

# JavaScript 王者归来

作者：月影

清华大学出版社

## 第一部分 概论

### 第一章 从零开始

程序设计之道无远弗届，御晨风而返

——杰弗瑞·詹姆士

在人类漫漫的历史长河里，很难找到第二个由简单逻辑和抽象符号组合而成的，具有如此宏大信息量和丰富多彩内涵的领域。从某种意义上说，当你翻开这本书的时候，你已经踏入了一个任由你制定规则的未知世界。尽管你面对的仅仅是程序设计领域的冰山一角，但你将透过它，去领悟“道”的奥秘。在接下来的一段时间内，你会同我一起，掌握一种简单而优雅的神秘语言，学会如何将你的意志作用于它。这种语言中所蕴涵着的亘古之力，将为你开启通往神秘世界的大门……

## 1.1 为什么选择 JavaScript?

在一些人眼里，程序设计是一件神秘而浪漫的艺术工作，对他们来说，一旦选定某种编程语言，就会像一个忠贞的信徒一样坚持用它来完成任何事情，然而我不是浪漫的艺匠，大多数人都都不是，很多时候我们学习一种新技术的唯一目的，只是为了把手中的事情做得更好。所以，当你面对一项陌生的技术时，需要问的第一个问题往往是，我为什么选择它，它对我来说，真的如我所想的那么重要吗？


好，让我们带着问题开始。

### 1.1.1 用户的偏好：B/S 模式


如果你坚持站在专业人员的角度，你就很难理解为什么 B/S 模式会那么受欢迎。如果你是一个资深的程序员，有时候你甚至会对那些 B/S 模式的东西有一点点反感。因为在你看来，浏览器、表单、DOM 和其他一切与 B/S 沾边的东西，大多是行为古怪而难以驾驭的。以你的经验，你会发现实现同样的交互，用 B/S 来做通常会比用任何一种客户端程序来做要困难得多。

如果你尝试站在用户的角度，你会发现为什么大多数最终用户对 B/S 模式却是如此的青睐。至少你不必去下载和安装一个额外的程序到你的电脑上，不必为反复执行安装程序而困扰，不必整天被新的升级补丁打断工作，不必理会注册表、磁盘空间和一切对普通用户来说有点头疼的概念。如果你的工作地点不是

固定的办公室，你日常工作的 PC 也不是固定的一台或者两台，那么，B/S 的意义对你而言或许比想象的还要大。

 大多数情况下，客户更偏好使用浏览器，而不是那些看起来比较专业的软件界面，专业人员则恰恰相反。

总之，用户的需求让 B/S 模式有了存在的理由，而迅速发展的互联网技术则加速了 B/S 应用的普及。随着一些优秀的 Web 应用产品出现，不但唤起了用户和业内人士对 Ajax 技术的关注，也令 Web 领域内的一个曾经被无数人忽视的脚本语言——JavaScript 进入了有远见的开发人员和 IT 经理人的视线。于是在你的手边，也多了现在这本教程。


 编写本书的时候，在 TIOBE 编程社区最新公布的数据中，JavaScript 在世界程序开发语言中排名第十，这意味着 JavaScript 已经正式成为一种被广泛应用的热门语言。

## 1.1.2 在什么情况下用 JavaScript

一路发展到今天，JavaScript 的应用范围已经大大超出一般人的想象，但是，最初的 JavaScript 是作为嵌入浏览器的脚本语言而存在，而它所提供的那些用以表示 Web 浏览器窗口及其内容的对象简单实用，功能强大，使得 Web 应用增色不少，以至于直到今天，在大多数人眼里，JavaScript 表现最出色的领域依然是用户的浏览器，即我们所说的 Web 应用的客户端。客户端浏览器的 JavaScript 应用也正是本书讨论的重点内容。

作为一名专业程序员，当你在面对客户的时候，经常需要判断哪些交互需求是适合于 JavaScript 来实现的。而作为一名程序爱好者或者是网页设计师，你也需要了解哪些能够带给人惊喜的特效是能够由 JavaScript 来实现的。总之一句话，除了掌握 JavaScript 本身，我们需要学会的另一项重要技能是，在正确的时候、正确的地方使用 JavaScript。对于 JavaScript 初学者来说学会判断正确使用的时机有时候甚至比学会语言本身更加困难。

作为项目经理，我经常接受来自客户的抱怨。因此我很清楚我们的 JavaScript 在带给客户好处的同时制造了太多的麻烦，相当多的灾难是由被错误使用的 JavaScript 引起的。一些代码本不应该出现在那个位置，而另一些代码则根本就不应当出现。我曾经寻访过问题的根源，发现一个主要的原因由于 JavaScript 的过于强大（在后面的小节中我们将会提到，另一个同样重要的原因是“脚本诱惑”），甚至超越了浏览器的制约范围，于是麻烦就不可避免的产生，这就好像你将一个魔鬼放入一个根本就不可能关住它的盒子里，那么你也就无法预料魔鬼会做出任何超出预期的举动。

 毫无疑问，正确的做法是：不要放出魔鬼。所以，JavaScript 程序员需要学会的第一个技巧就是掌握在什么情况下使用 JavaScript 才是安全的。

在什么情况下用 JavaScript? 给出一个简单的答案是：在任何**不得不用**的场合使用，除此以外，不要在任何场合使用！无懈可击的应用是不用，除非你确实无法找到一个更有效更安全的替代方案。也许这个答案会让读到这里的读者有些郁闷，但是，我要很严肃地提醒各位，由于 JavaScript 比大多数人想象的要复杂和强大得多，所以它也比大多数人想象得要危险得多。在我的朋友圈子里，许多资深的 JavaScript 程序员（包括我在内）偶尔也不得不为自己一时疏忽而做出的错误决定让整个项目团队在“脚本泥潭”中挣扎好一阵子。所以这个建议从某种意义上说也是专家们的血泪教训。最后向大家陈述一个令人欣慰的事实，即使是像前面所说的这样，在 Web 应用领域，JavaScript 的应用范围也仍然是相当广泛的。

➤ 在本节的最后三个小节里，我们将进一步展开讨论关于 JavaScript 使用的话题。

## 1.1.3 对 JavaScript 的一些误解

JavaScript 是一个相当容易误解和混淆的主题，因此在对它进一步研究之前，有必要澄清一些长期存在的有关该语言的误解。

### 1.1.3.1 JavaScript 和 Java

这是最容易引起误会的一个地方，这个 Java-前缀似乎暗示了 JavaScript 和 Java 的关系，也就是 JavaScript 是 Java 的一个子集。看上去这个名称就故意制造混乱，然后随之而来的是误解。事实上，这两种语言是完全不相干的。



JavaScript 和 Java 的语法很相似，就像 Java 和 C 的语法相似一样。但它不是 Java 的子集就像 Java 也不是 C 的子集一样。在应用上，Java 要远比原先设想的好得多（Java 原称 Oak）。

JavaScript 的创造者是 Brendan Eich，最早的版本在 NetScape 2 中实现。在编写本书时，Brendan Eich 在 Mozilla 公司任职，他本人也是 JavaScript 的主要革新者。而更加有名的 Java 语言，则是出自 Sun Microsystems 公司的杰作。

JavaScript 最初叫做 LiveScript，这个名字本来并不是那样容易混淆，只是到最后才被改名为 JavaScript，据说起同 Java 相似的名字纯粹是一种行销策略。

尽管 JavaScript 和 Java 完全不相干，但是事实上从某种程度上说它们是很好的搭档。JavaScript 可以控制浏览器的行为和内容，但是却不能绘图和执行连接（这一点事实上并不是绝对的，通过模拟是可以做到的）。而 Java 虽然不能在总体上控制浏览器，但是却可以绘图、执行连接和多线程。客户端的 JavaScript 可以和嵌入网页的 Java applet 进行交互，并且能够对它执行控制，从这一意义上来说，JavaScript 真的可以脚本化 Java。

### 1.1.3.2 披着 C 外衣的 Lisp

JavaScript 的 C 风格的语法，包括大括号和复杂的 for 语句，让它看起来好像是一个普通的过程式语言。这是一个误导，因为 JavaScript 和函数式语言如 Lisp 和 Scheme 有更多的共同之处。它用数组代替了列表，用对象代替了属性列表。函数是第一型的。而且有闭包。你不需要平衡那些括号就可以用  $\lambda$  算子。

► 关于 JavaScript 闭包和函数式的内容，在本书的第 23 章中会有更详细的介绍。

### 1.1.3.3 思维定势

JavaScript 是原被设计在 Netscape Navigator 中运行的。它的成功让它成为几乎所有浏览器的标准配置。这导致了思维定势。认为 JavaScript 是依赖于浏览器的脚本语言。其实，这也是一个误解。JavaScript 也适合很多和 Web 无关的应用程序。



早些年在学校的时候，我和我的实验室搭档曾经研究过将 JavaScript 作为一种 PDA 控制芯片的动态脚本语言的可行性，而在我们查阅资料的过程中发现一些对基于嵌入式环境的动态脚本语言实现的尝试，我们有理由相信，JavaScript 在某些特定的嵌入式应用领域中也能够表现得相当出色。

### 1.1.3.4 业余爱好者

一个很糟糕的认知是：JavaScript 过于简朴，以至于大部分写 JavaScript 的人都不是专业程序员。他们缺乏写好程序的修养。JavaScript 有如此丰富的表达能力，他们可以任意用它来写代码，以任何形式。

事实上，上面这个认知是曾经的现实，不断提升的 Web 应用要求和 Ajax 彻底改变了这个现实。通过学习本书，你也会发现，掌握 JavaScript 依然需要相当高的专业程序员技巧，而不是一件非常简单的事情。不过这个曾经的现实却给 JavaScript 带来了一个坏名声——它是专门为外行设计的，不适合专业的程序员。这显然是另一个误解。




推广 JavaScript 最大的困难就在于消除专业程序员对它的偏见，在我的项目团队中许多有经验的 J2EE 程序员却对 JavaScript 停留在一知半解甚至茫然的境地，他/她们不愿意去学习和掌握 JavaScript，认为这门脚本语言是和浏览器打交道的美工们该干的活儿，不是正经程序员需要掌握的技能。这对于 Web 应用开发

来说，无疑是一个相当不利的因素。

### 1.1.3.5 面向对象

JavaScript 是不是面向对象的？它拥有对象，可以包含数据和处理数据的方法。对象可以包含其它对象。它没有类（在 JavaScript 2.0 真正实现之前），但它却有构造器可以做类能做的事，包括扮演类变量和方法的容器的角色。它没有基于类的继承，但它有基于原型的继承。两个建立对象系统的方法是通过继承和通过聚合。JavaScript 两个都有，但它的动态性质让它可以在聚合上超越。

一些批评说 JavaScript 不是真正面向对象的因为它不能提供信息的隐藏。也就是，对象不能有私有变量和私有方法：所有的成员都是公共的。但随后有人证明了 JavaScript 对象可以拥有私有变量和私有方法。另外还有批评说 JavaScript 不能提供继承，但随后有人证明了 JavaScript 不仅能支持传统的继承还能应用其它的代码复用模式。

 说 JavaScript 是一种基于对象的语言，是一种正确而略显保守的判断，而说 JavaScript 不面向对象，在我看来则是错误的认知。事实上有充足的理由证明 JavaScript 是一种的面向对象的语言，只是与传统的 class-based OO（基于类的面向对象）相比，JavaScript 有它与众不同的地方，这种独特性我们称它为 prototype-based OO（基于原型的面向对象）。

➤ 关于 JavaScript 面向对象的内容，在本书的第 21 章中会有更详细的介绍。

### 1.1.3.6 其他误解


除了以上提到的几点之外，JavaScript 还有许多容易令人迷惑和误解的特性，这些特性使得 JavaScript 成为世界上最被误解的编程语言。

➤ 如果读者对这方面有兴趣，可以详细阅读下面这篇文章

<http://javascript.crockford.com/javascript.html> [Douglas Crockford]


## 1.1.4 警惕！脚本诱惑

前面我们提到过，许多专业程序员拒绝去了解如何正确使用 JavaScript，另一些则是缺乏对 JavaScript 足够的认知和应用经验。但是在 B/S 应用中，相当多的情况下，要求开发人员不得不采用 JavaScript。于是，一个问题产生了，大量的 JavaScript 代码拷贝出现在页面的这个或者那个地方，其中的大部分是不必要的，另一部分可能有缺陷。我们的开发人员没有办法（也没有意识到）去判断这些代码是否必要，以及使用它们会带来哪些问题。

 如果你的 B/S 应用中的 JavaScript 不是由专业的 JavaScript 程序员来维护的，那么当你对你的开发团队进行一次小小的代码走查时，你甚至可能会发现 90% 的 JavaScript 代码被错误地使用，这些错误使用的代码浪费了用户大量的网络带宽、内存和 CPU 资源，提升了对客户端配置的要求，降低了系统的稳定性，甚至导致许多本来可以避免的安全问题。

由于浏览器的 JavaScript 可以方便地被复制粘贴，因此，一个特效或者交互方式往往在真正评估它的必要性之前便被采用——客户想要它，有人使用过它，程序员复制它，而它就出现在那儿，表面上看起来很完美，于是，所谓的脚本诱惑就产生了。

事实上，在我们真正使用 JavaScript 之前，需要反复问自己一个重要问题是，究竟是因为有人想要它，还是因为真正有人需要它。在你驾驭 JavaScript 马车之前，你必须学会抵制脚本诱惑，把你的脚本用在必要的地方，永远保持你的 Web 界面简洁，风格一致。

 在用户眼里，简洁一致的风格与提供强大而不常用的功能和看起来很 COOL 而实际上没有什么功用的界面特效相比起来，前者更能令他们觉得专业。毕竟，大部分用户和你我一样，掌握一个陌生的环境和新的技能只是为了能够将事情做得更快更好。除非你要提供的是一个类似于 Qzone 之类的娱乐程序，你永远也不要大量地使用不必要的 JavaScript。

## 1.1.5 隐藏在简单表象下的复杂度

专业人员不重视 JavaScript 的一个重要原因是，他们觉得 JavaScript 是如此的简单，以至于不愿意花精力去学习（或者认为不用学习就能掌握）。前面提到过的，这实际上是一种误解。事实上，在脚本语言中，JavaScript 属于相当复杂的一门语言，它的复杂程度未必逊色于 Perl 和 Python。



另一个业内的偏见是脚本语言都是比较简单的，实际上，一门语言是否脚本语言往往是它的设计目标决定的，简单与复杂并不是区分脚本语言和非脚本语言的标准。JavaScript 即使放到非脚本语言中来衡量，也是一门相当复杂的语言。

之所以很多人觉得 JavaScript 过于简单，是因为他们大量使用的是一些 JavaScript 中看似简单的文法，解决的是一些看似简单的问题，真正复杂而又适合 JavaScript 的领域却很少有人选择 JavaScript，真正强大的用法很少被涉及。JavaScript 复杂的本质被一个个简单应用的表象所隐藏。

我曾经给一些坚持认为 JavaScript 过于简单的开发人员写过一段小代码，结果令他们中的大部分内行人大惊失色，那段代码看起来大致像下面这个样子：

```
var a = [-1,-1,1,-3,-3,-3,2,2,-2,-2,3,-1,-1];
function f(s, e)
{
    var ret = [];
    for(var i in s){
        ret.push(e(s[i]));
    }
    return ret;
}
var b = f(a, function(n){return n>0?n:0});
alert(b);
```



这是本书中出现的第一段 JavaScript 代码，也许现在你看来，它有那么一点点令人迷惑，但是不要紧，在本书后面的章节中，你会慢慢理解这段代码的含义以及它的无穷妙味。而现在你完全可以跳过它的实际内容，只要需要知道这是一段外表看起来简单的魔法代码就够了。

因为这段代码而尖叫的不仅仅包括我的这些程序员朋友，事实上，更兴奋的是另一些电子领域的朋友，他们写信给我反馈说，在此之前他们从来没有见到过如此形式简洁而优雅的数字高通滤波器，更令人欣喜的是，它的阈值甚至是可调节的：

```
var b = f(a, function(n){return n>=-1?n:0});
```

