XHR 从当

前运行的代码域之外请求数据，更何况早期版本的 IE 也不提供 readyState=3，不支持流。像对待一个字符串或者一个 XML 对象那样对待从请求返回的数据，这意味着处理大量数据将相当缓慢。

尽管有这些缺点，XHR 仍然是最常用的请求数据技术，也是最强大的，因此它应当成为异步数据通信的首选。当使用 XHR 请求数据时，可以选择 POST 或 GET。如果请求不改变服务器状态只是取回数据，则使用 GET。GET 请求被缓冲起来，多次提取相同的数据可提高性能。

只有当 URL 和参数的长度超过了 2048 个字符时才使用 POST 提取数据，这是因为 IE 限制 URL 的长度，过长将导致请求（参数）被截断。

（2）动态脚本标签插入

该技术克服了 XHR 的最大限制，可以从不同域的服务器上获取数据。这是一种黑客技术，而不是实例化一个专用对象，利用 JavaScript 创建了一个新脚本标签，并将它的源属性设置为一个指向不同域的 URL。

```
var scriptElement = document.createElement('script');
scriptElement.src = 'http://any-domain.com/Javascript/lib.js';
document.getElementsByTagName_r('head')[0].appendChild(scriptElement);
```

动态脚本标签插入与 XHR 相比只提供更少的控制。用户不能通过请求发送信息头。参数只能通过 GET 方法传递，不能用 POST。不能设置请求的超时或重试，并且必须等待所有数据返回之后才可以访问它们。也不能访问响应信息头或像访问字符串那样访问整个响应报文。

因为响应报文被用做脚本标签的源码，所以它必须是可执行的 JavaScript。任何数据，无论什么格式，必须在一个回调函数之中被组装起来。

```
var scriptElement = document.createElement('script');
scriptElement.src = 'http://any-domain.com/Javascript/lib.js';
document.getElementsByTagName_r('head')[0].appendChild(scriptElement);
function jsonCallback(jsonString) {
    var data = ('(' + jsonString + ')');
}
```

在上面示例中，lib.js 文件将调用 jsonCallback 函数组装数据：

```
jsonCallback({ "status": 1, "colors": [ "#fff", "#000", "#ff0000" ] });
```

尽管有这些限制，此技术在数据传输上仍然是非常迅速的。其响应结果是运行 JavaScript，而不是必须作为字符串被进一步处理。正因为如此，它是客户端获取并解析数据最快的方法。

由于 JavaScript 中没有权限或访问控制的概念，所以页面上任何使用动态脚本标签插入的代码都可以完全控制整个页面，包括修改任何内容、将用户重定向到另一个站点，以及跟踪在页面上的操作并将数据发送给第三方。使用外部来源的代码时要非常小心。

（3）多部分 XHR

多部分 XHR 允许只用一个 HTTP 请求就可以从服务器端获取多个资源。它将资源（可以是 CSS 文件、HTML 片段、JavaScript 代码，或 base64 编码的图片）打包成一个由特定分隔符界定的长字符串，将其从服务器端发送到客户端。JavaScript 代码处理此长字符串，根据它的媒体类型和其他“信息头”解析出每个资源。首先，发送一个请求向服务器索取几个图像资源：

```
var req = new XMLHttpRequest();
req.open('GET', 'rollup_images.php', true);
req.onreadystatechange = function() {
    if(req.readyState == 4) {
        splitImages(req.responseText);
    }
};
req.send(null);
```

这是一个非常简单的请求：向 rollup_images.php 请求数据，一旦收到返回结果，就将它交给函数 splitImages 处理。

然后，服务器读取图片并将它们转换为字符串：

```
$images = array('kitten.jpg', 'sunset.jpg', 'baby.jpg');
foreach ($images as $image) {
    $image_fh = fopen($image, 'r');
    $image_data = fread($image_fh, filesize($image));
    fclose($image_fh);
    $payloads[] = base64_encode($image_data);
}
$newline = chr(1);
echo implode($newline, $payloads);
```

这段 PHP 代码读取 3 张图片，并将它们转换成 base64 字符串。这些字符串之间用一个简单的字符连接起来，然后返回给客户端。

接下来回到客户端，此数据由 splitImages 函数处理：

```
function splitImages(imageString) {
    var imageData = imageString.split("\u0001");
    var imageElement;
    for(var i = 0, len = imageData.length; i < len; i++) {
        imageElement = document.createElement('img');
        imageElement.src = '' + imageData[i];
        document.getElementById('container').appendChild(imageElement);
    }
}
```

此函数将拼接而成的字符串分解为 3 段，每段用于创建一个图像元素，然后将图像元素插入页面中。图像不是从 base64 转换成二进制，而是使用 data:URL 并指定 image/jpeg 媒体类型。

最终结果：在一次 HTTP 请求中向浏览器传入了 3 张图片。也可以传入 20 张或 100 张图片，这样响应报文会更大，但也只是一次 HTTP 请求。这种响应也可以扩展至其他类型的资源，JavaScript 文件、CSS 文件、HTML 片段、不同类型的图片都可以合并成一次响应。任何数据类型都可作为一个 JavaScript 处理的字符串被发送。下面的函数用于将 JavaScript 代码、CSS 样式表和图片转换为浏览器可用的资源。

```
function handleImageData(data, mimeType) {
    var img = document.createElement('img');
    img.src = 'data:' + mimeType + ';base64,' + data;
    return img;
}
function handleCss(data) {
    var style = document.createElement('style');
    style.type = 'text/css';
    var node = document.createTextNode(data);
    style.appendChild(node);
    document.getElementsByTagName_r('head')[0].appendChild(style);
}
function handleJavascript(data) {(data);}
```

由于 MXHR 响应报文越来越大，有必要在每个资源收到响应时立刻处理，而不是等待整个响应报文接收完成后再处理。这可以通过监听 readyState=3 实现。

```
var req = new XMLHttpRequest();
var getLatestPacketInterval, lastLength = 0;
req.open('GET', 'rollup_images.php', true);
req.onreadystatechange = readyStateHandler;
req.send(null);
function readyStateHandler() {
    if(req.readyState === 3 && getLatestPacketInterval === null) {
        getLatestPacketInterval = window.setInterval(function() {
            getLatestPacket();
        }, 15);
    }
    if(req.readyState === 4) {
        clearInterval(getLatestPacketInterval);
        getLatestPacket();
    }
}
function getLatestPacket() {
    var length = req.responseText.length;
    var packet = req.responseText.substring(lastLength, length);
    processPacket(packet);
    lastLength = length;
}
```

当 readyState=3 第一次发出时，启动一个定时器，每隔 15 ms 检查一次响应报文中的新数据，数据片段被收集起来直到发现一个分隔符，然后一切都作为一个完整的资源处理。

以健壮的方式使用 MXHR 的代码很复杂，但值得进一步研究。完整的库可参见 http://

techfoolery.com/mxhr/。

使用这种技术有一些缺点，其中最大的缺点是以此方法获得的资源不能被浏览器缓存。使用 MXHR 获取一个特定的 CSS 文件，然后在下一个页面中正常加载它，会发现它不在缓存中，这是因为整批资源作为一个长字符串传输，然后由 JavaScript 代码分割。由于没有办法通过程序将文件放入浏览器缓存中，因此用这种方法获取的资源也无法存放在浏览器缓存中。另外，早期版本的 IE 不支持 readyState=3 或 data:URL，从 IE 8 开始支持这些技术，在 IE 6 和 IE 7 中必须设法变通。在某些情况下，MXHR 能够显著地提高整体页面的性能：

- 网页包含许多其他地方不会用到的资源，不需要缓存资源，尤其是图片。
- 网站为每个页面使用了独一无二的打包的 JavaScript 或 CSS 文件以减少 HTTP 请求，因为它们对每个页面来说是独特的，所以不需要从缓存中读取，除非重新载入特定页面。
- 由于 HTTP 请求是 Ajax 中最极端的瓶颈之一，减少其需求数量对整个页面性能有很大影响，尤其是将 100 个图片请求转化为一个 MXHR 请求时能够极大地提高响应速度。

建议 142：比较常用的服务器发送数据方法

有时可以不关心接收数据，只要将数据发送给服务器即可。可以发送用户的非私有信息以备日后分析，或者捕获所有脚本错误，然后将有关细节发送给服务器进行记录和提示。当数据只需发送给服务器时，有两种应用得非常广泛的技术：XHR 和灯标。

（1）XHR

XHR 主要用于从服务器获取数据，也可以用来将数据发回。可以用 GET 或 POST 方式发回数据，以及任意数量的 HTTP 信息头。在向服务器发回的数据量超过浏览器的最大 URL 长度时 XHR 特别有用。在这种情况下，可以用 POST 方式发回数据：

```
var url = '/data.php';
var params = ['id=934875', 'limit=20'];
var req = new XMLHttpRequest();
req.onerror = function() {
};
req.onreadystatechange = function() {
    if(req.readyState == 4) {
    }
};
req.open('POST', url, true);
req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
req.setRequestHeader('Content-Length', params.length);
req.send(params.join('&'));
```

在上面代码中，如果通信失败，则什么也不做。当使用 XHR 捕获登录用户统计信息时，这么做通常没什么问题，但是，如果发送到服务器的是至关重要的数据，那么可以添加代码在失败时重试：

```
function xhrPost(url, params, callback) {
    var req = new XMLHttpRequest();
    req.onerror = function() {
        setTimeout(function() {
            xhrPost(url, params, callback);
        }, 1000);
    };
    req.onreadystatechange = function() {
        if(req.readyState == 4) {
            if(callback && typeof callback === 'function') {
                callback();
            }
        }
    };
    req.open('POST', url, true);
    req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
    req.setRequestHeader('Content-Length', params.length);
    req.send(params.join('&'));
}
```

使用 XHR 将数据发回服务器比使用 GET 要快，这是因为对少量数据而言，向服务器发送一个 GET 请求要占用一个单独的数据包。另外，一个 POST 至少发送两个数据包，一个用于信息头，另一个用于 POST 体。POST 更适合于向服务器发送大量数据，既因为它不关心额外数据包的数量，又因为 IE 的 URL 长度限制，不可能使用过长的 GET 请求。

（2）灯标

灯标与动态脚本标签插入非常类似。JavaScript 创建一个新的 Image 对象，将 src 设置为服务器上一个脚本文件的 URL，该 URL 包含通过 GET 格式传回的键值对数据。注意，这里并没有创建 img 元素或将它们插入到 DOM 中。

```
var url = '/status_tracker.php';
var params = ['step=2', 'time=1248027314'];
(new Image()).src = url + '?' + params.join('&');
```

服务器得到此数据并保存下来，不必向客户端返回什么，因此没有实际的图像显示。这是将信息发回服务器的最有效方法，开销很小，而且任何服务器端错误都不会影响客户端。

由于简单的图像灯标不能发送 POST 数据，所以应将 URL 长度限制在一个相当小的字符数量上。当然也可以用非常有限的方法接收返回数据，可以监听 Image 对象的 load 事件，判断服务器端是否成功接收了数据。还可以检查服务器返回图片的宽度和高度（如果返回了一张图片）并用这些数字通知服务器的状态。例如，宽度为 1 表示成功，2 表示重试等。

如果不需要为此响应返回数据，那么应当发送一个 204 No Content 响应代码，表示无消息正文，从而阻止客户端继续等待永远不会到来的消息体。

```
var url = '/status_tracker.php';
var params = ['step=2', 'time=1248027314'];
var beacon = new Image();
```

```
beacon.src = url + '?' + params.join('&');
beacon.onload = function() {
    if(this.width == 1) {
        // 成功处理
    } else if(this.width == 2) {
        // 失败处理
    }
};
beacon.onerror = function() {
    // 错误处理
};
```

灯标是向服务器回送数据最快和最有效的方法。因为服务器根本不需要发回任何响应正文，所以不必担心客户端下载数据。使用灯标的唯一缺点是接收到的响应类型是受限的。如果需要向客户端返回大量数据，那么使用 XHR。如果只关心将数据发送到服务器端，那么使用图像灯标。

建议 143：避免使用 XML 格式进行通信

在数据传输实现中，可能需要考虑功能性、兼容性，以及其他性能和方向（发给服务器或从服务器接收）。在考虑数据格式时，唯一需要关注的问题就是速度。

没有哪种数据格式会始终比其他格式更好。根据传送数据的类型、在页面上的使用目的不同，某种格式数据可能下载更快，另一种格式可能解析更快。

与其他格式相比，XML 格式数据极其冗长。因为每个离散的数据片断需要大量 XML 结构，所以有效数据的比例非常低。XML 语法还有些轻微模糊。

在一般情况下，解析 XML 要占用 JavaScript 程序员相当一部分精力。除了要提前知道详细结构之外，还必须确切地知道如何解开这个结构，然后精心地将它们写入 JavaScript 对象中。下面的代码表示如何将特定 XML 报文解析到对象中。

```
function parseXML(responseXML) {
    var users = [];
    var userNodes = responseXML.getElementsByTagName_r('users');
    var node, usernameNodes, usernameNode, username, realnameNodes, realnameNode,
realname, emailNodes, emailNode, email;
    for(var i = 0, len = userNodes.length; i < len; i++) {
        node = userNodes[i];
        username = realname = email = '';
        usernameNodes = node.getElementsByTagName_r('username');
        if(usernameNodes && usernameNodes[0]) {
            usernameNode = usernameNodes[0];
            username = (usernameNodes.firstChild) ? usernameNodes.firstChild.nodeValue
: '';
        }
        realnameNodes = node.getElementsByTagName_r('realname');
        if(realnameNodes && realnameNodes[0]) {
```

```
                realnameNode = realnameNodes[0];
                realname = (realnameNodes.firstChild) ? realnameNodes.firstChild.nodeValue
: '';
            }
            emailNodes = node.getElementsByTagName_r('email');
            if(emailNodes && emailNodes[0]) {
                emailNode = emailNodes[0];
                email = (emailNodes.firstChild) ? emailNodes.firstChild.nodeValue : '';
            }
            users[i] = {
                id : node.getAttribute('id'),
                username : username,
                realname : realname,
                email : email
            };
        }
        return users;
    }
```

在上面代码中，JavaScript 引擎在读值之前，需要检查每个标签以保证该标签存在。这在很大程度上依赖于 XML 的结构。一个更有效的方式是将每个值都存储为 <user> 标签的属性，数据相同而文件尺寸却更小。

```
    <?xml version="1.0" encoding='UTF-8'?>
    <users total="4">
        <user id="1-id001" username="alice" realname="Alice Smith" email="alice@
alicesmith.com" />
        <user id="2-id001" username="bob" realname="Bob Jones" email="bob@bobjones.
com" />
        <user id="3-id001" username="carol" realname="Carol Williams" email="carol@
carolwilliams.com" />
        <user id="4-id001" username="dave" realname="Dave Johnson" email="dave@
davejohnson.com" />
    </users>
```

解析简化版 XML 数据要容易得多：

```
    function parseXML(responseXML) {
        var users = [];
        var userNodes = responseXML.getElementsByTagName_r('users');
        for(var i = 0, len = userNodes.length; i < len; i++) {
            users[i] = {
                id : userNodes[i].getAttribute('id'),
                username : userNodes[i].getAttribute('username'),
                realname : userNodes[i].getAttribute('realname'),
                email : userNodes[i].getAttribute('email')
            };
        }
        return users;
    }
```

XPath 在解析 XML 文档时比 getElementsByTagName 快得多。需要注意的是，由于

XPath 并未得到广泛支持，所以必须使用 DOM 遍历方法编写备用代码。现在，DOM 级别为 3 的 XPath 已经被 Firefox、Safari、Chrome 和 Opera 浏览器支持，IE 8 提供一个类似的接口。

建议 144：推荐使用 JSON 格式进行通信

JSON 是一个轻量级并易于解析的数据格式，它按照 JavaScript 对象和数组字面语法来编写。下面代码是用 JSON 编写的用户列表。

```
[{
    "id" : 1,
    "username" : "alice",
    "realname" : "Alice ",
    "email" : "alice@163.com"
}, {
    "id" : 2,
    "username" : "bob",
    "realname" : "Bob ",
    "email" : "bob@163.com"
}, {
    "id" : 3,
    "username" : "carol",
    "realname" : "Carol",
    "email" : "carol@163.com"
}, {
    "id" : 4,
    "username" : "dave",
    "realname" : "Dave",
    "email" : "dave@163.com"
}]
```

用户为一个对象，用户列表为一个数组，与 JavaScript 中其他数组或对象的写法相同。这意味着如果对象被包装在一个回调函数中，JSON 数据可以成为能够运行的 JavaScript 代码。在 JavaScript 中解析 JSON 可简单地使用 ()：

```
function parseJSON(responseText) {
   return ('(' + responseText + ')');
}
```

上面的 JSON 数据也可以提炼成一个更简单的版本，将名字缩短：

```
[{
    "i" : 1,
    "u" : "alice",
    "r" : "Alice ",
    "e" : "alice@163.com"
}, {
    "i" : 2,
    "u" : "bob",
```

```
        "r" : "Bob ",
        "e" : "bob@163.com"
    }, {
        "i" : 3,
        "u" : "carol",
        "r" : "Carol ",
        "e" : "carol@163.com"
    }, {
        "i" : 4,
        "u" : "dave",
        "r" : "Dave",
        "e" : "dave@163.com"
    }]
```

JSON 精简版本将相同的数据以更少的结构和更小的字节尺寸传送给浏览器。也可以完全去掉属性名，与原格式相比，这种格式可读性更差，但更利索，文件尺寸非常小：大约只有标准 JSON 格式的一半。

```
[
     [ 1, "alice", "Alice ", "alice@163.com" ],
     [ 2, "bob", "Bob", "bob@163.com" ],
     [ 3, "carol", "Carol ", "carol@163.com" ],
     [ 4, "dave", "Dave ", "dave@163.com" ]
]
```

解析过程需要保持数据的顺序，也就是说，这种精简格式在进行格式转换时必须保持和第一个 JSON 格式一样的属性名：

```
function parseJSON(responseText) {
    var users = [];
    var usersArray = ('(' + responseText + ')');
    for(var i = 0, len = usersArray.length; i < len; i++) {
        users[i] = {
            id : usersArray[i][0],
            username : usersArray[i][1],
            realname : usersArray[i][2],
            email : usersArray[i][3]
        };
    }
    return users;
}
```

在上面代码中，使用 () 将字符串转换为一个本地 JavaScript 数组，然后再将它转换为一个对象数组，用一个更复杂的解析函数换取了较小的文件尺寸和更快的时间。数组形式的 JSON 在每一项性能比较中均获胜，它文件尺寸最小，下载最快，平均解析时间最短。尽管解析函数不得不遍历列表中所有 5000 个单元，它的速度还是提高了 30%。

当使用 XHR 时，JSON 数据作为一个字符串返回。该字符串通过 () 转换为一个本地对象。然而，当使用动态脚本标签插入时，JSON 数据被视为另一个 JavaScript 文件并作为本

地码执行。为做到这一点，数据必须被包装在回调函数之中，这就是所谓的“JSON 填充”或 JSONP。下面用 JSONP 格式编写用户列表。

```
parseJSON([{
    "id" : 1,
    "username" : "alice",
    "realname" : "Alice ",
    "email" : "alice@163.com"
}, {
    "id" : 2,
    "username" : "bob",
    "realname" : "Bob ",
    "email" : "bob@163.com"
}, {
    "id" : 3,
    "username" : "carol",
    "realname" : "Carol",
    "email" : "carol@163.com"
}, {
    "id" : 4,
    "username" : "dave",
    "realname" : "Dave",
    "email" : "dave@163.com"
}]);
```

因为回调包装的原因，所以 JSONP 略微增加了文件的尺寸，但与其在解析性能上的改进相比这点增加微不足道。由于数据作为本地 JavaScript 处理，所以它的解析速度与本地 JavaScript 一样快。

JSONP 文件大小和下载时间与 XHR 测试基本相同，而解析时间几乎快了 10 倍。标准 JSONP 的解析时间为 0，因为根本用不着解析，它已经是本地格式了。简化版 JSONP 和数组 JSONP 也是如此，只是每种 JSONP 都需要转换成标准 JSONP 直接使用的格式。

最快的 JSON 格式是使用数组的 JSONP 格式，虽然这种格式只比使用 XHR 的 JSON 略快，但是这种差异随着列表尺寸的增大而增大。如果所从事的项目需要一个由 10 000 或 100 000 个单元构成的列表，那么使用 JSONP 比使用 JSON 好很多。

要避免使用 JSONP 还有一个与性能无关的原因：JSONP 必须是可执行的 JavaScript，利用动态脚本标签注入技术可在任何网站中被任何人调用。从另一个角度来说，JSON 在运行之前并不是有效的 JavaScript，使用 XHR 时只是被当做字符串获取。不要将任何敏感的数据编码为 JSONP，因为无法确定它是否包含私密信息、随机的 URL 或 cookie。

与 XML 相比，JSON 有许多优点：格式小得多；在总响应报文中，结构占用的空间更小；数据占用得更多，特别是在数据包含数组而不是对象时。JSON 与大多数服务器端语言的编/解码库之间有着很好的互操作性。JSON 在客户端的解析工作微不足道，可以将更多写代码的时间放在其他数据处理上。对网页开发者来说最重要的是，它是表现最好的格式之一，既因为在线传输相对较小，也因为解析十分快速。JSON 是高性能 Ajax 的基石，特别是在使用

动态脚本标签插入时。

建议 145：慎重使用 HTML 格式进行通信

JavaScript 虽然能够比较快地将一个大数据结构转化为简单的 HTML，但是服务器完成同样工作的速度更快。一种技术考虑是在服务器端构建整个 HTML，然后将其传递给客户端，JavaScript 只是简单地下载它并将其放入 innerHTML。下面使用 HTML 编码用户列表。

```
<ul class="users">
    <li class="user" id="1-id">
        <a href="#" class="username">alice</a>
        <span class="realname">Alice </span>
        <a href="mailto:alice@163.com" class="email">alice@163.com</a>
    </li>
    <li class="user" id="2-id">
        <a href="#" class="username">bob</a>
        <span class="realname">Bob </span>
        <a href="mailto:bob@163.com" class="email">bob@163.com</a>
    </li>
    <li class="user" id="3-id">
        <a href="#" class="username">carol</a>
        <span class="realname">Carol </span>
        <a href="mailto:carol@163.com" class="email">carol@163.com</a>
    </li>
    <li class="user" id="4-id">
        <a href="#" class="username">dave</a>
        <span class="realname">Dave </span>
        <a href="mailto:dave@163.com" class="email">dave@163.com</a>
    </li>
</ul>
```

使用 HTML 格式的问题在于，HTML 是一种详细的数据格式，比 XML 更加冗长。数据本身的最外层可以有嵌套的 HTML 标签，每个标签都具有 ID、类和其他属性。尽管 HTML 格式可通过尽量少用标签和属性来缓解占用空间多的问题，但是 HTML 格式还是可能比实际数据占用更多的空间。正因为如此，只有当客户端 CPU 比带宽更受限时才使用此技术。

一种极端情况是，有一种格式包含最少数量的结构，需要在客户端解析数据，如 JSON。将这种格式下载到客户机非常快，然而这一过程需要引擎花费更多的时间把它转化成 HTML 以显示在页面上。这需要很多字符串操作，而字符串操作也是 JavaScript 最慢的操作之一。

另一种极端情况是，在服务器上创建 HTML。这种格式在线传输数据量大，下载时间长，不过一旦下载完，只需一个操作就可以显示在页面上。这种格式与其他几种格式的差别："解析"在这种情况下指的是将 HTML 插入 DOM 的操作。此外，HTML 不能像本地 JavaScript 数组那样轻易且迅速地进行迭代操作。

HTML 传输数据量明显偏大，需要的解析时间也很长。将 HTML 插入到 DOM 的单一操

作看似简单，只有一行代码，却需要时间向页面加载很多数据。与其他测试相比，这些性能数据确实有轻微的偏差，最终结果不是数据数组，而是显示在页面上的 HTML 元素。无论如何，结果仍显示出 HTML 的一个事实：作为数据格式，它缓慢且臃肿。

建议 146：使用自定义格式进行通信

最理想的数据格式只包含必要的结构，并且能够分解出每个字段。可以自定义一种格式，只简单地用一个分隔符将数据连接起来。

```
d;Tyler;John
```

这些分隔符基本上创建了一个数据数组，类似于一个逗号分隔的列表。通过使用不同的分隔符，可以创建多维数组。这里是用自定义的字符分隔方式构造的用户列表：

```
alice:Alice:alice@163.com;
bob:Bob:bob@163.com;
carol:Carol:carol@163.com;
dave:Dave:dave@163.com
```

这种格式非常简洁，与其他格式相比（不包括纯文本），其数据 / 结构比例明显提高。自定义格式下载迅速，易于解析，只需调用字符串的 split() 将分隔符作为参数传入即可。更复杂的自定义格式具有多种分隔符，需要在循环中分解所有数据。注意，在 JavaScript 中这些循环处理起来是非常快的。split() 是最快的字符串操作之一，通常可以在数毫秒内处理具有超过 10 000 个元素的由分隔符分割的列表。下面的例子给出解析上述格式的方法。

```
function parseCustomFormat(responseText) {
    var users = [];
    var usersEncoded = responseText.split(';');
    var userArray;
    for(var i = 0, len = usersEncoded.length; i < len; i++) {
        userArray = usersEncoded[i].split(':');
        users[i] = {
            id : userArray[0],
            username : userArray[1],
            realname : userArray[2],
            email : userArray[3]
        };
    }
    return users;
}
```

当创建自定义格式时，最重要的决定是采用何种分隔符。在理想情况下，分隔符应当是一个单字符，而且不能存在于数据之中。ASCII 字符表中前面的几个字符在大多数服务器端语言中能够正常工作且容易书写。下面讲述如何在 PHP 中使用 ASCII 码。

```
function build_format_custom($users) {
    $row_delimiter = chr(1);      // \u0001 in JavaScript.
    $field_delimiter = chr(2);    // \u0002 in JavaScript.
    $output = array();
    foreach ($users as $user) {
        $fields = array($user['id'], $user['username'], $user['realname'], $user['email']);
        $output[] = implode($field_delimiter, $fields);
    }
    return implode($row_delimiter, $output);
}
```

这些控制字符在 JavaScript 中使用 Unicode 标注（如 \u0001）表示。split() 函数可以用字符串或正则表达式作为参数。如果希望数据中存在空字段，那么就使用字符串；如果分隔符是一个正则表达式，那么 IE 中的 split() 将跳过相邻两个分隔符中的第二个分隔符。这两种参数类型在其他浏览器上等价。

```
var rows = req.responseText.split(/\u0001/);
var rows = req.responseText.split("\u0001");
```

这是字符分隔的自定义格式的性能数据，使用 XHR 和动态脚本标签注入。

XHR 和动态脚本标签注入都可以使用这种格式。在两种情况下都要解析字符串，在性能上没有实质上的差异。对于非常大的数据集，自定义格式是最快的传输格式，甚至可以在解析速度和下载时间上“击败”本机执行的 JSON。用此格式向客户端传送大量数据只需要很少的时间。

总的来说，越轻量级的格式越好，最好是 JSON 和字符分隔的自定义格式。如果数据集很大或解析时间成问题，那么就使用以下两种格式之一：

- JSONP 数据，用动态脚本标签插入法获取。这种格式将数据视为可运行的 JavaScript 而不是字符串，解析速度极快。这种格式能够跨域使用，但不应涉及敏感数据。
- 字符分隔的自定义格式，使用 XHR 或动态脚本标签插入技术提取，使用 split() 解析。此技术在解析非常大数据集时比 JSONP 技术略快，而且通常文件尺寸更小。注意，这些数据只是在一个浏览器上进行一次测试获得的。测试结果可用做大概的性能指标，而不是确切的数字。

建议 147：Ajax 性能向导

一旦选择了最合适的数据传输技术和数据格式，就要开始考虑其他的优化技术。这些技术要根据具体情况使用，在考虑使用它们之前首先应确认应用程序是否适合这些概念。

有两种主要方法避免发出一个不必要的请求：

- 在服务器端，设置 HTTP 头，确保返回报文将被缓存在浏览器中。
- 在客户端，对于本地缓存已获取的数据，不要多次请求同一个数据。

第一种方法最容易设置和维护，而第二个方法提供了最大程度的控制。

如果希望 Ajax 响应报文能够被浏览器所缓存，必须在发起请求时使用 GET 方法。这还不够，必须在响应报文中发送正确的 HTTP 头。Expires 头告诉浏览器应当缓存响应报文多长时间，其值是一个日期，当过期之后任何对该 URL 发起的请求都不再从缓存中获得，而要重新访问服务器。一个 Expires 头如下：

```
Expires: Mon, 28 Jul 2014 23:30:00 GMT
```

这个特殊的 Expires 头告诉浏览器缓存此响应报文直到 2014 年 7 月 28 日。这就是所谓的遥远未来 Expires 头，用于缓存那些永不改变的内容，如图片和静态数据集。

Expires 头中的日期是 GMT 日期，它在 PHP 中使用如下代码设置：

```
$lifetime = 7 * 24 * 60 * 60; // 7 天，秒数
header('Expires:'.gmdate('D, d M Y H:i:s', time() + $lifetime) . ' GMT');
```

以上代码告诉浏览器缓存此数据 7 天。要设置一个遥远未来 Expires 头，可以将它的生命期设得更长。下面的例子告诉浏览器缓存文件 10 年：

```
$lifetime = 10 * 365 * 24 * 60 * 60;      // 10 年，秒数
header('Expires:'.gmdate('D, d M Y H:i:s', time() + $lifetime).'GMT');
```

一个 Expires 头是确保浏览器缓存 Ajax 响应报文最简单的方法。不需要改变客户端的任何代码，可继续正常地使用 Ajax 请求，确信只有当文件不在缓存之中时浏览器才将请求发送给服务器。这在服务器端也很容易实现，所有的语言都允许通过某种方法设置信息头。这是保证数据被缓存的最简单方法。

建议 148：使用本地存储数据

除了依赖浏览器处理缓存之外，还可以用手工方法直接存储那些从服务器收到的响应报文。可将响应报文存放在一个对象中，以 URL 为键值对它进行索引。以下是一个 XHR 封装，它首先检查一个 URL 此前是否被取用过。

```
var localCache = {};
function xhrRequest(url, callback) {
    if(localCache[url]) {
        callback.success(localCache[url]);
        return;
    }
    var req = createXhrObject();
    req.onerror = function() {
        callback.error();
    };
    req.onreadystatechange = function() {
        if(req.readyState == 4) {
```

```
            if(req.responseText ===''|| req.status == '404') {
                callback.error();
                return;
            }
            localCache[url] = req.responseText;
            callback.success(req.responseText);
        }
    };
    req.open("GET", url, true);
    req.send(null);
}
```

当然，设置一个 Expires 头是更好的解决方案，实现起来比较容易，而且其缓存内容可以跨页面或跨对话。而一个手工缓存可以利用程序废止缓存内容并获取新的数据。设想一种情况，为每个请求缓存数据，用户可能触发某些动作导致一个或多个响应报文作废。在这种情况下从缓存中删除报文十分简单：

```
delete localCache['/user/friendlist/'];
delete localCache['/user/contactlist/'];
```

本地缓存也可很好地工作于移动设备上。此类设备上的浏览器缓存小或根本不存在，手工缓存成为避免不必要请求的最佳选择。

值得注意的是，在大部分 XHR 技术中要用到流功能。通过监听 readyState=3，可以在一个大的响应报文没有完全接收之前就开始解析它，这时可以实时处理报文片断。这也是 MXHR 能够大幅度提高性能的原因之一。不过大多数 JavaScript 库不允许直接访问 readystatechange 事件，这意味着必须等待整个响应报文接收完才能使用它。