如果你想要，它也很容易支持低通滤波：

```
var b = f(a, function(n){return n<0?n:0});
```

用一个小小的堆栈或者其他伎俩，你也可以构造出一族差分或者其他更为复杂的数字设备，而它们明显形式相近并且结构优雅。

总之，不要被简单的表象所迷惑，JavaScript 的复杂度往往很大程度上取决于你的设计思路和你的使用技巧。JavaScript 的确是一门可以被复杂使用的程序设计语言。

## 1.1.6 令人迷惑的选择：锦上添花还是雪中送炭

本节最后的这个话题在前面已经被隐讳地提到过多次，实际上，本小节围绕的话题依然是什么时候使用 JavaScript。一种比较极端的观点是在必须的时候采用，也就是前面所说的不得不用场合，另一种比

较温和一点的观点则坚持在需要的时候使用，这种观点认为当我们可以依靠 JavaScript 令事情变得更好的时候，我们就采用它。

事实上，就我个人而言，比较支持“必须论”，这是因为从我以往的经验来看，JavaScript 是难以驾驭的，太多的问题由使用 JavaScript 不当而产生，其中的一部分相当令人困扰，彻底解决它们的办法就是尽可能降低 JavaScript 的使用频率，也尽可能将它用在真正适合它的地方。当然万事没有绝对，在何时使用 JavaScript 永远是一个难题，然而不管怎么说同“锦上添花”相比，JavaScript 程序员也许应当更多考虑的是如何“雪中送炭”。

## 1.1.7 回到问题上来

本节要解决的问题是为什么选择 JavaScript，然而在相当多的篇幅里，我们都在试图寻找一些少用和不用 JavaScript 的理由，尽管如此，抛开大部分不适合 JavaScript 的位置和时机，浏览器上依然会经常地见到 JavaScript 的身影，对于浏览器来说，JavaScript 实在是一个不可缺少的修饰。



你再也找不到任何一种优雅简朴的脚本语言如此适合于在浏览器中生存。在本书的第 2 章，我们将具体接触嵌入浏览器中的 JavaScript。

最后，用一句话小结本节的内容——我们之所以选择 JavaScript，是因为：Web 应用需要 JavaScript，我们的浏览器、我们的程序员和我们的用户离不开它。

## 1.2 JavaScript 的应用范围

我记得在前面依稀提到过，JavaScript 的应用范围相当广泛，除了最常见的客户端浏览器之外，JavaScript 还被应用在一部分服务器端的环境、桌面程序和其他一些应用环境中。

### 1.2.1 客户端的 JavaScript

目前绝大多数浏览器中都嵌入了某个版本的 JavaScript 解释器。当 JavaScript 被嵌入客户端浏览器后，就形成了客户端的 JavaScript。这是迄今为止最常见也最普通的 JavaScript 变体。大多数人提到 JavaScript 时，通常指的是客户端的 JavaScript，本书重点介绍的内容，也是 JavaScript 的客户端应用。

➤ 在后面的章节中提到的“浏览器中的 JavaScript”通常也是特指客户端的 JavaScript。

当一个 Web 浏览器嵌入了 JavaScript 解释器时，它就允许可执行的内容以 JavaScript 的形式在用户客户端浏览器中运行。下面的例子展示了一个简单的嵌入网页中的 JavaScript 程序。

例 1.1 经典程序 Hello World! 的 JavaScript 实现

```
<html>
<head>
<title>Example 1.1 Hello World!</title>
</head>
<body>
  <h1>
    <script type="text/JavaScript">
      <!--
        document.write("Hello World!");
      -->
    </script>
```

```
<noscript>您的浏览器不支持 JavaScript, 请检查浏览器版本或者安全设置, 谢谢! </noscript>
</h1>
<hr/>
<p>第一个例子展示了 document.write 是浏览器提供的一个方法, 用来向 document 文档对象输出内容,
至于什么是文档对象, 在本书的第三部分将有详细的介绍。</p>
</body>
</html>
```

把这个脚本装载进一个启用 JavaScript 的浏览器后, 就会产生如图 1.1 所示的输出。

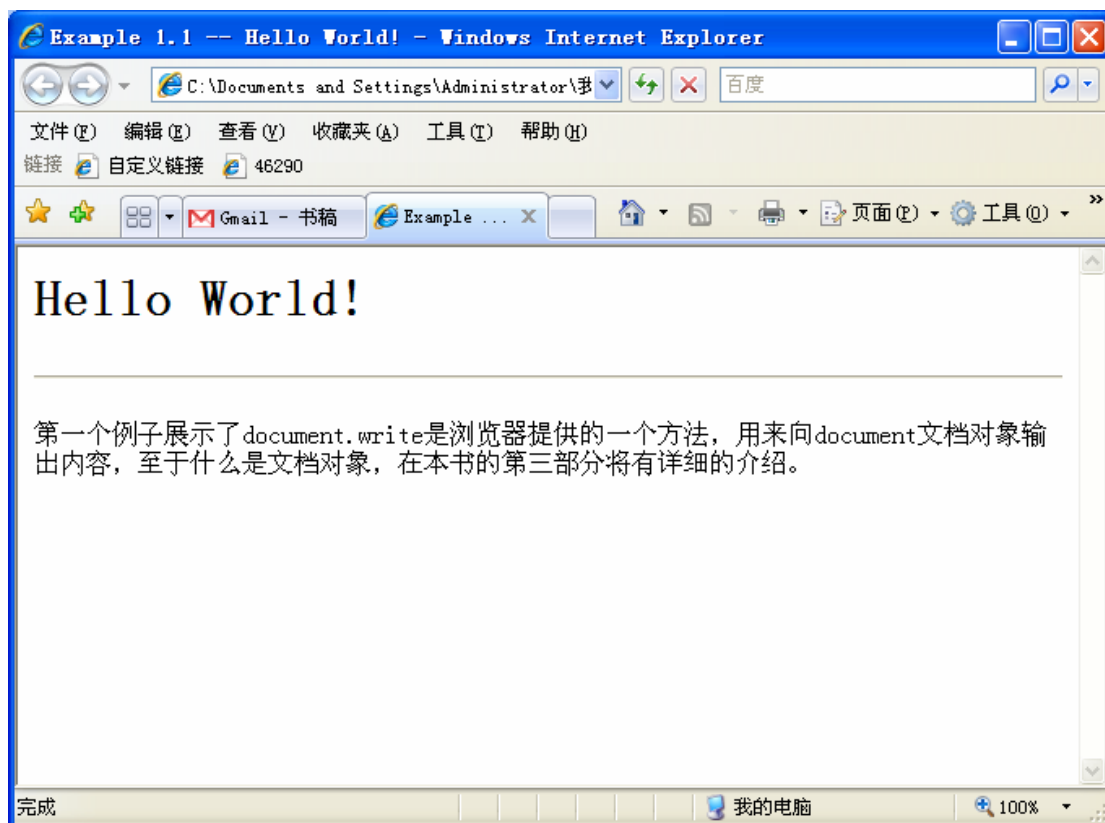


图 1.1 Hello World

如果你看到的是“您的浏览器不支持 JavaScript.....”的字样, 那么需要检查浏览器的版本和安全设置, 以确定你的浏览器正确支持 JavaScript。

**小技巧:** `<noscript>`和`</noscript>`是一种防御性编码, 如果用户的浏览器不支持 JavaScript 或者设置了过高的安全级别, 那么就会显示出相应的提示信息, 避免了在用户不知情的情况下停止运行或者得到错误结果。

从例子中可以看到, 标记`<script>`和`</script>`是用来在 HTML 中嵌入 JavaScript 代码的。

➤ 我们将在第 2 章和第 22 章中了解更多有关`<script>`标记的内容。

在这个例子中, 方法 `document.write()`用来向 HTML 文档输出文本, 在本书的后续章节中, 我们会多次见到它。

JavaScript 当然不仅仅是用来简单地向 HTML 文档输出文本内容的, 事实上它可以控制大部分浏览器相关的对象, 浏览器为 JavaScript 提供了强大的控制能力, 使得它不仅能够控制 HTML 文档的内容, 而且能够控制这些文档元素的行为。在后面的章节里, 我们会了解到 JavaScript 通过浏览器对象接口访问和控制浏览器元素, 通过 DOM 接口访问和控制 HTML 文档, 通过给文档定义“事件处理器”的方式响应由用户触发的交互行为。

## 1.2.2 服务器端的 JavaScript

相信大多数人对客户端执行的 JavaScript 并不陌生，而服务器端的 JavaScript 就鲜有人知了。不少应用服务器提供了对 JavaScript 的支持，比较典型的如 Microsoft 的 IIS，还有一些版本的 Java 应用服务器提供了在 Servlet 容器中执行 JavaScript 的能力。



在基于 IIS 的 asp 应用中，将一段 JavaScript 声明为服务器端代码，只需要在 <script> 标签中指定属性 `runat = "server"`，这样，这段代码将会在服务器端被执行。

Netscape 开发了一套用 Java 实现的 JavaScript 1.5 解释器，它是作为开放资源发布的，被称为 Rhino，目前通过 Mozilla 组织可以得到它。事实上正是 Rhino 的存在向 Java 应用服务器提供了容器对 JavaScript 的支持。另一套同样由 Mozilla 组织提供的 JavaScript 1.5 解释器是用 C 语言实现的，被称为 SpiderMonkey。

## 1.2.3 其他环境中的 JavaScript

除了 Web 应用的相关领域之外，JavaScript 还能够在多种不同的环境中运行。在较早一些的时候，Microsoft 已经在 Windows 系统中支持一种 HTA 应用，这可以看作是由 JavaScript + HTML 编写的类似 GUI 的应用程序。在 .net framework 的新版本中，Microsoft 更是直接支持了 Jscript.net。



Jscript.net 是一个较少人知的秘密，Microsoft 并未在 Visual Studio.net 中集成 Jscript.net 的可视化编辑器，却将 Jscript.net 在 .net 的核心环境中实现了。Jscript.net 可以看作是一种 CLR 托管的 JavaScript，实际上是 .net 家族的一种编程语言实现。安装了较新版本 .net framework 的读者可以试着编写 Jscript.net 并在命令行中编译执行，有关 Jscript.net 的更多内容可参考 Microsoft 的官方文档。

前面也提到过 Mozilla 组织提供的开源 JavaScript 解释器，实际上 Microsoft 公司和 Netscape 公司都向那些想把 JavaScript 解释器嵌入自己应用程序的公司和设计者开放了它们的 JavaScript 解释器。所以如果程序员在其他应用中需要 JavaScript 的支持，可以比较容易地获得 JavaScript 解释器的不同版本。随着计算机技术的发展，越来越多的应用程序将某种动态语言作为嵌入式脚本以增强系统的交互能力和扩展性，我们有理由相信，在可选择的动态语言中，JavaScript 是一种非常优秀的备选方案。我们期待着看到越来越多的应用程序将 JavaScript 作为嵌入式脚本语言。



应用程序支持脚本语言已经成为一种趋势。例如在 WinCVS 中直接引入了 Python 作为命令行脚本扩展，但那还不是一种真正的嵌入式脚本实现。在 AutoCAD 中引入了 Lisp 作为嵌入式脚本语言，而 LabView 则有自己的类 C 脚本实现。相对更为著名的 ActionScript 是 Macromedia 公司的 Flash 中所支持的动态脚本语言，有趣的是，ActionScript 是在 ECMAScript 标准发布后被模型化的，在后面的章节里，你会了解到 ECMAScript 实际上是标准化的 JavaScript。不过，有些遗憾的是，ActionScript 并不是真正的 JavaScript。

## 1.3 JavaScript 的版本

JavaScript 和其他的一些脚本语言一样，有着各种各样的实现版本，虽然早在 JavaScript 1.3 实现的时候（大约是 1999 年 12 月），ECMA 组织已经标准化了 ECMAScript v1 版本，但是，如同 ECMA 组织努力地标准化一样，JavaScript 从来也没有停止过它基于各种不同浏览器的差异化发展。一方面这种差异化 JavaScript 给开发者带来了不小的困扰，然而另一方面这种百花齐放式的差异化发展也将越来越多的优秀特性加入到 JavaScript 中，最终使得 JavaScript 迅速发展成为一种强大而优秀的脚本语言。



### 1.3.1 浏览器中的 JavaScript 版本

JavaScript 语言已经发展几年了，Netscape 公司发布了该语言的多个版本。Microsoft 公司也发布了 JavaScript 语言的相应版本，名为 Jscript。表 1.1 列出了这些版本

表 1.1 JavaScript 的版本

版本	说明
JavaScript 1.0	该语言的原始版本，目前已经基本上被废弃。由 Netscape 2 实现
JavaScript 1.1	引入了真正的 Array 对象，消除了大量重要的错误。由 Netscape 3 实现
JavaScript 1.2	引入了 switch 语句、正则表达式和大量其他特性，基本上符合 ECMA v1，但是还有一些不兼容性，由 Netscape 4 实现
JavaScript 1.3	修正了 JavaScript1.2 的不兼容性，符合 ECMA v1。由 Netscape 4.5 实现
JavaScript 1.4	只在 Netscape 的服务器产品中实现
JavaScript 1.5	引入了异常处理，符合 ECMA v3。由 Netscape 6 实现
Jscript 1.0	基本上相当于 JavaScript 1.0，由 IE 3 的早期版本实现
Jscript 2.0	基本上相当于 JavaScript 1.1，由 IE 3 的后期版本实现
Jscript 3.0	基本上相当于 JavaScript 1.3，符合 ECMA v1。由 IE 4 实现
Jscript 4.0	没有任何浏览器实现它
Jscript 5.0	支持异常处理，部分符合 ECMA v3。由 IE5 实现
Jscript 5.5+	基本上相当于 JavaScript 1.5。完全符合 ECMA v3。IE 5.5 实现 Jscript 5.5，IE 6 实现 Jscript 5.6，IE 7 实现 Jscript 5.7

### 1.3.2 其他版本

ECMA (<http://www.ecma.ch>) 组织发布了三个版本的 ECMA-262 标准，该标准标准化了 JavaScript 语言。ECMA 组织还整理了 ECMAScript 的第 4 个版本 (<http://www.mozilla.org/js/language/es4/index.html>)。几乎同一时间 Mozilla 组织开始设计 JavaScript 2.0，在本书开始编写的时候，还未听说 ECMAScript v4 和 JavaScript 2.0 在任何浏览器上实现。表 1.2 列出了 ECMAScript 的标准化版本和 JavaScript 2.0。

表 1.2 ECMAScript 标准化版本和 Mozilla JavaScript 2.0

版本	说明
ECMA v1	该语言的第一个标准版本，标准化了 JavaScript 1.1 的基本特性，并添加了一些新特性，没有标准化 switch 语句和正则表达式。与 JavaScript1.3 和 Jscript3.0 的实现一致
ECMA v2	该标准的维护版本，添加了说明，但没有定义任何新特性
ECMA v3	标准化了 switch 语句、正则表达式和异常处理，与 JavaScript 1.5 和 Jscript 5.5 的实现一致
ECMA v4	增加了强类型、名字空间、类修饰符、操作符重载等，极大强化了 JavaScript 面向对象的能力
JavaScript 2.0	本书编写的时候还未见在任何浏览器上实现的 JavaScript 未来版本，完全符合 ECMA v4，并增加了 include 语句

在本书的第 19 章里，将会更加详细地谈到这些版本和标准的相关内容。

## 1.4 一些值得留意的特性

JavaScript 为什么吸引着这么多的爱好者学习、研究和进行开发，一方面它确实拥有强大的功能，能够支持你开发出优秀的 Web 应用产品，另一方面它也是有趣的，它的某些特性本身就能够令人感受到某种乐趣。

### 1.4.1 小把戏：神奇的魔法代码

是什么使得 JavaScript 不同于其他程序设计语言，在浏览器修饰方面表现出其优异的特性？毫无疑问，JavaScript 在 Web 应用领域受到的好评，既源于它自身灵活的动态特性，也源于浏览器对它充分的支持。



JavaScript 是一种深受浏览器“宠爱”的语言，浏览器为其提供了丰富的资源和广阔的舞台。

下面的这段代码在网上广为流传，被众多 JavaScript 爱好者奉为代表 JavaScript 魔力的经典：

例 1.2 神奇的“魔法代码”

```
JavaScript:R=0; x1=.1; y1=.05; x2=.25; y2=.24; x3=1.6; y3=.24; x4=300; y4=200; x5=300; y5=200; DI=document.images; DIL=DI.length; function A(){for(i=0; i<DIL; i++){DIS=DI[ i ].style; DIS.position='absolute'; DIS.left=Math.cos(R*x1+i*x2+x3)*x4+x5; DIS.top=Math.sin(R*y1+i*y2+y3)*y4+y5;R++;}setInterval('A()', .5); void(0);
```

打开一个带有几张图片的网页（图片稍微多一些并且每张图片大小相当的话，效果会比较好），将上面这段代码输入到 IE 浏览器的地址栏（不要换行），敲回车，就会看到页面上的所有图片围成一圈绕着一个点旋转。事实上，这是一段有些故弄玄虚的指令，很容易让初学者觉得新奇和迷惑，而对于资深的 JavaScript 程序员来说，它几乎恰如其分地表现出了 JavaScript 大部分操作客户端浏览器的特性（除了故意的糟糕排版和蹩脚的变量命名方式之外）。

在这里，我们先简要地列举一下这些特性，而具体的内容将会在后续的章节里详细展开。

首先，一些浏览器（不是所有的）支持 JavaScript 伪协议，你可以在浏览器的地址栏里通过“JavaScript:”的形式来执行 JavaScript 代码。实际上这种良好的执行方式为 JavaScript 爱好者带来了一个便捷的测试手段，使得他们能够以类似命令行的方式来简易地测试一个没有用过的 JavaScript 特性，而不必写一大堆文本和 HTML 标签。

其次，JavaScript 支持缺省声明直接赋值的方式来使用全局变量，唯一的约束是命名规则和保留字，作为一种脚本语言，这个特性无疑提供了一种快速便利的执行手段，缺点则也是很明显的，缺乏严谨的约束，为不良代码的产生提供了可能。



大部分程序设计语言中，变量被设计为在声明之后引用，也就是说，要使用某个对象，必须先告知该对象存在之后才能赋值，即先“（在……之中）有一个 A”，然后才能说“A 是一个……”，在 JavaScript 中，如果对象的作用域是全局的，则不强制要求“有一个 A”的声明。关于变量定义和声明的内容，在第 4 章将会有详细的讨论。

作为程序员，如果你不管理好自己代码里的变量，那么总有一天你或者你的继任会为它们整天头疼不已。可能出现在任何地方的变量，像缺乏约束四处乱窜的野马，随时都可能导致整个系统崩溃。一个好的习惯是用良好的自我约束来限制变量的定义和使用，并且避免定义过多的全局变量。在 JavaScript 中，利用闭包是一种代替临时变量的好习惯，在后续的章节中，我们会详细讨论这些现在听起来有些深奥的技巧。

注意到 `document.images` 的用法，这个指令枚举出页面文档中所有的图片元素，并把这个元素集合的引用赋值给临时变量 `DI`。

```
DI=document.images;
```

`Document` 是一个非常有用的接口，它是 JavaScript 访问页面文档对象的主要方式。除了访问图片的

document.images 之外，document 提供的属性还能够方便地引用页面文档对象中的表单、链接和其他元素。

document 接口还提供了一组更为标准的方法来创建和访问文档元素，它们是 document.getElementById、document.getElementsByTagName 和 document.createElement，通常我们认为以上三个方法是 document 对象提供的最主要的 DOM 接口。关于 DOM 话题我们将会在第 12 章里详细讨论。



除了 Document 之外，另一个有用的接口是 Window，它提供了对浏览器、窗口、框架、对话框以及状态栏的访问方法，在第三部分里，我们会用很多篇幅仔细地讨论以上两个接口。

另一个需要重点关注的特性是函数定义，function A()声明了一个名字叫做“A”的函数，其后的一对大括号内的指令是对这个函数的定义。提供函数文法使得 JavaScript 成为一种完善的过程式语言。

```
function A(){for(i=0; i<DIL; i++){DIS=DI[ i ].style; DIS.position='absolute'; DIS.left=Math.cos(R*x1+i*x2+x3)*x4+x5; DIS.top=Math.sin(R*y1+i*y2+y3)*y4+y5}R++}
```



除了命名函数之外，JavaScript 提供了缺省函数名的定义方法，在某些特定情况下，定义在函数体内的匿名函数在运行的过程中形成“闭包”。除此以外，JavaScript 还提供了一种 new 操作符来实例化函数对象。以上的两个特性使得 JavaScript 同时兼有函数式和面向对象的特点，也使得函数成为了 JavaScript 的第一型。在第 6 章、第 22 章、第 23 章我们将会分别详细讨论 JavaScript 函数的各种特性和使用技巧。

在函数定义体内，我们可以看到像 Math.cos(R\*x1+i\*x2+x3)这样的用法，Math 是 JavaScript 的一个有用的内置对象，它为 JavaScript 的使用者提供了一组有用的数学函数，Math.cos 返回表达式的余弦值。

在这之后我们通过一个循环将数学计算的结果赋值给 document.images 集合中提供的图片样式属性，这里引用的是 style.top 和 style.left 属性，这两个属性分别定义了图片元素左上角距参照系原点的横坐标和纵坐标的值，默认的单位是像素点（关于元素的定位问题我们将会在后续的章节中有详细的讨论），这样我们相当于将页面文档中的图片元素抽取出来，重新计算了它们的位置，并按照新的位置进行排列。

最后，我们在排列的过程中改变参量 R 的值，并通过定时器函数 setInterval 每隔 5 个毫秒调用一次 A() 函数，就实现了例子中的图片旋转的特效。

```
setInterval('A()',5);
```



setInterval 是 JavaScript 中一个重要的系统函数，它提供了一种定时执行函数的方法，另一个类似的函数是 setTimeout，我们将在第 16 章里详细地讨论它们。在一些稍为复杂的应用中，setInterval 和 setTimeout 被大量用于实现动态效果、模拟异步执行、实现拦截器和一些控制型模式，以及实现自定义事件接口。

在结束话题之前，顺便提一个不太常用的特性。也许你已经注意到例子末尾的那个不起眼的 void(0)，如果你将它去掉，你会发现一切令人惊讶的特效都消失了，甚至连浏览器中的页面也不见踪迹，取而代之的是孤零零地显示在浏览器窗口左上角的一组奇怪的数字，这是怎么回事呢？

原来 JavaScript 伪协议默认将页面带到一个新的 document 中并显示程序返回结果，所以正常情况下运算的结果会在一个空文档对象内显示，这样也就没有图片可以展现特效，而 void(0)阻止了这个跳转动作。

void 是 JavaScript 的一个特殊的运算符，它的作用是舍弃任何参数表达式的值，这意味着要求解析器检验并计算参数表达式内容，但是却忽略其结果。如果你刻意去检查 void 运算的返回值，会发现它返回一个 undefined 标记（事实上任何一个不带 return 指令的函数运算的默认返回值都是 undefined）。在浏览器的缺省行为中，undefined 阻止了页面的跳转。



undefined 对于 JavaScript 来说是一个特殊的值，它令我联想到了某些宗教和物理学。如果说程序中的 null 代表着“空”的话，那么 undefined 则代表着“无”。“空”依然是一种存在，而“无”则是存在的对立面。JavaScript 的一个巧妙设计就在于把“无”概念化了，由于它没有强制检验对象存在的机制，所以它承认“无”的概念，任何一个未经定义和使用的标识，均可以用“无”来表示。这个“无”在 JavaScript 文法中即是 undefined。

typeof 操作符用来检查变量的类型，如果你直接引用一个未声明的标识，或者声明了一个变量却未对

其进行赋值，那么 `typeof` 操作返回的结果将是 `undefined`。事实上我觉得最好能够用一种新的标识来区分未声明和已声明未赋值的变量，如 `unknown`（未知）区别 `undefined`（无）。当然 JavaScript 并没有这么实现。尽管如此，大多数时候拥有 `undefined` 和 `null`，就已经足够了。

将以下各行代码分别输入到浏览器的地址栏，体会一下 `undefined` 和 `null` 的区别：

```
JavaScript:alert(typeof(x));
```

```
JavaScript:var x;alert(typeof(x));
```

```
JavaScript:var x=null;alert(typeof(x));
```

在例 1.2 代码中，我们用表达式 `undefined` 取代 `void(0)`，也能得到相同的结果。



实际上 `undefined` 远比想象得要有用得多，我们在后续章节里还会多次接触到 `undefined` 这个特殊的值。

至此，我们对例子代码的分析就告一段落。这段代码的经典之处不但在于它实现的效果令人惊叹，还在于它在短短的几行指令中体现了客户端 JavaScript 中大多数重要的特性，这些特性包括我们前面提到的伪协议、全局变量、文档接口、集合对象、函数、内置对象、元素样式属性、定时器以及 `void()` 和 `undefined`，除此以外还提到了代码中没有出现的闭包、函数实例化以及 `typeof` 操作符，这些特性几乎构成了客户端 JavaScript 的全部，在后面的章节中我们也将重点围绕着这些特性展开讨论，相信一段时间之后你再回头看这段代码，会有更加深刻的理解和新的收获。

## 1.4.2 为客户端服务

前面我们已经不止一次地提到过客户端浏览器的概念，那么一个典型的客户端应用究竟是怎样的？在这一节里，我们将概括地讨论 JavaScript 基于客户端的应用场景，简单介绍一下客户端应用的完整生命周期以及 JavaScript 程序在客户端生命周期过程中是如何作用的。这些知识有助于理解如何使得 JavaScript 更好地为客户端服务。



事实上，如果只是实现一个或者一组简单的特效和零散的增强交互，使用 JavaScript 并不需要了解客户端特质和完整的生命周期模型，然而如果你面对的是一个完全用 JavaScript 实现客户端交互的大型应用系统或者是一个 RIA 的网络娱乐系统，那么了解客户端的运作机制将是非常有帮助的。在这一节中我们接触的大部分概念在后续的章节中都会有更加详细的讨论，因此，先大致浏览过，等到时机成熟时再回过头来复习和理解，不失为一个非常好的学习方法。

在通常的 Web 应用中，Http 请求总是将页面文档以流的形式发送到客户端被浏览器所装载，不论后台应用的技术和服务部署的方式如何，客户端获得的总是以普通文本、`html`、`xhtml` 或者 `xml` 形式之一存在的数据。



极少数情况下，客户端也会获得二进制流或者其他格式的媒体流。我们在后续的章节里将会有相应的讨论。

我们通常定义的客户端生命周期起始于浏览器开始装载某个请求的特定数据，结束于浏览器发起一个新的请求（通常意味着页面的跳转或者刷新）。客户端的 JavaScript 则作用于这个完整的生命周期过程中。



很多开发人员不理解生命周期的含义，以至于经常有人犯一些尝试在 `jsp` 或者 `php` 页面解析的过程中执行客户端 JavaScript 的低级错误，这造成了前端和后端概念的混淆。

划分生命周期的意义除了避免概念混淆之外，还在于浏览器生命周期制约了大部分变量和作用域。通常情况下当一个浏览器生命周期结束时，绝大多数 JavaScript 变量和对象都会被销毁，资源得到释放。



通过采用特殊的处理方法，我们依然能够让部分对象跨越生命周期而存在，这对于我们实现一些特殊的功能是很有帮助的，当然其代价是容易造成内存泄露和其他一些潜在问题（在后续的章节里我们还有机会讨论这个话题）。

如果进一步细分，我们可以将客户端生命周期划分为从页面数据被装载到页面数据装载完毕的初始化阶段以及页面数据装载完毕一直到新的请求被发起之前的运行阶段。在前一个阶段里，JavaScript 代码被浏览器解析，运行环境被初始化，函数和闭包被建立，而那些可以被立即执行的指令被执行并实时地得到结果。在后一个阶段里，完成初始化的程序环境进入一个缺省的等待消息的循环，捕获用户操作引发的事件并作出正确响应，这种模式同经典的事件驱动模型非常接近。在这一阶段里，JavaScript 代码真正扮演一个界面交互行为处理者的角色。



很显然，被用作页面修饰的 JavaScript 代码通常在初始化阶段被执行完毕，而负责用户交互的 JavaScript 几乎总是要在运行阶段被触发和执行。区分这两者的作用和执行规律，有助于分解问题，优化我们的系统设计。

例 1.3 中的代码执行的效果与例 1-1 完全相同，区别是例 1-1 在生命周期的初始化阶段执行，而例 1-3 则是在运行期内执行。

例 1.3 经典程序 Hello World! 的另一种 JavaScript 实现


```
<html>
<html>
<head>
<title>Example 1.3 Hello World!</title>
<script type="text/JavaScript">
  <!--
    function PageLoad()
    {
      document.getElementsByTagName("h1")[0].innerHTML = "Hello World!";
    }
  -->
</script>
</head>
<body onload="PageLoad()">
  <h1>
    <noscript>您的浏览器不支持 JavaScript，请检查浏览器版本或者安全设置，谢谢！</noscript>
  </h1>
  <hr/>
  <p> document.getElementsByTagName 是我们接触到的 document 文档对象模型的第二个接口，它的作用通过它的名字很容易理解：它解析文档获取具有指定标记名称的一个列表，在这里 document.getElementsByTagName("h1")[0]得到文档中的第一个<h1> 标记。</p>
</body>
</html>
```

与例 1.1 比较，看起来略显繁琐的例 1.3 是一种更加安全的方式，在这种方式中，指令不会对装载期的文档内容产生影响，脚本指令被注册到 body 的 onload 事件中执行，这样确保了在执行前所有的文档元素都已经正确初始化完毕。



假如出现某种意外导致程序终止，例 1.1 可能因此而导致文档数据不能加载完全，而例 1.3 则不会有这样的风险。

例 1.3 中一个值得关注的特性是 `onload` 事件的注册: `<body onload="PageLoad()">`。这是到目前为止我们遇到的第一段事件注册代码, 它将函数 `PageLoad()` 注册到 `body` 的 `onload` 事件上, 在后续的章节里, 我们会了解到, 元素的 `onload` 事件将在元素被完全加载后由浏览器发起。除了 `onload` 之外, DOM 元素还有 `onclick`, `onkeydown`, `onchange`, `onblur` 等各种不同类型的事件, 这些事件共同构成了完整的客户端浏览器事件模型。在第 13 章中会就事件和事件模型展开详细讨论。

 一个比较好的习惯是把除声明之外的所有的脚本指令都放到运行阶段来执行, 这样避免了因为初始化期间的 DOM 元素加载失败或者低级的次序问题而导致脚本失效。

例 1.4 忽略了次序的失误

```
<html>
<head>
<title>Example 1.4 Hello World!</title>
</head>
<body>
<script type="text/JavaScript">
  <!--
    document.getElementsByTagName("h1")[0].innerText = "Hello World!";
  -->
</script>
<h1>
  <noscript>您的浏览器不支持 JavaScript, 请检查浏览器版本或者安全设置, 谢谢! </noscript>
</h1>
</body>
</html>
```

例 1.4 将产生一个脚本异常, 原因是当 `document.getElementsByTagName("h1")` 被执行时, 页面文档的 `h1` 标签还未被加载, 因此 `document.getElementsByTagName("h1")` 返回一个空值 `null`, 结果引发了异常。一个简单的修正方法是将 JavaScript 代码移至 `h1` 标签之后。当然, 而如果你事先将例 1.4 写成像例 1.3 那样的形式, 则根本不会遇到这个问题。

在本小节里, 我们讨论了浏览器客户端的生命周期, 对初始化和运行阶段进行了简单的划分, 并且对这两个阶段的特点进行了初步的探讨, 虽然相当一部分的 JavaScript 代码可以出现在上述两个阶段的任何一个当中, 但我的建议是将 JavaScript 代码尽可能多地放在运行阶段, 而不是尝试在装载阶段执行。在本书后续的章节里还会有更加深入的讨论, 第三部分和第四部分中一些稍微复杂和实用的例子也许能够帮助你更深刻的理解将 JavaScript 放在运行阶段执行的意义。而在这里, 只需要明确一个观点——优秀的 JavaScript 程序员总是善于利用浏览器特性让 JavaScript 代码更好地为客户端服务。