直接使用 XMLHttpRequest 对象并非像看起来那么不好。除一些个别行为之外，所有主流浏览器的最新版本均以同样方式支持 XMLHttpRequest 对象，均可访问不同的 readyState。要支持老版本的 IE，只需要多加几行代码。下面例子中的函数返回一个 XHR 对象，可以直接调用这个对象。

```
function createXhrObject() {
    var msxml_progid = ['MSXML2.XMLHTTP.6.0',
    'MSXML3.XMLHTTP',
    'Microsoft.XMLHTTP',
    'MSXML2.XMLHTTP.3.0'];
    var req;
    try {
        req = new XMLHttpRequest();
    } catch(e) {
        for(var i = 0, len = msxml_progid.length; i < len; ++i) {
            try {
                req = new ActiveXObject(msxml_progid[i]);
                break;
            } catch(e2) {
            }
```

```
            }
        } finally {
            return req;
        }
    }
```

此函数首先尝试支持 readyState =3 的 XMLHttpRequest，然后回落到那些不支持此状态的版本中。直接操作 XHR 对象减少了函数开销，进一步提高了性能。只是放弃使用 Ajax 库，可能会在极少数的浏览器上遇到一些问题。

建议 149：警惕基于 DOM 的跨域侵入

随着 Web 2.0 的发展及 Ajax 框架的普及，富客户端 Web 应用（Rich Internet Application，RIA）日益增多，越来越多的逻辑已经开始从服务器端转移至客户端，这些逻辑通常都是使用 JavaScript 语言编写的。但遗憾的是，目前开发人员普遍不太关注 JavaScript 代码的安全性。据 IBM X-Force 2011 年中期趋势报告揭示，在世界 500 强的网站及常见知名网站中有 40% 存在 JavaScript 安全漏洞。

客户端 JavaScript 安全漏洞与服务器端安全漏洞原理略为不同，自动化检测 JavsScript 安全漏洞目前存在较大的技术难题，不过利用 IBM Rational AppScan Standard Edition V8.0 新特性（JavaScript Security Analyzer，JSA）技术来自动检测 JavaScript 安全漏洞还是一个不错的选择。

2010 年 12 月，IBM 公司发布了关于 Web 应用中客户端 JavaScript 安全漏洞的白皮书，其中介绍了 IBM 安全研究机构曾做过的 JavaScript 安全状况调查。调查中，样本数据包括 675 家网站，其中有财富 500 强公司的网站以及另外 175 家著名网站，包括 IT 公司、Web 应用安全服务公司、社交网站等。为了不影响这些网站的正常运行，研究人员使用了非侵入式爬虫，仅扫描了无须登录即可访问的部分页面，每个站点被扫描的页面不超过 200 个，这些页面都被保存下来。研究人员采用 IBM 公司的 JavaScript 安全分析技术离线分析了这些页面，集中分析了基于 DOM 的跨域脚本编制及重定向两种漏洞。

测试结果令人吃惊，这些知名网站中有 14% 存在严峻的 JavaScript 安全问题，黑客可以利用这些漏洞植入流氓软件、植入钓鱼站点，以及劫持用户会话等。更令人吃惊的是，随着 IBM 公司的 JavaScript 安全分析技术的不断发展，IBM 公司重新测试了上述这些知名网站，2011 年中期的 X-Force 报告显示发现了更多的安全漏洞，大约有 40% 的网站存在 JavaScript 安全漏洞。

1. 基于 DOM 的跨域脚本编制

XSS（Cross Site Script，跨域脚本编制，也称为跨站脚本攻击）就是表示攻击者向合法的 Web 页面中插入恶意脚本代码（通常是 HTML 代码和 JavaScript 代码），然后提交请求给

服务器，随即向服务器响应页面植入攻击者的恶意脚本代码，攻击者可以利用这些恶意脚本代码进行会话劫持等攻击。

跨域脚本编制通常分为反射型和持久型，下面进行简单说明。

- 反射型：请求数据在服务器响应页面中呈现为未编码和未过滤。
- 持久型：包含恶意代码的请求数据被保存在 Web 应用的服务器上，每次用户访问某个页面时恶意代码都会被自动执行。这种攻击对于 Web 2.0 类型的社交网站来说尤为常见，威胁也更大。

应对跨域脚本编制的主要方法有两个：

- 不要信任用户的任何输入，尽量采用白名单技术来验证输入参数。
- 输出的时候对用户提供的内容进行转义处理。

鲜为人知的是，还有第三种跨域脚本编制漏洞。2005 年，Amit Klein 发表了白皮书“基于 DOM 的跨域脚本编制——第三类跨域脚本编制形式”（DOM Based Cross Site Scripting or XSS of the Third Kind），其中揭示了基于 DOM 的跨域脚本编制不需要依赖服务器端响应的内容。如果某些 HTML 页面使用了 document.location、document.URL 或者 document.referer 等 DOM 元素的属性，那么攻击者可以利用这些属性植入恶意脚本实施基于 DOM 的跨域脚本编制攻击。

存在 DOM based XSS 的 HTML 代码如下：

```
<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
Welcome to our system
</HTML>
```

按照该页面 JavaScript 代码的逻辑，上面代码接受 URL 中传入的 name 参数并展示欢迎信息。正常情况下的访问 URL 如下：

```
http://www.mysite/welcome.html?name=Jeremy
```

如果恶意攻击者输入类似如下的脚本，那么该页面会执行被注入的 JavaScript 脚本。

```
http://www. mysite/welcome.html?name=<script>alert(document.cookie)</script>
```

很明显，当受害者的浏览器访问以上 URL 时，服务器端会像在正常情况下一样返回 HTML 页面，然后浏览器会继续将这个 HTML 解析成 DOM，DOM 中包含 document 对象的 URL 属性将包含 <script>alert(document.cookie)</script> 注入的脚本内容，当浏览器解析到 JavaScript 时会执行这段被注入的脚本，导致跨站点脚本编制攻击成功实施。

值得关注的是，通过以上示例可以看出，恶意代码不必要嵌入服务器的响应中，基于 DOM 的跨站点脚本编制攻击也能成功实施。

可能部分读者会认为：目前主流浏览器会自动转义 URL 中的“<”和“>”符号，转义后的注入脚本就不会被执行了，基于 DOM 的跨站点脚本编制也就不再有什么威胁了。这句话前一半是对的，后一半就不准确了；因为攻击者可以很轻松地绕过浏览器对 URL 的转义。例如，攻击者可以利用锚点“#”来欺骗浏览器，浏览器会认为“#”后面的都是片段信息，将不做任何处理。

```
http://www. mysite/welcome.html#?name=<script>alert(document.cookie)</script>
```

2. 通过 URL 重定向钓鱼

网络钓鱼是一个通称，它表示通过欺骗的手段窃取用户私人信息。通过 URL 重定向钓鱼指的是 Web 页面采用 HTTP 参数来保存 URL 值，并且 Web 页面的脚本会将请求重定向到该保存的 URL 上，攻击者可以将 HTTP 参数中的 URL 值改为指向恶意站点，从而顺利启用网络钓鱼来欺骗当前用户并窃取用户凭证。

下面代码给出了较为常见的含有通过 URL 重定向钓鱼漏洞的行为方式。

```
<SCRIPT>
 var sData = document.location.search.substring(1);
 var sPos = sData.indexOf("url=") + 4;
 var ePos = sData.indexOf("&", sPos);
 var newURL;
 if (ePos< 0) {
     newURL = sData.substring(sPos);
 } else {
     newURL = sData.substring(sPos, ePos);
 }
 window.location.href = newURL;
</SCRIPT>
```

可以看出，这些 JavaScript 脚本负责执行重定向，新地址是从 document.location、document.URL 或 document.referer 等 DOM 元素的属性值中截取出来的。例如，类似下面的执行重定向的 URL：

```
http://www.vulnerable.site/redirect.html?url=http://www.phishing.site
```

显然，一旦执行了上面的 URL，将重定向到钓鱼网站。这个漏洞的原理很简单，比服务器端的重定向漏洞更好理解。但在通过 URL 重定向钓鱼的情况下，钓鱼站点的网址并不会被服务端拦截和过滤，因此，这个漏洞往往比服务器端重定向漏洞更具有隐蔽性。

3. 客户端 JavaScript Cookie 引用

Cookie 通常由 Web 服务器创建并存储在客户端浏览器中，用来在客户端保存用户的身份标识、Session 信息，甚至授权信息等。客户端 JavaScript 代码可以操作 Cookie 数据。如

果在客户端使用 JavaScript 创建或修改站点的 cookie，那么攻击者就可以查看到这些代码。通过阅读代码了解其逻辑，甚至可以根据自己所了解的知识来修改 cookie。一旦 cookie 包含了很重要的信息，如包含了权限信息等，攻击者很容易利用这些漏洞进行特权升级等攻击。

4．JavaScript 劫持

利用 JSON 作为 Ajax 的数据传输机制的 Web 应用程序通常都容易受到 JavaScript 劫持攻击，传统的 Web 应用程序反而不易受攻击。

JSON 实际上就是一段 JavaScript，通常是数组格式。攻击者在其恶意站点的页面中通过 <SCRIPT> 标签调用被攻击站点的一个 JSON 动态数据接口，并通过 JavaScript Function Hook 等技术取得这些 JSON 数据。如果用户在登录被攻击网站后（假定其身份认证信息是基于 Session Cookie 来保存的）又被攻击者诱引访问了恶意站点页面，那么，恶意站点会发送 JSON 数据获取请求至被攻击站点，被攻击站点服务器会认为当前请求是合法的，并向恶意站点返回当前用户的相关 JSON 数据，从而导致用户数据泄密，这是因为 <SCRIPTsrc=""> 这种标签的请求会带上 Cookie 信息。整个过程相当于一个站外类型的跨站点请求伪造（CSRF）攻击。

随着 Ajax 的进一步推广，以及 HTML 5 的逐步应用，还会有更多的客户端安全漏洞出现。目前对于 JavaScript 的安全研究尚不多，新推出的 HTML 5 客户端存储、跨域通信等新特型也都跟安全紧密相关，有兴趣的读者可以进一步研究。

建议 150：优化 Ajax 开发的最佳实践

优化 Ajax 开发的最佳实践，有助于编写更加高效且健壮的 Ajax 代码。

（1）最小化调用

最小化调用数量的方法之一是将大量调用合并成少量调用。如果数据量相对较小，那么在大多数网络中，主要问题就在于延迟。延迟是浏览器真正获取服务器之间的连接所需的时间，有时它会占去大部分连接时间。用户所感受到的总延迟由几个部分组成，包括浏览器的缓存设置、DNS 客户端，以及物理连接。

没有简易公式或代码片段供我们来了解如何减少 Web 应用程序调用。然而，只需一个简单的练习，就可以演示如何对从客户端到服务器的 Ajax 调用数量进行控制。考虑下面购买二手摩托车的 Web 应用程序。

首先，选择摩托车的年份，然后选择摩托车的构造，最后选择摩托车的型号。自始至终，Ajax 一直在后台运行，更新 Web 应用程序中的下拉框来为用户过滤清单，以方便用户选择。

要开始这一练习，首先要为客户端及服务器创建一个简单图表（有一个文本框），然后为浏览器进行的 Ajax 调用画线，以从服务器获取用户数据。

可将对品牌和型号进行的调用合并到一个调用中，从而实现优化设计。不是对品牌进

行一次调用，然后针对型号进行另一次调用，而是对型号进行缓存，这样，当用户选择品牌时，新代码只检索缓存中可用的型号列表。从本地缓存中获取数据要比从服务器获取相同数据快得多。回避额外的服务调用，避免服务调用的延迟。

新设计在浏览器与服务器之间的通信中去掉了一个调用。可利用下面代码进一步减少调用数量，其中的一些关键行可用于存储在数组中检索到的数据，供以后查找使用。

```
var choices = new Array();
function fillChoiceBoxes(year) {
    if(dojo.indexOf(choices, year) == -1) {
        // 开始
    } else {
        choices[year] = result;
        // Ajax 调用结果
    }
    // 调用函数
    fillSelect(dojo.byId('makes'), choices[year]);
}
```

在反复考虑两个不同的型号时，Web 应用程序会使用本地缓存数据，而不是发起附加服务调用。仅缓存静态数据，至少在用户会话持续阶段是这样的。不要因为缓存了不应缓存的数据，而引起一系列问题。通过减少客户端与服务器之间的交互次数，以及在可能的情况下缓存数据，可以最小化调用次数。

（2）让数组变得很小

为提高数据处理性能，需要让服务器与客户端之间传输的数据尽量的小。为了高效地完成这一任务，必须控制从服务层到能够指定从服务器到客户端的消息类型的部分。有充足的理由证明，XML 适合作为客户端到服务器的通用消息格式。理由之一是存在足够多的库或框架用于 XML 序列化。

然而，当与 JSON 对比时，XML 显得很冗长，JSON 则更加简明。目前已经有很多可以将消息方便地构建成 JSON 格式的库，这样就可以通过 JSON 的方式将数据从服务端传送到客户端。

如果服务器响应使用 JSON，那么可通过提供一个参数来使用相同的客户端对象。仔细研究下面代码，其展示了使用 XML 的摩托车对象的表示。

```
<motorcycle>
    <year>2012</year>
    <make>Moto</make>
    <model>Uberfast</model>
</motorcycle>
```

下面代码展示了使用 JSON 的摩托车对象。注意，它的代码量减少了大约 25%（如果去掉空格）。

```
{ "motorcycle" : {
```

```
        "year" : "2012",
        "make" : "Moto",
        "model" : "Uberfast"
        }
}
```

数据量变小了，不但从服务端到客户端的传输时间减少了，而且字符串的减小还节省了解析时间。在设计需要传输的数据时，其所包含的字符越少越好。

（3）预加载组件

可通过在 Ajax 调用中加载 JavaScript 文件与图像之类的组件来充分利用浏览器的缓存。需要注意的是，预加载 JavaScript 文件和图像，仅对那些开启缓存功能的用户有益，不过大多数用户的浏览器都开启了缓存功能。

要预加载外部 JavaScript 文件，可以将 JavaScript 文件包含在页面中，但是，只有当该页面很小且仅想优化少量资源时，才适合采用这一方式。例如，对于一个将工作流引入用户的相对轻量级的页面，预加载非常有用。考虑介绍最小化调用时的购买摩托车的例子，可在流的早期页面中预加载用于包含下拉框的页面的、包含全部 Ajax 代码的 JavaScript 代码。

在使用 Ajax 调用的方法更新图片时，预加载图像会提供很大便利。在预加载图像后，当用户将鼠标移动到元素时、从下拉框中进行选择时，或者单击按钮时，不必等待浏览器对图像进行检索。即使 Ajax 以异步方式发生，也需要花费一些时间将图像从服务器传送到客户端，并且图像在全部下载完毕之前不会在客户端中显示。例如，在下面示例中，当用户进行从清单中选择摩托车这一操作时，所采用的图像就是使用标准 JavaScript 代码预加载的。

```
<html>
<head><title> </title></head>
<body>
<script type="text/Javascript" language="Javascript">
    var img = new Image();
    img.src = "http://mysite/motocool.jpg";
</script>
</body>
</html>
```

在为页面预加载图像时，JavaScript 的位置很重要，当然不希望因为在 HTML 中加入了 JavaScript 代码而影响页面的加载速度。一般的规则是，可将 <script> 元素当中的 JavaScript 代码放到 HTML 页面的最后部分，因为在考虑可同时下载多少资源时会发现浏览器的能力相对有限。如果可能，将脚本加到 HTML 页面的最后部分，从而帮助浏览器更快速地加载图像和其他资源。

在 HTML 5 中，可使用 <script> 标记的新 async 属性。这将告诉浏览器可以异步运行 JavaScript 代码，这样，JavaScript 代码可以在页面中运行其他东西时执行。

（4）轻松处理错误

在 JavaScript 代码中定义的每个函数，都要假设会有恶意输入发生，因为防御性能强的

代码比使用 try catch 语句所编写的代码更善于处理错误。例如，想使用 JavaScript 函数来根据用户输入进行计算，要在计算前检查输入。

```
function caculateDistance(source,dest) {
    if (!isNaN(source)||!isNaN(dest)) {
        dojo.byId("errors").innerHTML = "提示错误";
    }
}
```

即使代码具有防御能力，在适当的时候，也可使用 try catch 语句与错误回调。在下面示例中，JavaScript 使用 try catch 语句来捕获错误。

```
function calculateDistance(source,dest) {
    try {
        // 执行计算...
    } catch (error) {
        dojo.byId("errors").innerHTML = "提示错误";
    }
}
```

下面示例演示的是在调用 Dojo Toolkit 中所提供的 xhrGet() 方法时对错误回调的使用。错误参数是可选的，因此可以很容易地跳过错误处理器的定义。

```
var args = {
    url: "/js/dojo//NoSuchFile",
    handleAs: "text",
    preventCache: true,
    load: function(data) {
        // 当加载成功时，执行计算
    },
    error: function(error) {
        dojo.byId("errors").innerHTML = "提示错误";
    }
}
var ajx = dojo.xhrGet(args);
```

如何处理页面上的错误，这既是个业务问题，又是个技术问题。适当的时候，客户能够提供在出现异常时有效的默认处理方式。

最后，不要在 JavaScript 提示对话框中显示错误描述。用户不是软件工程师，因此，这类提示信息对于用户来说没有任何意义。除了不要为用户提供无意义的信息之外，还要在提示对话框中要求客户取消该对话框，以返回页面。

```
function calculateDistance(source, dest) {
    try {
        // 执行计算
    } catch (error) {
        // 不要：
        // alert(error.message);
        // 最好：
```

```
            dojo.byId("errors").innerHTML = "提示错误";
        }
    }
```

（5）使用现有工具

通过使用现有工具（框架与平台），可有效利用相应资源。大多数成熟的技术人员，会使用已在多个平台上测试过的，具有跨浏览器兼容性的工具。现有工具的大部分特性可用于部署到自己的项目中。

很多现有的优秀工具，除了支持 Ajax 调用之外，还能支持很多其他函数与特性，如动画等。jQuery 是比较优秀的 JavaScript 库，它能提供全套的 Ajax 功能。jQuery 还支持不同的消息格式及其他基于 Ajax 的方法，如 getScript()，它可用于下载并执行 JavaScript 文件，是预载组件最佳实践的起源。

建议 151：数据存储要考虑访问速度

计算机科学的一个经典问题是确定数据应当存放在什么地方，以实现最佳的读写效率。数据存储在哪里，关系到代码运行期间数据被检索到的速度。在 JavaScript 中，此问题相对简单，因为进行数据存储只有少量方式可供选择。正如其他语言那样，数据存储位置关系到访问速度。JavaScript 中有 4 种基本的数据存储位置，下面分别进行介绍。

- 直接量：仅仅代表自己，而不存储于特定位置。JavaScript 的直接量包括字符串、数字、布尔值、对象、数组、函数、正则表达式、具有特殊意义的空值（null）和未定义值（undefined）。
- 变量：使用 var 关键字创建用于存储的数据值。
- 数组项：具有数字索引，存储一个 JavaScript 数组对象。
- 对象成员：具有字符串索引，存储一个 JavaScript 对象。

每一种数据存储位置都具有特定的读写操作负担。在大多数情况下，对一个直接量和一个局部变量数据进行访问的性能差异是微不足道的。访问数组项和对象成员的代价要高一些，具体高多少，很大程度上取决于浏览器。

早期版本的浏览器使用传统的 JavaScript 引擎，如 Firefox 3、IE 6 和 Safari 3.2，它们比优化后的 JavaScript 引擎耗费更多时间。总的来说，直接量和局部变量的访问速度要快于数组项和对象成员的访问速度。因此，如果关心运行速度，那么尽量使用直接量和局部变量，限制数组项和对象成员的使用。

在 JavaScript 中，数据存储位置可以对代码整体性能产生重要影响。数据访问类型有 4 种：直接量、变量、数组项、对象成员。它们有不同的性能考虑。

- 直接量和局部变量访问速度非常快，而数组项和对象成员需要更长时间。
- 局部变量的访问速度比域外变量快，因为它位于作用域链的第一个对象中。变量在作

用域链中的位置越深，访问所需的时间就越长。全局变量总是最慢的，因为它们总是位于作用域链的最后一环上。

- 避免使用 with 表达式，因为它改变了运行期上下文的作用域链。还应当小心对待 try catch 表达式的 catch 子句，因为 catch 子句具有同样效果。
- 嵌套对象成员会造成重大性能影响，尽量少用。
- 一个属性或方法在原型链中的位置越深，访问它的速度就越慢。

一般来说，可以通过这种方法提高 JavaScript 代码的性能：将经常使用的对象成员、数组项和域外变量存入局部变量中。然后，访问局部变量的速度会快于那些原始变量。通过使用这些策略，可以极大地提高那些需要大量 JavaScript 代码的网页应用的实际性能。

建议 152：使用局部变量存储数据

不论从性能的角度来看，还是从功能的角度来看，作用域概念都是理解 JavaScript 的关键。作用域对 JavaScript 有许多影响，从确定哪些变量可以被函数访问，到确定 this 的值。JavaScript 作用域也关系到性能，但要理解速度与作用域的关系，首先要理解作用域的工作原理。

（1）作用域链和标识符解析

每一个 JavaScript 函数都被表示为对象。函数对象如其他对象一样，拥有可以编程访问的属性和一系列不能被程序访问仅供 JavaScript 引擎使用的内部属性。其中一个内部属性是 scope，由 ECMA-262 标准第三版定义。

内部属性 scope 包含一个函数被创建的作用域中对象的集合。此集合被称为函数的作用域链，它决定哪些数据可由函数访问。此函数作用域链中的每个对象被称为一个可变对象，每个可变对象都以“键－值对”的形式存在。在一个函数被创建后，它的作用域链被填充以对象，这些对象代表创建此函数的环境中可访问的数据。例如，下面这个全局函数：

```
function add(num1, num2){
    var sum = num1 + num2;
    return sum;
}
```

在 add() 函数被创建后，它的作用域链中填入一个单独的可变对象，此全局对象代表了所有在全局范围内定义的变量。此全局对象包含诸如窗口、浏览器和文档之类的访问接口。Add() 函数的作用域链将会在运行时用到。例如，下面的代码：

```
var total = add(5, 10);
```

在运行 add() 函数时将建立一个内部对象，称之为“运行期上下文”。一个运行期上下文定义了一个函数运行时的环境。对函数的每次运行而言，由于每个运行期上下文都是独一的，所以多次调用同一个函数就会导致多次创建运行期上下文。在函数执行完毕时运行期上

下文就被销毁。

一个运行期上下文有它自己的作用域链，用于标识符解析。当运行期上下文被创建时，它的作用域链被初始化，连同运行函数的 scope 属性中所包含的对象也被初始化。scope 属性值按照它们出现在函数中的顺序，被复制到运行期上下文的作用域链中。这项工作一旦完成，一个被称做“激活对象”的新对象就为运行期上下文创建好了。此激活对象作为函数执行期的一个可变对象，包含访问所有局部变量、命名参数、参数集合和 this 的接口。然后，此对象被推入作用域链的前端。当作用域链被销毁时，激活对象也一同被销毁。

在函数运行过程中，每遇到一个变量，标识符识别过程要决定从哪里获得或存储数据。在此过程中搜索运行期上下文的作用域链，查找同名的标识符。搜索工作从运行函数的激活目标的作用域链的前端开始。如果找到了，那么就使用这个具有指定标识符的变量；如果没找到，那么搜索工作将进入作用域链的下一个对象。此过程持续运行，直到标识符被找到，或者没有更多对象可以搜索（在这种情况下标识符将被认为是未定义的）。在函数运行时每个标识符都要经过这样的搜索过程。例如，在上面示例中，函数访问 sum、num1、num2 时都会产生这样的搜索过程。

（2）标识符识别性能

标识符识别不是“免费”的。事实上没有哪种计算机操作可以不产生性能开销。在运行期上下文的作用域链中，一个标识符所处的位置越深，它的读写速度就越慢。因此，在函数中，局部变量的访问速度总是最快的，全局变量通常是最慢的。全局变量总是处于运行期上下文作用域链的最后一个位置上，总是最后才被访问到。

总的趋势是，对所有浏览器来说，一个标识符所处的位置越深，读写它的速度就越慢。采用了优化的 JavaScript 引擎的浏览器，如 Safari 4，在访问域外标识符时没有这种性能损失，而 IE 和其他浏览器则有较大幅度的性能损失。值得注意的是，早期浏览器（如 IE 6 和 Firefox 2）将会耗费更多的时间访问域外变量。

通过以上内容可以了解，在没有优化 JavaScript 引擎的浏览器中，最好尽可能使用局部变量。一个好的经验法则：用局部变量存储本地范围之外的变量值，如果它们在函数中的使用多于一次。

```
    function initUI() {
        var bd = document.body, links = document.getElementsByTagName_r("a"), i = 0,
len = links.length;
        while(i < len) {
            update(links[i++]);
        }
        document.getElementById("go-btn").onclick = function() {
            start();
        };
        bd.className = "active";
    }
```

上面示例中的函数包含 3 个对 document 的引用。document 是一个全局对象，搜索此变

量，必须遍历整个作用域链，直到在全局变量对象中找到它。可以通过这种方法减轻重复的全局变量访问对性能的影响：首先将全局变量的引用存储在一个局部变量中，然后使用这个局部变量代替全局变量。例如，上面的代码可以这样重写：

```
function initUI() {
    var doc = document, bd = doc.body, links = doc.getElementsByTagName_r("a"), i
= 0, len = links.length;
    while(i < len) {
        update(links[i++]);
    }
    doc.getElementById("go-btn").onclick = function() {
        start();
    };
    bd.className = "active";
}
```

在上面代码中，首先将 document 的引用存入局部变量 doc 中。现在访问全局变量的次数是 1 次，而不是 3 次。用 doc 替代 document 更快，因为它是一个局部变量。当然，因为数量的原因，这个简单的函数不会显示出巨大的性能改进，不过可以想象一下，如果几十个全局变量被反复访问，那么性能改进显然会很明显。

建议 153：警惕人为改变作用域链

一般来说，一个运行期上下文的作用域链不会被改变，但是，有两种表达式可以在运行时临时改变运行期上下文作用域链。

（1）with

with 表达式为所有对象属性创建一个默认操作变量。在其他语言中，类似的功能通常用来避免书写一些重复的代码。initUI() 函数可以重写成如下形式：

```
function initUI() {
    with(document) {
        var bd = body, links = getElementsByTagName_r("a"), i = 0, len = links.length;
        while(i < len) {
            update(links[i++]);
        }
        getElementById("go-btn").onclick = function() {
            start();
        };
        bd.className = "active";
    }
}
```

在上面代码中，重写的 initUI() 函数使用了一个 with 表达式，避免了多次书写 document。这样似乎更有效率，实际上却产生了一个性能问题。

当代码流执行到一个 with 表达式时，运行期上下文的作用域链被临时改变了。一个新的

可变对象将被创建，它包含指定对象的所有属性。此对象被插入到作用域链的前端，这意味着现在函数的所有局部变量都被推入第二个作用域链对象中，所以访问代价更高了。

通过将 document 对象传递给 with 表达式，一个新的可变对象容纳了 document 对象的所有属性，被插入到作用域链的前端。这使得访问 document 对象的速度非常快，但访问局部变量的速度却变慢了，如 bd 变量。正因为如此，最好不要使用 with 表达式，只要简单地将 document 存储在一个局部变量中，就可以获得性能上的提升。

（2）catch

在 JavaScript 中，不只是 with 表达式人为地改变运行期上下文的作用域链，try catch 表达式的 catch 子句也具有相同效果。当 try 块发生错误时，程序流程自动转入 catch 块，并将异常对象推入作用域链前端的一个可变对象中。在 catch 块中，函数的所有局部变量现在被放在第二个作用域链对象中，例如：

```
try {
        methodThatMightCauseAnError();
} catch (ex) {
        alert(ex.message);
}
```

注意，只要 catch 子句执行完毕，作用域链就会返回到原来的状态。如果使用得当，那么 try catch 表达式将是非常有用的语句，所以不建议完全避免使用 try catch 语句。如果计划使用一个 try catch 语句，那么一定要确保了解可能发生的错误。一个 try catch 语句不应该作为解决 JavaScript 错误的办法。如果一个错误会经常发生，那说明应当修正代码本身的问题。

可以通过精减代码的办法将 catch 子句对性能的影响降至最低。一个好的模式是将错误交给一个专用函数来处理，例如：

```
try {
        methodThatMightCauseAnError();
} catch (ex) {
        handleError(ex);
}
```

handleError() 函数是 catch 子句中运行的唯一代码。此函数以适当方法自由地处理错误，并接收由错误产生的异常对象。由于只有一条语句，没有局部变量访问，作用域链临时改变就不会影响代码的性能。

建议 154：慎重使用动态作用域

无论是 with 表达式，还是 try catch 表达式的 catch 子句，抑或是包含 () 的函数，都被认为是动态作用域。一个动态作用域只因代码运行而存在，因此无法通过静态分析（代码结构）来确定是否存在动态作用域，例如：

```
function execute(code) {(code);
    function subroutine() {
        return window;
    }
    var w = subroutine();
};
```

execute() 函数看上去像一个动态作用域，因为它使用了 ()。w 变量的值与 code 有关。在大多数情况下，w 将等价于全局的 window 对象，但下列情况除外。

```
execute("var window = {};")
```

在这种情况下，由于 () 在 execute() 函数中创建了一个局部 window 变量，所以 w 将等价于这个局部总的 window 变量而不等价全局中的那个 window 变量。所以说，不运行这段代码是没有办法了解具体情况的，因为标识符 window 的确切含义不能预先确定。

优化的 JavaScript 引擎，如 Safari 的 Nitro 引擎，企图通过分析代码来确定哪些变量应该在任意时刻被访问，以加快标识符识别过程。这些引擎企图避开传统作用域链查找，以标识符索引的方式进行快速查找。在涉及一个动态作用域后，此优化方法就不起作用了。正是这个原因，只在绝对必要时才推荐使用动态作用域。

建议 155：小心闭包导致内存泄漏

闭包是 JavaScript 最强大的一个方面，它允许函数访问局部范围之外的数据。在复杂的网页应用中闭包无处不在。有一种性能影响与闭包有关。为了分析与闭包有关的性能问题，考虑下面的例子：

```
function assignEvents() {
    var id = "xdi9592";
    document.getElementById("save-btn").onclick = function(event) {
        saveDocument(id);
    };
}
```

assignEvents() 函数为一个 DOM 元素指定了一个事件处理句柄。此事件处理句柄是一个闭包，当 assignEvents() 被执行时，可以访问其范围内部的 id 变量。要用这种方法封闭对 id 变量的访问，必须创建一个特定的作用域链。

当 assignEvents() 被执行时，一个激活对象被创建，其中包含了一些应有的内容，比如 id 变量，它将成为运行期上下文作用域链上的第一个对象，全局对象是第二个。当闭包被创建时，scope 属性与这些对象一起被初始化。

由于闭包的 scope 属性包含与运行期上下文作用域链相同的对象引用，因此会产生副作用。通常，一个函数的激活对象与运行期上下文一同被销毁。当涉及闭包时，激活对象就无法被销毁了，因为引用仍然存在于闭包的 scope 属性中。这意味着与脚本中的非闭包函数相

比，闭包需要更多内存开销。在大型网页应用中，这可能是个问题，尤其在 IE 中更被关注。IE 使用非本地 JavaScript 对象实现 DOM 对象，闭包可能导致内存泄漏。

当闭包被执行时，一个运行期上下文将被创建，它的作用域链与在 scope 中引用的两个相同的作用域链同时被初始化，然后一个新的激活对象作为闭包自身被创建。

注意，在闭包中使用的两个标识符：id 和 saveDocument，它们存在于作用域链第一个对象之后的位置上。这是闭包最主要的性能关注点：经常访问一些范围之外的标识符，每次访问都导致一些性能损失。