### 1.4.3 数据交互

除了页面修饰和完成交互行为之外, 执行数据交互也是 JavaScript 的一项强大功能。提供该功能的两个重要接口是 XML DOM 和 XML HTTP, 它们都可以被 JavaScript 很方便地操作。

► 关于 XML DOM 和 XML HTTP 的话题, 在本书的第 17 章中将会有详细的介绍。

这里所说的“数据交互”指的是客户端和服务器端不同系统之间的数据交换, 通常情况下, 数据流被以 HTTP 请求的形式发送到服务器端进行处理, 处理完毕的结果也被以流的形式发回客户端。


提交表单操作是浏览器提供了一种数据交互的默认方式, 传统的 Web 应用也是以提交表单为主要数据交互方式的。但是这种传统方式不能很方便地满足 Web2.0 下对用户体验的更高要求, 因此越来越多的应用系统采用 XML HTTP 作为主要的数据交互方式。XML HTTP 逐渐取代传统的提交表单, 标志着 Ajax 技

术的日趋成熟。


- 关于 Ajax 技术，在本书的第 18 章会有比较详细的讨论。

## 1.4.4 表面上的禁忌

前面我们提到了 JavaScript 的种种能力，在本小节里，我们将讨论 JavaScript 依赖于环境的一些“禁忌”，也即 JavaScript 不是万能的，它也有许多不能够做的事情。在章节的标题上用了“表面”一词，是因为这些“禁忌”中的相当一部分实际上不是绝对的，由于 JavaScript 具有出色的灵活性，以及相对可变的环境，对于资深程序员来讲，仍然不乏突破其中某些禁忌的手段。


 谈到禁忌和反禁忌就不得不触碰“安全性”这一高压线，所以在下一小节里，我们将进一步讨论 JavaScript 的安全问题。

通常情况下，客户端 JavaScript 只限于完成浏览器相关的任务。换句话说，客户端 JavaScript 的运行环境在相当大程度上是受到浏览器限制的，所以在这个环境中的 JavaScript 缺少一些独立的语言所必需的特性。

 这里所说的是客户端 JavaScript 受到浏览器的制约，并不意味着 JavaScript 本身不具备独立特性，事实上，减少浏览器禁忌或者离开浏览器，JavaScript 将变得更加强大。

由于客户端 JavaScript 受制于浏览器，而浏览器的安全环境和制约因素并不是绝对的，操作系统、用户权限、应用场合都会对其产生影响，因此，熟悉一些安全特性，就能够在一定程度上具备有突破 JavaScript 禁忌的手段。

- 另一种突破 JavaScript 禁忌的手段是利用 JavaScript 语言本身的灵活性，稍后将会进一步解释。

 理解 JavaScript 禁忌并不意味着鼓励程序员去突破它们，事实上，任何一种禁忌的存在都有它值得存在的理由，一个优秀的程序员总是让自己的代码尽可能地去遵守禁忌，而不是去打破它们，学会合理利用禁忌所带来的安全性，只有在必要的时候才去破解它们，是成为一个优秀程序员所必需掌握的技能。

除了能够动态生成浏览器要显示的 HTML 文档（包括图像、表格、框架、表单和元素样式等等）之外，JavaScript 并不具有任何图形图像处理能力。

客户端 JavaScript 虽然并不具备直接的图形图像处理 API，但是浏览器对图形图像处理提供了足够丰富的样式，而几乎所有的样式都能够被 JavaScript 随心所欲地控制。另外，简单的 2D、3D 绘图可以利用 JavaScript 动态生成 HTML 元素的特性，让 JavaScript 在浏览器上绘制点和曲线。如图 1.2 所示：

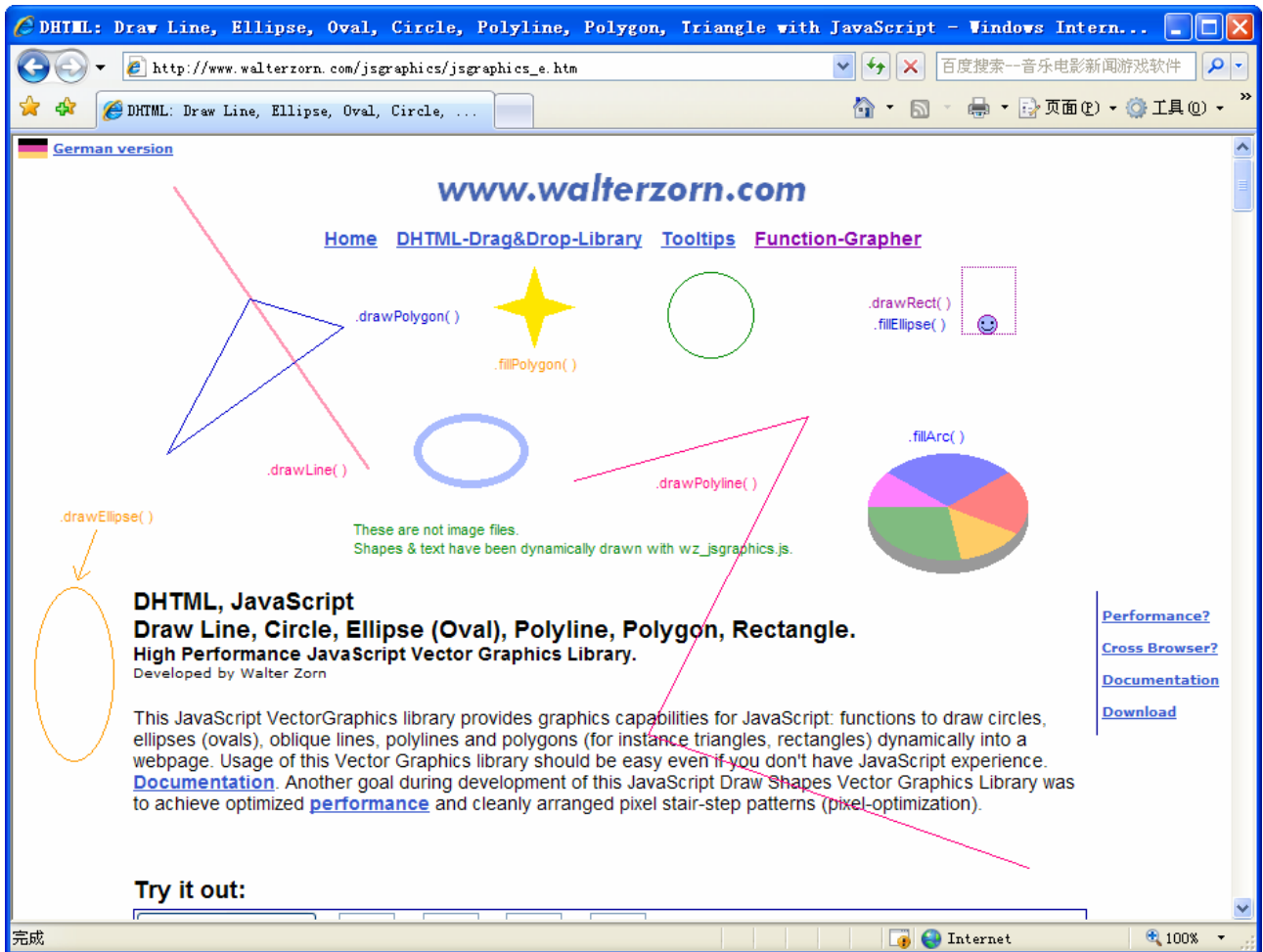


图 1.2 walterzorn 提供的 js 绘图程序，页面上的图形都是 JavaScript 绘制的



利用浏览器支持的元素样式 JavaScript 可以方便地缩放、旋转图片、着色、移动位置以及设定滤镜。

关于 JavaScript 控制元素样式的话题，在本书的第 14 章会有详细的讨论。

通过程序控制在页面上生成由 HTML 元素构成的点和直线，能够设计出一组完善的绘图函数，从而实现 JavaScript 绘图功能，在这方面，<http://www.walterzorn.com> 等组织提供了相当不错的开源组件。

要实现稍微复杂一些的 2D、3D 矢量图绘制功能，可以借助其它一些浏览器支持的第三方插件，比如 IE 支持的 VML，以及标准的 CSV 插件等。JavaScript 对 VML 和 CSV 的控制同操作标准的 HTML DOM 元素一样方便。

出于安全方面的考虑，客户端 JavaScript 一般不允许对文件进行读写操作。



显然，你一定不想让一个来自某个站点的不可靠程序在自己的计算机上运行，并且随意篡改你的文件。

实际上，依然有一些手段能够突破客户端 JavaScript 对文件进行读写操作的禁忌。在本地运行的 JavaScript 可以通过 windows 系统提供的一组被称作 FSO (File System Objects) 的 API 来操作本地文件，另外通过某些安装插件的方式可以在一些安全级别设定比较低的客户端上进行有限的文件读写。第三种比较安全的方式是通过浏览器对 XML 文本的支持把数据以 XML 文本的形式进行读写。

这三种突破方式各有利弊，其中第一种方式明确限制了应用必须在客户端执行，妄想在 Web 应用中直接调用客户端的 FSO 是不切实际的。第二种方式利用在客户端安装插件的方式来取消浏览器对客户端的保护，一般来说对于客户端而言这种方式是相当危险的，所以在安装插件前一定要通过恰当的询问和充分的理由得到用户的许可。




我想大多数用户对于国内一些知名或者非知名公司开发的插件以各种方式强制劫持浏览器的流氓行




为深恶痛绝，如果你是有良知和在乎颜面的程序员，一定不愿意让自己的程序被人称为流氓软件。而且，事实上，大多数用户不愿意在自己的浏览器中装各种各样的插件。所以不到万不得已，**一定不要**用这种方式来增强浏览器的文件处理能力。

最后一种稍微安全一些的方式是利用 XML，事实上 XML 是 JavaScript 唯一可以安全操作的一种文件格式，你可以让程序通过询问的方式将 XML 格式的文档保存到客户端或者从客户端读取文档进行处理。

 XML 文档是一种描述能力相当强的文件形式，利用它很容易以一种既易于被计算机处理又容易被理解的格式来组织数据，你可以把需要的任何格式的数据组织成 XML 文件来方便地存取。在本书的后续章节里，还会陆续地讨论到有关 XML 的内容，特别是第 17 章对 XML 格式会有一个相对比较完整的介绍。尽管如此，XML 并不是本书关注的技术，如果你需要更为深入地了解 XML 技术，可以参考其它相关的文档和技术教程。

除了能够引发浏览器下载任意 URL 所指的文档以及把 HTTP 请求、邮件和 FTP 请求发送给服务器端之外 JavaScript 不支持任何形式的联网技术。

客户端 JavaScript 可控制的数据传输只限于应用层，它不支持 TCP/IP、UDP 等传输层协议和 Socket 接口，这显然是因为它仅能利用浏览器实现数据传输而本身不具有发送和接收数据的功能。


 HTTP 请求可以看作是一种应用层上的数据传输协议，与之处于同一层次上的传输协议还有 FTP、邮件协议和一些流媒体传输协议等。在这个层次上，一般来说数据以流的形式被发送和接收，一个封装好的头部（对于 HTTP 请求来说是 Http Header）反映了数据流的基本信息，你不必关注数据包的划分、链路的选择和其它一些底层的東西。数据流通常有文本流和二进制流两种形式，HTTP 请求的数据一般是一种文本流，如果要利用客户端 JavaScript 传输二进制数据，一个比较可靠的方式是预先用程序将它们进行编码（通常采用 Base64 或者其它一些标准编码）。关于 HTTP 请求的话题，在本书的第四和第五部分会有进一步的探讨。

这个禁忌从一定程度上限制了客户端数据在毫无征兆的情况下被发送，从而尽力避免用户隐私的外泄。遗憾的是，这种限制并不彻底，实际上，由于利用浏览器传输数据本身并不安全，所以客户端 JavaScript 的数据交互安全性无法在浏览器层面上得到保证。

## 1.7 学习和使用 JavaScript 的几点建议

在本节里，我们正式开始接触“道”的本质。

通读本书完成学业之前，迫切需要做的一件事情是，冷静下来思考如何学习和使用 JavaScript。真正掌握有效的学习方法，是提高学习效率和改进学习效果的重要途径。学习效率和学习效果则直接影响到对 JavaScript 本质的掌握程度，从而决定 JavaScript 在你手中能够发挥出来的威力。

 就像武侠小说中描述的那样，决定一个人武功强弱，不仅看招式，更为重要的是看内功。即使是普普通通的招式，在内功练到炉火纯青的高手手中，也会发挥出极大的威力。JavaScript 的学习也是这样。希望本书不仅仅教会你众多的 JavaScript “招式”，也能成为帮助你修炼程序设计“内功”的秘籍。


### 1.7.1 像程序员一样地思考

随着软件技术不断发展，从事软件行业的人员日渐增多。你发现身边多了这么一群人，他们有的西装革履，有的穿着随意，有的不苟言笑，有的风趣幽默，有的博学多才，有的质朴木讷，唯一的共同点是，他们的名片上都印着“程序员”这样的字。

现在大街上所谓的“程序员”是如此之多，他们中有真正的高手，也有只会写几行蹩脚代码的滥竽充


数者。在这里我无意贬低程序员同僚，只是想通过我的经验说明什么样的人才是真正的程序员。

### 程序员是怎样炼成的？

 一些人认为，掌握一门计算机语言，会编写几行代码并且能够让这些代码在计算机上运行起来的人，就可以称为程序员。事实上，软件行业里，要成为真正意义上的程序员，对得起 programmer 这个称号，还是要花费一番功夫的。

在成为程序员的道路上，要经历四个坎坷，让我们用四个境界来标明他们。

第一境界，就是前面所说的，掌握一门或者几门编程语言，会模仿例子来实现程序代码，并且让代码在计算机系统中运行起来。达到这个境界人，还不能算是真正意义上的程序员，而仅仅是掌握了一种或者几种工具的工匠，他们中的熟练者能够快速模仿现成的例子，以实现自己或者用户需要的软件模块。


 非常遗憾，许多“程序员”仅仅达到第一个境界，他们根据手中的文档和参考资料，通过“模仿”来完成工作，他们实现的程序只是无数个前人已经实现过的代码的翻版组合，虽然其中的熟练者以快速高效率完成任务著称，然而他们的作品中毫无新意，日复一日地重复代码，罕有任何可以称之为“创新”的东西。

第一境界的特质是对语言工具的掌握，在这个境界的高手，会强调自己对语言如何如何熟练，因此这个境界可以用“**知器**”来表示。


第二境界里，我们要学习的是分解问题和推理的技巧，学会用逻辑的语言来精确地表达一个命题。在这个境界里，软件工作者掌握的是一种分析具体事物的方法，他们不再一味地模仿，而是开始对一个又一个具体问题思考并尝试用自己的方法来更好地解决。

在这个层次里的“程序员”开始关注解决问题的思路，并且关注分析和推理的数学技巧，他们中的优秀者熟知各种算法善用各种各样的命题推理来分析并解决问题。他们同样善于借鉴前人的例子，但是往往能够根据问题的特点进行有效的改进，并且能够在尝试改进的过程中得到创新的成就感和新的经验。在这个层次里的人，对语言工具的认识比第一境界更加深刻，他们是真正知道如何利用手中语言工具的特点更好地解决问题的人。但是他们并不会强调自己对于语言如何熟悉，也不再热衷于宣扬掌握如何如何多的语言，在他们眼里，语言仅仅是一种工具而已，真正重要的是分析问题的方法。

第二境界的特质是对具体问题的分析，在这个境界的高手，往往善于从具体问题中分析出合理有效的解决方法。因此这个境界用“**格物**”来表示。

 第二境界里有真正对如何用程序来解决问题经验丰富的人，这些人能够出色地胜任编码工作，因此我们称他们为 Coder，或者初级程序员。

第三境界里，我们要学习的是抽象思维和找出事物表象后面的规律。在这个境界里，软件工作者不再针对一件一件具体的事物来分析，而是尝试理解事物表象下的本质。在这个层次里的人，开始关注事物的共性，并且逐渐掌握归纳和总结的方法。“模式”开始出现在他们的头脑里。

 “设计模式”是软件领域的“三十六计”，是经过抽象总结而归纳出来的真正的思想精华。第三个境界的软件工作者开始接触并且理解“模式”，学会灵活运用模式和抽象思维来解决“某一类”问题。与表象相比，他们更关注的事物的本质，他们的代码里充满思想和对事物规律的深刻认识，他们熟知各种类型问题的特点和解决技巧。对事物本质规律的认识使他们不再依赖于语言工具，任何一种熟悉或者陌生的程序设计语言在他们的手中都能够发挥到极致，完美地解决问题。

第三境界的特质是对事物本质规律的认识，在这个境界的高手，往往能够快速地抽象出问题的本质，从而用最合适的方法来解决问题。这个境界我用“**明理**”来表示。

达到第三境界的程序员，是天生的设计师，他们对问题本质的领悟能力帮助他们用优美简洁代码来解决问题，他们的代码中充满设计思想。他们是真正能够享受到程序设计的艺术魅力并且充满成就感的一群人。



第三境界里的程序员真正当得起 Programmer 称号，他们在外人眼里看起来是天生的设计师、艺术家和技术牛人，他们是为软件创作而生的。

前面说到了三个境界，层层深入，并且第三个境界的程序员已经当之无愧地成为软件领域的专家领袖，然而，“程序员”是否只有以上三个境界了呢？答案是否定的。在这三个境界之上，依然存在有——

#### 第四个境界

第三境界程序员中的极优秀者，并不满足于专家的地位和高薪，他们开始向第四境界艰难地前进。第四境界是程序设计领域的最高境界，要达到这个境界，只需要掌握一样东西，然而这个东西并不是寻常之物，而是许多人穷尽一生也无法得到的，这个世界最为深邃的秘密。



自古以来，有这样一群僧人，他们遵守戒律，不吃肉，不喝酒，整日诵经念佛，而与其他和尚不同的是，他们往往几十年坐着不动，甚至有的鞭打折磨自己的身体，痛苦不堪却依然故我。

有这样一群习武者，经过多年磨练，武艺已十分高强，但他们却更为努力地练习，坚持不辍。

有这样一群读书人，他们有的已经学富五车，甚至功成名就，却依然日夜苦读，不论寒暑。

他们并不是精神错乱，平白无故给自己找麻烦的白痴，如此苦心苦行，只是为了寻找一样东西。

传说这个世界上存在着一种神奇的东西，它无影无形，却又无处不在，轻若无物，却又重如泰山，如果能够获知这样东西，就能够了解这个世界上的所有的奥秘，看透所有伪装，通晓所有知识，天下万物皆可归于掌握！

这并不是传说，而是客观存在的事实。

引自《明朝的那些事儿》 作者：当年明月

这样的东西，叫做“道”。



静寂虚无中有奥秘，不静不动，乃程序之源，无以名之，故曰：程序设计之道。若道至大，则操作系统至大；若操作系统至大，编译程序亦然；若编译程序至大，应用程序亦复如是。是以用者大悦，世之和諧存焉。

——杰弗瑞·詹姆士

所谓道，是天下所有规律的总和，是最根本的法则，只要能够了解道，就可以明了世间所有的一切。掌握了“道”的程序员，才是真正的程序设计大师，能够创作出流芳百世的作品。

然而怎样才能“悟道”，我并不知道，也无法描述，因为“道”实在不是一个能够轻易得到和理解的东西。

对第四境界的程序员来说，“思想”已经不再是很重要的东西，因为他们对程序本质的理解已经超越了问题本身，在他们的代码里，有的只是自然，现实和虚幻的边界都已经模糊，一个完美自治的系统在刹那间诞生，却仿佛从亘古时刻起便存在着、运动着，从简单而质朴的规律中涵盖着世间万物的本质。因此，这个境界，我称之为“成道”。

学习 JavaScript 不应该游离于程序员之外，JavaScript 程序员也是真正的程序员，因此摆在我们面前的道路也是从“知器”、“格物”、“明理”到“成道”的艰难过程，像程序员一样地思考，扎扎实实地向着更高的层次迈进，才是正确的学习方法。只要坚持不懈，迟早有一天，JavaScript 会在你的手中大放异彩。

## 1.7.2 吝惜你的代码

代码对于程序员来说就像剑客手中的剑。对于高手来说，剑的长短不是决定因素，剑招的犀利才是胜负的关键。相对来说越短的剑，破绽反而越少。

要知道，你写的每段代码，在将来都有可能需要花费精力去维护，代码越多，将来需要维护的工作量就会越大。



程序大师如是说：“虽然程序只有三行，但总有需要维护的一天。”

聪明的程序员总是用简洁的代码来证明自己的才华，通常情况下，优秀的代码总是比较短的那一段。

吝惜代码的另一个含义是“不要轻易动手编写代码”。真正优秀的程序员永远在深思熟虑之后才动手写代码，因为他们知道，要在实际动手之前避开可能的陷阱，尽量让自己的代码不要有破绽。



“入界宜缓”，不仅仅是正确的棋理，也是程序设计中的真理，甚至是充满人生智慧的格言，它是“道”的一部分。

程序大师深思一天，只写三行代码，而一年积累下来的千行代码却成为整个软件时代的灵魂。一个百万行代码的大型程序将因为它而成为不朽的经典。

本书中的例子秉承这样的原则——用最少的代码做最多的事情。相信在阅读本书后续章节的过程中你将渐渐理解和领悟其中的奥妙。

### 1.7.3 学会在环境中调试

虽然大师可以不依赖于环境让思维自由飞翔，可是在“求道”的路上，调试环境却能为你扫除许多障碍。

除非你确信自己可以一遍写出正确无误的代码来，否则你就需要一个强大调试工作的帮助。



我的一些同事曾经因为缺乏对运行环境和调试工具的认识，而在一些小障碍面前显得束手无策，这对于项目组来说实在不是一个好消息。在软件领域，你可以犯错误，但是你不能在错误面前不知所措，或者选择逃避。如果你自身的能力有限，请用好的调试工具来武装自己，它们的的确确可以迅速地帮助你定位问题的所在。

➤ 在本书的第3章里，将详细地讨论 JavaScript 调试技巧与调试工具。

### 1.7.4 警惕小缺陷

许多人知道 JavaScript 能做什么，却对它的缺点视而不见。相信通过前面章节的叙述，已经给大家暗示了一个道理，那就是——JavaScript 不是万能的，它有很多缺陷，一定要谨慎地使用它。

用 JavaScript 进行浮点数计算很容易造成精度问题；用 JavaScript 操作 DOM，内存泄露永远存在；各个版本的浏览器下，总有一些 JavaScript 代码行为诡异；而某些场合下 JavaScript 性能慢到无法忍受……即使抛却这些缺陷，JavaScript 的大量使用如果不够谨慎，失控的代码依然很容易使你泥足深陷无法自拔。



我和我的一些同事经历过那样刻骨铭心的痛，系统的问题诡异得无法捉摸，大量随机出现的问题像张牙舞爪的魔鬼，恶作剧般的手法折腾得你筋疲力尽，你却依然对摆在面前一团乱麻似的脚本无可奈何。最终你或许赌气地说，我这辈子再也不用 JavaScript 了，可是客户还等着你实现他们的需求，于是你摇摇头强打精神去重新一遍一遍地梳理那些凌乱的代码。

小心你手中的魔鬼，学会正确地控制它们，切记不要放任任何一个哪怕是无伤大雅的小缺陷在你一段代码中，否则你总有一天将会受到惩罚。

### 1.7.5 思考先于实践

前面已经说过，在经过充分思考之前，不要轻易动手写代码。不管是学习还是工作，一味模仿他人的

作品或者书中的例子，都算不上是一个好习惯。

即使本书是面向实战的，要透彻理解 JavaScript 的本质，还是要建立在深入理解和分析的基础上。在每一个章节里，概览范例，重视细节，深入领悟精髓，才会得到最大的收获。

真正掌握 JavaScript 的标志是要能够写出属于自己的代码，而这，显然是要先思考而后动手才能做到的。



急于动手，是许多投身程序设计领域的爱好者们的一个通病，也往往是职业和业余的分别。也许一个效果的实现能够带给你惊喜和成就感，然而你必须知道，你在这里，绝对不是为了仅仅实现这一个小小的效果，你必须确保你的代码现在不会出问题，将来也不会出问题，在你手里不会出问题，在别人手里也不会出问题。而这一切，都要求你经过充分的思考。

软件过程包括分析、设计、实现和验证等若干个阶段，动手去写，仅仅是其中很小的一部分工作，而为了保证这部分工作的出色品质，往往要在之前投入大量精力去思考。就像要创作一件优秀的艺术品，必须要经过严谨的构思一样。



在用你的键盘敲出每一行代码前，请三思。优秀的代码是思想的结晶，蹩脚的代码才是呆板的模仿和毫无章法的拼凑。

## 1.7.6 时刻回头

即使是伟大的贤哲，也无法完全预知未来。

就算你确信目前的代码无懈可击，你也不可能保证它们在将来永远能够不加改变地正常工作。在软件实现的过程中，要习惯时刻回头完善你之前所创作的代码。如果你觉得一个接口将来有可能变化，你就去完善它，不要吝惜走回头路，这一小段回头路将令你避免将来走入一条歧路。



许多开发人员忽视维护系统原有结构的重要性，他们在实现新功能时，宁愿自己编写一个模块而不愿意去了解和完善原有的模块，这样做的结果使得每个开发人员的成果彼此孤立，而且系统出现大量功能相近的冗余模块，严重地影响了系统的完整性和可重用性。

不愿意回头看的人永远也不会真正掌握未来。

任何一个努力的过程总是循序渐进的，发现过去的完善是好事情，这意味着进步。这个时候，学会回过头去，稍微回顾和改变一下过去的成果，你会有许多新的收获。

在学习本书的任何一个阶段，请学会回头，当你尝试着往前翻时，往往便是你从书的字里行间得到更多收获的时候。

## 1.8 关于本书的其余部分

本书的章节共分为五个部分，

本章和第 2、3 章共同构成了概论部分，在这个部分，主要介绍 JavaScript 的特点、学习方法、编写和调试环境以及一些有趣的例子。绪论部分独立地给读者一个完整的 JavaScript 概貌，并且尝试着达到一定的深度，揭示今后的学习过程充满挑战性。

第 4~10 章是第二部分，这一部分系统性地讲述 JavaScript 语言的核心，包括基础的词法和语法、程序结构、数据类型和其他语言基本特征。这些知识虽然描述起来比较乏味，但却是初学者接触一门编程语言所必须要掌握的内容。

第 11~15 章构成了本书的第三部分，这一部分的各章介绍了客户端 JavaScript 的核心——浏览器的各个对象，并且还提供了有关这些对象的用法的示例程序。可以说，第三部分是 JavaScript 客户端应用的基

础，是相当有趣和有用的内容。

第四部分由第 16~20 章构成，讨论的是 JavaScript 的数据交互和信息安全问题，在这一章里，我们将真正接触到目前 Web 应用领域的热门技术——Ajax。学习完第四部分，才可以说真正掌握了使用 JavaScript 构建 Web 应用系统的能力。

第 21~26 章是本书的最后一个部分，在这里我们探讨一些稍微高级的话题，这一部分的内容最贴近于 JavaScript 的特征，它们是开启 JavaScript 最后封印的钥匙。可以这么说，要真正让 JavaScript 在你手中发挥出化腐朽为神奇般的亘古之力，深入学习和深刻理解这一部分是必须的。不要让 JavaScript 成为一种平庸的语言，依靠你的努力，让 JavaScript 在你的手中放出光彩！

最后，祝你学习 JavaScript 愉快，使用 JavaScript 时一帆风顺，度过一个个有趣的探秘之旅。

## 第二十一章 面向对象

如果你曾经是一名 C++ 程序员或者 Java 程序员，面向对象这个概念对于你来说应该并不陌生。即使你刚刚接触程序设计，本书通过前面的安排，也已经为你理解面向对象做了充分的准备。在第二部分，你已经了解了什么是对象，并且通过一些例子对面向对象有了感性的认识。而在这一章里，我会试图向你阐述面向对象的思想本质及隐藏在各种表象下的深层规律。需要注意的是，面向对象只是过程化程序设计方法的一个层次，它是目前我们所知的一种比较高级的过程化境界（但不是最高的），面向对象的代码有较好的组织结构和重用性，从而适用于比较大型的应用程序开发中。面向对象是一种思想而不是一种固定的套路，请牢记这一点以免自己陷入不必要的思维定势中去。

### 21.1 什么是面向对象

早在第 7 章，我们就系统地讨论了 JavaScript 的对象，并且在多处提到过“面向对象”这个概念。然而，除了第一章中一段简单的解释之外（回顾 1.1.3 节），我们并没有直接将“面向对象”和 JavaScript 放到一起，或者说，我们并没有明确地认为，JavaScript 是一种**面向对象**的编程语言。

JavaScript 是否面向对象，是一个有争议的话题，本书也不能妄下定论。有人说，JavaScript 是一种**基于对象**的语言，这种说法基本上是正确的，但是，另一些人坚持 JavaScript 是面向对象的，而这个看法，在后面我们会分析，应该说是更加准确的。不过需要注意，“面向对象”和“基于对象”是两个不同层次的概念。



面向对象的三大特点（封装，延展，多态）缺一不可。在后面的章节里我们会分别谈到它们。通常“基于对象”是使用对象，但是不一定支持利用现有的对象模板产生新的对象类型，继而产生新的对象，也就是说“基于对象”不要求拥有继承的特点。而“多态”表示为父类类型的子类对象实例，没有了继承的概念也就无从谈论“多态”。现在的很多流行技术都是基于对象的（例如 DOM），它们使用一些封装好的对象，调用对象的方法，设置对象的属性。但是它们无法让程序员派生新对象类型。他们只能使用现有对象的方法和属性。所以当你判断一个新的技术是否是面向对象的时候，通常可以使用后两个特性来加以判断。“面向对象”和“基于对象”都实现了“封装”的概念，但是面向对象实现了“继承和多态”，而“基于对象”可以不实现这些。

通常情况下，面向对象的语言一定是基于对象的，而反之则不成立。

从本质上说，面向对象既是一种思想，也是一种技术，它是过程式程序设计方法的一个高级层次。面向对象思想利用对问题的高度抽象来提升代码的可重用性，从而提高生产力。尤其是在较为复杂的规模较大的系统实现中，面向对象通常比传统的过程式方法产生更高的效能。而且，随着软件规模的增大，面向对象相对于传统的过程式的优势就更加凸现。可以说，是软件产业化最终促进了面向对象技术的产生和发

展。

下面，我们将介绍与 JavaScript 相关的面向对象技术，而关于面向对象思想本身更加深入的内容，可以参考相关的面向对象分析、设计和开发教程。

## 21.1.1 类和对象

我们对 JavaScript 对象已经并不陌生，下面的例子展示了三种构造对象的方法：

例： 21.1 对象的三种基本构造法

//第一种构造法： new Object

```
var a = new Object();
```

```
a.x = 1, a.y = 2;
```

//第二种构造法：对象直接量

```
var b = {x : 1, y : 2};
```

//第三种构造法：定义类型

```
function Point(x, y)
```

```
{
```

```
    this.x = x;
```

```
    this.y = y;
```

```
}
```

```
var p = new Point(1,2);
```

其中，第一种方式是通过实例化一个 Object 来生成对象，第二种方式是通过对象常量，而第三种方式比较特殊，我们先构造了一个 function，这个 function 代表了一“类”特殊的对象，这类对象描述二维平面上的点，new Point(1,2)表示二维平面上坐标为（1，2）的点，要得到二维平面上坐标为（3，4）的点则可以用 new Point(3,4)。

现在我们比较一下这三种方法的差别，第一种方法是通过构造基本对象直接添加属性的方法来实现的。我们说 JavaScript 是一种弱类型的语言，一方面体现在 JavaScript 的变量、参数和返回值可以是任意类型，另一方面也体现在，JavaScript 可以对对象任意添加属性和方法，这样无形中就淡化了“类型”的概念。例如：

```
var a1 = new Object();
```

```
var a2 = new Object();
```

```
a1.x = 1, a1.y = 2;
```

```
a2.x = 3, a2.y = 4, a2.z = 5;
```