在脚本中最好小心地使用闭包，内存和运行速度都值得去关注。但是，可以将常用的域外变量存入局部变量中，然后直接访问局部变量，这样能够减小对运行速度的影响。

建议 156：灵活使用 Cookie 存储长信息

Cookie 适合在客户端存储用户的简单个人信息，因为 Cookie 既不是通用的通信机制，也不是通用的数据传输机制。从客观角度分析，Web 浏览器存储的 Cookie 总数不能够超过 300 个。如果在同一个域中存储 Cookie 信息，那么 Cookie 个数不能够超过 20，并且每个 Cookie 字符串长度不能够超过 4KB，即每个 Cookie 文本文件的大小不能够超过 4KB，因此使用 Cookie 存储信息时一定要适度。Cookie 用来存储需要长期保存的大量数据，一般用来保存用户的状态信息和访问足迹等小数据。

由于每个 Web 服务器最多只能够存储 20 个 Cookie，为了避免超出这个限制，可以把多个用户信息保存到一个 Cookie 中，而不是为每个用户信息新建一个 Cookie。由于 Cookie 可存储的字符串最大长度为 4KB（即 4096 个字符），在实际应用中，这个字符串长度完全能够满足各种用户信息的存储。

这里介绍一种方法，即在 Cookie 值中存储一组子名 - 值对。子名 - 值对的形式可以自由约定，确保不引发歧义即可。例如，使用冒号作为子名和子值之间的分隔符，而使用逗号作为子名 - 值对之间的分隔符，这种写法类似于对象直接量：

```
subName1 : subValue1, subName2 : subValue2, subName3 : subValue3
```

然后把这组子名 - 值串作为值传递给 Cookie 的名称，例如：

```
name=subName1:subValue1,subName2:subValue2,subName3:subValue3
```

为了确保子名 - 值串不引发歧义，建议使用 escape() 方法对其进行编码，读取时再使用 unescape() 方法转码即可。下面的示例演示了如何在 Cookie 中存储更多的信息：

```
// 定义有效期
var d = new Date();
d.setMonth(d.getMonth() + 1);
d = d.toGMTString();
// 定义 Cookie 字符串
```

```
var a = "name:a,age:20,addr:beijing"
var c = "user=" + escape(a)
c += ";" + "expires=" + d;
document.Cookie = c; // 写入 Cookie 信息
```

当读取 Cookie 信息时，首先要获取 Cookie 值，然后调用 unescape() 方法对 Cookie 值进行解码，最后再访问 Cookie 值中每个子 Cookie 值。因此，对于 document.Cookie 来说，需要分解 3 次才能得到精确的信息。要获取 Cookie 的函数如下：

```
// 读取所有 Cookie 信息，包括子 Cookie 信息
// 参数：无
// 返回值：对象，存储的是子 Cookie 信息，其中名称作为对象的属性而存在，而值作为属性值存在
function getSubCookie(){
    var a = document.Cookie.split(";");
    var o ={};
    for (var i = 0; i < a.length; i++ ){ // 遍历 Cookie 信息数组
        a[i] && (a[i] = a[i].replace(/^\s+|\s+$/, ""));
        // 清除头部空格符
        var b = a[i].split("=");
        var c = b[1];
        c && (c = c.replace(/^\s+|\s+$/, ""));  // 清除头部空格符
        c = unescape(c);          // 解码 Cookie 值
        if(!/\,/gi.test(c)){
            o[b[0]] = b[1];
        }
        else{
            var d = c.split(",");
            for (var j = 0; j < d.length; j++ ){// 遍历子 Cookie 数组
                var e = d[j].split(":");
                o[e[0]] = e[1];  // 把子 Cookie 信息写入返回对象
            }
        }
    }
    return o; // 返回包含 Cookie 信息的对象
}
```

目前的浏览器都支持 Cookie，不过也有例外。例如，个别老式浏览器不支持 Cookie，或者用户禁止浏览器使用 Cookie。为了安全起见，在使用 Cookie 之前，应该探测客户端是否启用 Cookie，如果没有启用，则可以采取应急措施，避免不必要的损失或导致网站部分功能无法实现。

一般可以使用下面的方法来探测客户端浏览器是否支持 Cookie。

```
if(navigator.CookieEnabled){ /* 如果存在 CookieEnabled 属性，则说明浏览器支持 Cookie，可以安全写入或读取 Cookie 信息 */
    setCookie();
    // 或
    getCookie();
}
```

如果浏览器启用了 Cookie，则 CookieEnabled 属性值为 true；如果浏览器禁用了 Cookie，

则该属性值为 false。

建议 157：推荐封装 Cookie 应用接口

在默认状态下，存取 Cookie 信息是比较麻烦的。由于 Cookie 是通过字符串来存取信息的，所以容易导致在执行赋值运算时需要转换读取信息的数据类型。另外，烦琐的构造和解析 Cookie 信息字符串本身就令人生厌，在经常使用 Cookie 信息的 Web 应用中格外不便。因此，建议读者在进行 Web 开发前，封装 Cookie 以便能够提高开发效率。

Cookie 封装的原则：接口高度统一，参数自由、灵活。Cookie 存取功能比较强大，具体包括写入、读取和删除 Cookie 信息，能够满足日常开发的需要。下面介绍一下 Cookie 封装的具体设计思路。

定义函数 Cookie()，利用该函数既可以写入指定的 Cookie 信息，删除指定的 Cookie 信息，也能够读取指定名称的 Cookie 值，另外，在该函数中还可以指定 Cookie 信息的有效期、有效路径、作用域和安全性选项设置。如果在调用 Cookie() 函数时，仅指定一个参数值，则表示读取指定名称的 Cookie 值；如果指定两个参数，则表示写入 Cookie 信息，其中第一个参数表示名称，第二个参数表示值。在第三个参数中还可以传递选项信息，这些信息以字典形式存储在对象中进行传递。前两个参数以字符串形式进行传递。完整的封装代码如下：

```
// 封装 Cookie 存取功能，可以写入 Cookie 信息，读取 Cookie 信息，也可以删除 Cookie 信息
/* 参数：name 表示 Cookie 的名称，value 表示 Cookie 值，都以字符串形式传递。options 参数是一个对象，该对象可以包含多项信息，用来指定 Cookie 信息的有效期、路径、作用域和完全性设置 */
// 返回值：当仅有一个参数时，该函数获读取并返回 Cookie 值
function Cookie(name, value, options){
    if (typeof value != 'undefined'){ // 如果第二个参数存在
        options = options ||{};
        if (value === null){
            options.expires = - 1;        // 设置失效时间
        }
        var expires = '';
        // 如果存在时间参数项，并且类型为 number，或者具体时间，那么分别设置时间
        if (options.expires && (typeof options.expires == 'number' || options.expires.toUTCString)) {
            var date;
            if (typeof options.expires == 'number'){
                date = new Date();
                date.setTime(date.getTime() + (options.expires * 24 * 60 * 60 * 1000));
            }
            else{
                date = options.expires;
            }
            expires = '; expires=' + date.toUTCString();
        }
        var path = options.path ? '; path=' + options.path : '';  // 设置路径
        var domain = options.domain ? '; domain=' + options.domain : ''; // 设置域
        var secure = options.secure ? '; secure' : ''; // 设置安全措施，为 true 则直接设置，
```

```
否则为空
            // 把所有字符串信息都存入数组，然后调用 join() 方法转换为字符串，并写入 Cookie 信息
            document.Cookie = [name,'=',encodeURIComponent(value),expires,path,domain,
secure].join('');
        }
        else{ // 如果第二个参数不存在，则表示读取指定 Cookie 信息
            var CookieValue = null;
            if (document.Cookie && document.Cookie != ''){
                var Cookie = document.Cookie.split(';');
                for (var i = 0; i < Cookies.length; i ++ ){
                    var Cookie = (Cookies[i] || "").replace( /^\s+|\s+$/g, "" );
                    if (Cookie.substring(0, name.length + 1) == (name + '=')){
                        CookieValue = decodeURIComponent(Cookie.substring
                            (name.length + 1));
                        break;
                    }
                }
            }
            return CookieValue;        // 返回查找的 Cookie 值
        }
    }
```

下面是关于该封装函数的应用示例。

1）写入 Cookie 信息：

```
Cookie("user", "baidu"); // 简单写入一条 Cookie 信息
Cookie("user", " baidu ", {// 写入一条 Cookie 信息，并设置更多选项
                    expires:10,                    // 有效期为 10 天
                    path:"/",                      // 整个站点有效
                    domain:"www.css8.cn",          // 有效域名
                    secure:true                    // 加密数据传输
});
```

2）读取 Cookie 信息：

```
Cookie("user")
```

3）删除 Cookie 信息：

```
Cookie("user",null);
```

通过上面代码，可以实现对 Cookie 信息的读写操作，这样用户就可以方便地在客户端存储简短的页面信息，避免因为在服务器端存储这些信息而增加服务器运行负担和浪费服务器端资源。

第 8 章

JavaScript 引擎与兼容性

JavaScript 兼容性一直是 Web 开发中面临的一个主要问题。在正式规范、实施标准及各种实现之间的现实差异让很多开发者十分迷茫。这种差异的结果就是很多网页不能在各种浏览器上呈现一致的行为或效果，甚至根本不能跨浏览器。

为了实现浏览器解析的一致性，首先需要找出不同引擎的分歧点在哪里。为此，开发人员必须清楚浏览器实现的分歧在哪里，不同 JavaScript 引擎在哪个地方把规范解读错了，或者习惯性定义了哪些怪异的私自标准。本章不仅挖掘 IE 的 JScript 解析的歧义，还囊括了来自 Firefox、Opera 和 Safari 的 JavaScript 实现，逐一比较标准的规定和实际的情形。

建议 158：比较主流浏览器内核解析

浏览器可以分为两部分：外壳（shell）和内核（core），其中外壳的种类相对较多，内核则较少。浏览器的外壳包括菜单、工具栏等，主要向用户提供界面操作、参数设置等。它是调用内核来实现各种功能的。内核才是浏览器的核心。内核是基于标记语言显示内容的程序或模块。也有一些浏览器并不区分外壳和内核。Mozilla 在将 Gecko 独立出来后，才有了外壳和内核的明确划分。目前主流的浏览器有 IE、Firefox、Opera、Safari、Chrome、Netscape 等。

浏览器内核又可以分成两部分：渲染引擎和 JavaScript 引擎。浏览器内核负责取得网页的内容（HTML、XML、图像等）、整理信息（如加入 CSS 等），以及计算网页的显示方式，然后将结果输出至显示器或打印机。由于不同浏览器内核对网页的语法解释会有不同，因此渲染的效果也不相同。所有网页浏览器、电子邮件客户端，以及其他需要编辑、显示网络内容的应用程序都需要内核。JavaScript 引擎则是解析、执行 JavaScript 语言来实现网页的动态效果。最开始渲染引擎和 JavaScript 引擎并没有明确区分，后来 JavaScript 引擎越来越独立，内核就倾向于只指渲染引擎。有一个网页标准计划小组制作了一个 ACID 来测试引擎的兼容

性和其他性能。内核的种类很多，包括免费内核大约有十多种，但常见的浏览器内核可以分为 4 种：Trident、Gecko、Presto、Webkit。

- Trident 又称 MSHTML，是微软公司开发的渲染引擎，包含了 JavaScript 引擎 JScript，它已经深入到 Windows 操作系统中，如 Windows Media Player、Windows Explorer、Outlook Express 等都使用它。目前很多浏览器都使用这个引擎，如 IE、Maxthon（最新版已经不使用）等。
- Gecko 是使用 C++ 语言开发的开源渲染引擎（Mozilla 开源项目），包括了 SpiderMonkey(Rhino)。主要的使用者有 Firefox。
- Presto 是由 Opera 公司开发、用于 Opera 的渲染引擎。Macromedia Dreamweaver（MX 版本及以上）和 Adobe Creative Suite 也使用了 Presto 的内核。
- Webkit 是苹果公司基于 KHTML 开发的，包括 Webcore 和 JavaScriptCore（SquirrelFish、V8）两个引擎。主要的使用者有 Safari 和 Chrome。

下面是主流浏览器所使用的内核的简单分类。

- Trident 内核：IE、Maxthon、腾讯浏览器、The World、360、搜狗浏览器等。
- Gecko 内核：Netscape 6 及以上版本、Firefox、MozillaSuite/SeaMonkey 等。
- Presto 内核：Opera 7 及以上版本。
- Webkit 内核：Safari、Chrome 等。

随着互联网十多年的高速发展，近几年市场上也推出了很多新的浏览器，但由于它们采用的并不是自主开发的内核，所以浏览器内核本身没有实质上的突破。下面对主流浏览器内核的优缺点进行比较。

- Trident：这种浏览器内核是 IE 使用的内核，因为在早期 IE 占有大量的市场份额，所以这种内核比较流行，以前有很多网页都是根据这个内核的标准来编写的。实际上这个内核对真正的网页标准支持得不是很好，甚至在 2005 年，与网页标准制定组织（W3C）所制定的标准发生了脱节。同时 Trident 内核本身的 Bug 比较多，对一些符合 W3C 标准的网页代码支持得不是很好，这在早期的 IE 版本中比较明显。微软公司从 IE 9 版本开始全面升级了 IE 内核，使其完全符合标准设计要求。
- Gecko：这是 Firefox 和 Flock 所采用的内核，这个内核的优点就是功能强大、丰富，可以支持很多复杂网页效果和浏览器扩展接口，但会消耗很多的资源，如内存。
- Presto：这是 Opera 采用的内核。Presto 内核被公认为目前浏览网页速度最快的内核，这得益于它在开发时的优势，在处理 JavaScript 等脚本语言时，会比其他的内核快 3 倍左右，缺点就是为了达到很快的速度而丢掉了一部分网页兼容性。
- Webkit：这是 Safari 采用的内核，属于苹果系统下的浏览器。优点是网页浏览速度较快，虽然不及 Presto 但也胜于 Gecko 和 Trident；缺点是对于网页代码的容错性不高，也就是说，对网页代码的兼容性较低，会使一些编写不标准的网页无法正确显示。

不同 JavaScript 引擎在解析相同 JavaScript 代码时的实现逻辑和算法可能存在分歧，当

然运行的结果也会不同，最明显的就是 DOM 实现部分。常用 JavaScript 引擎说明见表 8.1。

表 8.1 主流浏览器的 JavaScript 引擎

JavaScript 引擎	应用浏览器
JScript	IE
SpiderMonkey	Firefox 3.0 及其以下版本
TraceMonkey	Firefox 3.1 及其以上版本
JavaScriptCore	Safari 3.1 及其以下版本
SquirrelFish	Safari 4.0
Futhark	Opera 9.5 及其以上版本
V8	Chrome

建议 159：推荐根据浏览器特性进行检测

不同浏览器对 JavaScript 的兼容性是不同的，尤其是 IE 与其他浏览器之间，甚至不同版本的 IE 之间，也会有所不同。解决办法有以下 3 种：

- 检测浏览器的名称、版本等属性或其他辨别特性，根据不同引擎编写不同的代码。
- 坚持编写符合 JavaScript 标准的代码，运行在支持标准的浏览器上。
- 使用第三方技术，如 jQuery 等 JavaScript 框架，这样就不用考虑兼容性问题。

第二种方法目前来说还不太现实，因为 IE 市场份额还很大，并且 IE 与其他浏览器在对标准的支持方面存在很大的差异，特别是 IE 定义了很多私有属性或用法，兼容问题靠标准不能够完全解决，只能寄希望于未来浏览器开发商都自觉遵循国际标准。

第三种方法其实是迂回做法，把兼容的事情交给第三方开发商去做。很多中小企业会选择这种做法，一些大企业也会使用这一类框架。但在一些场合可能不会使用框架，或许是因为应用很简单，不想用框架，也或许其他什么原因需要考虑兼容性问题。

第一种方法是最通常的做法，不过这种做法也有缺陷，即有些时候判断的浏览器的类型版本号并不准确。那么，更好做法是判断运行当前代码的浏览器是否支持正在使用的 JavaScript 特性。

（1）检测浏览器的名称

不同的浏览器对 JavaScript 标准的支持也不同，有时希望脚本能够在不同的浏览器上都运行良好，这时需要对浏览器进行检测，确定其名称，以针对不同的浏览器编写相应的脚本。

一般使用 navigator 对象的 appName 属性，并且通过正则表达式检测返回字符串标识符中是否包含该浏览器特定的字符标志。

（2）检测浏览器的版本号

随着浏览器版本的更迭，浏览器所支持的脚本特性也在变化，有时需要针对不同的版本编

写相应的脚本，一般通过解析 navigator 对象的 userAgent 属性来获得浏览器的完整版本号。

（3）检测客户端的操作系统类型

navigator.userAgent 属性通常含有操作系统的基本信息，但没有统一的规则去根据 userAgent 获取准确的操作系统信息，因为这些值与浏览器的种类、浏览器的版本甚至浏览器的 OEM 版本都有关系。

通常可以检测一些更为通用的信息，如操作系统是 Windows 还是 Mac，而不是去检测操作系统是 Windows 7 还是 Windows X，其规则是所有的 Windows 版本都含有“Win”，所有的 Macintosh 版本都含有“Mac”，所有的 UNIX 则含有“X11”，而在 Linux 下则同时包含“X11”和“Linux”。

```
var isWin = (navigator.userAgent.indexOf("Win") != -1);
var isMac = (navigator.userAgent.indexOf("Mac") != -1);
var isUnix = (navigator.userAgent.indexOf("X11") != -1);
```

上面代码适合检测不同的操作系统，并根据不同的操作系统为应用设置不同的字体或位置等样式。

（4）检测浏览器对特定对象的支持

要编写对多种浏览器或浏览器的多个版本都能适用的脚本，就要检测一下浏览器是否支持某个对象。当然这种检测主要是针对那些潜在的不兼容对象的语句。

例如，早期的浏览器对 img 元素的支持差别很大，因此，要在脚本中操作 img 元素，需要检测浏览器是否支持。这时不需要对所有可能的浏览器一一检测，只需在必要的地方用下面的方式进行检测。

```
function rollover(imgName, imgSrc) {
    if(document.images) {
        // 执行语句
    }
}
```

如果 document.images 对象不存在，那么 if 求值的结果为 false。这种方法使对象的检测变得简单易行，但要注意，对于那些不支持该对象的浏览器需要更好地进行处理，例如：

```
function getImgAreas() {
    var result = 0;
    for(var i = 0; i < document.images.length; i++) {
        result += (document.images[i].width * document.images[i].height);
    }
    return result;
}
function reportImageArea() {
    document.form1.imgData.value = getImgAreas();
}
```

这里没使用对象支持的检测。如果浏览器支持 document.images，那么这两个函数运行

正常，否则就会抛出异常。下面对上面的代码进行优化：

```
function getImgAreas(){
    var result;
    if (document.images){
        result = 0;
        for (var i = 0; i < document.images.length; i++){
            result += (document.images[i].width * document.images[i].height);
        }
        return result;
    }
function reportImageArea(){
    var imgArea = getImgAreas();
    var output;
    if (imgArea == null){
        output = "Unknown";
    } else {
        output = imgArea;
    }
    document.reportForm.imgData.value = output;
}
```

这样不管浏览器是否支持该对象，都能向用户提供比较合理的信息，而不会只简单地跳出错误信息。

（5）检测浏览器对特定属性和方法的支持

检测一个对象是否含有某个特定的属性或方法，可以使用如下代码进行判断：

```
if(objectTest && objectPropertyTest) {
    // 执行语句
}
```

先检测对象是否存在，然后检测对象的属性是否存在。如果对象确实不存在，那么该方法有效；如果属性存在，但其值为 null、0、false，那么 if 语句的求值结果也将是 false，因此，这种方法并不安全，最好的方法是这样的：

```
if(objectReference && typeof (objectReference.propertyName) != "undefined") {
    // 执行语句
}
```

对方法的检测也可使用如下方法：

```
function myFunction() {
    if(document.getElementById) {
        // 这里可以使用 getElementById 方法
    }
}
```

建议 160：关注各种引擎对 ECMAScript v3 的分歧

在下面描述中，IE 表示 Internet Explorer，FF 表示 Mozilla Firefox 浏览器。

1. 空白符

IE 不支持 \v 字符为空白符，会把它解析为字母 v。

示例：

```
alert('\v supported ' + (String.fromCharCode(11) == '\v'));
```

输出：

```
IE: false                                    FF: true
Opera: true                                  Safari: true
```

2. 保留字

ECMA Script v3 定义了 25 个关键字：break、else、new、var、case、finally、return、void、catch、for、switch、while、continue、function、this、with、default、if、throw、delete、in、try、do、instanceof、typeof。

同时还预留了 31 个保留字用于未来版本的功能扩展：abstract、enum、int、short、boolean、export、interface、static、byte、extends、long、super、char、final、native、synchronized、class、float、package、throws、const、goto、private、transient、debugger、implements、protected、volatile、double、import、public。

所有保留字可以作为标识符在代码中使用，而 IE 仅允许下面 23 个保留字作为标识符使用：abstract、int、short、boolean、interface、static、byte、long、char、final、native、synchronized、float、package、throws、goto、private、transient、implements、protected、volatile、double、public。

3. 字面量

IE 借用 C 语言风格的转义字符，设置文字的换行符，但根据 ECMAScript v3 标准，这种行为将引发一个未结束的字符串常量的语法错误。

示例：

```
var s = "this is a \
multiline string";
```

输出：

```
IE、FF、Opera、Safari: "this is a multiline string"
```

IE 会把上面字符串视为单行字符串，FF、Opera、Safari 与 IE 解析一致。

IE 会忽略“\”及其后面的字符。s.length 将返回 34（这包括在第二行中的前导空格）。

4.Arguments 对象

不同引擎没有针对函数的 arguments 变量名实现统一的处理方式。在 IE 中，arguments 并没有包含在变量所在的上下文环境中，调用 eval 动态执行代码将无法改变 arguments 的值。

示例：

```
function foo() {
    document.write(arguments);
    document.write(arguments[0]);
    eval("arguments = 10;");
    document.write(arguments);
    document.write(arguments[0]);
}
foo("test");
```

输出：

```
IE: [object Object]test[object Object]test
FF: [object Object]test10undefined
Opera: testtest10undefined
Safari: [object Arguments]test10undefined
```

针对上面示例，如果不使用 eval 动态执行字符串，而是直接修改 arguments 变量的值：

```
function foo() {
    document.write(arguments);
    document.write(arguments[0]);
    arguments = 10;
    document.write(arguments);
    document.write(arguments[0]);
}
foo(42);
```

则不同引擎的输出结果如下：

```
IE: [object Object]4210undefined
FF: [object Object]4210undefined
Opera: 424210undefined
Safari: [object Arguments]4210undefined
```

5.Global 对象

在 IE 中，全局对象（Global）不能使用 this 进行迭代。

示例：

```
var __global__ = this;
function invisibleToIE() {
    document.write("IE can't see me");
}
__global__.visibleToIE = function() {
    document.write("IE sees me");
}
```

```
for(func in __global__) {
    var f = __global__[func];
    if(func.match(/visible/)) {
        f();
    }
}
```

输出：

```
IE: IE sees me
FF: IE sees meIE can't see me
Opera: IE can't see meIE sees me
Safari: IE can't see meIE sees me
```

在 IE 中，在使用 delete 运算符通过 this 指针删除全局成员时，将会产生运行时错误，而在 FF 等浏览器中删除返回 false。根据标准，FF 的行为是正确的，例如：

```
var __global__ = this;
function invisibleToIE() {
    document.write("IE can' t see me");
}
__global__.visibleToIE = function() {
    document.write("IE sees me");
}
document.write(delete this.invisibleToIE);
```

输出：

```
IE: runtime error (object doesn' t support this action)
FF、Opera、Safari: false
```

在 IE 中全局对象（Global）不继承 Object.prototype，即使它的类型是对象。根据 JavaScript 原型继承规则，全局对象应该继承 Object.prototype，当然这些都必须依赖于实现的浏览器。作为内置的原型，必须遵循 Object.prototype 的标准方法。

```
var __global__ = this;
document.write( typeof (__global__) + '<br>');
var f=['toString', 'toLocaleString', 'valueOf', 'hasOwnProperty', 'isPrototypeOf',
'propertyIsEnumerable'];
for( i = 0; i < f.length; i++) {
    test(f[i]);
}
function test(s) {
    if(__global__[s]) {
        document.write(s + ' supported' + '<br>');
    }
}
```

输出：

```
IE:
object
```

```
toString supported
FF、Opera、Safari:
object
toString supported
toLocaleString supported
valueOf supported
hasOwnProperty supported
isPrototypeOf supported
propertyIsEnumerable supported
```

6. 初始化数组

在 IE 中，向数组尾部添加逗号分隔符将会增加数组的长度。IE 把尾部逗号后的空白视为一个值为 undefined 的元素。实际上，这是一个非法解析的错误。

示例：

```
document.write([1, 2, 3,].length);
```

输出：

```
IE: 4
FF、Opera、Safari: 3
```

7. 函数表达式

在 IE 中，函数表达式中的标识符在闭包的上下文环境中是可见的，因为这种表达被视为函数声明。

示例：

```
var foo = function bar(b) {
    if(b == true) {
        bar();     // 可以工作，因为在函数内部 bar 标识符是可见的
    } else {
        document.write("hello");
    }
}
foo();     // 可以工作，因为 foo 标识符指向一个函数对象
bar(false);        // 失败，因为在全局环境中 bar 是不可见的
```

输出：

```
IE:"hellohello"
FF: "hello"  接着显示一个语法错误 (bar is not defined)
Opera: "hello"  接着显示一个引用错误 (Reference to undefined variable: bar)
Safari: "hello"
```

IE 把嵌套在函数表达式中的一个函数名称作为一个函数声明，这个函数声明位于封闭的上下文环境中。在下面的示例中，IE 可以向前引用 x，而在其他浏览器中将显示语法错误。

```
function f(x) {
    x();
```

```
        y = function x() {
            document.write("inner called ")
        };
        document.write(x);
        document.write(arguments[0]);
    }
    document.write("test 4 ");
    f("param");
```

输出：

```
    IE: test 4 inner called function x() {document.write("inner called ")}function
x() {document.write("inner called ")}
    FF、Opera、Safari: test 4
```

再如：

```
    function foo() {
        function bar() {}
        var x = function baz(z) {
            document.write("baz" + z);
            if(z)  baz(false);
        };
        x(true);        //合法的
        bar();          //合法的
        baz(true);      //原来是非法的，现在是合法的
    }
    foo();
```

输出：

```
    IE: baztruebazfalsebaztruebazfalse FF: baztruebazfalse (followed by an error - baz
not defined) Opera: same as FF
    FF、Opera、Safari: baztruebazfalse (followed by an error-baz not defined)
```

8. 抽象关系比较算法

IE 使用 or 而不是 and 进行字符串比较计算。

示例：

```
    document.write('1 < 10 == ' + (1 < 10) + '<br>');
    document.write('NaN < 1 == ' + (NaN < 1) + '<br>');
    document.write('1 < Infinity == ' + (1 < Infinity) + '<br>');
    document.write('"10" < 1 == ' + ("10" < 1) + '<br>');
    document.write('1 < "a" == ' + (1 < "a") + '<br>');
    document.write('"a" < "b" == ' + ("a" < "b") + '<br>');
```

输出：

```
    IE、FF、Opera、Safari:
    1 < 10 == true
    NaN < 1 == false
    1 < Infinity == true
```

```
"10" < 1 == false
1 < "a" == false
"a" < "b" == true
```

9. 函数体内的函数声明

当使用函数声明定义一个函数时，不管怎样使用 with 语句修改函数的作用域，IE 都会把它绑定到全局作用域上。

示例：

```
var v = 'value 1';
var o = {v : 'value 2'};
function f1() {
    alert('v == ' + v);
};
with(o) {
    function f2() {
        alert('v == ' + v);
    };
}
f1();
f2();
// 修改变量的值
v = 'modified value 1';
o.v = 'modified value 2';
f1();
f2();
```

输出：

```
IE、Opera:
v == value 1
v == value 1
v == modified value 1
v == modified value 1
FF、Safari:
v == value 1
v == value 2
v == modified value 1
v == modified value 2
```

针对上面示例，如果使用函数表达式定义 f1() 和 f2() 函数（代码如下），那么输出结果与 FF 相同，说明在 IE、Opera 浏览器中 with 作用域会影响函数表达式，但不会影响函数声明。

```
var f1 = function() {
    alert('v == ' + v);
};
with(o) {
    var f2 = function() {
        alert('v == ' + v);
    };
}
```

10. 枚举和属性

IE 不支持通过 for in 语句枚举类型的自定义属性，这些自定义属性通过 Object.prototype 进行映射进而实现继承。

示例：

```
function cowboy() {
    this.toString = function() {
        return "cowboy";
    }
    this.shoot = function() {
        return "bang!";
    }
}
var p = new cowboy();
document.write("Enumerable properties:");
for(var i in p) {
    document.write(" ", i);
}
document.write("<br/>cowboy propertyIsEnumerable(\"toString\"): ", p.propertyIsEnumerable("toString"));
document.write("<br/>cowboy hasOwnProperty(\"toString\"): ", p.hasOwnProperty("toString"));
```

输出：

```
IE:
Enumerable properties: shoot
cowboy propertyIsEnumerable("toString"): false
cowboy hasOwnProperty("toString"): true
FF、Opera、Safari:
Enumerable properties: toString shoot
cowboy propertyIsEnumerable("toString"): true
cowboy hasOwnProperty("toString"): true
```

11.try 语句

在 IE 中，用于保存捕获异常的变量在当前上下文环境是可见的，在 catch 子句执行完毕后此变量依然存在，但在该上下文环境被注销后会随之消失。

示例：

```
function foo() {
    try {
        throw "hello";
    } catch(x) {
        document.write(x);
    }
    document.write(x);    // x 在这里应该是不可见的
}
foo();
```

输出：

```
IE: hellohello
FF、Opera、Safari:
hello (然后抛出一个错误，x is not defined)
```

try 语句包含自己的作用域，当抛出异常时，这个作用域是封闭的，但在 IE 和 FF 浏览器中可以看到一些特殊的情况。示例如下：

```
function foo() {
    this.x = 11;
}
x = "global.x";
try {
    throw foo;
} catch(e) {
    document.write(x)      // 应该输出 "global.x"
    e();
    document.write(x)      // 应该把 x 添加到 e 对象上，但 IE 和 FF 却修改了全局变量 x
}
document.write(x);         // 应该输出 "global.x"
```

输出：

```
IE、FF: global.x1111
Opera、Safari: global.x11global.x
```

12. jion 数组原型

当分隔符为 undefined 时，IE 会使用“undefined”字符串作为分隔符来连接数组成员值。示例：

```
var array = [1, 2];
alert(array.join());
alert(array.join(undefined));
alert(array.join('-'));
```

输出：

```
IE:
1,2
1undefined2
1-2
FF、Opera、Safari:
1,2
1,2
1-2
```

13.unshift 数组原型

Array.unshift 方法能够把它的参数添加到数组的起始位置，同时返回结果数组的长度，但 IE 在调用 Array.unshift 方法时的返回值为 undefined。

示例：

```
var a = new Array(1, 2, 3);
var l = a.unshift();
document.write(l, " ");
document.write(a.length, " ");
document.write(a, " ");
l = a.unshift(2);
document.write(l, " ");
document.write(a.length, " ");
document.write(a, " ");
```

输出：

```
IE: undefined 3 1,2,3 undefined 4 2,1,2,3
FF、Opera、Safari:
3 3 1,2,3 4 4 2,1,2,3
```

14. 函数 length 属性

Object.length、String.fromCharCode.length、String.Prototype.indexOf.length、String.Prototype.lastIndexOf.length、String.prototype.slice.length 等的返回值与标准值存在差异。

- Object.length：IE 返回值为 0，标准解析为 1。
- String.fromCharCode.length：IE 返回值为 0，标准解析为 1。
- String.prototype.indexOf.length：IE 返回值为 2，标准解析为 1。
- String.prototype.lastIndexOf.length：IE 返回值为 2，标准解析为 1。
- String.prototype.slice.length：IE 和 FF 返回值为 0，标准解析为 2。

15.split 字符串原型

IE 可以忽略捕获括号，FF 能够使用空字符代替 undefined。

示例：

```
alert("A<B>bold</B>and<CODE>coded</CODE>".split(/<(\/)?([^<>]+)>/));
```

输出：

```
IE: A,bold,and,coded
FF、Opera、Safari:
A,,B,bold,/,B,and,,CODE,coded,/,CODE,
```

16.toPrecision 数值原型

如果参数的精度不确定，则 IE 将抛出 RangeError 异常。

示例：

```
var number = 123.456;
document.write('number.toString() == ' + number.toString() + '<br>');
document.write('number == ' + number + '<br>');
try {
    document.write('number.toPrecision(undefined) == ' + number.toPrecision(undefined)
```

```
+ '<br>');
    } catch (e) {
        document.write('Exception thrown. ' + e.name + ': ' + e.message + '<br>');
    }
```

输出：

```
IE:
number.toString() == 123.456
number == 123.456
Exception thrown. RangeError: The precision is out of range
FF、Opera、Safari:
number.toString() == 123.456
number == 123.456
number.toPrecision(undefined) == 123.456
```

17.valueOf 日期原型

直接调用日期原型的 valueOf 方法，IE 将返回 0，而标准规定为 NaN。

示例：

```
document.write('Date.prototype.valueOf() == ' + Date.prototype.valueOf());
```

输出：

```
IE: 0
FF、Opera、Safari: NaN
```

18.Disjunction

IE 将使用空字符代替 undefined 值。

示例：

```
// 重写 Array.prototype.toString ，使字符串包含在引号中，同时显示 undefined 值
Array.prototype.toString = function() {
    var s = '';
    for(var i = 0; i < this.length; i++) {
        if(s) {
            s += ',';
        }
        switch (typeof this[i]) {
            case 'string':
                s += "'" + this[i] + "'";
                break;
            case 'undefined':
                s += 'undefined';
                break;
            default:
                s += this[i];
                break;
        }
    }
```

```
    return '[' + s + ']';
}
var a = /((a)|(ab))((c)|(bc))/.exec('abc');
document.write(a);
```

输出：

```
IE: ['abc','a','a','','bc','','bc']
FF、Opera、Safari: NaN
['abc','a','a',undefined,'bc',undefined,'bc']
```

19. 数列

IE 不清除子表达式中重复匹配的选项。

示例：

```
Array.prototype.toString = function() {
    //执行代码
}
var a1 = /(z)((a+)?(b+)?(c))*/.exec('zaacbbbcac');
var a2 = /(a*)*/.exec('b');
document.write(a1);
document.write('<br>');
document.write(a2);
```

输出：

```
IE:
['zaacbbbcac','z','ac','a','bbb','c']
['','']
FF、Opera、Safari:
['zaacbbbcac','z','ac','a',undefined,'c']
['',undefined]
```

20. 正则表达式实现

如果正则表达式的标志中包含任何字符，那么超出了“g”、“i”、“m”，或者这几个字符重复都将抛出 SyntaxError 异常。但 IE 不会抛出 SyntaxError 或 TypeError 异常，它会抛出一个 TypeError 异常，由此可知，IE 对正则表达式字符串的要求是宽松的，在标志重复的情况下，不抛出任何异常。

示例：

```
function test(p, f) {
    try {
        var r = new RegExp(p, f);
        document.write(r.toString() + '<br>');
    } catch (e) {
        document.write(e.name + ': ' + e.message + '<br>');
    }
} test(new RegExp('foo'));                  // ok
test(new RegExp('foo'), undefined);         // ok
```

```
test(new RegExp('foo'), 'gim');// TypeError
test('foo');                   // ok
test('foo', undefined);        // ok
test(undefined, 'gim');        // ok
test('foo', 'gimgim');         // SyntaxError
test('foo', 'pvl');            // SyntaxError
```

21.getYear 日期原型

对于 IE 来说，Date.prototype.getYear 类似于 Date.prototype.getFullYear。

示例：

```
var d = new Date("10 July 2001");
var y = d.getYear();
document.write(y + '<br>');
```

输出：

```
IE: 2001
FF、Opera、Safari: 101
```

22.setYear 日期原型

对于 IE 来说，Date.prototype.setYear 类似于 Date.prototype.setFullYear。

示例：

```
var d = new Date(+0);
d.setYear(95);
y = d.getYear();
document.write("setYear: " + y + " ");
d.setFullYear(95);
y = d.getYear();
document.write("setFullYear: " + y);
```

输出：

```
IE: setYear: 95 setFullYear: 95
FF、Opera、Safari: setYear: 95 setFullYear: -1805
```

建议 161：关注各种引擎对 ECMAScript v3 的补充

1. 数值类型

ECMAScript v3 提供了检测各种 NaN 值的方法，而 IE 没有提供任何方法来检测各种 NaN 值。

2. 数值类型的字符串应用

ECMAScript v3 规范提供了浮点数转换成字符串的精度表示，但最终显示结果将依赖于

实现的代理设备。代表浮点数作为一个字符串时，IE 支持 16 位精度。而其他引擎与 IE 略有不同，下面通过一个示例进行比较。

```
var x = 0x8000000000000800;
document.write(x, " ");
var y = 9223372036854777856;
document.write(y);
```

输出：

```
IE:
9223372036854777000              9223372036854777000
FF、Opera、Safari:
9223372036854778000              9223372036854778000
```

3.typeof 运算符

在使用 typeof 运算符检测宿主对象的类型时，将依赖于实现引擎返回不同的值。

示例：

```
alert(typeof(window.alert));
```

输出：

```
IE: object
FF、Opera、Safari: function
```

不过下面的用法在 IE 中将抛出异常，而其他引擎返回 45。

```
var a = window.alert; a.apply(null, [45]);
```

4.for in 语句

ECMAScript v3 规范的枚举依赖于实现属性的机制。在 IE 中，在列举属性时，以相反的顺序枚举原型属性（相对于在其中被添加的顺序），按顺序枚举当前对象的属性，其余对象的属性按原型链中的顺序枚举。

示例：

```
function dough() {
    this.da = "da";
    this.db = "db";
    this.dc = "dc";
}
bread.prototype = new dough();
function bread() {
    this.ba = "ba";
    this.bb = "bb";
    this.bc = "bc";
}
pizza.prototype = new bread();
function pizza() {
```

```
    this.pa = "pa";
    this.pb = "pb";
    this.pc = "pc";
}
function getProperties(obj, objName) {
    var res = "";
    for(var i in obj) {
        res += objName + "." + i + " = " + obj[i] + "<br>";
    }
    return res;
}
var p = new pizza();
document.write(getProperties(p, "pizza"));
```

输出：

```
IE:
pizza.bc = bc
pizza.bb = bb
pizza.ba = ba
pizza.da = da
pizza.db = db
pizza.dc = dc
pizza.pa = pa
pizza.pb = pb
pizza.pc = pc
FF、Opera、Safari:
pizza.pa = pa
pizza.pb = pb
pizza.pc = pc
pizza.ba = ba
pizza.bb = bb
pizza.bc = bc
pizza.da = da
pizza.db = db
pizza.dc = dc
```

5. 对象连接

当不同变量连接对象时，由于实现的差异，运行结果可能也会不同。

考虑下面的示例：

```
function foo() {
    function bar() {}
    return bar;
}
var x = foo();
var y = foo();
x.blah = 1;
y.blah = 2;
document.write(x.blah + y.blah);
```

上面代码的输出值为 3，但另一种实现的输出值为 4，这种实现也符合标准。

6. 函数创建

与标准不同，IE 使用最后一次出现的函数声明定义函数。

示例：

```
if(true) {
    function bar() {
        document.write("bar1");
    }
} else {
    function bar() {
        document.write("bar2");
    }
}
bar();
function foo() {
    if(true) {
        function baz() {
            document.write("baz1");
        }
    } else {
        function baz() {
            document.write("baz2");
        }
    }
    baz();
}
foo();
```

输出：

```
IE: bar2   baz2
FF、Opera、Safari:
bar1           baz1
```

7.eval() 方法

ECMAScript v3 规范不允许通过其他变量间接调用全局函数 eval()，否则将抛出一个 EvalError 异常，但 IE、FF、Safari 引擎允许间接调用 eval()。

示例：

```
var sum = eval("1 + 2");
alert(sum);
var myeval = eval;
try {
    sum = myeval("1 + 2");
    alert(sum);
} catch (e) {
    alert("indirect eval not supported!");
}
```

输出：

```
IE、FF、Safari: 3  3
Opera: 3  "indirect eval not supported"
```

8.parseInt() 方法

当基数为 0，或者未定义的字符串值以 0 开始，后面不是 x 或 X 时，这个数值可根据不同的实现转换为八进制或十进制的数字。IE 和 FF、Safari 引擎会把它解释为八进制数字。

示例：

```
alert(parseInt("08", undefined));
alert(parseInt("08", 0));
alert(parseInt("011", undefined));
alert(parseInt("011", 0));
```

输出：

```
IE、FF、Safari:
0, 0, 9, 9
Opera: 8, 8, 11, 11
```

9.toString 函数原型

当使用 toString 函数原型来处理其他函数中的空白时，IE 将忽略换行符和 Tab 空格，而 FF、Opera 和 Safari 浏览器忽略换行符，这些浏览器有一个细微的差别。

示例：

```
function foo() {
    var e = {
        name : "value",
        id : 11
    };
    return e;
}
alert(foo.toString().indexOf(','));
```

输出：

```
IE: 45
FF: 42
Opera: 43
Safari: 45
```

10. 对象构造器和宿主对象

当使用 Object 构造器构建宿主对象时，IE 把它视为一个 object 对象，而其他引擎把它视为一个 DOM 对象。

示例：

```
alert(new Object(window.document));
```

输出：

```
IE: [object]
FF、Safari、Opera:[object HTMLDocument]
```

11. 宿主对象的值

当使用 valueOf 方法读取宿主对象的值时，IE 将会抛出错误。

示例：

```
var x = new Object(window.document);
alert(x.valueOf());
```

输出：

```
IE: 抛出运行时错误 (object doesn't support this property or method)
FF、Safari、Opera:[object HTMLDocument]
```

12.toLocaleString 数组原型

不同引擎对于 toLocaleString 方法的返回值是不同的。

示例：

```
var n = Number(123456789.00);
var d = new Date();
var t = new Date().getTime();
var s = "hello world";
var a = new Array();
a.push(n);
a.push(d);
a.push(d);
a.push(s);
document.write(a.toString() + "<br>");
document.write(a.toLocaleString() + "<br>");
```

输出：

```
IE:
123456789,Fri Jul 13 01:13:45 UTC+0530 2007,Fri Jul 13 01:13:45
UTC+0530 2007,hello world
123,456,789.00, Friday, July 13, 2007 1:13:45 AM, Friday, July
13, 2007 1:13:45 AM, hello world
FF
123456789,Fri Jul 13 2007 01:14:50 GMT+0530 (India Standard
Time),Fri Jul 13 2007 01:14:50 GMT+0530 (India Standard Time),hello world
123,456,789,Friday, July 13, 2007 1:14:50 AM,Friday, July 13, 2007 1:14:50
AM,hello world
Opera:
123456789,Fri, 13 Jul 2007 01:16:07 GMT+0530,Fri, 13 Jul 2007
01:16:07 GMT+0530,hello world
123456789,7/13/2007 1:16:07 AM,7/13/2007 1:16:07 AM,hello world
Safari
123456789,Fri Jul 13 2007 01:16:24 GMT+0530 (India Standard Time),Fri Jul 13 2007
```

```
01:16:24 GMT+0530 (India Standard Time),hello world
    123456789,Friday, July 13, 2007 01:16:24,Friday, July 13, 2007 01:16:24,hello
world
```

13. Number() 构造器

IE 和 Opera 会把所有字符串带有负号的十六进制数字转换为 NaN，而其他引擎会把它转换为负数。

示例：

```
alert('Number("0x10") == ' + Number("0x10"));
alert('Number("-0x10") == ' + Number("-0x10"));
```

输出：

```
IE、Safari: 16, NaN
FF、Opera: 16, -16
```

14. toString 数值原型

当使用无效的基数时，IE 将抛出一个 TypeError 异常。

示例：

```
var val = 42;
document.write('val.toString() == ' + val.toString() + '<br>');
document.write('val.toString(2) == ' + val.toString(2) + '<br>');
document.write('val.toString(8) == ' + val.toString(8) + '<br>');
document.write('val.toString(16) == ' + val.toString(16) + '<br>');
document.write('val.toString(36) == ' + val.toString(36) + '<br>');
try {
    document.write('val.toString(100) =='+ val.toString(100) + '<br>');
} catch(e) {
    document.write('Invalid Radix Error: ' + e.name + ': ' + e.message + '<br>');
}
document.write('val.toString() == ' + val.toString() + '<br>');
try {
    document.write('val.toString(undefined) =='+ val.toString(undefined) + '<br>');
} catch(e) {
    document.write('Invalid Radix Error: ' + e.name + ':' + e.message + '<br>');
}
```

输出：

```
IE:
val.toString() == 42
val.toString(2) == 101010
val.toString(8) == 52
val.toString(16) == 2a
val.toString(36) == 16
Invalid Radix Error: TypeError: Invalid procedure call or argument
val.toString() == 42
Invalid Radix Error: TypeError: Invalid procedure call or argument
```

```
FF:
val.toString() == 42
val.toString(2) == 101010
val.toString(8) == 52
val.toString(16) == 2a
val.toString(36) == 16
Invalid Radix Error: Error: illegal radix 100
val.toString() == 42
Invalid Radix Error: Error: illegal radix 0
Opera、Safari:
val.toString() == 42
val.toString(2) == 101010
val.toString(8) == 52
val.toString(16) == 2a
val.toString(36) == 16
val.toString(100) == 42
val.toString() == 42
val.toString(undefined) == 42
```

15.UTC 日期对象

在 IE 和 FF 浏览器中，如果 UTC 日期对象只是作为参数调用，那么返回一个数字，代表在这一年的 1 月 1 日的当前语言环境。

示例：

```
document.write(Date.UTC(1995));
```

输出：

```
IE、FF: 788918400000
Opera: syntax error
Safari: NaN
```

16.toString 日期原型

IE 将使用 UTC 进行日期字符串转换。

示例：

```
var d = new Date("4 July 1776");
document.write(d.toString());
```

输出：

```
IE: Thu Jul 4 00:00:00 UTC+0530 1776
FF、Safari: Thu Jul 04 1776 00:00:00 GMT+0530 (India Standard Time)
Opera: Thu, 04 Jul 1776 00:00:00 GMT+0530
```

17.toDateString 日期原型

使用 toDateString 方法，不同引擎返回的字符串信息是不同的。

示例：

```
var d = new Date("July 10 1995");
document.write(d.toDateString());
```

输出：

```
IE、FF、Safari: Mon Jul 10 1995
Opera: Mon, 10 Jul 1995
```

18. toTimeString 日期原型

使用 toTimeString 方法，不同引擎返回的字符串信息是不同的。

示例：

```
var d = new Date("July 10 1995");
document.write(d.toTimeString());
```

输出：

```
IE: 00:00:00 UTC+0530
FF、Safari: 00:00:00 GMT+0530 (India Standard Time)
Opera: 00:00:00 GMT+0530
```

19.toLocaleString 日期原型

使用 toLocaleString 方法，不同引擎返回的字符串信息是不同的。

示例：

```
var d = new Date("July 10 1995");
document.write(d.toLocaleString());
```

输出：

```
IE、FF: Monday, July 10, 1995 12:00:00 AM
Opera: 7/10/1995 12:00:00 AM
Safari: Monday, July 10, 1995 00:00:00
```

20. toLocaleDateString 日期原型

使用 toLocaleDateString 方法，不同引擎返回的字符串信息是不同的。

示例：

```
var d = new Date("July 10 1995");
document.write(d.toLocaleDateString());
```

输出：

```
IE、FF: Monday, July 10, 1995
Opera: 7/10/1995
Safari: same as IE
```

21.toLocaleTimeString 日期原型

使用 toLocaleTimeString 方法，不同引擎返回的字符串信息是不同的。

示例：

```
var d = new Date("July 10 1995");
document.write(d.toLocaleTimeString());
```

输出：

```
IE、FF、Opera: 12:00:00 AM
Safari: 00:00:00
```

22.toUTCString 日期原型

使用 toUTCString 方法，不同引擎返回的字符串信息是不同的。

示例：

```
var d = new Date("July 10 1995");
document.write(d.toUTCString());
```

输出：

```
IE: Sun, 9 Jul 1995 18:30:00 UTC
FF、Opera、Safari: Sun, 09 Jul 1995 18:30:00 GMT
```

23.toString 正则表达式原型

当使用 RegExp 构造正则表达式对象时，如果传递的参数为空、未定义或 /，则不同引擎返回对象的字符串值是不同的。

示例：

```
document.write(new RegExp().toString() + '<br>');
document.write(new RegExp(undefined).toString() + '<br>');
document.write(new RegExp("/").toString() + '<br>');
```

输出：

```
IE、Opera、Safari:
//
//
///
FF:
/(?:)/
/undefined/
/\//
```

24.message 错误原型

在 IE、FF 中，默认错误信息是一个空字符串，如果明确将未定义值作为参数信息，则将初始值设置为 undefined。

示例：

```
function test(e) {
```

```
    document.write('e.message: ' + e.message + '<br>');
}
test(new Error());
test(new Error(undefined));
```

输出：

```
IE、FF:
e.message:
e.message: undefined
Opera:
e.message: Generic error
e.message: undefined
Safari:
e.message: Unknown error e
.message: Unknown error
```

25.toString 错误原型

不同引擎对于错误对象的字符串返回值是不同的。

示例：

```
function test(e) {
    document.write('e.toString(): ' + e.toString() + '<br>');
}
test(new Error());
test(new Error(undefined));
test(new Error('Complex error'));
```

输出：

```
IE:
e.toString(): [object Error]
e.toString(): [object Error]
e.toString(): [object Error]
FF:
e.toString(): Error
e.toString(): Error: undefined
e.toString(): Error: Complex error
Opera:
e.toString(): [Error: name: Error message: Generic error ]
e.toString(): [Error: name: Error ]
e.toString(): [Error: name: Error message: Complex error ]
Safari:
e.toString(): Error: Unknown error
e.toString(): Error: Unknown error
e.toString(): Error: Complex error
```

26.message 本地错误原型

不同错误类型的默认抛出信息是不同的。

示例：

```
document.write(new EvalError().message + '<br>');
document.write(new RangeError().message + '<br>');
document.write(new ReferenceError().message + '<br>');
document.write(new SyntaxError().message + '<br>');
document.write(new TypeError().message + '<br>');
document.write(new URIError().message + '<br>');
```

输出：

```
IE、FF：输出皆为空
Opera:
Use of eval as a value
Illegal manipulation of array or string length
Undefined variable or property
Mis-constructed program text
Incorrect value to a primitive operation
Generic error in a URI
Safari:
EvalError
RangeError
ReferenceError
SyntaxError
TypeError
URIError
```

27. 错误对象

如果值、对象、属性等超出 ECMAScript v3 规范中所描述的功能，那么可能会导致返回一个结构。例如，寻找一个在全局范围内的变量，而不是抛出一个错误（如 ReferenceError），可以降低互操作性。

示例：

```
var x = {};
x.__proto__ = 3;
document.write(x.__proto__);
```

输出：

```
IE、Opera、Safari: 3
FF: [object Object]
```

建议 162：关注各种引擎对 Event 解析的分歧

1.Event 使用

IE 可以直接使用 event 对象，但其他引擎不可以直接使用。

```
<input type="button" value="clickMe" onclick="doIt()">
<script>
```

```
function doIt(){
    alert(event);
}
</script>
```

上面代码在 Firefox 等浏览器中不能正常工作，这是因为 Firefox 浏览器中没有默认的 event 对象，只能在事件发生的现场使用 event 对象。兼容的方法如下：

```
<input type="button" value="clickMe" onclick="doIt(event)">
<script>
function doIt(oEvent){
    alert(oEvent);
}
</script>
```

2. event.srcElement[IE] 和 event.target[Moz]

在 Firefox 等引擎下的 event.target 相当于 IE 下的 event.srcElement，不过在细节上有区别，event.srcElement 返回一个 Html Element，而 event.target 返回的是个节点，该节点包括文本节点。比较下面示例代码的区别和联系。

（1）IE 使用 event.srcElement

```
<table border="1" width="50%" onclick="doIt()">
    <tr><td>1</td><td>2</td></tr>
    <tr><td>3</td><td>4</td></tr>
</table>
<script>
function doIt() {
    alert(event.srcElement.tagName);
}
</script>
```

（2）非 IE 浏览器使用 event.target

```
<table border="1" width="50%" onclick="doIt()">
    <tr><td>1</td><td>2</td></tr>
    <tr><td>3</td><td>4</td></tr>
</table>
<script>
function doIt(oEvent) {
    var Target = oEvent.target;
    while(oTarget.nodeType != 1)
    Target = oTarget.parentNode;
    alert(oTarget.tagName);
}
</script>
```

3. 获取键盘值

不同引擎获取键盘值的途径和方法是不同的。

（1）IE 使用 event.keyCode

```
<input type="text" onkeypress="doIt()">
<script>
function doIt() {
    alert(event.keyCode);
}
</script>
```

(2) 非 IE 浏览器使用 event.which

```
<input type="text" okeypress="doIt(event)">
<script>
function doIt(oEvent) {
    alert(oEvent.which)
}
</script>
```

4. 获取鼠标指针的绝对位置

IE 通过 event 对象的 event.x 和 event.y 属性获取鼠标指针的绝对位置，而 Firefox 等引擎通过 event 对象的 event.pageX 和 event.pageY 获取鼠标指针的绝对位置。例如，通过如下方法可以兼容不同引擎的用法。

```
    <div id="myDiv" onclick="doIt(event)" style="position:absolute;top:100;left:100;
width:100;height:100;background-color:orange;border:1px solid black">
    <script>
    function doIt(oEvent) {
        var posX = oEvent.x ? oEvent.x : oEvent.pageX;
        var posY = oEvent.y ? oEvent.y : oEvent.pageY;
        alert("X:" + posX + "\nY:" + posY)
    }
    </script>
```

5. 获取鼠标指针的相对位置

IE 通过 event 对象的 event.offsetX 和 event.offsetY 属性获取鼠标指针的相对位置，而 Firefox 等引擎通过 event 对象的 event.layerX 和 event.layerY 获取鼠标指针的相对位置。例如，通过如下方法可以兼容不同引擎的用法。

```
    <div id="myDiv" onclick="doIt(event)" style="position:absolute;top:100;left:100;
width:100;height:100;background-color:orange;border:1px solid black">
    <script>
    function doIt(oEvent) {
        var posX = oEvent.offsetX ? oEvent.offsetX : oEvent.layerX;
        var posY = oEvent.offsetY ? oEvent.offsetY : oEvent.layerY;
        alert("X:" + posX + "\nY:" + posY)
    }
    </script>
```

6. 绑定事件

IE 通过 attachEvent、detachEvent 方法为 DOM 对象绑定事件、注销事件，而 Firefox 等

引擎通过 addEventListener、removeEventListener 方法为 DOM 对象绑定事件、注销事件。

（1）IE

```
<input type="button" value="test" id="testBT">
<script>
var oButton = document.getElementById("testBT");
oButton.attachEvent("onclick", clickEvent);
function clickEvent() {
    alert("Hello, World!");
}
</script>
```

（2）Firefox

```
<input type="button" value="test" id="testBT">
<script>
var oButton = document.getElementById("testBT");
oButton.addEventListener("click", clickEvent, true);
function clickEvent() {
    alert("Hello, World!");
}
</script>
```

注意，在 IE 中要在事件前加 on 前缀，而在 Firefox 等引擎中不用加。

7. 获取事件目标源

在 IE 下，event 对象使用 srcElement 属性获取当前事件的目标元素，不支持 target 属性。而其他引擎正好相反，event 对象使用 target 属性获取当前事件的目标元素，不支持 srcElement 属性。可以使用下面代码进行兼容。

```
obj = event.srcElement ? event.srcElement : event.target;
```

建议 163：关注各种引擎对 DOM 解析的分歧

1. 通过 ID 访问 HTML 元素

对于 IE 来说，可以使用 eval("idName") 或 getElementById("idName") 获取 ID 为 idName 的 HTML 元素，但非 IE 浏览器只能使用 getElementById("idName") 来获取 ID 为 idName 的 HTML 元素。通过 ID 访问 HTML 元素一般直接使用 document.getElementById 就可以了。

```
<input type="button" value="clickMe" id="myButton">
<script>
    alert(document.getElementById("myButton").value);
</script>
```

为了兼容低版本的 IE，可以加上 document.all。

提示： IE 还支持把 HTML 元素的 ID 名作为 document 对象的属性名直接使用，如 document.idName。但其他引擎是不支持这种用法的，建议禁止使用。同时不要定义与 HTML 元素 ID 相同的变量名，以减少错误。在声明变量时一律加上 var 关键字，以避免产生歧义。

2. 集合类对象问题

在 IE 中可以使用 () 或 [] 获取集合类对象，而在非 IE 浏览器中只能使用 [] 获取集合类对象，解决方法就是统一使用 [] 获取集合类对象。

例如，访问表单元素 form，传统方法支持使用 document.form.item 访问表单对象，但 IE 允　许 document.formName.item("itemName") 或 document.formName.elements ["elementName"] 用法，Firefox 等引擎仅支持 document.formName.elements["elementName"] 用法。若想要兼容 IE、Firefox 等浏览器都能够正常运行，那么需要规范化访问表单对象的方法，统一使用 document.formName.elements["elementName"]。

```
<body>
        <form. name="myForm">
                <input value="test" id="txt" />
        </form>
</body>
<script>
        alert(document.myForm.elements["txt"].value);
</script>
```

在 Firefox 中，在访问数组时不能使用类似 arr("itemName") 的形式，必须使用中括号，而在 IE 中两者都可以。在使用时应该注意代码的规范性，例如，下面两种不同的写法在 Firefox 等引擎中有不同的返回结果。

（1）错误写法

```
<form. name="myForm">
        <input value="test" id="txt" />
</form>
<script>
        alert(document.myForm);
        alert(document.forms.length);     //返回 0，错误
</script>
```

（2）正确写法

```
<body>
        <form. name="myForm">
                <input value="test" id="txt" />
        </form>
</body>
<script>
        alert(document.myForm);
```

```
        alert(document.forms.length);    //返回1，正常
</script>
```

3. 删除节点

IE 支持使用 removeNode 方法删除节点。

```
<input type="button" value="clickMe" id="myButton">
<script>
document.getElementById("myButton").removeNode();
</script>
```

但 Firefox 等引擎没有这个方法，只能先找到父节点，然后调用 DOM 的 removeChild 方法才可以删除节点。不过 IE 也支持这种方法。

```
<input type="button" value="clickMe" id="myButton">
<script>
var Node = document.getElementById("myButton");
oNode.parentNode.removeChild(oNode);
</script>
```

4. 交换节点

IE 支持使用 swapNode 私有方法来交换两个 HTML 元素节点。

```
<input type="button" value="first" id="firstButton">
<input type="button" value="second" id="secondButton">
<script>
var First = document.getElementById("firstButton");
var Second = document.getElementById("secondButton");
oFirst.swapNode(oSecond);
</script>
```

但 Firefox 不支持这种方法，要实现相同的操作，需要自定义函数。

```
<input type="button" value="first" id="firstButton">
<input type="button" value="second" id="secondButton">
<script>
if(window.Node) {
    Node.prototype.swapNode = function(node) {
        var nextSibling = this.nextSibling;
        var parentNode = this.parentNode;
        node.parentNode.replaceChild(this, node);
        parentNode.insertBefore(node, nextSibling);
    }
}
var First = document.getElementById("firstButton");
var Second = document.getElementById("secondButton");
oFirst.swapNode(oSecond);
</script>
```

5. 插入节点

IE 支持使用 insertAdjacentHTML 和 insertAdjacentElement 两个私有方法来插入 HTML 节点。

```
<div id="div1" style="border:1px solid black"></div>
<script>
var Div = document.getElementById("div1");
var htmlInput = "<input>";
oDiv.insertAdjacentHTML('beforeEnd', htmlInput);
</script>
```

但 Firefox 中没有这两个方法，为了兼容它们，可以统一使用 DOM 的 insertBefore 方法实现，IE 也支持该方法。

```
<div id="div1" style="border:1px solid black"></div>
<script>
var Div = document.getElementById("div1");
var Element = document.createElement("input");
oElement.type = "text";
oDiv.insertBefore(oElement, null);
</script>
```

6. 访问自定义属性

对于 IE 来说，可以使用获取常规属性的方法来获取自定义属性，也可以使用 getAttribute() 获取自定义属性。而非 IE 浏览器只能使用 getAttribute() 获取自定义属性。解决方法是统一通过 getAttribute() 获取自定义属性。

7.input.type 属性

在 IE 中，input.type 属性为只读，但在其他浏览器中，input.type 属性为读 / 写。

8. window.location.href

IE 支持使用 window.location 或 window.location.href 属性，但其他引擎只能使用 window.location。解决方法是使用 window.location 来代替 window.location.href。

9. 模态窗口和非模态窗口

IE 可以通过 showModalDialog 和 showModelessDialog 来分别打开模态窗口和非模态窗口，其他引擎则不能。解决方法是直接使用 window.open(pageURL,name,parameters) 方式打开新窗口。要将子窗口中的参数传递回父窗口，可以在子窗口中使用 window.opener 来访问父窗口。例如：

```
var parWin = window.opener;
parWin.document.getElementById("Aqing").value = "Aqing";
```

10.frame

以下面的 frame 为例。

```
<frame src="xxx.html" id="frameId" name="frameName" />
```

（1）访问 frame 对象

IE 使用 window.frameId 或 window.frameName 访问这个 frame 对象，其他引擎只能使用 window.frameName 访问这个 frame 对象。另外，所有引擎都可以使用 window.document.getElementById("frameId") 来访问这个 frame 对象。

（2）切换 frame 内容

所有引擎都可以使用如下代码来切换 frame 内容。

```
window.document.getElementById("testFrame").src = "xxx.html";
```

或

```
window.frameName.location = "xxx.html";
```

如果需要将 frame 中的参数传回父窗口，可以在 frme 中使用 parent 来访问父窗口，例如：

```
parent.document.form1.filename.value="Aqing";
```

如果在 <frame> 标签中设置如下属性：

```
<frame src="xx.htm" id="frameId" name="frameName" />
```

那么 IE 可以通过 id 或 name 访问该 frame 对应的 window 对象，其他引擎只能够通过 name 来访问该 frame 对应的 window 对象。

如果将上述 <frame> 标签写在顶层窗口的 HTML 中，那么 IE 可以使用 window.top.frameId 或 window.top.frameName 访问该 window 对象，其他引擎只能使用 window.top.frameName 访问该 window 对象。不过，所有引擎都可以使用 window.top.document.getElementById("frameId") 访问 <frame> 标签。

11.body

在 IE 中，当 <body> 标签被浏览器完全读入之后，body 对象才存在，而其他引擎的 body 对象，在 <body> 标签没有被浏览器完全读入之前就已存在。例如：