你既没有办法说明 a1、a2 是同一种类型，也没有办法说明它们是不同的类型，而在 C++和 Java 中，变量的类型是很明确的，在声明时就已经确定了它们的类型和存储空间。

第二种方法和第一种方法大同小异，实际上你可以将它看成是第一种方法的一种快捷表示法。

比较有趣的是第三种方法：

```
function Point(x,y)
```

```
{
```

```
    this.x = x;
```

```
    this.y = y;
```

```
}
```

```
var p1 = new Point(1,2);
```

```
var p2 = new Point(3,4);
```

你现在知道了 p1 和 p2 是同一种类型，它们都是 Point 的实例。而对于 p1 和 p2 来说，Point 是它们的

“类”，p1、p2 和 Point 之间的关系是创建与被创建的关系，这种关系是面向对象中最重要的一种关系，它是“泛化”关系的一个特例。



通常在讨论面向对象时，构造对象时采用的是上面第三种方法，因为“创建”是面向对象中不可缺少的一种“泛化”关系。

## 21.1.2 公有和私有：属性的封装

前面已经说过，封装性是面向对象的一个重要特性。所谓的封装，指得是属性或方法可以被声明为公有或者私有，只有公有的属性或方法才可以被外部环境感知和访问。曾经有人说 JavaScript 不具备封装性，它的对象的属性和方法都是公有的，其实，持这个观点的人只看到了 JavaScript 函数的对象特征，而忽视了 JavaScript 函数的另一个特征——闭包。



在这里要再一次强调，JavaScript 中，函数是绝对的“第一型”，JavaScript 的对象和闭包都是通过函数实现的。关于闭包的内容，在稍后的第 22 章会有深入的讨论。

利用闭包的概念，JavaScript 中不但有公有和私有的特性，而且它的公有和私有性，比起其它各种面向对象语言毫不逊色。下面给出一个例子：

例 21.2 对象的公有和私有特性

```
function List()
{
    var m_elements = []; //私有成员，在对象外无法访问

    m_elements = Array.apply(m_elements, arguments);

    //公有属性，可以通过“.”运算符或下标来访问
    this.length = {
        valueOf:function(){
            return m_elements.length;
        },
        toString:function(){
            return m_elements.length;
        }
    }

    this.toString = function()
    {
        return m_elements.toString();
    }

    this.add = function()
    {
        m_elements.push.apply(m_elements, arguments);
    }
}
```

function List 定义了一个 List 类，该类接受一个参数列表，该列表中的成员为 List 的成员。m\_elements



是一个私有成员，在类的定义域外部是无法访问的。`this.length`、`this.toString` 和 `this.add` 是公有成员，其中 `this.length` 是私有成员 `m_elements` 的 `length` 属性的 `getter`，在外部我们可以通过对象名的“.”运算符对这些属性进行访问，例如：

```
var alist = new List(1,2,3);
alert(alist);
alert(alist.length);
alist.push(4,5,6);
alert(alist);
alert(alist.length);
```



小技巧：对象的 `getter` 是一种特殊的属性，它形式上像是变量或者对象属性，但是它的值随着对象的某些参数改变而变化。在不支持 `getter` 的语言中，我们通常用 `get<Name>` 方法来代替 `getter`，其中 `<Name>` 是 `getter` 的实际名字，这种用法产生的效果和 `getter` 等价，但是形式上不够简洁。ECMAScript v3 不支持 `getter`，但是可以用上面这种构造带有自定义 `valueOf` 和 `toString` 方法的对象来巧妙地模拟 `getter`。

例如，下面的两段代码基本上等价：

```
//第一段代码：使用 getName()方式
function Foo(a, b)
{
    this.a = a;
    this.b = b;
    this.getSum = function()
    {
        return a+b;
    }
}
alert((new Foo(1,2)).getSum()); //得到 3
```

```
//第二段代码：模拟 getter
function Foo(a, b)
{
    this.a = a;
    this.b = b;
    this.sum = {
        valueOf:function(){return a+b},
        toString:function(){return a+b}
    }
}
alert((new Foo(1,2)).sum); //同样得到 3
```

对象的 `setter` 是另一个相对应的属性，它的作用是通过类似赋值的方式改变对象的某些参数或者状态，遗憾的是，ECMAScript v3 不支持 `setter`，并且目前为止也没有什么好的办法可以在 JavaScript 上模拟 `setter`。要实现 `setter` 的效果，只有通过定义 `set<Name>` 方法来实现。

### 21.1.3 属性和方法的类型

JavaScript 里，对象的属性和方法支持 4 种不同的类型，第一种类型就是前面所说的私有类型，它的特点是对外界完全不具备访问性，要访问它们，只有通过特定的 `getter` 和 `setter`。第二种类型是动态的公有

类型，它的特点是外界可以访问，而且每个对象实例持有一个副本，它们之间不会相互影响。第三种类型是静态的公有类型，或者通常叫做原型属性，它的特点是每个对象实例共享唯一副本，对它的改写会相互影响。第四种类型是类属性，它的特点是作为类型的属性而不是对象实例的属性，在没有构造对象时也能够访问，下面通过例子说明这四种属性类型各自的特点和区别：

### 例 21.3 类型的四种属性

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 21.3</title>
</head>
<body>
<h1 id="output"></h1>
<script language="javascript" type="text/javascript">
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function myClass()
    {
        var p = 100; //private property; 私有属性
        this.x = 10; //dynamic public property 动态公有属性
    }
    myClass.prototype.y = 20; //static public property or prototype property 原型属性
    myClass.z = 30; //static property //静态属性

    var a = new myClass();
    dwn(a.p); //undefined 私有属性对象无法访问到
    dwn(a.x); //10 公有属性
    dwn(a.y); //20 公有属性
    a.x = 20;
    a.z = 40;
    dwn(a.x); //20
    dwn(a.y); //40 //动态公有属性 y 覆盖了原型属性 y
    delete(a.x);
    delete(a.y);
    dwn(a.x); //undefined 动态公有属性 x 被删除后不存在
    dwn(a.y); //20 动态公有属性 y 被删除后还原为原型属性 y
    dwn(a.z); //undefined 类属性无法通过对象访问
    dwn(myClass.z); //30 类属性应该通过类访问
-->
</script>
</body>
</html>
```

执行结果如图 21.1 所示：

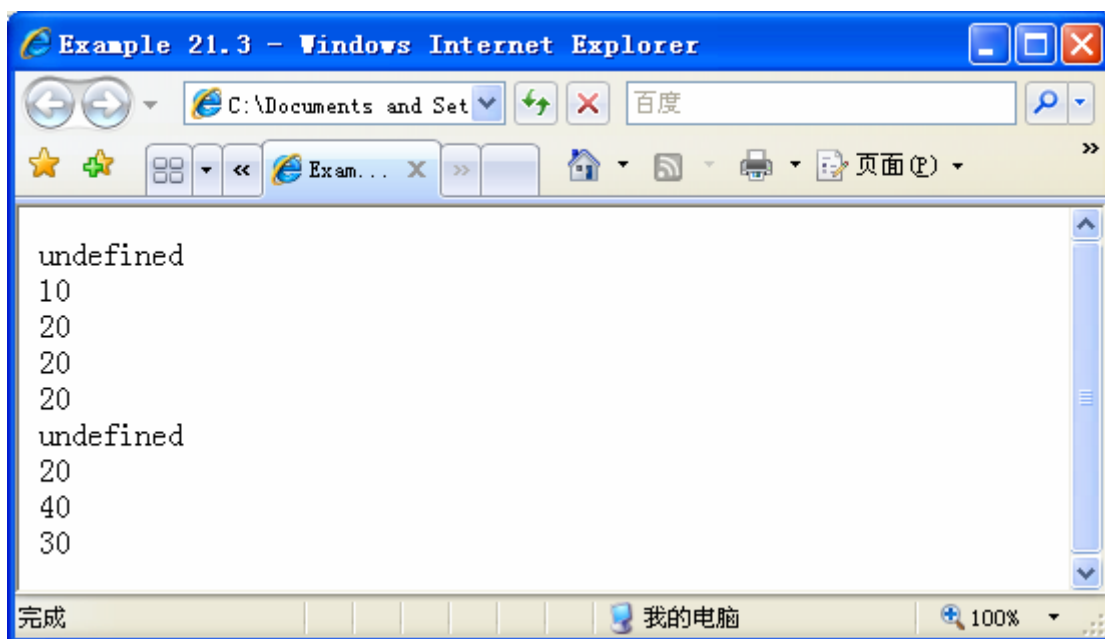


图 21.1 类型的四种属性

## 21.2 神奇的 prototype

对于初学 JavaScript 的人来说 prototype 是一种很神奇的特性，而事实上，prototype 对于 JavaScript 的意义重大，prototype 不仅仅是一种管理对象继承的机制，更是一种出色的设计思想。

### 21.2.1 什么是 prototype

JavaScript 中对象的 prototype 属性，可以返回对象类型原型的引用。这是一个相当拗口的解释，要理解它，先要正确理解对象类型（Type）以及原型（prototype）的概念。

前面我们说，对象的类（Class）和对象实例（Instance）之间是一种“创建”关系，因此我们把“类”看作是对象特征的模型化，而对象看作是类特征的具体化，或者说，类（Class）是对象的一个类型（Type）。例如，在前面的例子中，p1 和 p2 的类型都是 Point，在 JavaScript 中，通过 instanceof 运算符可以验证这一点：

```
p1 instanceof Point
```

```
p2 instanceof Point
```

但是，Point 不是 p1 和 p2 的唯一类型，因为 p1 和 p2 都是对象，所以 Object 也是它们的类型，因为 Object 是比 Point 更加泛化的类，所以我们说，Object 和 Point 之间有一种衍生关系，在后面我们会知道，这种关系被叫做“继承”，它也是对象之间泛化关系的一个特例，是面向对象中不可缺少的一种基本关系。

在面向对象领域里，实例与类型不是唯一的一对可描述的抽象关系，在 JavaScript 中，另外一种重要的抽象关系是类型（Type）与原型（prototype）。这种关系是一种更高层次的抽象关系，它恰好和类型与实例的抽象关系构成了一个三层的链，图 21.2 描述了这种关系：

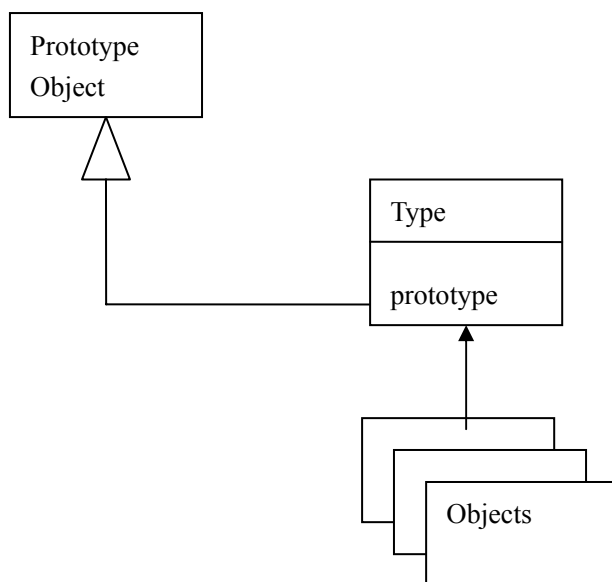


图 21.2 对象、类型与原型的关系

在现实生活中，我们常常说，某个东西是以另一个东西为原型创作的。这两个东西可以是同一个类型，也可以是不同类型。习语“照猫画虎”，这里的猫就是原型，而虎就是类型，用 JavaScript 的 prototype 来表示就是“虎.prototype=某只猫”或者“虎.prototype=new 猫()”。

💡 “原型”是描述自然界事物之间“归类”关系的一种，另外几种关系包括“继承”和“接口”。一般来说，“继承”描述的是事物之间固有的衍生关系，能被“继承”所描述的事物之间具有很强的关联性（血缘）。“接口”描述的是事物功用方面的共同特征。而“原型”则倾向于描述事物之间的“相似性”。从这一点来看，“原型”在描述事物关联性的方面，比继承和接口更加广义。

如果你是 Java 程序员，上面的例子从继承的角度来考虑，当然不可能用“猫”去继承“虎”，也不可能用“虎”去继承“猫”，要描述它们的关系，需要建立一个涵盖了它们共性的“抽象类”，或者你会叫它“猫科动物”。可是，如果我的系统中只需要用到“猫”和“老虎”，那么这个多余的“猫科动物”对于我来说没有任何意义，我只需要表达的是，“老虎”有点像“猫”，仅此而已。在这里，用原型帮我们成功地节省了一个没有必要建立的类型“猫科动物”。

要深入理解原型，可以研究关于它的一种设计模式——prototype pattern，这种模式的核心是用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。JavaScript 的 prototype 就类似于这种方式。

关于 prototype pattern 的详细内容可以参考《设计模式》（《Design Patterns》）它不是本书讨论的范围。

注意，原型模式要求一个类型在一个时刻只能有一个原型（而一个实例在一个时刻显然可以有多个类型）。对于 JavaScript 来说，这个限制有两层含义，第一是每个具体的 JavaScript 类型有且仅有一个原型（prototype），在默认的情况下，该原型是一个 Object 对象（注意不是 Object 类型!）。第二是，这个类型的实例的所有类型，必须是满足原型关系的类型链。例如 p1 所属的类型是 Point 和 Object，而一个 Object 对象是 Point 的原型。假如有一个对象，它所属的类型分别为 ClassA、ClassB、ClassC 和 Object，那么必须满足这四个类构成某种完整的原型链，例如：

例 21.4 原型关系的类型链

```
function ClassA()
{
    .....
}
ClassA.prototype = new Object(); //这个可以省略
function ClassB()
```

```

{
    .....
}
ClassB.prototype = new ClassA(); //ClassB 以 ClassA 的对象为原型
function ClassC()
{
    .....
}
ClassC.prototype = new ClassB(); //ClassC 以 ClassB 的对象为原型

```

```

var obj = new ClassC();
alert(obj instanceof ClassC); //true
alert(obj instanceof ClassB); //true
alert(obj instanceof ClassA); //true
alert(obj instanceof Object); //true

```

图 21.3 简单描述了它们之间的关系：

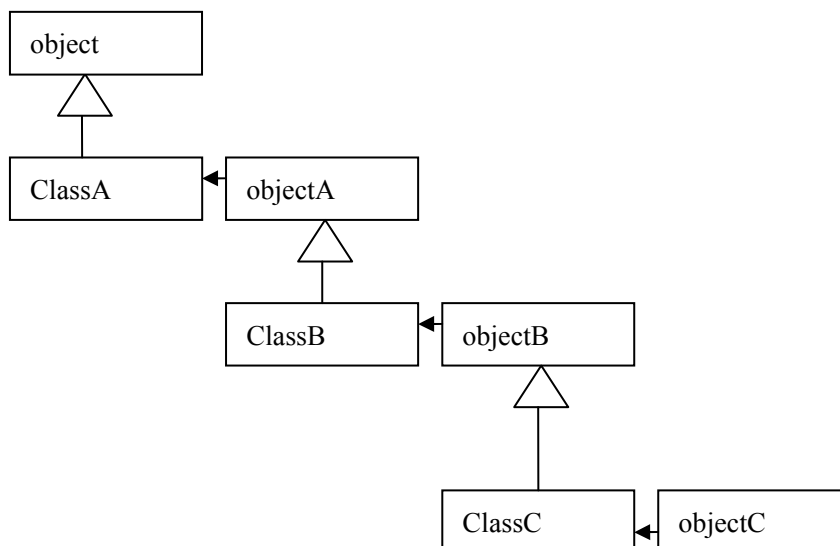


图 21.3 原型关系的类型链

有意思的是，JavaScript 并没有规定一个类型的原型的类型（这又是一段非常拗口的话），因此它可以是任何类型，通常是某种对象，这样，对象-类型-原形（对象）就可能构成一个环状结构，或者其它有意思的拓扑结构，这些结构为 JavaScript 带来了五花八门的用法，其中的一些用法不但巧妙而且充满美感。下面的一节主要介绍 prototype 的用法。

## 21.2.2 prototype 使用技巧

在了解 prototype 的使用技巧之前，首先要弄明白 prototype 的特性。JavaScript 为每一个类型(Type)都提供了一个 prototype 属性，将这个属性指向一个对象，这个对象就成为了这个类型的“原型”，这意味着由这个类型所创建的所有对象都具有这个原型的特性。另外，JavaScript 的对象是动态的，原型也不例外，给 prototype 增加或者减少属性，将改变这个类型的原型，这种改变将直接作用到由这个原型创建的所有对象上，例如：

例 21.5 给原型对象添加属性