（1）非 IE 引擎

```
<body>
    <script>
        document.body.onclick = function(evt) {
            evt = evt || window.event;
            alert(evt);
        }
    </script>
```

```
</body>
```

（2）兼容性用法

```
<body></body>
<script type="text/javascript">
    document.body.onclick = function(evt) {
        evt = evt || window.event;
        alert(evt);
    }
</script>
```

12. 访问父节点

对于父元素（parentElement），IE 与其他引擎的访问方法是不同的。

- IE：obj.parentElement
- Firefox：obj.parentNode

由于所有引擎都支持 DOM，因此可以使用 obj.parentNode 兼容不同浏览器。

13. 插入文本

IE 支持使用 innerText 属性为 DOM 元素插入文本，但其他引擎不支持这个属性，却支持 textContent 属性。因此，解决方法如下：

```
if(navigator.appName.indexOf("Explorer") > -1) {
    document.getElementById('element').innerText = "my text";
} else {
    document.getElementById('element').textContent = "my text";
}
```

14. 文本操作

不同浏览器对于 <table> 标签的操作各不相同。IE 不允许对 table 和 tr 对象的 innerHTML 赋值，也不支持使用 appendChild 方法添加 tr 对象。 解决方法就是为 table 追加一个空行。

```
var row = otable.insertRow(-1);
var cell = document.createElement("td");
cell.innerHTML = " ";
cell.className = "className";
row.appendChild(cell);
```

建议 164：关注各种引擎对 CSS 渲染的分歧

1.float 值

访问一个给定 CSS 值的最基本句法如下：

```
object.style.property
```

使用驼峰写法来替换有连接符的值。例如，访问某个 ID 为“header”的 的 background-color 值，可以使用如下方法：

```
document.getElementById("header").style.backgroundColor= "#ccc";
```

但由于 float 这个词是 JavaScript 保留字，因此不能用 object.style.float 来访问，这里可以根据不同引擎选择使用不同的方法。

（1）IE 使用 styleFloat

```
document.getElementById("header").style.styleFloat = "left";
```

（2）非 IE 浏览器使用 cssFloat

```
document.getElementById("header").style.cssFloat = "left";
```

2. 计算样式

在 JavaScript 中，元素的计算样式可以使用如下语法：

```
object.style.property
```

利用该语法可以方便地在外部访问和修改某个 CSS 样式，但该语法的限制是这些方法只能取出已设的行内样式或直接由 JavaScript 设定的样式，并不能访问某个外部的样式表。为了访问元素的计算样式，可以使用下面的代码。

（1）IE 使用 currentStyle

```
var myObject = document.getElementById("header");
var myStyle = myObject.currentStyle.backgroundColor;
```

（2）非 IE 浏览器使用 defaultView.getComputedStyle

```
var myObject = document.getElementById("header");
var myComputedStyle = document.defaultView.getComputedStyle(myObject, null);
var myStyle = myComputedStyle.backgroundColor;
```

3. 访问类样式

class 是 JavaScript 的一个保留字，因此在不同引擎中需要使用不同的方法来访问类样式。

（1）IE 使用 getAttribute("className")

```
var myObject = document.getElementById("header");
var myAttribute = myObject.getAttribute("className");
```

（2）非 IE 浏览器使用 getAttribute("class")

```
var myObject = document.getElementById("header");
var myAttribute = myObject.getAttribute("class");
```

该方法也适用于 setAttribute。

4. 访问 for 属性

对于 <label> 标签中的 for 属性来说，由于 for 是 JavaScript 的保留字，因此需要使用不同的方法来分别访问 <label> 标签中的 for。

（1）IE 使用 getAttribute("htmlFor")

```
var myObject = document.getElementById("myLabel");
var myAttribute = myObject.getAttribute("htmlFor");
```

（2）非 IE 浏览器使用 getAttribute("for")

```
var = document.getElementById("myLabel");
var myAttribute = myObject.getAttribute("for");
```

5. 获取鼠标指针的位置

在 Web 开发中，经常需要计算鼠标指针的位置，不过不同引擎对此采取的用法和标准不同，需要分别编写代码进行兼容。下面代码是最基本的，可以解释其中的异同点。同时必须指出，它们获取的结果相对于不同引擎会有所不同。

（1）IE 使用 event.clientX 和 event.clientY

```
var myCursorPosition = [0, 0];
myCursorPosition[0] = event.clientX;
myCursorPosition[1] = event.clientY;
```

（2）非 IE 浏览器使用 event.pageX 和 event.pageY

```
var myCursorPosition = [0, 0];
myCursorPosition[0] = event.pageX;
myCursorPosition[1] = event.pageY;
```

6. 获取可见区域窗口的大小

在程序设计中，经常需要获取浏览器的可见区域大小，具体代码如下。

（1）IE

```
var myBrowserSize = [0, 0];
myBrowserSize[0] = document.documentElement.clientWidth;
myBrowserSize[1] = document.documentElement.clientHeight;
```

（2）非 IE 浏览器

```
var myBrowserSize = [0, 0];
myBrowserSize[0] = window.innerWidth;
myBrowserSize[1] = window.innerHeight;
```

7.Alpha 透明

Alpha 透明不是 JavaScript 句法问题，而是源于 CSS 的 Alpha 透明。不过在使用 JavaScript 设计淡入或淡出效果时，需要通过访问 CSS 的 Alpha 透明设置来完成，通常使用

for 循环来实现。

（1）IE

```
var myObject = document.getElementById("myElement");
myObject.style.filter = "alpha(opacity=80)";
```

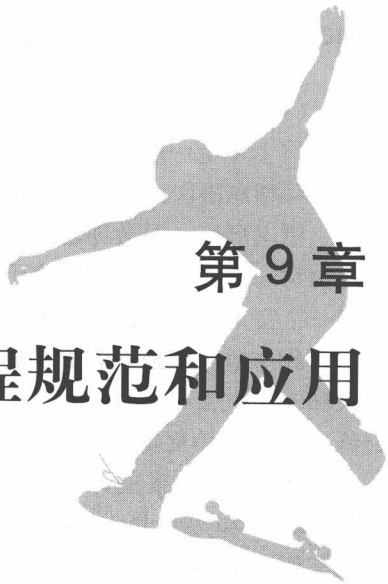
（2）非 IE 浏览器

```
var myObject = document.getElementById("myElement");
myObject.style.opacity = "0.5";
```

8. 元素尺寸

IE 支持“obj.style.height = imgObj.height”语句，但其他引擎对此视为无效。解决方法是为其他引擎添加单位。

```
obj.style.height = imgObj.height + 'px';
```



第9章 JavaScript 编程规范和应用

每种语言都有自己的不足之处，都存在低效模式。随着越来越多的人选用 JavaScript 语言，它的应用边界也在不断扩展。自 2005 年以来，Ajax 应用对 JavaScript 和浏览器的推动作用远超过以前，其结果是出现了一些非常具体的模式，这其中有优秀的做法，也有糟糕的做法。

也许，很多程序员会无休止地讨论良好的语言风格是由什么构成的。有些程序员坚定地拥护曾经的用法，比如在学校或在开始工作时养成的编程习惯，完全没有代码风格的意识。事实证明，代码风格在编程中是非常重要的，就像文字风格对于写作非常重要一样。好的代码风格能够更好地阅读，更为关键的是能够提高代码的执行效率。

建议 165：不要混淆 JavaScript 与浏览器

语言和环境是两个不同的概念。提及 JavaScript，很多读者可能会想到浏览器。脱离环境的 JavaScript 代码是不能够运行的，这与其他系统级的语言有着很大的不同。例如，C 语言可以用于开发系统和制造环境，而 JavaScript 只能够寄生在某个具体的环境中才能够工作。

JavaScript 运行环境一般都由宿主环境和执行期环境构成。其中宿主环境是由外壳程序生成的，如浏览器就是一个外壳程序，它提供了一个可控制浏览器窗口的宿主环境。执行期环境则由嵌入到外壳程序中的 JavaScript 引擎（或称为 JavaScript 解释器）生成，在这个环境中 JavaScript 能够生成内置静态对象和初始化执行环境等（如图 9.1 所示）。

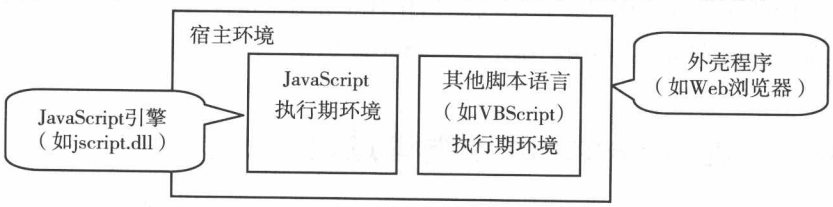


图 9.1　JavaScript 运行环境

（1）宿主环境

宿主环境一般由外壳程序创建和维护，它不仅为 JavaScript 语言提供服务，还可以运行很多种脚本语言。这与 Java 虚拟机功能类似。

宿主环境一般会创建一套公共对象系统，这套对象系统对所有脚本语言开放，允许它们自由访问。同时，宿主环境还会提供公共接口，用来装载不同的脚本语言引擎。这样可以在同一个宿主环境中装载不同的脚本引擎，并允许它们共享宿主对象。

脚本语言与独立的语言是有区分的，JavaScript 是一种脚本语言，它本身不提供 I/O（输入和输出）接口，也没有与系统和外界通信的能力，更不能够有操作外围设备、管理内存、修改注册表等更低级的行为，这些功能全部交给宿主环境来完成。例如，在客户端浏览器（外壳程序）的宿主环境中，JavaScript 通过 window 对象的 alert() 方法以及 document 对象的 write() 和 writeln() 方法输出信息，而借助 window 对象的 prompt() 方法接收信息。当然，在其他宿主环境中可能会使用不同的宿主对象来完成以上功能。例如，在 Windows 环境中，微软公司开发的 WSH（Windows Script Host）就是一种脚本语言的宿主环境，它就定义了 wscript 对象（类似 window 对象）来表示全局对象。

不仅 Web 浏览器是外壳程序，只要能够提供 JavaScript 引擎执行的环境都可以算做外壳程序。很多服务器、桌面应用系统也都提供能够允许 JavaScript 引擎执行的运行环境，这些运行环境也是宿主环境。同时，大部分 JavaScript 引擎自身也带有一个用于代码调试的程序，在这个简单的程序被运行时，也会创建一个宿主环境。

另外，外壳程序还可以通过符合标准的扩展接口接纳更多的插件、组件或 ActiveX 控件等 .DLL 扩展应用。例如，Web 浏览器自定义的 DOM 组件，这个符合 W3C 标准的 DOM 组件就是通过宿主环境与 JavaScript 引擎进行联系的。Web 浏览器允许 JavaScript 引擎对其进行控制，并通过 DOM 组件实现对 HTML 或 XML 文档的操作。

（2）执行期环境

执行期环境是由宿主环境通过脚本引擎创建的，实际上就是由 JavaScript 引擎创建的一个代码解析初始化环境。初始化内容主要包括：

- 一套与宿主环境相联系的规则。
- JavaScript 引擎内核（基本语法规范、逻辑、命令和算法）。
- 一组内置对象和 API。
- 其他约定。

当然，不同的 JavaScript 引擎定义的初始化环境是不同的，这就形成了所谓的浏览器兼容性问题，因为不同的浏览器使用不同的 JavaScript 引擎。

建议 166：掌握 JavaScript 预编译过程

JavaScript 解析过程可以分为编译和执行两个阶段。编译就是常说的 JavaScript 预处理

（即预编译）。在预编译期，JavaScript 解释器将完成对 JavaScript 代码的预处理，也就是说，把 JavaScript 脚本代码转换成字节码。在执行期，JavaScript 解释器借助执行期环境将字节码生成机械码，并按顺序执行，完成程序设计的任务。

JavaScript 是一种解释型语言，而不是编译型语言。所谓解释型语言，就是代码在执行时才被解释器逐行动态编译和执行，而不是在执行之前就完成编译。而编译型语言是先编译后执行，两者的操作过程不同。当程序被编译时，需要一个被称为编译器的程序来完成所有工作。一般编译器可以包括下面一些组件（如图 9.2 所示）。

- 符号表：其中存储所有的符号及其信息，如类型、范围等。
- 词法分析器：其功能是将字符流（即脚本字符串）转换为记号（如关键词、操作符等）。
- 语法分析器：其功能是读取记号流，并建立语法树。
- 语义检查器：用来检查语法树的语义错误。
- 中间代码生成器：用来把语法树转换为中间代码。
- 代码优化器：用来优化中间代码。
- 代码生成器：用来将中间代码生成二进制字节码。

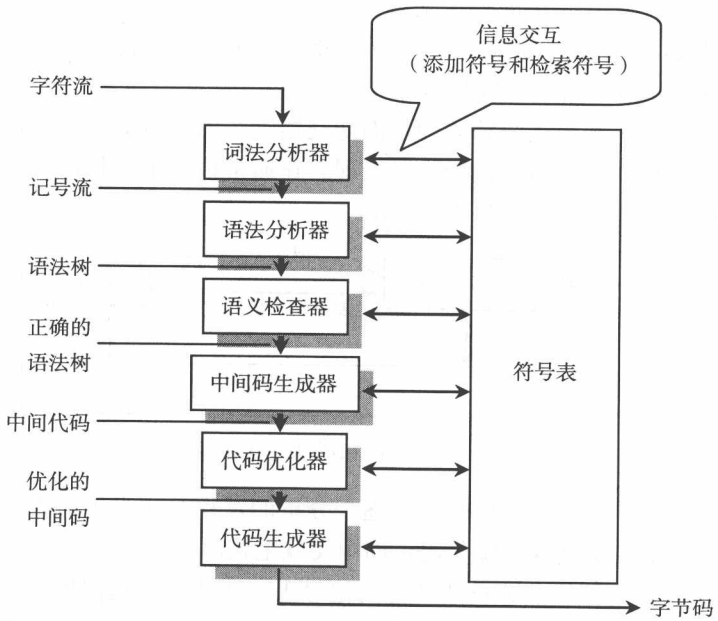


图 9.2　编译器的构成和工作流程示意图

编译程序的一般步骤分为：词法分析、语法分析、语义检查、代码优化和生成字节码。但是，对于 JavaScript 这类解释型语言来说，通过词法分析和语法分析，并建立语法树之后，就开始解释执行了，而不是在完全生成字节码之后，再调用虚拟机来执行这些编译好的字节码。在词法分析过程中，JavaScript 解释器先把脚本代码的字符流转换为记号流。例如，把

字符流：

```
a = (b - c);
```

转换为记号流：

```
NAME "a"
EQUALS
OPEN_PARENTHESIS
NAME "b"
MINUS
NAME "c"
CLOSE_PARENTHESIS
SEMICOLON
```

词法分析器是编译器中与源程序直接接触的部分，因此，词法分析器可以实现：

- 去掉注释，自动生成文档。
- 提供错误位置（可以通过记录行号来提供），当字符流变成词法记号流以后，就没有了行的概念。
- 完成预处理，如 C 语言中的宏定义等。

词法结构是 JavaScript 语言基础，至于词法分析的实现就比较复杂，这里就不再深入研究，读者只需要简单了解它的工作机制即可。

词法分析和语法分析不是完全独立的，而是交错进行的，也就是说，词法分析器不会在读取所有的词法记号后再使用语法分析器来处理。在通常情况下，每取得一个词法记号，就将其送入语法分析器进行分析（如图 9.3 所示）。

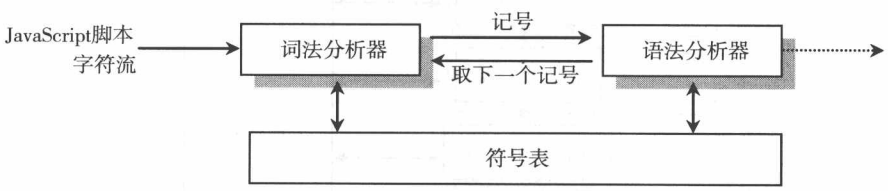


图 9.3 词法分析和语法分析示意图

词法分析是对 JavaScript 脚本代码进行逐一分析的过程，相当于语言翻译。例如，把英文逐词逐句地译成中文，英文就是源代码，而中文就是代码的记号。

语法分析的过程就是把词法分析所产生的记号生成语法树，通俗地说，就是把从程序中收集的信息存储到数据结构中。注意，在编译中用到的数据结构有两种：符号表和语法树。

- 符号表：就是在程序中用来存储所有符号的一个表，包括所有的字符串变量、直接量字符串，以及函数和类。
- 语法树：就是程序结构的一个树形表示，用来生成中间代码。

下面是一个简单的条件结构和输出信息代码段，被语法分析器转换为语法树之后，如图 9.4 所示。

```
if(typeof a == "undefined" ){
    a = 0;
}
else{
    a = a;
}
alert(a);
```

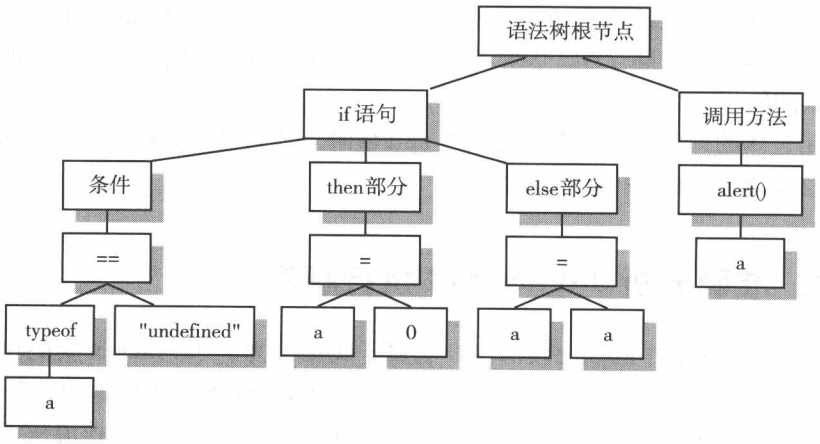


图 9.4　语法树结构示意图

如果 JavaScript 解释器在构造语法树的时候发现无法构造，就会报语法错误，并结束整个代码块的解析。对于传统强类型语言来说，在通过语法分析构造出语法树后，翻译出来的句子可能还会有模糊不清的地方，需要进一步的语义检查。语义检查的主要部分是类型检查。例如，函数的实参和形参类型是否匹配。但是，对于弱类型语言来说，就没有这一步。

经过编译阶段的准备，JavaScript 代码在内存中已经被构建为语法树，然后 JavaScript 引擎就会根据这个语法树结构边解释边执行。

在解释过程中，JavaScript 引擎是严格按照作用域机制来执行的。JavaScript 语法采用的是词法作用域，也就是说，JavaScript 的变量和函数作用域是在定义时决定的而不是在执行时决定的。由于词法作用域取决于源代码结构，因此 JavaScript 解释器只需要通过静态分析就能确定每个变量、函数的作用域，这种作用域也称为静态作用域。

当 JavaScript 解释器执行每个函数时，先创建一个执行环境，在这个虚拟环境中创建一个调用对象，在这个对象内存储当前域中所有局部变量、参数、嵌套函数、外部引用和父级引用列表 upvalue 等语法分析结构。

实际上，通过声明语句定义的变量和函数，在预编译期的语法分析中就已经存储到符号表中了，只要把它们与调用对象中的同名属性进行映射即可。调用对象的生命周期与函数的生命周期是一致的，在函数调用完毕且没有外部引用的情况下，调用对象会自动被 JavaScript 引擎当做垃圾进行回收。

另外，JavaScript 解释器通过作用域链把多个嵌套的作用域连在一起，并借助这个链条帮助 JavaScript 解释器检索变量的值。这个作用域链相当于一个索引表，通过编号来存储作用域的嵌套关系。JavaScript 解释器在检索变量的值时会按着这个索引编号进行快速查找，直到找到全局对象为止，如果没有找到，则传递一个特殊的 undefined 值。

如果函数引用了外部变量的值，则 JavaScript 解释器会为该函数创建一个闭包体。闭包体是一个完全封闭和独立的作用域，它不会在函数调用完毕后就被 JavaScript 引擎当做垃圾进行回收。闭包体可以长期存在，因此开发人员常把闭包体当做内存中的蓄水池，专门用于长期保存变量的值。

只有当闭包体的外部引用被全部设置为 null 值时，该闭包才会被回收。当然，也容易引发“垃圾泛滥”，甚至出现内存外溢现象。

建议 167：准确分析 JavaScript 执行顺序

JavaScript 引擎的工作机制比较深奥，它属于底层行为。JavaScript 代码执行顺序就比较形象了，用户可以直观感受到这种执行顺序，当然，JavaScript 代码的执行顺序是比较复杂的。HTML 文档在浏览器中的解析过程是这样的：浏览器按着文档流从上到下逐步解析页面结构和信息。JavaScript 代码作为嵌入的脚本也算做 HTML 文档的组成部分，因此，JavaScript 代码在装载时的执行顺序也是根据脚本标签 <script> 的出现顺序来确定的。

例如，浏览下面文档页面，所有代码都是从上到下被逐步解析的。

```
<script>
alert("顶部脚本");
</script>
<html><head>
<script>
alert("头部脚本");
</script>
<title></title>
</head>
<body>
<script>
alert("页面脚本");
</script>
</body></html>
<script>
alert("底部脚本");
</script>
```

对于通过脚本标签 <script> 的 src 属性导入的外部 JavaScript 文件脚本，它也将按照其语句出现的顺序来执行，而且执行过程是文档装载的一部分，不会因为是外部 JavaScript 文件而延期执行。例如，把上面文档中的头部和主体区域的脚本移到外部 JavaScript 文件中，然后通过 src 属性导入进来，继续预览页面文档，会看到相同的执行顺序。

```
<script>
alert("顶部脚本");
</script>
<html>
<head>
<script src="head.js"></script>
<title></title>
</head>
<body>
<script src="body.js"></script>
</body>
</html>
<script>
alert("底部脚本");
</script>
```

当 JavaScript 引擎解析脚本时，它会在预编译期对所有声明的变量和函数进行处理，因此，就会出现当 JavaScript 解释器执行下面脚本时不会报错的情况。

```
alert(a);  //undefined
var a =1;
alert(a);  //1
```

由于变量声明是在预编译期进行的，因此，在执行期间，变量对所有代码来说都是可见的。执行上面的代码，提示的值是 undefined，而不是 1，这是因为变量初始化过程发生在执行期，而不是预编译期。在执行期，JavaScript 解释器是按着代码先后顺序进行解析的，如果在前面代码行中没有为变量赋值，那么 JavaScript 解释器会使用默认值 undefined。由于在第二行中为变量 a 赋值了，因此在第三行代码中会提示变量 a 的值为 1，而不是 undefined。

同理，在下面示例中，在函数声明前调用函数也是合法的，并能够被正确解析，因此，返回值为 1。

```
f();        // 1
function f(){
    alert(1);
}
```

但是，如果按下面方式定义函数，那么 JavaScript 解释器会提示语法错误。

```
f();        //返回语法错误
var f = function(){
    alert(1);
}
```

在上面代码中定义的函数仅作为值赋值给变量 f，因此，在预编译期，JavaScript 解释器只能为声明变量 f 进行处理，而对于变量 f 的值，只能等到执行期按顺序进行赋值，自然就会出现语法错误，提示找不到对象 f。

虽然变量和函数声明可以在文档任意位置，但是良好的习惯应该是在所有 JavaScript 代码之前声明全局变量和函数，并且对变量进行初始化赋值。在函数内部也是先声明变量，然

后再引用。

同时，JavaScript 是分块执行代码的。所谓代码块就是使用 <script> 标签分隔的代码段。例如，下面两个 <script> 标签分别代表两个 JavaScript 代码块。

```
<script>
// JavaScript 代码块 1
var a =1;
</script>
<script>
// JavaScript 代码块 2
function f(){
    alert(1);
}
</script>
```

JavaScript 在执行脚本时是按块来执行的。浏览器在解析 HTML 文档流时，如果遇到一个 <script> 标签，则 JavaScript 会等到这个代码块都加载完后再对代码块进行预编译，然后再执行。执行完毕后，浏览器会继续解析下面的 HTML 文档流，同时 JavaScript 也准备好处理下一个代码块。

由于 JavaScript 是按块执行的，因此在一个 JavaScript 块中调用后面块中声明的变量或函数就会提示语法错误。例如，当 JavaScript 解释器执行下面代码时就会提示语法错误，显示变量 a 未定义，对象 f 找不到。

```
<script>
// JavaScript 代码块 1
alert(a);
f();
</script>
<script>
// JavaScript 代码块 2
var a =1;
function f(){
    alert(1);
}
</script>
```

虽然 JavaScript 是按块执行的，但是不同块都属于同一个全局作用域，也就是说，块之间的变量和函数是可以共享的。

由于 JavaScript 是按块处理代码，同时又遵循 HTML 文档流的解析顺序，因此在上面示例中会看到语法错误。但是，在文档流加载完毕后再次访问就不会出现这样的错误。例如，把访问第 2 个代码块中的变量和函数的代码放在页面初始化事件函数中，这样就不会出现语法错误了。

```
<script>
// JavaScript 代码块 1
window.onload = function(){          // 页面初始化事件处理函数
```

```
    alert(a);
    f();
}
</script>
<script>
// JavaScript 代码块 2
var a =1;
function f(){
    alert(1);
}
</script>
```

为了安全起见，一般在页面初始化完毕之后才允许 JavaScript 代码执行，这样就可以避免网速对 JavaScript 执行的影响。同时，也避开了 HTML 文档流对 JavaScript 执行的限制。

如果一个页面中存在多个 window.onload 事件处理函数，那么只有最后一个才是有效的，为了解决这个问题，可以把所有脚本或调用函数都放在同一个 onload 事件处理函数中。

```
window.onload = function(){
    f1();
    f2();
    f3();
}
```

通过这种方式还可以改变函数的执行顺序，方法是简单地调整 onload 事件处理函数中调用函数的排列顺序。

除了页面初始化事件外，还可以通过各种交互事件来改变 JavaScript 代码的执行顺序，如鼠标事件、键盘事件，以及时钟触发器等方法。

在 JavaScript 开发中，经常会使用 document 对象的 write() 方法输出 JavaScript 脚本。

```
document.write('<script type="text/javascript">');
document.write('f();    ');
document.write('function f(){    ');
document.write('    alert(1);    ');
document.write('}    ');
document.write('<\/script>    ');
```

运行以上代码，document.write() 方法先把输出的脚本字符串写入到脚本所在的文档位置，浏览器在解析完 document.write() 所在文档的内容后，继续解析 document.write() 输出的内容，然后按顺序解析后面的 HTML 文档。也就是说，JavaScript 脚本输出的代码字符串会在输出后马上被执行。

使用 document.write() 方法输出的 JavaScript 脚本字符串必须放在同时输出的 <script> 标签中，否则 JavaScript 解释器会因为不能识别这些合法的 JavaScript 代码而作为普通的字符串显示在页面文档中。例如，下面的代码就会把 JavaScript 代码显示出来，而不是执

行它。

```
document.write('f();    ');
document.write('function f(){    ');
document.write('    alert(1);    ');
document.write('};    ');
```

但是，通过 document.write() 方法输出并执行脚本也存在一定的风险，因为不同 JavaScript 引擎对其执行顺序不同，同时不同浏览器在解析时也会出现 Bug。

❑ 找不到通过 document.write() 方法导入的外部 JavaScript 文件中声明的变量或函数，例如：

```
document.write('<script type="text/javascript" src="test.js"><\/script>');
document.write('<script type="text/javascript">    ');
document.write('alert(n);    ');    // IE 提示找不到变量 n
document.write('<\/script>    ');
alert(n+1);                          // 所有浏览器都会提示找不到变量 n
```

外部 JavaScript 文件（test.js）的代码如下：

```
var n = 1;
```

分别在不同浏览器中进行测试，均会发现提示语法错误，找不到变量 n。也就是说，如果在 JavaScript 代码块中访问本代码块使用 document.write() 方法输出的脚本导入的外部 JavaScript 文件所包含的变量，会显示语法错误。同时，如果在 IE 中，不仅在脚本中，在输出的脚本中也会提示找不到输出的导入外部 JavaScript 文件的变量（表述有点长且有点拗口，不懂的读者尝试运行上面代码即可明白）。

❑ 不同 JavaScript 引擎对输出的外部导入脚本的执行顺序略有不同，例如：

```
<script type=" text/javascript" >
document.write('<script type="text/javascript" src="test1.js"><\/script>');
document.write('<script type="text/javascript">    ');
document.write('alert(2);')
document.write('alert(n+2);');
document.write('<\/script>');
</script>
<script type="text/javascript">
alert(n+3);
</script>
```

外部 JavaScript 文件（test1.js）的代码如下：

```
var n = 1;
alert(n);
```

在 IE 中的执行顺序如图 9.5 所示。

在符合 DOM 标准的浏览器中的执行顺序与 IE 不同，并且没有语法错误。在 Firefox 浏览器中的执行顺序如图 9.6 所示。

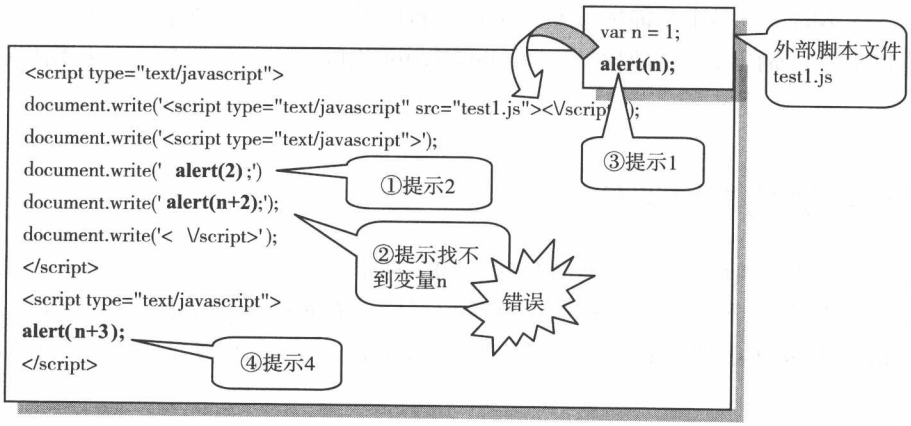


图 9.5　在 IE 中的执行顺序和提示的语法错误

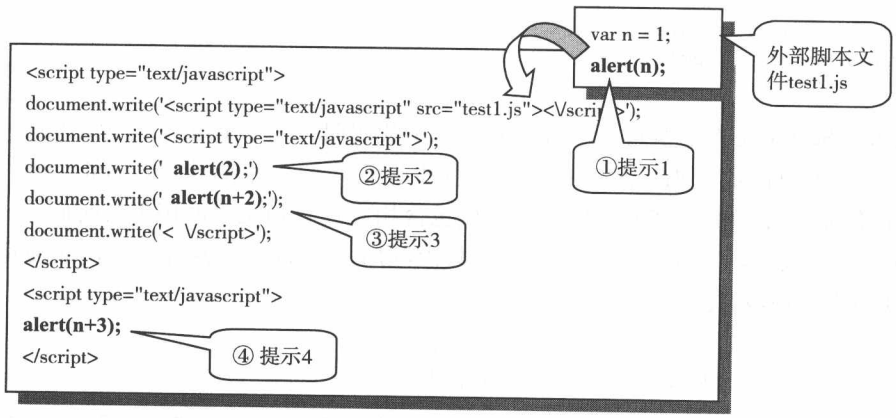


图 9.6　在 Firefox 浏览器中的执行顺序

可以把凡是使用输出脚本导入的外部文件，都放在独立的代码块中，这样根据上面介绍的 JavaScript 代码块执行顺序，就可以解决不同浏览器的不同执行顺序的问题，以及可能存在的 Bug。例如，针对上面示例，可以这样设计：

```
<script type="text/javascript">
document.write('<script type="text/javascript" src="test1.js"><\/script>');
</script>
<script type="text/javascript">
document.write('<script type="text/javascript">    ');
document.write('alert(2);    ') ;          // 提示 2
document.write('alert(n+2);    ');         // 提示 3
document.write('<\/script>    ');
alert(n+3);                                // 提示 4
</script>
<script type="text/javascript">
alert(n+4);                                // 提示 5
</script>
```