```

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 21.5 给原型对象添加属性</title>
</head>
<body>
<h1 id="output"></h1>
<script language="javascript" type="text/javascript">
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function Point(x, y)
    {
        this.x = x;
        this.y = y;
    }
    var p1 = new Point(1,2);
    var p2 = new Point(3,4);
    Point.prototype.z = 0; //动态为 Point 的原型添加了属性
    dwn(p1.z);
    dwn(p2.z); //同时作用于 Point 类型创建的所有对象
-->
</script>
</body>
</html>

```

如果给某个对象的类型的原型添加了某个名为 **a** 的属性,而这个对象本身又有一个名为 **a** 的同名属性,则在访问这个对象的属性 **a** 时,对象本身的属性“覆盖”了原型属性,但是原型属性并没有消失,当你用 **delete** 运算符将对象本身的属性 **a** 删除时,对象的原型属性就恢复了可见性。利用这个特性,可以为对象的属性设定默认值,例如:

例 21.6 带默认值的 Point 对象

```

<html>
<head>
    <title>Example-21.6 带默认值的 Point 对象</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function Point(x, y)
    {
        if(x) this.x = x;

```

```

        if(y) this.y = y;
    }
    //设定 Point 对象的 x、y 默认值为 0
    Point.prototype.x = 0;
    Point.prototype.y = 0;
    //p1 是一个默认(0,0)的对象
    var p1 = new Point;
    //p2 赋予值(1,2)
    var p2 = new Point(1,2);
    dwn(p1.x+","+p1.y);
    dwn(p2.x+","+p2.y);
-->
</script>
</body>
</html>

```

执行结果如图 21.4 所示:

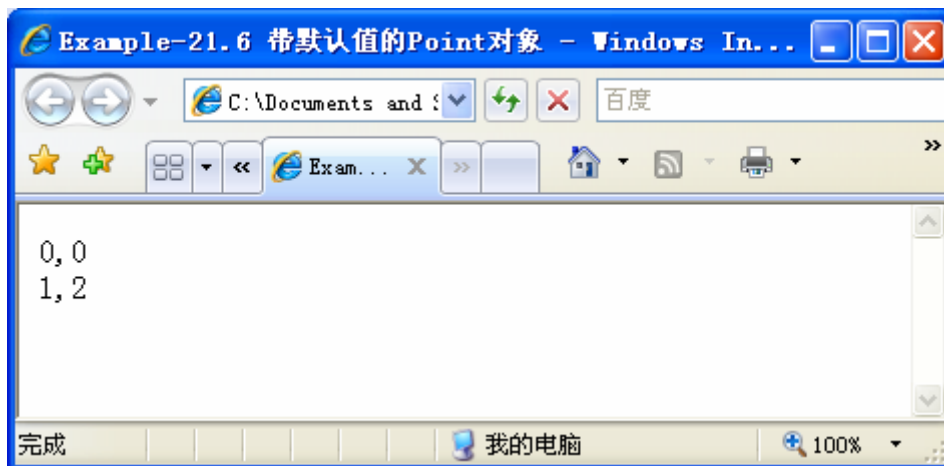


图 21.4 带默认值的 Point 对象

上面的例子通过 prototype 为 Point 对象设定了默认值(0,0)，因此 p1 的值为(0,0)，p2 的值为(1,2)，通过 delete p2.x, delete p2.y; 可以将 p2 的值恢复为(0,0)。下面是一个更有意思的例子:

例 21.7 delete 操作将对象属性恢复为默认值

```

<html>
<head>
    <title>Example-21.7 delete 操作将对象属性恢复为默认值</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function ClassA()
    {
        this.a = 100;

```

```

        this.b = 200;
        this.c = 300;
    }
    ClassA.prototype = new ClassA(); //将 a、b、c 同时设为 ClassA 的默认值

    //这个方法可将自身的非原型属性删除，达到 reset 的效果
    ClassA.prototype.reset = function()
    {
        for(var each in this)
        {
            delete this[each];
        }
    }

    //构造一个 ClassA 对象
    var a = new ClassA();
    dwn(a.a);
    //改变 a、b、c 属性的值
    a.a *= 2;
    a.b *= 2;
    a.c *= 2;
    //显示改变后的值
    dwn(a.a);
    dwn(a.b);
    dwn(a.c);
    //调用 reset 方法将对象的值恢复为默认值
    a.reset();
    //显示恢复后的默认值
    dwn(a.a);
    dwn(a.b);
    dwn(a.c);
-->
</script>
</body>
</html>

```

执行结果如图 21.5 所示：



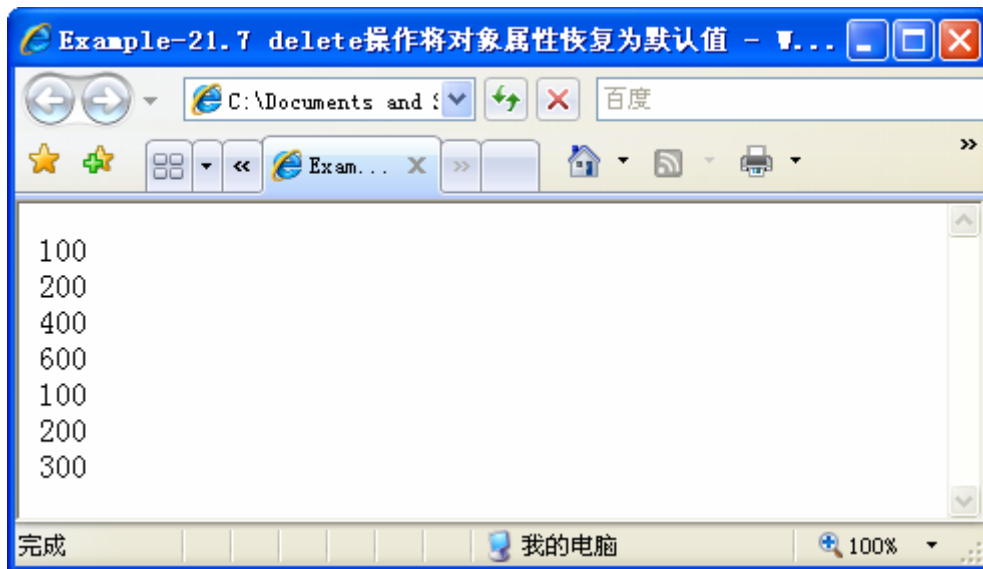


图 21.5 delete 操作将对象属性恢复为默认值

利用 prototype 还可以为对象的属性设置一个只读的 getter，从而避免它被改写。下面是一个例子：

#### 例 21.8 prototype 巧设 getter

```
function Point(x, y)
{
    if(x) this.x = x;
    if(y) this.y = y;
}
Point.prototype.x = 0;
Point.prototype.y = 0;

function LineSegment(p1, p2)
{
    //私有成员
    var m_firstPoint = p1;
    var m_lastPoint = p2;
    var m_width = {
        valueOf : function(){return Math.abs(p1.x - p2.x)},
        toString : function(){return Math.abs(p1.x - p2.x)}
    }
    var m_height = {
        valueOf : function(){return Math.abs(p1.y - p2.y)},
        toString : function(){return Math.abs(p1.y - p2.y)}
    }

    //getter
    this.getFirstPoint = function()
    {
        return m_firstPoint;
    }
    this.getLastPoint = function()
    {
```

```

        return m_lastPoint;
    }

    //公有属性
    this.length = {
        valueOf : function(){return Math.sqrt(m_width*m_width + m_height*m_height)},
        toString : function(){return Math.sqrt(m_width*m_width + m_height*m_height)}
    }
}
//构造 p1、p2 两个 Point 对象
var p1 = new Point;
var p2 = new Point(2,3);
//用 p1、p2 构造 line1 一个 LineSegment 对象
var line1 = new LineSegment(p1, p2);
//取得 line1 的第一个端点（即 p1）
var lp = line1.getFirstPoint();
//不小心改写了 lp 的值，破坏了 lp 的原始值而且不可恢复
//因为此时 p1 的 x 属性发生了变化
lp.x = 100;
alert(line1.getFirstPoint().x);
alert(line1.length); //就连 line1.length 都发生了改变

```

将 this.getFirstPoint()改写为下面这个样子：

```

this.getFirstPoint = function()
{
    function GETTER(){}; //定义一个临时类型
    //将 m_firstPoint 设为这个类型的原型
    GETTER.prototype = m_firstPoint;
    //构造一个这个类型的对象返回
    return new GETTER();
}

```

则可以避免这个问题，保证了 m\_firstPoint 属性的只读性。

实际上，将一个对象设置为一个类型的原型，相当于通过实例化这个类型，为对象建立只读副本，在任何时候对副本进行改变，都不会影响到原始对象，而对原始对象进行改变，则会影响到每一个副本，除非被改变的属性已经被副本自己的同名属性覆盖。用 delete 操作将对象自己的同名属性删除，则可以恢复原型属性的可见性。下面再举一个例子：

例 21.9 delete 操作恢复原型属性的可见性

```

<html>
<head>
    <title>Example-21.9</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {

```

```

    document.write(s + "<br/>");
}
//定义一个多边形（Polygon）类型
function Polygon()
{
    //存放多边形的顶点
    var m_points = [];
    m_points = Array.apply(m_points, arguments);

    //用上面介绍的那种方式定义 getter
    function GETTER(){};
    GETTER.prototype = m_points[0];
    this.firstPoint = new GETTER();

    //公有属性
    this.length = {
        valueOf : function(){return m_points.length},
        toString : function(){return m_points.length}
    }

    //添加一个或多个顶点
    this.add = function(){
        m_points.push.apply(m_points, arguments);
    }

    //取得序号为 idx 的顶点
    this.getPoint = function(idx)
    {
        return m_points[idx];
    }

    //设置特定需要的顶点
    this.setPoint = function(idx, point)
    {
        if(m_points[idx] == null)
        {
            m_points[idx] = point;
        }
        else
        {
            m_points[idx].x = point.x;
            m_points[idx].y = point.y;
        }
    }
}
//构造一个三角形 p

```

```

var p = new Polygon({x:1, y:2},{x:2, y:4},{x:2, y:6});
dwn(p.length);
dwn(p.firstPoint.x);
dwn(p.firstPoint.y);
p.firstPoint.x = 100; //不小心写了它的值
dwn(p.getPoint(0).x); //不会影响到实际的私有成员
delete p.firstPoint.x; //恢复
dwn(p.firstPoint.x);

p.setPoint(0, {x:3,y:4}); //通过 setter 改写了实际的私有成员
dwn(p.firstPoint.x); //getter 的值发生了改变
dwn(p.getPoint(0).x);
-->
</script>
</body>
</html>

```

执行结果如图 21.6 所示:

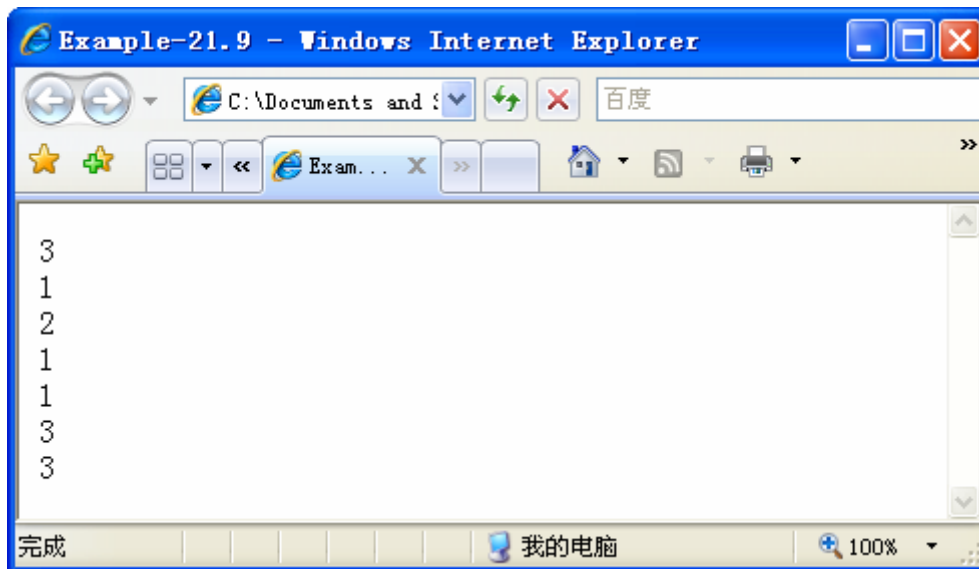


图 21.6 delete 操作恢复原型属性的可见性

注意, 上面的两个例子还说明了用 prototype 可以快速创建对象的一个或多个副本, 一般情况下, 利用 prototype 来创建大量的复杂对象的副本, 要比用其他任何方法来 copy 对象快得多。注意到, 以一个对象为原型, 来创建大量的新对象, 这正是 prototype pattern 的本质。下面是一个例子:

例 21.10 构建大量副本

```

var p1 = new Point(1,2);
var points = [];
var PointPrototype = function(){};
PointPrototype.prototype = p1;
for(var i = 0; i < 10000; i++)
{
    points[i] = new PointPrototype();
    //由于 PointPrototype 的构造函数是空函数, 因此它的构造要比直接构造//p1 副本快得多。
}

```

除了以上作用, prototype 更常见的用处是声明对象的方法。因为, 在一般情况下, 和属性相比, 对象

的方法不会轻易改变，正好利用 `prototype` 的静态特性来声明方法，这样避免了在构造函数中每次对方法进行重新赋值，节省了时间和空间。例如：

例 21.11 定义静态方法

```
function Point(x, y)
{
    this.x = x;
    this.y = y;
}
Point.prototype.distance = function(){
    return Math.sqrt(this.x * this.x + this.y * this.y);
}
```

上面的例子中，也可以用 `this.distance = function(){}` 的形式来定义 `Point` 对象的 `distance()` 方法，但是用 `prototype` 避免了每次调用构造函数时对 `this.distance` 的赋值操作和函数构造，如果程序里构造对象的次数很多的话，时间和空间的节省是非常明显的。



**小技巧：**尽量采用 `prototype` 定义对象方法，除非该方法要访问对象的私有成员或者返回某些引用了构造函数上下文的闭包。

习惯上，我们把采用 `prototype` 定义的属性和方法称为静态属性和静态方法，或者原型属性和原型方法，把用 `this` 定义的属性和方法称为公有属性和公有方法。

尽管采用 `prototype` 和采用 `this` 定义的属性和方法在对象调用的形式上是一致的，以至于在一段代码中甚至很难严格区分，但是用“静态”两个字还是很好地诠释了 `prototype` 在数据存储上的特质，即所有的实例共享唯一的副本。这一点和 C++ 中的 `static` 成员非常相似，但是和 C# 不同，C# 中的 `static` 方法的调用形式是通过类型的“.”运算符，这相当于 JavaScript 中的类属性和类方法。

- 关于“公有属性”、“私有属性”、“静态属性”和“类属性”的话题，在后续的章节中还会有更为详细的介绍。

除了上面所说的这些使用技巧之外，`prototype` 因为它独特的特性，还有其它一些用途，被用作最广泛和最广为人知的可能是用它来模拟继承，关于这一点，留待下一节中去讨论。

### 21.2.3 `prototype` 的实质

上面已经说了 `prototype` 的作用，现在我们来透过规律揭示 `prototype` 的实质。

我们说，`prototype` 的行为类似于 C++ 中的静态域，将一个属性添加为 `prototype` 的属性，这个属性将被该类型创建的所有实例所共享，但是这种共享是只读的。在任何一个实例中只能用自己的同名属性覆盖这个属性，而不能改变它。换句话说，对象在读取某个属性时，总是先检查自身域的属性表，如果有这个属性，则会返回这个属性，否则就去读取 `prototype` 域，返回 `prototype` 域上的属性。另外，JavaScript 允许 `prototype` 域引用任何类型的对象，因此，如果对 `prototype` 域的读取依然没有找到这个属性，则 JavaScript 将递归地查找 `prototype` 域所指向对象的 `prototype` 域，直到这个对象的 `prototype` 域为它本身或者出现循环为止。

而下面的这个代码揭示了对对象属性查找的 `prototype` 规律：

例 21.12 对象属性查找的 `prototype` 规律