这样，在不同浏览器中都能够按顺序执行上面代码，并且输出顺序都是 1、2、3、4 和 5。存在问题的原因：输出导入的脚本与当前 JavaScript 代码块之间的矛盾。单独输出就不会发生冲突了。

建议 168：避免二次评估

避免二次评估是实现 JavaScript 性能优化的一种措施。与许多脚本语言一样，JavaScript 允许在程序中获取一个包含代码的字符串后运行它，有 4 种标准函数可以实现这一过程：eval_r()、Function() 构造器、setTimeout() 和 setInterval()。每个函数允许传入一串 JavaScript 代码后运行它。

```
var num1 = 5, num2 = 6,
result = eval_r("num1 + num2"),
sum = new Function("arg1", "arg2", "return arg1 + arg2");
setTimeout("sum = num1 + num2", 100);
setInterval("sum = num1 + num2", 100);
```

当在 JavaScript 代码中执行另一段 JavaScript 代码时，将会付出二次评估的代价。以上代码首先被评估为正常代码，然后在执行过程中运行字符串中的代码，此时发生另一次评估。二次评估是一项代价较高的操作，与直接包含相应代码相比将占用更长时间。

作为一个比较点，不同浏览器访问一个数组项所占用的时间各有不同，而使用 eval_r() 访问所占用的时间差别更大。

```
var item = array[0];              // 比较快
var item = eval_r("array[0]");    // 比较慢
```

如果使用 eval_r() 代替直接代码访问 10 000 个数组项，那么在不同浏览器上将会有非常巨大的差异。

在访问数组项时间上存在巨大差异，原因是每次调用 eval_r() 时要创建一个新的解释 / 编译实例。同样的过程也发生在 Function()，setTimeout() 和 setInterval() 上，自动使代码执行速度变慢。

在大多数情况下，没必要使用 eval_r() 或 Function()，如果可能，尽量避免使用它们。至于另外两个函数，setTimeout() 和 setInterval()，建议通过第一个参数传入一个函数而不是一个字符串，例如：

```
setTimeout(function() {
    sum = num1 + num2;
}, 100);
setInterval(function() {
    sum = num1 + num2;
}, 100);
```

建议 169：建议使用直接量

在 JavaScript 中有多种方法创建对象和数组，但没有什么比创建对象和数组直接量更快的了。如果不使用直接量，那么典型的对象创建和赋值如下：

```
// 创建对象
var myObject = new Object();
myObject.name = "Nicholas";
myObject.count = 50;
myObject.flag = true;
myObject.pointer = null;
// 创建数组
var myArray = new Array();
myArray[0] = "Nicholas";
myArray[1] = 50;
myArray[2] = true;
myArray[3] = null;
```

虽然在技术上这种做法没有什么不对，但是直接量赋值更快，在代码中占用空间较少，整个文件尺寸可以更小。上面的代码可用直接量重写：

```
// 创建对象
var myObject = {
    name: "Nicholas",
    count: 50,
    flag: true,
    pointer: null
};
// 创建数组
var myArray = ["Nicholas", 50, true, null];
```

上面代码与前面代码的效果相同，但在几乎所有浏览器上运行更快。随着对象属性和数组项数量的增加，使用直接量的速度优势会更加明显。

建议 170：不要让 JavaScript 引擎重复工作

在进行 JavaScript 性能优化时，需要注意两件事：不要做不必要的工作，不要重复做已经完成的工作。不要做不必要的工作就要重构代码，降低冗余率。不要重复做已经完成的工作通常难以确定，因为工作可能因为各种原因而在很多地方被重复。

也许最常见的重复工作是浏览器检测。大量代码依赖于浏览器的功能。以事件句柄的添加和删除为例，典型的跨浏览器代码如下：

```
function addHandler(target, eventType, handler) {
    if(target.addEventListener) {//DOM2 Events
        target.addEventListener(eventType, handler, false);
    } else {//IE
```

```
            target.attachEvent("on" + eventType, handler);
        }
    }
    function removeHandler(target, eventType, handler) {
        if(target.removeEventListener) {//DOM2 Events
            target.removeEventListener(eventType, handler, false);
        } else {//IE
            target.detachEvent("on" + eventType, handler);
        }
    }
```

在上面代码中，通过测试 addEventListener() 和 removeEventListener() 来检查 DOM2 的事件支持情况，它能够被除 IE 之外的所有现代浏览器所支持。如果这些方法不存在于 target 中，那么就认为当前浏览器是 IE，并且使用 IE 特有的方法。

这两个函数隐藏着性能问题：每次函数调用时都执行重复工作，检查某种方法是否存在。在每次调用中重复同样的工作是一种浪费。假设 target 的唯一值就是 DOM 对象，而且用户不可能在页面加载时改变浏览器，如果在调用 addHandler() 时首先调用了 addEventListener()，那么每个后续调用都要调用 addEventListener()。

（1）延迟加载

延迟加载就是在信息被使用之前不做任何工作。例如，在上面示例中不需要判断使用哪种方法附加或分离事件句柄，直到函数被调用。

```
function addHandler(target, eventType, handler) {
    if(target.addEventListener) {//DOM2 Events
        addHandler = function(target, eventType, handler) {
            target.addEventListener(eventType, handler, false);
        };
    } else {//IE
        addHandler = function(target, eventType, handler) {
            target.attachEvent("on" + eventType, handler);
        };
    }
    addHandler(target, eventType, handler);
}
function removeHandler(target, eventType, handler) {
    if(target.removeEventListener) {//DOM2 Events
        removeHandler = function(target, eventType, handler) {
            target.addEventListener(eventType, handler, false);
        };
    } else {//IE
        removeHandler = function(target, eventType, handler) {
            target.detachEvent("on" + eventType, handler);
        };
    }
    removeHandler(target, eventType, handler);
}
```

这两个函数在第一次被调用时检查一次并决定使用哪种方法附加或分离事件句柄，然后

原始函数就被包含适当操作的新函数覆盖了，最后调用新函数并将原始参数传给它。以后再调用 addHandler() 或 removeHandler() 时不会再次检测，因为检测代码已经被新函数覆盖了。

调用一个延迟加载函数总是在第一次时需要较长时间，因为必须在运行检测后调用另一个函数以完成任务。后续调用同一函数将快很多，因为不再执行检测了。延迟加载适用于函数不会在页面上立即用到的场合。

（2）条件预加载

条件预加载就是在脚本加载之前提前进行检查，而不用等待函数调用。这样的检测仍然只进行一次，但在此过程中来得更早，例如：

```
    var addHandler = document.body.addEventListener ? function(target, eventType,
handler) {
        target.addEventListener(eventType, handler, false);
    } : function(target, eventType, handler) {
        target.attachEvent("on" + eventType, handler);
    };
    var removeHandler = document.body.removeEventListener ? function(target,
eventType, handler) {
        target.removeEventListener(eventType, handler, false);
    } : function(target, eventType, handler) {
        target.detachEvent("on" + eventType, handler);
    };
```

在上面代码中，先检查 addEventListener() 和 removeEventListener() 是否存在，然后根据信息指定最合适的函数。三元操作符返回 DOM 级别 2 的函数，如果它们不存在，则返回 IE 特有的函数。虽然检测功能提前了，但是接下来调用 addHandler() 和 removeHandler() 同样很快。条件预加载确保所有函数调用时间相同，其代价是在脚本加载时进行检测。预加载适用于一个函数马上就会被用到且在整个页面生命周期中经常会被使用的场合。

建议 171：使用位操作符执行逻辑运算

JavaScript 引擎由低级语言构建，在处理过程中它的执行速度是最快的。在 JavaScript 中，位操作运算符经常被误解，很少被开发者使用，同时经常在布尔表达式中被误用。尽管位操作运算符具有优势，还是会导致在 JavaScript 开发中不能经常使用它们。

JavaScript 中的数字按照 IEEE-754 标准 64 位格式存储。在位运算中，数字被转换为有符号 32 位格式。每种操作均直接在这个 32 位数上实现结果。尽管需要转换，这个过程与 JavaScript 中其他数学和布尔运算相比还是非常快的。

如果对数字的二进制表示法不熟悉，那么使用 JavaScript 可以很容易地将数字转换为字符串形式的二进制表达式，通过调用 toString() 方法传入数字 2，例如：

```
var num1 = 25,
num2 = 3;
```

```
alert(num1.toString(2));          //"11001"
alert(num2.toString(2));          // "11"
```

该表达式消隐了数字高位的零。JavaScript 中有 4 种位逻辑操作符。

- AND（位与）：只有两个操作数的位都是 1，结果才是 1。
- OR（位或）：有一个操作数的位是 1，结果就是 1。
- XOR（位异或）：两个操作数的位中只有一个是 1，结果才是 1。
- NOT（位非）：遇 0 返回 1，遇 1 返回 0。

具体用法如下：

```
var result1 = 25 & 3;             //1
alert(result.toString(2));         //"1"
var result2 = 25 | 3;             //27
alert(resul2.toString(2));        //"11011"
var result3 = 25^3;               //26
alert(resul3.toString(2));        //"11000"
var result = ~25;                 //-26
alert(resul2.toString(2));        //"-11010"
```

有许多方法可以通过位运算符提高 JavaScript 的速度。首先可以用位运算符替代纯数学操作。例如，通常采用对 2 取模运算实现颜色交替显示：

```
for(var i = 0, len = rows.length; i < len; i++) {
    if(i%2) {
        className = "even";
    } else {
        className = "odd";
    }
    //...
}
```

在上面代码中，计算某个数对 2 取模，需要用这个数除以 2 然后查看余数。对 32 位数字的底层（二进制）表示法进行分析会发现，偶数的最低位是 0，奇数的最低位是 1。如果此数为偶数，那么它和 1 进行位“与”操作的结果就是 0；如果此数为奇数，那么它和 1 进行位“与”操作的结果就是 1。也就是说，上面的代码可以重写如下：

```
for(var i = 0, len = rows.length; i < len; i++) {
    if(i & 1) {
        className = "odd";
    } else {
        className = "even";
    }
    //...
}
```

虽然代码改动不大，但是位“与”运算的速度比原始代码提升了约 50%。

将使用位操作的技术称为位掩码。位掩码在计算机科学中是一种常用的技术，可用于

同时判断多个布尔选项，快速地将数字转换为布尔标志数组。掩码中每个选项的值都是 2 的幂。例如：

```
var OPTION_A = 1;
var OPTION_B = 2;
var OPTION_C = 4;
var OPTION_D = 8;
var OPTION_E = 16;
```

通过定义这些选项，可以用位“或”操作创建一个数字来包含多个选项：

```
var options = OPTION_A | OPTION_C | OPTION_D;
```

使用位“与”操作检查一个给定的选项是否可用。如果该选项未设置，那么运算结果为 0；如果设置了该选项，那么运算结果为 1。

```
if(options & OPTION_A) {
    // 执行代码
}
if(options & OPTION_B) {
    // 执行代码
}
```

上面代码中的位掩码操作非常快，原因前面提到过，因为操作发生在系统底层。对于许多选项保存在一起并经常检查的情况，位掩码有助于提高整体性能。

建议 172：推荐使用原生方法

无论怎样优化 JavaScript 代码，永远都不会比 JavaScript 引擎提供的原生方法更快。其原因是 JavaScript 的原生部分在执行代码之前已经存在于浏览器中，原生方法都是用原生语言写的，如 C++。这意味着这些方法被编译成机器码，作为浏览器的一部分，没有 JavaScript 代码的限制多。

经验不足的 JavaScript 开发者经常犯的一个错误是在代码中进行复杂的数学运算，却没有使用内置 Math 对象中那些性能更好的方法。Math 对象包含专门设计的属性和方法，使数学运算更容易。使用 JavaScript 内置函数比同样功能的 JavaScript 代码更快。

另外，借助各种框架或自定义函数可以像 CSS 选择器那样查询 DOM 文档。CSS 查询被 JavaScript 原生实现并由 jQuery 这个 JavaScript 库来推广。虽然 jQuery 引擎被认为是最快的 CSS 查询引擎，但是它仍比原生方法慢。原生的 querySelector() 和 querySelectorAll() 方法在完成它们的任务时，平均只需要基于 JavaScript 的 CSS 查询的时间的 10%。大多数 JavaScript 库已经使用了原生函数来提高整体性能。

当原生方法可用时，尽量使用它们，尤其是数学运算和 DOM 操作。用编译后的代码做越多的事情，代码就会越快。

建议 173：编写无阻塞 JavaScript 脚本

管理 JavaScript 代码是一个棘手的问题，因为执行代码阻塞了其他处理过程，如用户界面绘制。当 JavaScript 引擎遇到 <script> 标签时，页面必须停下来等待下载和执行代码（如果是外部的），然后再继续处理页面其他部分。但是，有几种方法可以减少这种操作对性能的影响。

- 将所有 <script> 标签放置在页面的底部，紧靠 body 关闭标签 </body> 的上方，这样可以保证页面在脚本运行之前完成解析。
- 将脚本分组打包。页面的 <script> 标签越少，页面的加载速度就越快，响应也更加迅速。不论外部脚本文件，还是内联代码都应该如此。

也可以使用非阻塞方式下载 JavaScript。

- 为 <script> 标签添加 defer 属性（只适用于 IE 和 Firefox 3.5 以上版本）。
- 动态创建 <script> 元素，并且用它下载并执行代码。
- 用 XHR 对象下载代码，并且将其注入到页面中。

通过使用上述策略，可以极大地提高那些大量使用 JavaScript 代码的网页性能。

JavaScript 在浏览器中的性能，是开发者所要面对的最重要的可用性问题。此问题因 JavaScript 的阻塞特征而变得复杂。大多数浏览器使用单进程处理 UI 更新和 JavaScript 运行等多个任务，并且同一时间只能执行一个任务。当 JavaScript 运行时而其他的事情不能被浏览器处理时，JavaScript 运行了多长时间，在浏览器空闲之后响应用户输入之前的等待时间就有多长。

因此，<script> 标签的出现使整个页面因脚本解析和运行而出现等待。不论实际的 JavaScript 代码是内联的还是包含在一个不相干的外部文件中的，页面下载和解析过程必须停下，等待脚本完成这些处理才能继续。这是页面生命周期必不可少的部分，因为脚本可能在运行过程中修改页面内容。

例如，最典型的是 document.write() 函数。

```
<html>
<head>
<title>Script Example</title>
</head>
<body>
<p>
<script type="text/javascript">
document.write((new Date()).toDateString());
</script>
</p>
</body>
</html>
```

当浏览器遇到一个 <script> 标签时，无法预知 JavaScript 是否在 <p> 标签中添加内容，

因此，浏览器会停下来，运行此 JavaScript 代码，然后再继续解析和翻译页面。同样的事情也发生在使用 src 属性加载 JavaScript 的过程中，浏览器必须首先下载外部文件的代码，然后解析并运行此代码。这个过程会占用一些时间，使页面解析和用户交互完全阻塞。

<script> 标签用于加载外部 JavaScript 文件，该标签可以放在 HTML 文档的 <head> 或 <body> 标签中，可以在其中多次出现。除此类代码外，<head> 部分还包含 <link> 标签用于加载外部 CSS 文件等。因此，最好把样式和行为所依赖的部分放在一起，先加载它们，使页面可以得到正确的外观和行为，例如：

```
<html>
<head>
<title></title>
<script type="text/javascript" src="file1.js"></script>
<script type="text/javascript" src="file2.js"></script>
<script type="text/javascript" src="file3.js"></script>
<link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
<p> </p>
</body>
</html>
```

虽然这些代码看起来是正确的，但是它们确实存在性能问题：在 <head> 部分加载了 3 个 JavaScript 文件。由于每个 <script> 标签阻塞了页面的解析过程，因此直到它完整地下载并运行了外部 JavaScript 代码之后，页面处理才能继续进行。如果脚本文本很大，用户会察觉到延迟。

浏览器在解析 <body> 标签之前，不会渲染页面的任何部分。用这种方法把脚本放到页面的顶端，将导致一个可以察觉的延迟：在页面打开之前，显示为一幅空白的页面，此时用户既不能阅读，也不能与页面进行交互操作。

在上面代码中，第一个 JavaScript 文件开始下载并阻塞了其他文件的下载过程。接下来，在 file1.js 下载完之后和 file2.js 开始下载之前有一个延时，这是 file1.js 完全运行所需的时间。每个文件必须等待前一个文件下载完成并运行完之后，才能开始自己的下载过程。当这些文件正在下载时，用户面对一个空白的屏幕。这是大多数浏览器的行为模式。

一般读者希望 <script> 标签正在下载外部资源时，不必阻塞其他 <script> 标签。不幸的是，JavaScript 的下载仍然要阻塞其他资源（比如图片）的下载过程。即使脚本之间的下载过程互不阻塞，页面仍要等待所有 JavaScript 代码下载并执行完之后才能继续。因此，在浏览器允许并行下载提高性能之后，该问题并没有完全解决。脚本阻塞仍然是一个问题。

因为脚本阻塞其他页面资源的下载过程，所以推荐的办法为：将所有 <script> 标签放在尽可能接近 <body> 标签底部的位置，尽量减少对整个页面下载的影响。

```
<html>
<head>
```

```
<title> </title>
<link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
<p> </p>
<script type="text/javascript" src="file1.js"></script>
<script type="text/javascript" src="file2.js"></script>
<script type="text/javascript" src="file3.js"></script>
</body>
</html>
```

上面代码展示了 <script> 标签在 HTML 文件中的推荐位置。尽管脚本下载之间互相阻塞，不过页面已经下载完成并显示在用户面前了，进入页面的速度不会太慢。

由于每个 <script> 标签下载时会阻塞页面解析过程，因此限制页面的 <script> 总数也可以改善性能。这个规则对内联脚本和外部脚本同样适用。每当页面解析碰到一个 <script> 标签时，紧接着有一段时间用于代码执行。最小化这些延迟时间可以改善页面的整体性能。

这个问题与外部 JavaScript 文件处理过程略有不同。每个 HTTP 请求都会产生额外的性能负担，下载一个 100KB 的文件比下载 4 个 25KB 的文件要快。总之，减少引用外部脚本文件的数量可以降低性能损耗。在一个大型网站或网页应用需要多次请求 JavaScript 文件时，可以将这些文件整合成一个文件，只需要一个 <script> 标签引用，就可以减少性能损失。

建议 174：使脚本延迟执行

JavaScript 会阻塞浏览器的某些处理过程，如 HTTP 请求和界面刷新，这是开发者面临的最重要的性能问题。保持 JavaScript 文件短小，并限制 HTTP 请求的数量，只是创建反应迅速的网页应用的第一步。一个应用程序所包含的功能越多，所需要的 JavaScript 代码就越大，保持源代码短小并不总是最佳选择。尽管下载一个大 JavaScript 文件只产生一次 HTTP 请求，还是会锁定浏览器一大段时间。为了避开这种情况，需要逐步添加 JavaScript。

（1）非阻塞脚本技巧

等页面完成加载之后，再加载 JavaScript 源代码。这意味着在 window 的 load 事件发出之后开始下载 JavaScript 源代码。

HTML 4 为 <script> 标签定义了一个扩展属性：defer。这个 defer 属性指明元素中所包含的脚本暂时不修改 DOM，因此代码可以稍后执行。defer 属性只被 IE 4 和 Firefox 3.5 及其更高版本的浏览器所支持，它不是一个理想的跨浏览器解决方案。在其他浏览器上，defer 属性被忽略，<script> 标签按照默认方式处理，这样又会造成阻塞。如果浏览器支持 defer 属性，那么这种方法仍是一种有用的解决方案。

```
<script type="text/javascript" src="file1.js" defer></script>
```

一个带有 defer 属性的 <script> 标签可以放置在文档的任何位置，对应的 JavaScript 文

件将在 <script> 被解析时启动下载，但直到 DOM 加载完成，也就是在 load 事件被调用之前代码不会被执行。

当一个 defer 的 JavaScript 文件被下载时，由于它不会阻塞浏览器的其他处理过程，所以这些文件可以与页面的其他资源一起并行下载。

任何带有 defer 属性的 <script> 元素在 DOM 加载完成之前不会被执行，不论是内联脚本还是外部脚本文件。例如，下面代码展示了 defer 属性如何影响脚本行为。

```
<html>
<head>
<title> </title>
</head>
<body>
<script defer>alert("defer");</script>
<script>alert("script");</script>
<script>
window.onload = function(){
    alert("load");
};
</script>
</body>
</html>
```

这些代码在页面处理过程中弹出 3 个对话框。如果浏览器不支持 defer 属性，那么弹出对话框的顺序是 defer、script 和 load。如果浏览器支持 defer 属性，那么弹出对话框的顺序是 script、defer 和 load。

注意：标记为 defer 的 <script> 元素不是跟在第二个 <script> 后面运行，而是在 load 事件处理之前被调用。

（2）动态脚本加载

如果用户浏览器只包括 IE 和 Firefox，那么 defer 脚本确实有用。如果需要支持跨领域的多种浏览器，那么还有与 defer 更一致的实现方式：动态脚本加载。动态脚本加载是非阻塞 JavaScript 下载中最常用的模式，因为它可以跨浏览器，而且简单易用。

文档对象模型（DOM）允许使用 JavaScript 动态创建 HTML 文档内容。<script> 元素与页面其他元素没有什么不同：引用变量可以通过 DOM 进行检索，可以从文档中移动、删除，也可以被创建。一个新的 <script> 元素可以非常容易地通过标准 DOM 函数创建。

```
var script = document.createElement ("script");
script.type = "text/javascript";
script.src = "file1.js";
document.getElementsByTagName_r("head")[0].appendChild(script);
```

以上代码通过新的 <script> 元素加载 file1.js 源文件。此文件在元素被添加到页面之后立刻开始下载。这样无论在何处启动下载，文件的下载和运行都不会阻塞其他页面处理过

程，甚至可以将这些代码放在 <head> 部分也不会对其余部分的页面代码造成影响，下载文件的 HTTP 连接的情况除外。

当文件使用动态脚本节点下载时，返回的代码通常立即执行（但 Firefox 和 Opera 将等待此前的所有动态脚本节点执行完毕）。当脚本是自运行类型时这一机制运行正常，如果脚本只包含供页面其他脚本调用的接口，则会带来问题，在这种情况下，需要跟踪脚本下载完成并准备妥善的情况。可以使用动态 <script> 节点发出事件得到相关信息。

Firefox、Opera、Chrome 和 Safari 都会在 <script> 节点接收完成之后发出一个 load 事件，这样可以监听 <script> 标签的 load 事件，以获取脚本准备好的通知。

```
var script = document.createElement ("script")
script.type = "text/javascript";
//Firefox、Opera、Chrome、Safari 3+
script.onload = function(){
    alert("Script loaded!");
};
script.src = "file1.js";
document.getElementsByTagName_r("head")[0].appendChild(script);
```

IE 不支持标签的 load 事件，却支持另一种实现方式，它会发出一个 readystatechange 事件。<script> 元素有一个 readyState 属性，它的值随着下载外部文件的过程而改变。readyState 有 5 种取值：

- ❑ uninitialized，默认状态。
- ❑ loading，下载开始。
- ❑ loaded，下载完成。
- ❑ interactive，下载完成但尚不可用。
- ❑ complete，所有数据已经准备好。

在 <script> 元素的生命周期中，readyState 的这些取值不一定全部出现，也并没有指出哪些取值总会被用到。不过在实践中 loaded 和 complete 状态值很重要。在 IE 中这两个 readyState 值所表示的最终状态并不一致，有时 <script> 元素会得到 loader，却从不出现 complete，而在另外一些情况下出现 complete 而用不到 loaded。最安全的办法就是在 readystatechange 事件中检查这两种状态，并且当其中一种状态出现时，删除 readystatechange 事件句柄，保证事件不会被处理两次。

```
var script = document.createElement ("script")
script.type = "text/javascript";
script.onreadystatechange = function(){  //IE
     if (script.readyState == "loaded" || script.readyState == "complete"){
         script.onreadystatechange = null;
         alert("Script loaded.");
     }
};
script.src = "file1.js";
```

```
document.getElementsByTagName_r("head")[0].appendChild(script);
```

下面的函数封装了标准实现和 IE 实现所需的功能：

```
function loadScript(url, callback) {
    var script = document.createElement("script")
    script.type = "text/javascript";
    if(script.readyState) {//IE
        script.onreadystatechange = function() {
            if(script.readyState == "loaded" || script.readyState == "complete") {
                script.onreadystatechange = null;
                callback();
            }
        };
    } else {// 其他浏览器
        script.onload = function() {
            callback();
        };
    }
    script.src = url;
    document.getElementsByTagName_r("head")[0].appendChild(script);
}
```

上面的封装函数接收两个参数：JavaScript 文件的 URL 和当 JavaScript 接收完成时触发的回调函数。属性检查用于决定监视哪种事件。最后设置 src 属性，并将 <script> 元素添加至页面。此 loadScript() 函数的使用方法如下：

```
loadScript("file1.js", function(){
    alert("File is loaded!");
});
```

可以在页面中动态加载很多 JavaScript 文件，只是要注意，浏览器不保证文件加载的顺序。在所有主流浏览器之中，只有 Firefox 和 Opera 保证脚本按照指定的顺序执行，其他浏览器将按照服务器返回次序下载并运行不同的代码文件。可以将下载操作串联在一起以保证它们的次序：

```
loadScript("file1.js", function() {
    loadScript("file2.js", function() {
        loadScript("file3.js", function() {
            alert("All files are loaded!");
        });
    });
});
```

此代码待 file1.js 可用之后才开始加载 file2.js，待 file2.js 可用之后才开始加载 file3.js。虽然此方法可行，但是如果要下载和执行的文件很多，还是有些麻烦。如果多个文件的次序十分重要，那么更好的办法是将这些文件按照正确的次序连接成一个文件。独立文件可以一次性下载所有代码，由于这是异步执行，因此使用一个大文件并没有什么损失。

建议 175：使用 XHR 脚本注入

使用 XMLHttpRequest（XHR）对象将脚本注入到页面中，以非阻塞方式获得脚本。这种方法需要首先创建一个 XHR 对象，然后下载 JavaScript 文件，接着用一个动态 <script> 元素将 JavaScript 代码注入页面。下面是一个简单的例子：

```
var xhr = new XMLHttpRequest();
xhr.open("get", "file1.js", true);
xhr.onreadystatechange = function() {
    if(xhr.readyState == 4) {
        if(xhr.status >= 200 && xhr.status < 300 || xhr.status == 304) {
            var script = document.createElement("script");
            script.type = "text/javascript";
            script.text = xhr.responseText;
            document.body.appendChild(script);
        }
    }
};
xhr.send(null);
```

在上面代码中，先向服务器发送一个获取 file1.js 文件的 GET 请求。onreadystatechange 事件处理函数检查 readyState 是不是 4，然后检查 HTTP 状态码是不是有效（大于或等于 200 表示有效的回应，304 表示一个缓存响应）。如果收到了一个有效的响应，那么就创建一个新的 <script> 元素，将它的文本属性设置为从服务器接收到的 responseText 字符串。这样做实际上会创建一个带有内联代码的 <script> 元素。一旦新 <script> 元素被添加到文档，代码将被执行，并准备使用。

这种方法的一个优点是可以下载不立即执行的 JavaScript 代码。由于代码返回在 <script> 标签之外，它下载后不会自动执行，这使得可以推迟执行，直到一切都准备好了。另一个优点是，同样的代码在所有现代浏览器中都不会引发异常。

这种方法的主要缺点：JavaScript 文件必须与页面放置在同一个域内。正因为如此，大型网站通常不采用 XHR 脚本注入技术。

建议 176：推荐最优化非阻塞模式

如果向页面加载大量 JavaScript，则推荐的方法可分为两个步骤：

第 1 步，动态加载 JavaScript 所需的代码。

第 2 步，加载页面初始化所需的除 JavaScript 之外的部分。这部分代码要尽量小，可能只包含 loadScript() 函数，下载和运行非常快，不会对页面造成很大干扰。当初始代码准备好之后，用它来加载其余的 JavaScript。

```
<script type="text/javascript" src="loader.js"></script>
```

```
<script type="text/javascript">
loadScript("the-rest.js", function(){
    Application.init();
});
</script>
```

在上面代码中，将脚本放置在 body 的关闭标签 </body> 之前。这样做有几点好处：首先，可以确保 JavaScript 运行不会影响页面其他部分显示；其次，当第二部分 JavaScript 文件完成下载时，所有应用程序所必须的 DOM 已经创建好了，并且做好被访问的准备，避免使用额外的事件处理来得知页面是否已经准备好了。

向页面加载大量 JavaScript 的另一个方法是直接将 loadScript() 函数嵌入在页面中，这可以避免另一次 HTTP 请求，例如：

```
<script type="text/javascript">
    function loadScript(url, callback) {
        var script = document.createElement("script")
        script.type = "text/javascript";
        if(script.readyState) {//IE
            script.onreadystatechange = function() {
                if(script.readyState == "loaded" || script.readyState == "complete") {
                    script.onreadystatechange = null;
                    callback();
                }
            };
        } else {// 其他浏览器
            script.onload = function() {
                callback();
            };
        }
        script.src = url;
        document.getElementsByTagName_r("head")[0].appendChild(script);
    }
    loadScript("the-rest.js", function() {
        Application.init();
    });
</script>
```

建议 177：避免深陷作用域访问

JavaScript 代码的解释执行在进入函数内部时会预先分析当前的变量，并且将这些变量归入不同的层级。局部变量放入层级 1（浅），全局变量放入层级 2（深）。如果进入 with 或 try/catch 代码块，则会增加新的层级，也就是将 with 或 catch 中的变量放入最浅层（层级 1），并且将之前的层级依次加深。

```
var myObj;
function process() {
     var images = document.getElementsByTagName("img"), widget = document.
```

```
getElementsByTagName("input"), combination = [];
        for(var i = 0; i < images.length; i++) {
            combination.push(combine(images[i], widget[2 * i]));
        }
        myObj.container.property1 = combination[0];
        myObj.container.property2 = combination[combination.length - 1];
    }
```

在上面代码中，images、widget、combination 属于局部变量，位于层级 1。document、myObj 属于全局变量，位于层级 2。变量所在的层越浅，访问速度越快，变量所在的层越深，访问速度越慢，所以这里对 images、widget、combination 的访问速度比 document、myObj 要快一些。推荐使用局部变量，例如：

```
    var myObj;
    function process() {
        var doc = document;
          var images = doc.getElementsByTagName("img"), widget = doc.
getElementsByTagName("input"), combination = [];
        for(var i = 0; i < images.length; i++) {
            combination.push(combine(images[i], widget[2 * i]));
        }
        myObj.container.property1 = combination[0];
        myObj.container.property2 = combination[combination.length - 1];
    }
```

使用局部变量 doc 取代全局变量 document，这样可以改进性能，尤其是在大量使用全局变量的函数中。再看下面代码：

```
    var myObj;
    function process() {
        var doc = document;
        var images = doc.getElementsByTagName("img"), widget = doc.getElementsByTagName("input"),
combination = [];
        for(var i = 0; i < images.length; i++) {
            combination.push(combine(images[i], widget[2 * i]));
        }
        with(myObj.container) {
            property1 = combination[0];
            property2 = combination[combination.length - 1];
        }
    }
```

加上 with 语句结构，可以让代码更加简洁清晰，但这样做会使性能受影响。正如之前说的，当进入 with 代码块时，combination 便从原来的层级 1 变到了层级 2，这样效率会大大降低。因此，优化 with 语句，推荐使用下面代码。

```
    var myObj;
    function process() {
        var doc = document;
          var images = doc.getElementsByTagName("img"), widget = doc.
```

```
getElementsByTagName("input"), combination = [];
        for(var i = 0; i < images.length; i++) {
            combination.push(combine(images[i], widget[2 * i]));
        }
        myObj.container.property1 = combination[0];
        myObj.container.property2 = combination[combination.length - 1];
    }
```

但这样并不是最好的方式，JavaScript 有个特点，对于 object 对象来说，其属性访问层级越深，效率越低。如果这里的 myObj 已经访问到了第 3 层，那么可以这样改进一下来缩小对象访问层级：

```
    var myObj;
    function process() {
        var doc = document;
        var images = doc.getElementsByTagName("img"), widget = doc.getElementsByTagName("input"),
combination = [];
        for(var i = 0; i < images.length; i++) {
            combination.push(combine(images[i], widget[2 * i]));
        }
        var ctn = myObj.container;
        ctn.property1 = combination[0];
        ctn.property2 = combination[combination.length - 1];
    }
```

用局部变量来代替 myObj 第 2 层的 container 对象。如果存在大量的这种对对象深层属性的访问，那么可以参照以上方式提高性能。

建议 178：推荐的 JavaScript 性能调优

由于 JavaScript 语言的单线程和解释执行的两个特点决定了它本身有很多地方存在性能问题，因此需要进行性能优化的地方还是很多，下面就 JavaScript 存在的性能障碍节点进行说明。

（1）eval 问题

比较下面两段代码：

```
    // 方法 1
    var reference = {}, props = "p1";
    eval("reference." + props + "=5")
    // 方法 2
    var reference = {}, props = "p1";
    reference[props] = 5
```

没有 eval 的代码要比有 eval 的代码快 100 倍以上，这是因为 JavaScript 代码在执行前会进行类似预编译的操作：首先会创建一个当前执行环境下的活动对象，并将那些用 var 声明的变量设置为活动对象的属性，但此时这些变量都是 undefined，还会将那些以 function 定

义的函数也添加为活动对象的属性，而且它们的值正是函数的定义。如果使用了 eval，那么 eval 中的代码（实际上为字符串）无法预先识别其上下文，无法被提前解析和优化，即无法进行预编译的操作，所以，代码性能也会大幅度降低。

（2）Function 用法

比较下面两段代码：

```
//方法 1
var func1 = new Function("return arguments[0] + arguments[1]");
func1(10, 20);
//方法 2
var func2 = function() {
    return arguments[0] + arguments[1]
};
func2(10, 20);
```

第一段代码的效率会比第二段代码的效率差很多，故推荐使用第二种方式。

（3）字符串拼接

经常看到如下形式的简单字符串拼接代码：

```
str += "str1" + "str2"
```

这是拼接字符串常用的方式，但这种方式会有一些临时变量的创建和销毁，使性能受到影响，所以推荐使用如下方式拼接：

```
var str_array = [];
str_array.push("str1");
str_array.push("str2");
str = str_array.join("");
```

这里利用数组（array）的 join 方法实现字符串的拼接，尤其是在早期的 IE 版本（如 IE 6）上运行时，会有非常明显的性能上的改进。

当然，最新的浏览器（如 Firefox、IE 8 及其以上版本等）对字符串的拼接做了优化，也可以这样实现字符串快速拼接：

```
str +="str1"
str +="str2"
```

新的浏览器对“+=”做了优化，其性能略好于数组的 join 方法。在不久的将来，更新版本的浏览器可能对“+”进行优化，那时可以直接写成：str += "str1" +"str2"。

（4）隐式类型转换

参考如下隐式类型转换代码：

```
var str = "12345678", arr = [];
for(var i = 0; i <= s.length; i++) {
    arr.push(str.charAt(i));
}
```

在上面代码中，每次循环时都会调用字符串的 charAt 方法，但由于将常量“12345678”赋值给 str，因此这里的 str 并不是一个字符串对象，每次调用 charAt 方法时都会临时构造值为“12345678”的字符串对象，然后调用 charAt 方法，最后再释放这个字符串临时对象。可以通过一些改进来避免隐式类型转换。

```
var str = new Stirng("12345678"), arr = [];
for(var i = 0; i <= s.length; i++) {
    arr.push(str.charAt(i));
}
```

这样，变量 str 作为一个字符串对象，就不会有这种隐式类型转换的过程了，效率会显著提高。

（5）字符串匹配

JavaScript 具有 RegExp 对象，支持对字符串的正则表达式匹配。它是一个很好的工具，但性能并不是非常理想。相反，字符串对象（String）本身的一些基本方法的效率是非常高的，如 substring、indexOf、charAt 等，在需要用正则表达式匹配字符串时，可以考虑下面的因素。

- 是否能够通过字符串对象本身支持的基本方法解决问题。
- 是否可以通过 substring 来缩小需要用正则表达式的范围。

这些方式都能够有效地提高程序的运行效率。关于正则表达式对象，还有一点需要注意：

```
for(var i = 0; i <= str_array.length; i++) {
    if(str_array[i].match(/^s*extra\s/)) {
        //...
    }
}
```

在这里向 match 方法传入 /^s*extra\s/ 是会影响执行效率的，因为在这一过程中会构建临时值为 /^s*extra\s/ 的正则表达式对象，执行 match 方法，然后销毁临时的正则表达式对象。可以这样进行优化：

```
var sExpr = /^s*extra\s/;
for(var i = 0; i <= str_array.length; i++) {
    if(str_array[i].match(sExpr)) {
        //...
    }
}
```

这样就不会有临时对象了。

（6）setTimeout 和 setInterval

setTimeout 和 setInterval 这两个函数可以接受字符串变量，但会带来和之前谈到的 eval 类似的性能问题，所以建议还是直接传入函数对象本身。

（7）利用提前退出

参考如下两段代码：

```
// 代码 1
if(source.match(//)){

}
// 代码 2
if(name.indexOf( ) && source.match(//)){

}
```

代码 2 中多了一个对 name.indexOf() 的判断，这使程序每次执行到这一段时会先执行 indexOf 的判断，再执行后面的 match，在 indexOf 比 match 效率高很多的前提下，这样做会减少 match 的执行次数，从而在一定程度上提高效率。

建议 179：减少 DOM 操作中的 Repaint 和 Reflow

由于 JavaScript 开发离不开 DOM 操作，所以对 DOM 操作的性能优化在 Web 开发中是非常重要的。Repaint（或称为 Redraw）是一种不会影响当前 DOM 结构和布局的一种重绘动作。下面动作都会产生 Repaint 动作：

- 不可见到可见（visibility 样式属性）。
- 颜色或图片变化（background、border-color、color 样式属性）。
- 不改变页面元素大小、形状和位置，但改变其外观的变化。

Reflow 主要发生在 DOM 树被操作的时候。任何改变 DOM 的结构和布局的操作都会产生 Reflow。当一个元素的 Reflow 操作发生时，它的所有父元素和子元素都会产生 Reflow，最后 Reflow 必然会导致 Repaint 的产生。例如，如下动作会产生 Repaint 动作：

- 浏览器窗口的变化。
- DOM 节点的添加和删除操作。
- 一些改变页面元素大小、形状和位置的操作的触发。

每次 Reflow 会比 Repaint 带来更多的资源消耗，应该尽量减少 Reflow 的发生，或者将其转化为只会触发 Repaint 操作的代码，例如：

```
var pDiv = document.createElement("div");
document.body.appendChild(pDiv);     // Reflow
var cDiv1 = document.createElement("div");
var cDiv2 = document.createElement("div");
pDiv.appendChild(cDiv1);    //Reflow
pDiv.appendChild(cDiv2);    // Reflow
```

在上面代码中，总共产生 3 次 Reflow。下面进行优化：

```
var pDiv = document.createElement("div");
var cDiv1 = document.createElement("div");
var cDiv2 = document.createElement("div");
pDiv.appendChild(cDiv1);
pDiv.appendChild(cDiv2);
document.body.appendChild(pDiv);    //Reflow
```

这样只有一次 Reflow，因此，推荐这种 DOM 节点操作的方式。

关于上述较少 Reflow 操作的解决方案，还有一种可以参考的模式：利用 display 减少 Reflow。

```
var pDiv = document.getElementById("parent");
pDiv.style.display = "none" ;  // reflow
pDiv.appendChild(cDiv1);
pDiv.appendChild(cDiv2);
pDiv.appendChild(cDiv3);
pDiv.appendChild(cDiv4);
pDiv.appendChild(cDiv5);
pDiv.style.width = "100px";
pDiv.style.height = "100px";
pDiv.style.display = "block" ;  // reflow
```

先隐藏 pDiv，再显示，这样隐藏和显示之间的操作便不会产生任何的 Reflow，提高了效率。

DOM 元素中有一些特殊的测量属性的访问和方法的调用，也会触发 Reflow，比较典型的就是 offsetWidth 属性和 getComputedStyle 方法。例如，下面代码都会产生 Reflow。

```
var width = element.offsetWidth;
var scrollleft = element.scrollleft;
var display = window.getComputerStyle(div,'').getPropertyValue('display');
```

这些测量属性和方法如下：

- offsetLeft
- offsetTop
- offsetHeight
- offsetWidth
- scrollTop/Left/Width/Height
- clientTop/Left/Width/Height
- getComputedStyle()
- currentStyle(in IE)

对这些测量属性和方法的访问或调用都会触发 Reflow 的产生，应该尽量减少对这些属性和方法的访问或调用。

```
var pe = document.getElementById("pos_element");
var result = document.getElementById("result_element");
```

```
var pOffsetWidth = pe.offsetWidth;
result.children[0].style.width  = pOffsetWidth;
result.children[1].style.width  = pOffsetWidth;
result.children[2].style.width  = pOffsetWidth;
```

在上面代码中，使用临时变量将 offsetWidth 的值缓存起来，这样就不用每次都访问 offsetWidth 属性。这种方式在循环中非常适用，可以极大地提高性能。

经常见到如下的代码：

```
var sElement = document.getElementById("pos_element");
sElement.style.border ='1px solid red'
sElement.style.backgroundColor ='silver'
sElement.style.padding ='2px 3px'
sElement.style.marginLeft ='5px'
```

从中可以看到，这里的每一个样式的改变，都会产生 Reflow。要减少这种情况的发生，可以这样做：

```
.class1 {
    border: '1px solid red'
    background-color: 'silver'
    padding: '2px 3px'
    margin-left: '5px'
 }
 document.getElementById("pos_element").className = 'class1';
```

用 class 替代 style，可以将原有的所有 Reflow 或 Repaint 的次数都缩减到一次。

```
var sElement = document.getElementById("pos_element");
var newStyle = 'border: 1px solid red;'+'background-color: silver;' +'padding: 2px 3px;
'+"margin-left: 5px;"
sElement.style.cssText += newStyle;
```

一次性设置所有样式，也是减少 Reflow、提高性能的方法。

建议 180：提高 DOM 访问效率

一个页面一般包含 1000 多个页面元素，在定位具体元素时，往往需要花费一定的时间。如果用 ID 或 name 定位，效率可能比较快，而用元素的一些其他属性（如 className 等）定位，效率就不理想了。可能只有通过遍历所有元素（getElementsByTagName），然后过滤才能找到相应元素，这就更加低效了。为了提高 DOM 访问效率，这里推荐使用 XPath 查找元素。很多浏览器已支持该功能，具体用法如下：

```
if(document.evaluate){
    var tblHeaders = document.evaluate("//body/div/table//th");
    var result = tblHeaders.iterateNext();
    while(result){
        result.style.border = "1px dotte"d blue";
        result = xpathResult.iterateNext();
```

```
    }
    } else{ //getElementsByTagName()
        // 处理浏览器不支持 XPath 的情况
    }
}
```

浏览器 XPath 的搜索引擎会提高搜索效率，大大缩短结果返回时间。

HTMLCollection 对象是一类特殊的对象，类似数组，但不完全是数组。下列方法的返回值一般都是 HTMLCollection 对象。

- document.images、document.forms
- getElementsByTagName()
- getElementsByClassName()

这些 HTMLCollection 对象并不是一个固定的值，而是一个动态的结果。它们是一些比较特殊的查询返回值。在 HTMLCollection 对象为下面两种情况时，它们会重新执行之前的查询而得到新的返回值（查询结果），不过在多数情况下会和前一次或几次的返回值一样。

- length 属性
- 具体的某个成员

HTMLCollection 对象对这些属性和成员的访问，比数组要慢很多。当然也有例外，Opera 和 Safari 对这种情况就处理得很好，不会有太大性能问题。例如：

```
var items = ["test1", "test2", "test3", ..., "testn"];
for(var i = 0; i < items.length; i++){
}
var items = document.getElementsByTagName("div");
for(var i = 0; i < items.length; i++){
}
```

与上面一段代码相比较，下面代码频繁读取 items 的 length 属性值，执行效率要低很多，因为每一个循环都会读取 items.length 的值，也会导致 document.getElementsByTagName() 方法的再次调用，这便是效率大幅度下降的原因。可以这样解决：

```
var items = document.getElementsByTagName("div");
var len = items.length
for(var i = 0; i < len; i++){
}
```

这样一来，效率基本与普通数组一样。

加载并执行一段 JavaScript 脚本是需要一定时间的，有时候 JavaScript 脚本被加载后基本没有被使用过，如脚本中的函数从来没有被调用等。加载这些脚本只会占用 CPU 时间和增加内存消耗，降低 Web 应用的性能，所以推荐动态加载 JavaScript 脚本文件，尤其是那些内容较多、消耗资源较大的脚本文件。

```
if(needXHR){
    document.write("<script type= 'test\/Javascript' src= 'dojo_xhr.js' >");
```

```
}
if(dojo.isIE){
    document.write("<script type= 'test\/Javascript' src= 'vml.js' >");
}
```

建议 181：使用 setTimeout 实现工作线程

在 Ajax 应用中，有时候需要在后台执行一些耗时较长且与页面主要逻辑无关的操作。例如，对于一个在线文档编辑器来说，会需要定期自动备份用户当前所编辑的内容，这样当应用异常崩溃时，用户还能恢复所编辑的内容。这样的定期备份任务可能需要花费一些时间，但优先级较低。类似这样的任务还有页面内容的预先加载和日志记录等。对于这些任务，最好的实现方式是在后台工作线程中执行它们，这样不会对用户在主页面上的操作造成影响。用户并不会希望由于后台备份正在进行，而无法对当前的文档进行编辑。

浏览器中 JavaScript 引擎是单线程执行的，也就是说，在同一时间内，只能有一段代码被 JavaScript 引擎执行。如果在同一时间内还有其他代码需要执行，那么这些代码需要等待 JavaScript 引擎执行完当前的代码之后才有可能获得被执行的机会。在正常情况下，作为页面加载过程中的重要一步，JavaScript 引擎会顺序执行页面上的所有 JavaScript 代码。在页面加载完成之后，JavaScript 引擎会进入空闲状态。用户在页面上的操作会触发一些事件，这些事件的处理方法会交给 JavaScript 引擎来执行。鉴于 JavaScript 引擎的单线程特性，一般会在内部维护一个待处理的事件队列，每次从事件队列中选出一个事件处理方法来执行。如果在执行过程中，有新的事件发生，那么新事件的处理方法只会被加入到队列中等待执行。如果当前正在执行的事件处理方法非常耗时，那么队列中的其他事件处理方法可能长时间无法得到执行，造成用户界面失去响应，严重影响用户的使用体验。

JavaScript 引擎的这种工作方式类似于早期的单核 CPU 的调度方式。单核 CPU 虽然也支持多任务同时运行，但实际上同一时间只能有一个任务在执行。CPU 通过时间片的轮转来保证每个任务都有一定的执行时间。JavaScript 并没有原生提供与操作系统中的线程类似的结构，但可以通过定时器机制来模拟。JavaScript 提供了两个基本的方法来执行与定时相关的操作，分别是 setTimeout() 和 setInterval()。

- setTimeout() 用来设置在指定的间隔时间之后执行某个 JavaScript 方法。setTimeout() 的方法声明非常简单：setTimout(func, time)，其中参数 func 表示的是要执行的 JavaScript 方法，可以是 JavaScript 方法对象或方法体的字符串；参数 time 表示的是以毫秒为单位的间隔时间。
- setInterval() 用来设置根据指定的间隔重复执行某个 JavaScript 方法。setInterval() 的方法声明与 setTimeout() 相同：setInterval(func, time)，这里参数 time 指定的是方法 func 重复执行的间隔。当 setTimeout() 或 setInterval() 被调用的时候，浏览器会根据设置的时间间隔来触发相应的事件。

如果代码的调用方式是 setTimeout(func, 100)，那么该代码被执行 100 ms 后，定时器的事件被触发。如果这个时候 JavaScript 引擎中没有正在执行的其他代码，那么与此定时器对应的 JavaScript 方法 func 就可以被执行，否则，该 JavaScript 方法的执行就被加入到等待的队列中。当 JavaScript 引擎空闲的时候，会从这个队列中选择一个等待的 JavaScript 方法来执行。也就是说，虽然在调用 setTimeout() 时设置的间隔时间是 100 ms，但与之对应的 JavaScript 方法实际被执行的间隔有可能大于设定的 100 ms，这取决于是否有其他代码正在被执行和执行所花费的时间。因此，setTimeout() 实际生效的间隔时间可能大于设定的时间。

setInterval() 的执行方式与 setTimeout() 有很大不同。如果代码的调用方式是 setInterval(func, 100)，那么每隔 100 ms，定时器的事件就会被触发。与 setTimeout() 相同的是，如果当前 JavaScript 引擎空闲，那么定时器对应的方法 func 会被立即执行，否则，该 JavaScript 方法的执行就会被加入到等待队列中。由于定时器的事件是每隔 100 ms 就触发一次，有可能某一次事件触发的时候，上一次事件的处理方法还没有机会执行，仍然在等待队列中。这个时候，这个新的定时器事件就被丢弃。需要注意的是，由于 JavaScript 引擎的这种执行方式，两次执行定时器事件处理方法的实际时间间隔小于设定的时间间隔。例如，在上一个定时器事件的处理方法触发之后，等待了 50 ms 才获得被执行的机会，而第二个定时器事件的处理方法被触发后马上就被执行了。也就是说，这两者之间的时间间隔实际上只有 50 ms。因此，setInterval() 并不适合实现精确的按固定间隔的调度操作。

总的来说，使用 setTimeout() 和 setInterval() 都不能满足精确的时间间隔。通过 setTimeout() 设置的 JavaScript 方法的实际执行间隔不小于设定的时间，而通过 setInterval() 设置的重复执行的 JavaScript 方法的间隔可能会小于设定的时间。

setTimeout() 可用于设置在某个时间间隔之后再执行某个 JavaScript 方法。setTimeout() 的另外一个作用是可以将某些操作推迟执行，让出 JavaScript 引擎来处理其等待队列中的其他事件，以提高用户体验。例如，某个操作需要进行大量计算，平均耗时在 3 s 左右，当这个操作开始执行之后，就会一直占用 JavaScript 引擎，直到执行结束。在这个过程中，其他的 JavaScript 方法就被放置到 JavaScript 引擎的等待队列中。如果用户在这个过程中单击了页面上的某个按钮，那么相应的事件处理方法并不能马上执行，给用户的感觉就是整个 Web 应用暂时失去了响应，给用户带来不好的用户体验。

如果使用 setTimeout()，就可以把一个需要较长执行时间的任务分成若干个小任务，这些小任务之间用 setTimeout() 串联起来。在这些小任务的执行间隔中，就可以给其他正在等待的 JavaScript 方法以执行机会。

这里以计算 100 000 以内的质数的个数为例进行介绍。求质数的方法有不少，这里使用一种简单的方法。对于每一个正整数，通过判断其是否能被小于或等于其平方根的整数整除，就可以确定其是否为质数。实现的代码如下：

```
function isPrime(n) {
    if (n == 0 || n == 1) {
```

```
        return false;
    }
    var bound = Math.floor(Math.sqrt(n));
    for (var i = 2; i <= bound; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

如果使用一般的计算方式，那么只需要通过一个很大的循环对范围之内的每个整数都进行判断即可。

```
function calculateNormal() {
    var count = 0;
    for (var i = 2; i <= MAX; i++) {
        if (isPrime(i)) {
            count++;
        }
    }
    alert("计算完成，质数个数为：" + count);
}
```

上面代码的问题在于：在整个计算过程中，JavaScript 引擎被全部占用，整个页面无法响应用户的其他请求，页面会呈现“假死”的状态。而通过 setTimeout() 可以把计算任务分成若干个小任务来执行，提高页面的响应能力。实现的代码如下：

```
function calculateUsingTimeout() {
    var jobs = 10, numberPerJob = Math.ceil(MAX / jobs);
    var count = 0;
    function calculate(start, end) {
        for (var i = start; i <= end; i++) {
            if (isPrime(i)) {
                count++;
            }
        }
    }
    var start, end, timeout, finished = 0;
    function manage() {
        if (finished == jobs) {
            window.clearTimeout(timeout);
            alert("计算完成，质数个数为：" + count);
        }
        else {
            start = finished * numberPerJob + 1,
                end = Math.min((finished + 1) * numberPerJob, MAX);
            timeout = window.setTimeout(function() {
                calculate(start, end);
                finished++;
                manage();
```

```
            }, 100);
        }
    }
    manage();
}
```

通过 setTimeout() 把耗时较长的计算任务分成了 10 个小任务，每个任务之间的执行间隔是 100 ms。在这些小任务的执行间隔中，JavaScript 引擎是可以处理其他事件的，这样就保证了对用户的响应时间。虽然整个任务总的执行时间变长了，但是带来了用户体验的提升。

建议 182：使用 Web Worker

随着 HTML 5 的流行，当 JavaScript 工作线程对 Ajax 应用是一个比较迫切的实现要求时，Web Worker 得到更多浏览器的支持，从而可以在 Ajax 应用中得到更多的使用。对于 Ajax 应用开发者来说，必须紧跟 HTML 5 标准的发展潮流。

WHATAG 工作组借鉴了 Google Gears 的成功经验，创建了 Web Worker 规范，并将其作为 HTML 5 标准的一部分。Web Worker 规范定义了一套 API，允许 Web 应用创建在后台执行 JavaScript 代码的工作线程。工作线程在执行过程中不会对主页面造成影响。

在 Web Worker 规范中定义了两类工作线程，分别是专属工作线程和共享工作线程。专属工作线程只能为一个页面所使用，而共享工作线程则可以被多个页面所共享。通过 Worker 构造方法可以创建一个新的专属工作线程，在创建时要指定需要执行的 JavaScript 文件的 URL。Web Worker 规范并不支持从 JavaScript 代码文本中直接创建工作线程。创建工作线程的页面和工作线程之间通过简单的消息传递来进行交互。通过工作线程对象的 postMessage() 方法就可以发送消息给工作线程，而通过 onmessage 属性就可以设置接收到消息时的处理方法。在工作线程的 JavaScript 代码中做法也是相同的。例如，下面给出了使用 Web Worker 计算质数个数的实现方法。

（1）使用专属工作线程的主页面代码

```
var worker = new Worker("prime_worker.js");
worker.onmessage = function(event) {
    var result = event.data;
    alert("计算完成，质数个数为：" + result);
};
function calculate() {
    var limit = parseInt(document.getElementById("limit").value) || 100000;
    worker.postMessage(limit);
}
```

在主页面中创建了一个专属工作线程，让它在后台执行 JavaScript 文件 prime_worker.js。下面给出在 prime_worker.js 中与主页面通信部分的代码。

（2）专属工作线程所执行的 JavaScript 代码

```
onmessage = function(event) {
    var limit = event.data;
    var count = calculateNormal(limit);
    postMessage(count);
}
```

共享工作线程的使用方式与专属工作线程有所不同。共享工作线程允许多个页面共享使用，每个页面都是连接到该共享工作线程的某个端口（port）上，页面通过此端口与共享工作线程进行通信。使用 SharedWorker 构造方法可以创建出共享工作线程的实例。在发送消息时，需要使用的是工作线程的 port 对象。下面是使用共享工作线程的主页面代码：

```
var worker = new SharedWorker("prime_worker.js");
worker.port.onmessage = function(event) {
    var result = event.data;
    alert("计算完成，质数个数为：" + result);
};
function calculate() {
    var limit = parseInt(document.getElementById("limit").value) || 100000;
    worker.port.postMessage(limit);
}
```

在共享工作线程所执行的 JavaScript 代码中，不能使用 onmessage 来直接定义接收到消息时的处理方法，而需要使用 onconnect 来定义接收到连接时的处理逻辑。

（3）共享工作线程所执行的 JavaScript 代码

```
onconnect = function(event) {
    var port = event.ports[0];
    port.onmessage = function(event) {
        var limit = event.data;
        var count = calculateNormal(limit);
        port.postMessage(count);
    };
}
```

如上所示，在 onconnect 的处理方法中，设置了当从某个端口上接收到消息时应该执行的处理逻辑。

除了通过 onmessage 属性来设置消息处理方法之外，还可以使用 addEventListener() 来添加消息处理方法，这样做的好处是可以添加多个事件处理方法。需要注意的是，如果使用 addEventListener() 来添加事件处理方法，在添加完成之后，需要调用 port.start() 方法来显式地启动端口上的通信。

Web Worker 目前只有一些较新的浏览器上支持 Web Worker，如 Firefox 3.5、Safari 4、Google Chrome 4 和 Opera 10.6 等以及它们的更高版本。由于 IE 的主流版本不支持 Web Worker，使用 Web Worker 编写的 Web 应用是不能在 IE 上运行的。

建议 183：避免内存泄漏

JavaScript 是一种垃圾收集式语言，其对象的内存是根据对象的创建分配给该对象的，并且会在没有对该对象的引用时由浏览器收回。JavaScript 的垃圾收集机制本身并没有问题，但浏览器在为 DOM 对象分配和恢复内存的方式上有些出入。

IE 和 Firefox 均使用引用计数来为 DOM 对象处理内存。在引用计数系统中，每个所引用的对象都会保留一个计数，以获悉有多少对象正在引用它。如果计数为零，那么该对象就会被销毁，其占用的内存也会返回给堆。虽然这种解决方案总的来说还算有效，但是在循环引用方面却存在一些盲点。

当两个对象互相引用时，就构成了循环引用，其中每个对象的引用计数值都被赋为 1。在纯垃圾收集系统中，循环引用问题不大：如果涉及的两个对象中有一个对象被任何其他对象引用，那么这两个对象都将被垃圾收集。而在引用计数系统中，这两个对象都不能被销毁，原因是引用计数永远不能为零。在同时使用了垃圾收集和引用计数的混合系统中，将会发生泄漏，因为系统不能正确识别循环引用。在这种情况下，DOM 对象和 JavaScript 对象均不能被销毁，例如：

```
<html>
<body>
<script type="text/javascript">
    var obj;
    window.onload = function() {
        obj = document.getElementById("DivElement");
        document.getElementById("DivElement").expandoProperty = obj;
        obj.bigString = new Array(1000).join(new Array(2000).join("XXXXX"));
    };
</script>
<div id="DivElement">Div Element</div>
</body>
</html>
```

在上面代码中，JavaScript 对象 obj 拥有到 DOM 对象的引用，表示为 DivElement。而 DOM 对象也拥有到此 JavaScript 对象的引用，由 expandoProperty 表示。可见 JavaScript 对象和 DOM 对象间就产生了一个循环引用。由于 DOM 对象是通过引用计数管理的，因此两个对象将都不能销毁。

另一种内存泄漏模式：通过调用外部函数 myFunction 创建循环引用。同样，JavaScript 对象和 DOM 对象间的循环引用也会导致内存泄漏。

```
<html>
<head>
<script type="text/javascript">
    document.write(" objects between Javascript and DOM!");
    function myFunction(element) {
        this.elementReference = element;
```

```
        element.expandoProperty = this;
    }
    function Leak() {
        new myFunction(document.getElementById("myDiv"));
    }
</script>
</head>
<body onload="Leak()">
<div id="myDiv"></div>
</body>
</html>
```

循环引用很容易创建。在 JavaScript 最为方便的编程结构之一——闭包中，循环引用尤其突出。JavaScript 的优势在于它允许函数嵌套。一个嵌套的内部函数可以继承外部函数的参数和变量，并由该外部函数私有。

```
<html>
<body>
<script type="text/javascript">
    window.onload = function closureDemoParentFunction(paramA) {
        var a = paramA;
        return function closureDemoInnerFunction(paramB) {
            alert(a + " " + paramB);
        };
    };
    var x = closureDemoParentFunction("outer x");
    x("inner x");
</script>
</body>
</html>
```

在上面代码中，closureDemoInnerFunction 是在父函数 closureDemoParentFunction 中定义的内部函数。当用外部的 x 对 closureDemoParentFunction 进行调用时，外部函数变量 a 就会被赋值为外部的 x。外部函数会返回指向内部函数 closureDemoInnerFunction 的指针，该指针包括在变量 x 内。

外部函数 closureDemoParentFunction 的本地变量 a 即使在外部函数返回时仍会存在。这一点与 C/C++ 这样的编程语言不同。在 C/C++ 中，一旦函数返回，本地变量也将不复存在。在 JavaScript 中，在调用 closureDemoParentFunction 时，带有属性 a 的范围对象将会被创建。该属性包括值 paramA，又称为“外部 x”。同样，当 closureDemoParentFunction 返回时，它将会返回内部函数 closureDemoInnerFunction，该函数包括在变量 x 中。

由于内部函数持有对外部函数的变量的引用，因此这个带属性 a 的范围对象将不会被垃圾收集。当对具有参数值 inner x 的 x 进行调用时，即执行 x("inner x") 时，将会弹出警告消息——“outer x innerx”。

JavaScript 闭包功能非常强大，原因是它们使内部函数在外部函数返回时也仍然可以保留对此外部函数的变量的访问。不幸的是，闭包非常易于隐藏 JavaScript 对象和 DOM 对象

间的循环引用。

例如，在下面代码中的闭包内，JavaScript 对象（obj）包含到 DOM 对象的引用（通过 id="element" 被引用），而 DOM 元素则拥有到 JavaScript obj 的引用，这样建立起来的 JavaScript 对象和 DOM 对象间的循环引用将会导致内存泄漏。

```
<html>
<body>
<script type="text/javascript">
    window.onload = function outerFunction() {
        var obj = document.getElementById("element");
        obj.onclick = function innerFunction() {
            alert("Hi! I will leak");
        };
        obj.bigString = new Array(1000).join(new Array(2000).join("XXXXX"));
    };
</script>
<button id="element">Click Me</button>
</body>
</html>
```

JavaScript 内存泄漏是可以避免的。例如，以上述由事件处理引起的内存泄漏模式为例来展示 3 种应对已知内存泄漏的方式。

❑ 主动设置 JavaScript 对象 obj 为空，显式打破此循环引用。

```
<script type="text/javascript">
    window.onload = function outerFunction() {
        var obj = document.getElementById("element");
        obj.onclick = function innerFunction() {
            alert("Hi! I have avoided the leak");
        };
        obj.bigString = new Array(1000).join(new Array(2000).join("XXXXX"));
        obj = null;
    };
</script>
```

❑ 通过添加另一个闭包来避免 JavaScript 对象和 DOM 对象间的循环引用。

```
<script type="text/javascript">
    document.write("Avoiding a memory leak by adding another closure");
    window.onload = function outerFunction() {
        var anotherObj = function innerFunction() {
            alert("Hi! I have avoided the leak");
        }; (function anotherInnerFunction() {
            var obj = document.getElementById("element");
            obj.onclick = anotherObj
        })();
    };
</script>
```

❑ 通过添加另一个函数来避免闭包本身，进而阻止内存泄漏。

```
<script type="text/javascript">
    window.onload = function() {
        var obj = document.getElementById("element");
        obj.onclick = doesNotLeak;
    }
    function doesNotLeak() {
        alert("Hi! I have avoided the leak");
    }
</script>
```

建议 184：使用 SVG 创建动态图形

与传统的客户端编程相比，JavaScript 操作的对象被限制在 DOM 模型之内，无法进行图形编程。所以长久以来，在设计网页时都仅仅是在“搭积木”，并且这些积木只有一种形状——长方形。这些“长方形的积木”就是应用在 HTML 元素上的“盒子”模型（box model）。每个盒子有边框（border）、边缘（margin）和填充（padding）。我们只能控制这些盒子的大小和有限的样式。这些方块的集合对于构建一个传统的文档页面已经足够了，但 Web 的流行已经使网页承担的任务远远超出了传递文字信息。

SVG（Scalable Vetor Graphics，可缩放矢量图）作为一种通用的数据格式，属于 XML 语言的一个分支，主要负责描述矢量图的数据结构关系。实际上，SVG 不是第一种用 XML 描述图片的格式，甚至也不是第一种在 Web 上提出的 XML 与矢量图的组合的标准。在它之前的 VML（Vector Markup Language）和 PGML（Precision Graphics Markup Language）都是基于 XML 的矢量图规范。

SVG 中各种元素和属性的详细说明可以参考相关规范文档。下面代码是一个简单的 SVG 文件。

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/
SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" width="100%" height="100%" >
    <circle cx="100" cy="100" r="40" fill="red"/>
</svg>
```

第一条语句为 XML 指令定义版本，并说明此文件引用到其他文件。第二条语句是文档类型定义，规定此 XML 中哪些是有效的 SVG 元素。这里引用的 http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd 正是第一条语句中 standalone 属性为 no 的原因。从第三条语句开始是 SVG 的真正定义，circle 元素指定画一个圆，cx、cy 和 r 属性分别指定圆心的横坐标、纵坐标和半径，fill 属性指定用红色填充此圆内部的区域。

将这段代码粘贴进任何一个文本编辑器，然后将文件保存为一个 SVG 文件，如 sun.svg，就可以画出一个红色的太阳。各种浏览器对 SVG 的支持情况不同。总的来说，IE 6、IE 7 和 IE 8 对 SVG 都没有原生的支持，需要专门的插件（如 Adobe SVG Viewer）才能显示，

IE 9 支持 SVG，不过 IE 有自定义的 VML 技术可以实现相同的效果。其他主流浏览器都对 SVG 标准有不同程度的支持。

（1）使用 <img> 标签显示

```
<img src= 'sun.svg' >
```

将 SVG 与传统的互联网图片格式同等使用（现在只有 Chrome、Safari 和 Opera 支持）。

（2）使用 <embed> 标签显示

```
<embed src="sun.svg" width="300" height="100"
type="image/svg+xml"
pluginspage="http://www.adobe.com/svg/viewer/install/" />
```

pluginspage 属性的值是 Adobe 公司为不原生支持 SVG 的浏览器开发的插件 Adobe SVG Viewer 的安装地址。2009 年 1 月 1 日，Adobe 公司已经终止对该产品的支持。

（3）使用 <object> 标签显示

```
<object data="sun.svg" width="300" height="100"
 type="image/svg+xml"
 codebase="http://www.adobe.com/svg/viewer/install/" />
```

（4）使用 <iframe> 标签显示

```
<iframe src="sun.svg" width="300" height="100" border="0" style="border-width:0">
 </iframe>
```

下面是对在 HTML 中各种使用 SVG 的方式是否支持进行判断的页面代码，sun.svg 文件与该页面保存于同一目录。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
 <HTML>
 <HEAD>
<TITLE> SVG in HTML </TITLE>
 </HEAD>
 <BODY>
 1. 使用 &lt;img&gt; 标签
 <br>
 <img src="sun.svg" width="300" height="100">
 <br>
 2. 使用 &lt;embed&gt; 标签
 <br>
 <embed src="sun.svg" width="300" height="100"
type="image/svg+xml"
pluginspage="http://www.adobe.com/svg/viewer/install/" />
<br>
3. 使用 &lt;object&gt; 标签
<br>
<object data="sun.svg" width="300" height="100"
type="image/svg+xml"
codebase="http://www.adobe.com/svg/viewer/install/" />
```

```
<br>
4. 使用 &lt;iframe&gt; 标签
<br>
<iframe src="sun.svg" width="300" height="100" border="0" style="border-width:0">
</iframe>
</BODY>
</HTML>
```

如果仅将 SVG 作为图片引用，则只发挥了它的静态功能。SVG 的动态功能包括两个方面：支持动画和支持脚本编程。

SVG 在设计时就加入了对动画的支持，这是通过另一种 W3C 颁布的动画语言 SMIL（Synchronized Multimedia Integration Language）实现的。SMIL 在应用时与 SVG 结合得非常紧密，它与 SVG 一样，是一种声明性（declarative）的标记语言，通过元素（element）和属性（attribute）来定义动画的行为。这里只给出一个简单的例子，不做详细介绍，因为浏览器对它的支持还很有限，另外 SMIL 声明性的本质也使其表现力受到限制，不如使用脚本自定义动画灵活。

（5）用 SMIL 实现的动画

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg">
   <polygon points="50,100  100,100  75,50" stroke="#660000" fill="#cc3333">
        <animateTransform
           attributeName="transform"
           begin="0s"
           dur="10s"
           type="rotate"
           from="0 0 0"
           to="360 60 60"
           repeatCount="indefinite"
       />
    </polygon>
</svg>
```

polygon 元素指定画一个多边形，由于这里给定了三个顶点，所以是一个三角形。将上面的代码保存成一个 SVG 文件，在一个页面中引用该文件，如果此时的浏览器支持 SMIL，屏幕上会显示一个不断旋转的红色三角形；如果此时的浏览器只支持 SVG，将看到一个静止的红色三角形。

（6）脚本可编程性

SVG 是一个 XML 文件，用于 XML 编程的两种模型 DOM 和 SAX 也适用于它。因为 SVG 是被设计用于互联网的，所以通过 JavaScript 和 DOM 访问它就是最重要的应用模式。我们已经熟悉了通过 JavaScript 和 DOM 动态地修改 HTML，同样也可以在浏览器中动态地创建、修改和删除图片，这也是接下来要介绍的在 SVG 方面的重点。

为了演示这些动态功能，可以采取和前面的在页面中使用 SVG 不同的方式——在 XHTML 中直接写入 SVG 的源文本，而在前面的 4 种方式中，SVG 的定义都保存在和页面不同的另一个文件中。这样做有两个原因：一是在支持 XHTML 和 SVG 的浏览器中，可以通过 JavaScript 直接访问和修改 SVG；二是在互联网的未来标准 HTML 5 中，SVG 可以这样直接在 HTML 中定义，就像其他 HTML 元素一样。

下面设计一个进度条，显示一个绿色的运动的进度条。

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title> 进度条 </title>
        <script language='Javascript'>
            /* <![CDATA[ */
            function ProgressBar(info){
                var stem = {};              // 此函数最后返回的代表进度条的对象
                var done = 0, length, outline, bar;     // 声明内部变量
                bar = document.getElementById('done');  // 进度条中绿色的变化部分
                length = 80;
                // 重置进度到零
                function reset(){
                    return to(0);
                }
                // 设置进度到某个值
                function to(value){
                    if (value >= 100) {
                        done = 100;
                        bar.setAttribute('width', length);
                    }
                    else {
                        done = value;
                        bar.setAttribute('width', Math.round(done * length / 100));
                    }
                    return stem;
                }
                // 进度变化某个值
                function advance(step){
                    return to(done + step);
                }
                // 以下为进度条对象添加方法
                // 获得当前进度值
                stem.getDone = function(){
                    return done
                };
                stem.reset = reset;
                stem.to = to;
                stem.advance = advance;
                return stem;      // 返回可供脚本使用的进度条对象
            }
            // 测试进度条对象
            function testBar(){
                var bar = ProgressBar();
```

```
                // 此内部函数每运行一次，增加进度值 1，直到进度值为 100
                function test(){
                    if (bar.getDone() === 100) {
                        clearInterval(id);
                    }
                    else {
                        bar.advance(1);
                    }
                }
                // 每 0.1 秒改变一次进度
                var id = setInterval(test, 100);
            }
            // 页面载入后开始测试
            window.addEventListener('load', testBar, true);
            /* ]]> */
        </script>
    </head>
    <body>
        <div id='svgDiv'>
            <svg xmlns="http://www.w3.org/2000/svg" version="1.1"
            viewBox="0 0 100 100" style="border:1px solid; width:100px; height:100px; ">
            <g id='progBar'>
            <rect x='10' y='45' width='80' height='10' stroke='grey' fill='white'/>
            <rect id='done' x='10' y='45' width='0' height='10' fill='green'/>
                </g>
            </svg>
        </div>
    </body>
</html>
```

<svg xmlns="http://www.w3.org/2000/svg" version="1.1" viewBox="0 0 100 100" style="border:1px solid; width:100px; height:100px; "> 表示在 XHTML 中直接插入 svg 元素，并且指定命名空间等其他属性。

viewBox 定义矢量图可见的坐标空间，4 个数字依次是原点的 x 坐标、y 坐标以及平面的宽度、高度。SVG 的坐标空间符合计算机中指定屏幕空间的惯例，x 坐标轴的正方向向右，y 坐标轴的正方向向下。

style 属性指定 svg 元素的各种外观特性。SVG 与 HTML 一样，可以应用 CSS 定义外观，并且有一些专门的特性——XHTML 中的 JavaScript 代码被包含在 /* <![CDATA[*/ 和 /*]]> */ 之间。

在 HTML 文件中不需要这样做，因为在 HTML 中 <script> 标签内的 JavaScript 代码被解释为 CDATA（Character Data，XML 中的一种类型，用于包含任意的字符数据）；而在 XHTML 中 <script> 标签内的部分被解释为 PCDATA（Parsed Character Data，也是 XML 中的一种类型，为字符数据和元素的混合内容），也要通过 XML 的语法检查，而 JavaScript 代码显然不符合 XML 的标签的定义语法。解决方法就是在代码外人工加上 ![CDATA[和]]> 标注，使 XML 的语法校验器忽略这段内容。但这样会带来另一个问题，有些浏览器不认识

CDATA 标注，导致这些代码又无法通过 JavaScript 的语法检查。因此在 CDATA 标注两侧再加上 JavaScript 的注释标记，这样 <script> 标签内的代码既能通过 XML 的语法检查，又能被 JavaScript 引擎识别。

<svg> 标签内有一个 <g> 标签和两个 <rect> 标签。g 元素用于分组。分组不仅可以使 SVG 的内容结构清晰，还可以使同一组内的对象被集体操作。rect 元素代表一个矩形，x、y、width 和 height 属性分别指定矩形左上顶点的横坐标、纵坐标，以及矩形的宽度和长度；stroke 属性指定图形外框的线条颜色。我们用第一个空心的矩形显示进度条的外框，第二个实心的绿色矩形显示变化的进度。为了在脚本中方便地访问，我们设置了绿色矩形的 id 属性。

在 JavaScript 脚本中用 DOM 先后获得绿色矩形对象并修改它的宽度属性。getElementById 和 setAttribute 的用法和在 HTML 中没有两样。注意，有些在操作 HTML 时使用的方法，在 XML 中是不存在的，如根据名称获取元素的 getElementsByName。

在这个例子中，前三点特别设定有些麻烦，不过这些在正在获得越来越多支持并且很快将成为互联网的现实标准的 HTML 5 中都不是必须的。在 HTML 5 中，不需要在 html 和 svg 元素中指定命名空间，svg 和其中的各种标签会被自动识别，JavaScript 代码也会和在目前的 HTML 页面中一样，不需要在两侧加上 CDATA 标注。

SVG 中的元素同样支持用户界面的事件，因此可以通过鼠标和键盘触发的各种事件改变 SVG 中的图形，使得在整个页面上可以进行丰富的图形互动，而不需要借助于 Flash 插件。

建议 185：减少对象成员访问

大多数 JavaScript 代码是以面向对象的形式编写的，无论是创建的自定义对象还是使用内置的对象，如文档对象模型（DOM）和浏览器对象模型（BOM）之中的对象，因此，存在很多对象成员访问。

对象成员包括属性和方法，在 JavaScript 中，二者差别甚微。对象成员可以包含任何数据类型。既然函数也是一种对象成员，那么对象成员除了包含传统数据类型外，也可以包含一个函数。当一个命名成员引用了一个函数时，该成员又被称做方法，而一个非函数类型的数据则被称做属性。

访问对象成员比访问直接量或局部变量速度慢，在某些浏览器上比访问数组项还要慢。要弄清其中的原因，首先要理解 JavaScript 中对象的性质。

JavaScript 中的对象是基于原型的。原型是其他对象的基础，定义并实现了一个新对象所必须具有的成员。这一概念完全不同于传统面向对象编程中类的概念，它定义了创建新对象的过程。原型对象为所有给定类型的对象实例所共享，因此所有实例共享原型对象的成员。

一个对象通过一个内部属性绑定到它的原型。Firefox、Safari 和 Chrome 浏览器向开发人员开放这一属性（__proto__），其他浏览器不允许脚本访问这一属性。在任何时候创建一个内置类型的实例，如 Object 或 Array，它们自动拥有一个 Object 作为它们的原型。

因此，对象可以有两种类型的成员：实例成员和原型成员。实例成员直接存在于实例自身中，而原型成员则从对象原型继承。例如：

```
var book = {
    title: "Javascript",
    publisher: "机械工业出版社"
};
alert(book.toString());    //"[object Object]"
```

在此代码中，book 对象有两个实例成员：title 和 publisher。注意这里并没有定义 toString() 接口，但这个接口却被调用了，而且并没有抛出错误。toString() 函数就是一个 book 对象继承的原型成员。

处理对象成员的过程与变量处理十分相似。当 book.toString() 被调用时，对成员进行名为“toString”的搜索，首先从对象实例开始，如果 book 没有名为 toString 的成员，那么就转向搜索原型对象，在那里发现 toString() 方法并执行它。通过这种方法，book 可以访问它的原型所拥有的每个属性或方法。

可以利用 hasOwnProperty() 函数确定一个对象是否具有特定名称的实例成员，它的参数就是成员名称。要确定对象是否具有某个名称的属性，可以使用操作符 in，例如：

```
var book = {
    title: "Javascript",
    publisher: "机械工业出版社"
};
alert(book.hasOwnProperty("title"));      //true
alert(book.hasOwnProperty("toString")); //false
alert("title" in book);                   //true
alert("toString" in book);                //true
```

在上面代码中，当 hasOwnProperty() 传入 title 时，返回 true，因为 title 是一个实例成员。当 hasOwnProperty() 传入 toString 时，返回 false，因为 toString 不在实例之中。如果使用 in 操作符检测这两个属性，那么返回都是 true，因为它既搜索实例又搜索原型。

对象的原型决定了一个实例的类型。在默认情况下，所有对象都是 Object 的实例，并且继承了所有基本方法，如 toString()。可以用构造器创建另外一种类型的原型，例如：

```
function Book(title, publisher) {
    this.title = title;
    this.publisher = publisher;
}
Book.prototype.sayTitle = function() {
    alert(this.title);
};
var book1 = new Book("CSS", "电子");
var book2 = new Book("HTML", "清华");
alert( book1 instanceof Book);    //true
alert( book1 instanceof Object);    //true
book1.sayTitle();    //" CSS "
```

```
alert(book1.toString());    //"[object Object]"
```

Book 构造器用于创建一个新的 Book 实例。book1 的原型（__proto__）是 Book.prototype，Book.prototype 的原型是 Object。这就创建了一个原型链，book1 和 book2 继承了它们的成员。

注意：两个 Book 实例共享同一个原型链，每个实例拥有自己的 title 和 publisher 属性，但其他成员均继承自原型。当 book1.toString() 被调用时，搜索工作必须深入原型链才能找到对象成员 toString。深入原型链越深，搜索的速度就会越慢。

虽然采用优化 JavaScript 引擎的新式浏览器在此任务中表现良好，但是对于旧版本的浏览器，特别是 IE 和 Firefox 3.5，每深入原型链一层都会增加性能损失。记住，搜索实例成员的过程比访问直接量或局部变量负担更重，增加遍历原型链的开销正好放大了这种效果。

由于对象成员可能包含其他成员，因此对于不太常见的写法，例如 window.location.href 这种模式，每遇到一个点号，JavaScript 引擎就要在对象成员上执行一次解析过程。

成员嵌套越深，访问速度越慢。location.href 总是快于 window.location.href，而 hasOwnProperty() 也要比 window.location.href.toString() 更快。如果这些属性不是对象的实例属性，那么成员解析还要在每个点上搜索原型链，这将需要更长时间。

由于所有这些性能问题与对象成员有关，因此如果可能避免使用它们。更确切地说，只在必要情况下使用对象成员。例如，没有理由在一个函数中多次读取同一个对象成员的值。

```
function hasEitherClass(element, className1, className2){
    return element.className == className1 || element.className == className2;
}
```

在上面代码中，element.className 被访问了两次。很明显，在这个函数的执行过程中 element.className 的值是不会改变的，但仍然引起两次对象成员搜索过程。可以将 element.className 的值存入一个局部变量，消除一次搜索过程。

```
function hasEitherClass(element, className1, className2){
    var currentClassName = element.className;
    return currentClassName == className1 || currentClassName == className2;
}
```

在重写后的代码中成员搜索只进行了一次。既然两次对象搜索都在读属性值，因此有理由只读一次并将值存入局部变量中。局部变量的访问速度要快得多。

一般来说，要在同一个函数中多次读取同一个对象属性，最好将它存入一个局部变量。以局部变量替代属性，避免多余的属性查找带来性能开销。在处理嵌套对象成员时这点特别重要，因为多次属性查找会对运行速度产生难以想象的影响。

JavaScript 的命名空间，如 YUI 所使用的技术，是经常访问嵌套属性的来源之一，例如：

```
function toggle(element){
    if (YAHOO.util.Dom.hasClass(element, "selected")){
        YAHOO.util.Dom.removeClass(element, "selected");
        return false;
    } else {
        YAHOO.util.Dom.addClass(element, "selected");
        return true;
    }
}
```

此代码重复 YAHOO.util.Dom 3 次以获得 3 种不同的方法。每个方法都产生 3 次成员搜索过程，总共 3 次，导致此代码相当低效。一个更好的方法是将 YAHOO.util.Dom 存储在局部变量中，然后访问局部变量。

```
function toggle(element){
    var Dom = YAHOO.util.Dom;
    if (Dom.hasClass(element, "selected")){
        Dom.removeClass(element, "selected");
        return false;
    } else {
        Dom.addClass(element, "selected");
        return true;
    }
}
```

总的成员搜索次数从 9 次减少到 5 次。在一个函数中，绝不应该对一个对象成员进行超过一次的搜索，除非该值可能改变。

建议 186：推荐 100 ms 用户体验

确保网页应用程序的响应速度是一个重要的性能关注点。总的来说，大多数浏览器有一个单独的处理进程，它由两个任务所共享：JavaScript 任务和用户界面更新任务。每一刻只有其中的一个操作得以执行，也就是说，当 JavaScript 任务运行时用户界面不能对输入产生反应，反之亦然。或者说，当 JavaScript 任务运行时，用户界面就被锁定了。管理好 JavaScript 运行时间对网页应用的性能很重要。

JavaScript 和 UI 更新共享的进程通常称做浏览器 UI 线程。UI 线程围绕着一个简单的队列系统工作，任务被保存到队列中。一旦进程空闲，队列中的下一个任务将被检索和运行。这些任务不是运行 JavaScript 代码，就是执行 UI 更新，包括重绘和重排版。此进程中最令人感兴趣的部分是每次输入均导致一个或多个任务被加入队列。

在下面示例中页面包含一个按钮，按下该按钮，屏幕上显示一个消息。

```
<html>
    <head>
        <title></title>
```

```
    </head>
    <body>
        <button onclick="handleClick()">
            Click Me
        </button>
        <script type="text/javascript">
            function handleClick() {
                var div = document.createElement("div");
                div.innerHTML = "Clicked!";
                document.body.appendChild(div);
            }
        </script>
    </body>
</html>
```

当页面中的按钮被单击时，将触发 UI 线程创建两个任务并将它们添加到队列中。第一个任务是按钮的 UI 更新，需要改变按钮外观以指示它被按下了；第二个任务是 JavaScript 运行任务，包含 handleClick() 的代码，被运行的唯一代码就是这个方法和所有被它调用的方法。假设 UI 线程空闲，第一个任务被检查并运行以更新按钮外观，然后 JavaScript 任务被检查和运行。在运行过程中，handleClick() 创建了一个新的 元素，并追加在 <body> 元素上，其效果是引发另一次 UI 改变。也就是说，在 JavaScript 运行过程中，一个新的 UI 更新任务被添加到队列中，当 JavaScript 运行完之后，UI 还会再更新一次。

当所有 UI 线程任务执行之后，进程进入空闲状态，并等待更多任务被添加到队列中。空闲状态是理想的，因为所有用户操作都会立刻引发一次 UI 更新。如果用户企图在任务运行时与页面交互，不仅没有即时的 UI 更新，而且不会有新的 UI 更新任务被创建和加入队列。事实上，大多数浏览器在 JavaScript 运行时停止 UI 线程队列中的任务，也就是说，JavaScript 任务必须尽快结束，以免对用户体验造成不良影响。

浏览器在 JavaScript 运行时间上采取了限制。这是一个有必要的限制，确保恶意代码编写者无法通过无尽的密集操作锁定用户浏览器或计算机。此类限制有两个：栈尺寸限制和长时间运行脚本限制。长时间运行脚本限制有时被称做长时间运行脚本定时器或失控脚本定时器，其基本思想是浏览器记录一个脚本的运行时间，一旦到达一定限度时就终止它。当到达此限制时，浏览器会向用户显示一个对话框。

有两种方法测量脚本的运行时间。第一个方法是统计自脚本开始运行以来执行过多少语句。此方法意味着脚本在不同的机器上可能会运行不同的时间长度，可用内存和 CPU 速度可以影响一条独立语句运行所花费的时间。第二种方法是统计脚本运行的总时间。在特定时间内可运行的脚本数量也因用户机器性能差异而不同，但脚本总是停在固定的时间上。毫不奇怪，每个浏览器在对长时间运行脚本检查方法上略有不同。

- IE 设置默认限制为 500 万条语句，此限制存放在 Windows 注册表中，叫做 HKEY_CURRENT_USER\Software\Microsoft\InternetExplorer\Styles\MaxScriptStatements。
- Firefox 默认限制为 10 s，此限制存放在浏览器的配置设置中（在地址栏中输入

about:config），键名为 dom.max_script_run_time。

- Safari 默认限制为 5 s，此设置不能改变，但可以关闭，通过启动 Develop 菜单并选择“禁止失控 JavaScript 定时器”来关闭此定时限制。
- Chrome 没有独立的长时间运行脚本限制，代之以依赖它的通用崩溃检测系统来处理此类实例。
- Opera 没有长运行脚本限制，将继续运行 JavaScript 代码直至完成。由于 Opera 的结构特点，当脚本运行结束时并不会导致系统不稳定。

当浏览器的长时间运行脚本限制被触发时，不管页面上的任何其他错误处理代码，都会有一个对话框显示给用户。这是一个主要的可用性问题，因为大多数互联网用户并不精通技术，会被错误信息所迷惑，不知道应该选择哪个选项（停止脚本或允许它继续运行）。

如果脚本在浏览器上触发了此对话框，意味着脚本用太长的时间来完成任务。它还表明用户浏览器在 JavaScript 代码继续运行状态下无法响应输入。从开发者的观点来看，没有办法改变长运行脚本对话框的外观，不能检测到它，因此不能用它来提示可能出现的问题。显然，长运行脚本最好的处理办法是避免它们。

浏览器允许脚本继续运行直至某个固定的时间，这并不意味着可以这样做。事实上，JavaScript 代码持续运行的总时间应当远小于浏览器实施的限制，以创建良好的用户体验。

如果几秒钟对 JavaScript 运行来说太长了，那么多长时间是适当的呢？事实证明，即使一秒钟对脚本运行来说也太长了。一个单一的 JavaScript 操作应当使用的总时间（最大）是 100 ms。如果某接口在 100 ms 内响应用户输入，那么用户认为自己是在直接操作用户界面中的对象。响应时间超过 100 ms 意味着用户认为与接口断开了。由于 UI 在 JavaScript 运行时无法更新，因此运行时间大于 100 ms 就会使用户感受不到对接口的控制。

更复杂的是有些浏览器在 JavaScript 运行时不将 UI 更新放入队列。例如，在某些 JavaScript 代码运行时单击按钮，浏览器可能不会将重绘按钮的 UI 更新任务放入队列，也不会将由这个按钮启动的 JavaScript 任务放入队列。其结果是一个无响应的 UI，表现为挂起或冻结。

每种浏览器的行为大致相同。当脚本执行时，UI 不随用户交互而更新。此时 JavaScript 任务作为用户交互的结果被创建并放入队列，然后当原始 JavaScript 任务完成时队列中的任务被执行。用户交互导致的 UI 更新被自动跳过，因为优先考虑的是页面上的动态部分。因此，当一个脚本运行时单击一个按钮，将看不到它被按下的样子，即使它的 onclick 句柄被执行了。

尽管浏览器尝试在这些情况下做一些符合逻辑的事情，所有这些行为还是会导致一个间断的用户体验。因此，最好的方法是，通过限制任何 JavaScript 任务在 100 ms 或更少时间内完成，避免此类情况出现。这种测量应当在支持得最慢的浏览器上执行。

建议 187：使用接口解决 JavaScript 文件冲突

在同一个页面中导入多个外部 JavaScript 脚本文件时，由于这些外部文件由不同人员编

写，可能会存在重名的对象、函数、方法或变量等。当它们在一起执行时就会出现冲突，并且会产生重名覆盖现象，最终影响 JavaScript 脚本的顺利执行。例如，新建两个 JavaScript 文件，在 average_floor.js 文件中定义一个 average() 函数，该函数计算两个参数的平均值。

```
function average(a, b){
    return Math.floor((a + b)/2);
}
```

在 average_round.js 文件中也定义了一个同名函数 average()，该函数计算两个参数的平均值。

```
function average(a,b){
    return Math.round((a+b)/2);
}
```

如果在一个页面中同时导入这两个文件，那么就会出现重名函数冲突问题。解决 JavaScript 文件冲突的最好方法就是利用接口。所谓接口技术就是对函数进行封装，然后定义一个对外接口，其他文件只能通过这个接口来访问被封装的函数。JavaScript 代码封装的方法是，把函数放在如下的结构体内：

```
(function(){
    // 接口对象；
    // 被封装的函数
})()
```

这种结构体实际上就是一个闭包，创建一个封闭的结构可以有效地避免外部引入冲突。下面对上面两个文件中的函数进行封装。

封装 average_floor.js 文件中的 average() 函数的代码如下：

```
// 求两个数字的平均值——向下取整计算
// 封装 average() 函数，接口为 average_floor.average()
(function(){
    average_floor = {
        average: average
    };
    function average(a, b){
        return Math.floor((a + b)/2);
    }
})()
```

在上面这个封闭结构中，average_floor 表示接口对象，该对象拥有一个私有属性 average，并指向结构内部的 average() 函数。

封装 average_round.js 文件中的 average() 函数的代码如下：

```
// 求两个数字的平均值——四舍五入法
// 封装 average() 函数，接口为 average_round.average()
 (function(){
    average_round = {
        average: average
```

```
    };
    function average(a,b){
        return Math.round((a+b)/2);
    }
})()
```

注意，定义的接口对象名不能相同，否则也会出现冲突。

完成 JavaScript 文件的封装工作后就可以放心地在同一个页面中引用这些文件了。例如，在下面示例中同时导入这两个外部 JavaScript 脚本文件，重名函数 average() 并没有发生冲突。

```
<html>
<head>
<script type="text/javascript" src="average_floor.js"></script>
<script type="text/javascript" src="average_round.js"></script>
<title> </title></head>
<body>
<script type="text/javascript" language="javascript">
alert(average_floor.average(7.3,8.6));    // 为 7
alert(average_round.average(7.3,8.6));    // 为 8
</script>
</body>
</html>
```

建议 188：避免 JavaScript 与 CSS 冲突

虽然 JavaScript 文件与 CSS 文件差别很大，但是由于利用 JavaScript 可以控制 CSS 样式，因此它们之间仍然存在某些关联和容易混淆的概念性操作。

对于 CSS 文件来说，样式所引用的外部文件的路径都是以代码所在位置作为参考来进行设置的，而 JavaScript 恰恰相反，它是以所引用的网页位置作为参考进行设置的。

例如，有这么一个简单的站点结构，网页文件位于根目录，而 CSS 文件、JavaScript 文件和图像文件都位于根目录下 images 文件夹中，如图 9.7 所示。

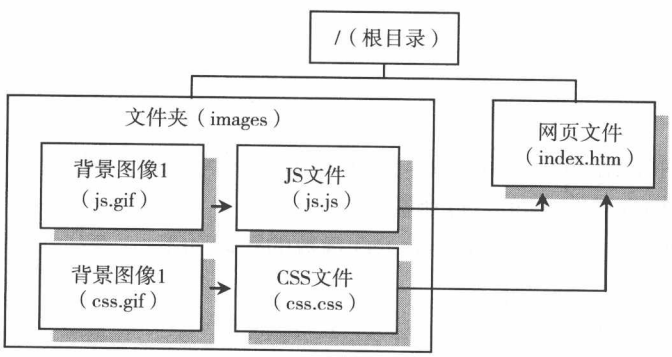


图 9.7　一个简单站点结构

下面分别使用 CSS 文件和 JavaScript 文件为网页文件中的盒子（ ）定义背景图像。在 CSS 样式表文件（css.css）中的定义方法如下：

```
#box {
    background:url(css.gif);
}
```

由于 CSS 文件与背景图像文件都在同一目录（images 文件夹）下，所以可以直接引用，而不用考虑网页文件的位置。但是，要使用 JavaScript 文件定义网页文件中盒子的背景图像，就必须考虑网页文件的具体位置。实现的 JavaScript 代码如下：

```
window.onload = function(){
    document.getElementById("box").style.backgroundImage="url(images/js.gif)";
}
```

JavaScript 文件所引用的背景图像路径是以网页文件的位置为参考进行设置的，而不用考虑 JavaScript 文件的具体位置，只要网页文件不动，JavaScript 文件所引用的路径是不会变化的。

总之，对于 JavaScript 文件与 CSS 文件，在引用外部图像文件时，它们的 URL 设置是不同的，具体区分如下。

- CSS 文件：考虑 CSS 文件与导入的外部图像文件之间的位置关系。
- JavaScript 文件：考虑网页文件与导入的外部图像文件的位置关系。

另外，当使用 CSS 和 JavaScript 同时为页面中某个元素定义样式时，JavaScript 脚本定义的样式优先级要大于 CSS 样式的优先级。例如，以上面示例为基础，然后在网页文件中同时引用 CSS 文件和 JavaScript 文件：

```
<html>
<head>
<style type="text/css">
#box {
    width:200px;
    height:200px;
}
</style>
<script type="text/javascript"  src="images/js.js"></script>
<link href="images/css.css" rel="stylesheet" type="text/css" />
</head>
<body>
<div id="box"></div>
</body>
</html>
```

此时，如果在浏览器中预览，则会显示 JavaScript 脚本定义的背景图像效果。