```
<html>
<head>
    <title>Example-21.12</title>
</head>
<body>
```

```

<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }

    //定义 Point2D 对象
    function Point2D(x, y)
    {
        this.x = x;
        this.y = y;
    }
    Point2D.prototype.x = 0;
    Point2D.prototype.y = 0;

    //定义 ColorPoint2D 对象
    function ColorPoint2D(x, y, c)
    {
        this.x = x;
        this.y = y;
    }

    //ColorPoint2D 以 Point2D 对象为原型
    ColorPoint2D.prototype = new Point2D();
    ColorPoint2D.prototype.x = 1;
    ColorPoint2D.prototype.y = 1;

    //构造一个 ColorPoint2D 对象
    var cp = new ColorPoint2D(10,20,"red");
    dwn(cp.x); //10-先查找 cp 本身的属性
    delete cp.x;
    dwn(cp.x); //1-删除后查找上层原型链上的属性
    delete ColorPoint2D.prototype.x;
    dwn(cp.x); //0-删除后继续查找更上层原型链上的属性
-->
</script>
</body>
</html>

```

执行结果如图 21.7 所示:

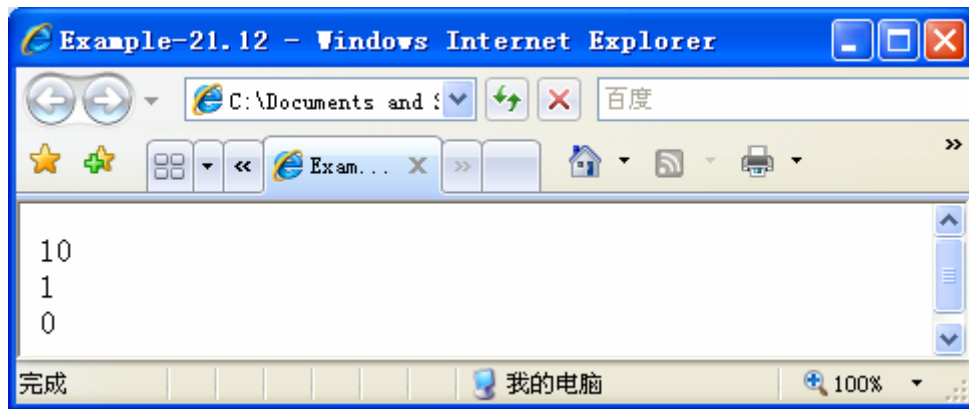


图 21.7 对象属性查找的 prototype 规律

## 21.2.4 prototype 的价值与局限性

从上面的分析我们理解了 prototype，通过它能够以一个对象为原型，安全地创建大量的实例，这就是 prototype 的真正含义，也是它的价值所在。后面我们会看到，利用 prototype 的这个特性，可以用来模拟对象的继承，但是要知道，prototype 用来模拟继承尽管也是它的一个重要价值，但是绝对不是它的核心，换句话说，JavaScript 之所以支持 prototype，绝对不是仅仅用来实现它的对象继承，即使没有了 prototype 继承，JavaScript 的 prototype 机制依然是非常有用的。

由于 prototype 仅仅是以对象为原型给类型构建副本，因此它 also 具有很大的局限性。首先，它在类型的 prototype 域上并不是表现为一种值拷贝，而是一种引用拷贝，这带来了“副作用”。改变某个原型上引用类型的属性的属性值（又是一个相当拗口的解释:P），将会彻底影响到这个类型创建的每一个实例。有的时候这正是我们需要的（比如某一类所有对象的改变默认值），但有的时候这也是我们所不希望的（比如在类继承的时候），下面给出了一个例子：

例 21.13 prototype 的局限性

```
<html>
<head>
  <title>Example-21.13</title>
</head>
<body>
<script>
<!--
  function dwn(s)
  {
    document.write(s + "<br/>");
  }
  //定义一个类型 ClassA，它的对象有一个引用类型的属性 a（a 是一个数组）
  function ClassA()
  {
    this.a=[];
  }
  //定义一个类型 ClassB
  function ClassB()
  {
    this.b=function(){};
  }
-->
```

```

//ClassB 以 ClassA 的对象为原型
ClassB.prototype=new ClassA();
//创建两个 ClassB 类型的对象
var objB1=new ClassB();
var objB2=new ClassB();
//改变 objB1 对象中的 a 属性的值
objB1.a.push(1,2,3);
dwn(objB2.a);
    //所有 b 的实例中的 a 成员全都变了!! 这并不是这个例子所希望看到的。
    //原因是 ClassA 类型的对象中的 a 属性的类型是一个数组
    //而数组是一个引用类型的属性（回忆什么是值类型和引用类型）
    //ClassB 的原型又引用 ClassA 的一个对象，因此
    //objB1 和 objB2 共享了引用类型的原型属性 a
    //于是通过两个中的任何一个操作 a 数组中的元素，结果都会导致 a 的值改变
-->
</script>
</body>
</html>

```

执行结果如图 21.8 所示：

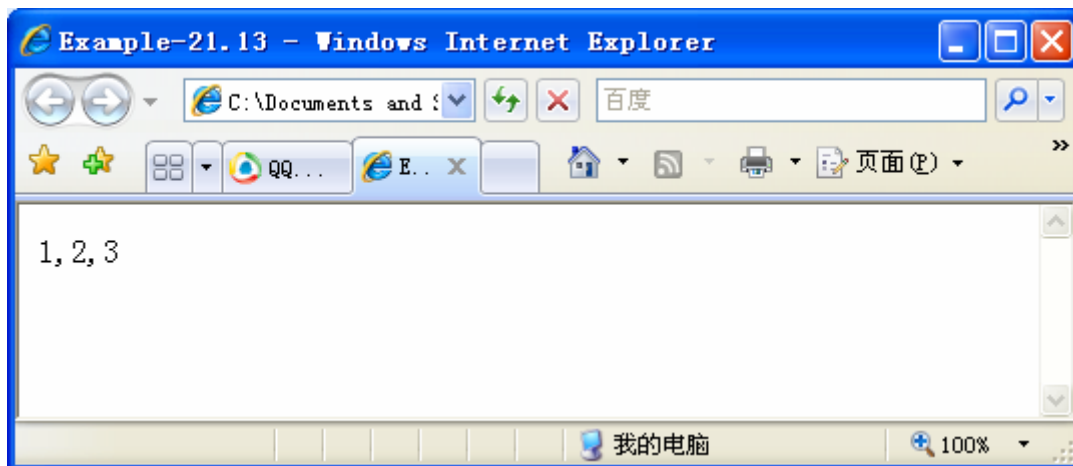


图 21.8 prototype 的局限性

总之，prototype 是一种面向对象的机制，它通过原型来管理类型与对象之间的关系，prototype 的特点是能够以某个类型为原型构造大量的对象。以 prototype 机制来模拟的继承是一种原型继承，它是 JavaScript 多种继承实现方式中的一种（在下一节我们会详细讨论她）。尽管 prototype 和传统的 Class 模式不同，但是我们仍然可以认为 prototype-based 是一种纯粹的面向对象机制。

## 21.3 继承与多态

继承与多态是面向对象最重要的两个特征，JavaScript 能用语言本身的特性来实现它们。

### 21.3.1 什么是继承

前面已经说过，如果两个类都是同一个实例的类型，那么它们之间存在着某些关系，我们把同一个实例的类型之间的泛化关系称为“继承”。



这很容易理解，例如，“白马”是一种“马”，“白马”和“马”之间的关系就是继承关系。“一匹白马”是“白马”的一个实例，“白马”是“一匹马”的类型，而“马”同样是“一匹马”的类型，“马”是“白马”的泛化，所以“白马”继承自“马”。上述听起来复杂的关系可以简单地用图 21.9 来表示：

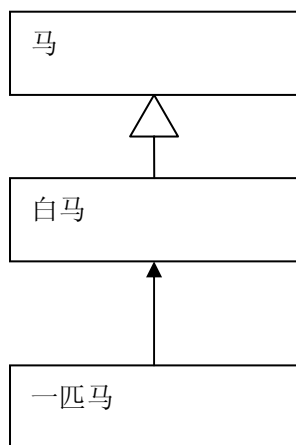


图 21.9 继承关系

一旦确定了两个类的继承关系，就至少意味着三层含义，一是子类的实例可以共享父类的方法，二是子类可以覆盖父类的方法或者扩展新的方法，三是子类和父类都是的子类实例的“类型”。

在 JavaScript 中，并不直接从文法上支持继承，换句话说，JavaScript 没有实现“继承”的语法，从这个意义上来说，JavaScript 并不是直接的面向对象的语言。

在 JavaScript 中，继承是通过模拟的方法来实现的，在下一小节里，我们将讨论 JavaScript 中具体的实现继承的方法。

## 21.3.2 实现继承的方法

从上一小节我们知道，要实现继承，其实就是实现上面所说的三层含义，即子类的实例可以共享父类的方法，子类可以覆盖父类的方法或者扩展新的方法，以及子类和父类都是子类实例的“类型”。

对于 JavaScript 来说要实现上面这三层含义，其实既简单又不简单。这个结论听起来很矛盾，但是你会发现它是有道理的。下面将介绍几种 JavaScript 中具体的实现继承的方法，并详细分析它们的利与弊。

### 21.3.2.1 构造继承法

JavaScript 中实现继承的第一种方法被称作构造继承法。顾名思义，这种继承方法的形式是在子类中执行父类的构造函数，例如：

例 21.14 构造继承法

```
<html>
<head>
  <title>Example-21.14 构造继承法</title>
</head>
<body>
<script>
<!--
  function dwn(s)
  {
    document.write(s + "<br/>");
  }
-->
```

```

//定义一个 Collection 类型
function Collection(size)
{
    this.size = function(){return size}; //公有方法，可以被继承
}

Collection.prototype.isEmpty = function(){ //静态方法，不能被继承
    return this.size() == 0;
}

//定义一个 ArrayList 类型，它“继承” Collection 类型
function ArrayList()
{
    var m_elements = []; //私有成员，不能被继承
    m_elements = Array.apply(m_elements, arguments);

    //ArrayList 类型继承 Collection
    this.base = Collection;
    this.base.call(this, m_elements.length);

    this.add = function()
    {
        return m_elements.push.apply(m_elements, arguments);
    }
    this.toArray = function()
    {
        return m_elements;
    }
}

ArrayList.prototype.toString = function()
{
    return this.toArray().toString();
}

//定义一个 SortedList 类型，它继承 ArrayList 类型
function SortedList()
{
    //SortedList 类型继承 ArrayList
    this.base = ArrayList;
    this.base.apply(this, arguments);

    this.sort = function()
    {
        var arr = this.toArray();
        arr.sort.apply(arr, arguments);
    }
}

```

```

}

//构造一个 ArrayList
var a = new ArrayList(1,2,3);
dwn(a);
dwn(a.size()); //a 从 Collection 继承了 size()方法
dwn(a.isEmpty()); //但是 a 没有继承到 isEmpty()方法

//构造一个 SortedList
var b = new SortedList(3,1,2);
b.add(4,0); //b 从 ArrayList 继承了 add()方法
dwn(b.toArray()); //b 从 ArrayList 继承了 toArray()方法
b.sort(); //b 自己实现的 sort()方法
dwn(b.toArray());
dwn(b);
dwn(b.size()); //b 从 Collection 继承了 size()方法
-->
</script>
</body>
</html>

```

执行结果如图 21.10 所示:

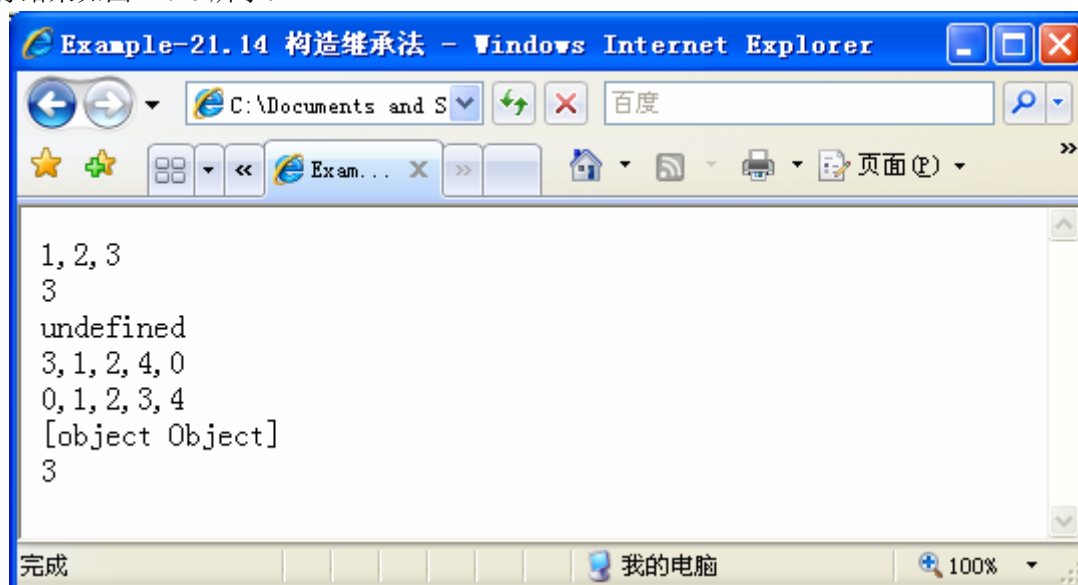


图 21.10 构造继承法

上面的这个例子中，类 `ArrayList` 继承了 `Collection`，而类 `SortedList` 继承了 `ArrayList`。注意到，这种继承关系是通过在子类中调用父类的构造函数来维护的。如，`ArrayList` 中调用了 `this.base.call(this.base, m_members.length)`；而 `SortedList` 中则调用了 `this.base.apply(this.base, arguments)`。

从继承关系上看，`ArrayList` 继承了 `Collection` 公有的 `size()`方法，但是却无法继承 `Collection` 静态的 `isEmpty()`方法。`ArrayList`定义了自己的 `add()`方法。`SortedList`继承了 `ArrayList`的 `add()`方法，以及从 `Collection`继承下来的 `size()`方法，但是却不能继承 `ArrayList`重写的静态 `toString()`方法。`SortedList`定义了自己的 `sort()`方法。注意到 `SortedList`的 `size()`方法实际上有一个 BUG，用 `add()`方法添加了元素之后，并没有变更 `size()`的值，这是因为 `Collection`类中定义的 `size()`返回的是一个外部环境的参数（具体的奥妙在第 22 章中会有详细的解释），它不会受到子类 `ArrayList`和 `SortedList`的影响。所以要维持 `size()`的正确性，只能在 `ArrayList`

类中重写 size()方法，如下：

```
this.size = function(){return m_elements.length}
```

注意到实际上构造继承法并不能满足继承的第三层含义，无论是 a instanceof Collection 还是 b instanceof ArrayList，返回值总是 false。其实，这种继承方法除了通过调用父类构造函数将属性复制到自身之外，并没有作其他任何的事情，严格来说，它甚至算不是上继承。尽管如此用它的特性来模拟常规的对象继承，也已经基本上达到了我们预期的目标。这种方法的优点是简单和直观，而且可以自由地用灵活的参数执行父类的构造函数，通过执行多个父类构造函数方便地实现多重继承（下一小节里将讨论多重继承的概念），缺点主要是不能继承静态属性和方法，也不能满足所有父类都是子类实例的类型这个条件，这样对于实现多态将会造成麻烦（多态的概念 21.3.5 小节将有详细的讨论）。

### 21.3.2.2 原型继承法：

原型继承法是 JavaScript 中最流行的一种继承方式。以至于有人说，JavaScript 的面向对象机制实际上是基于原型的一种机制，或者说，JavaScript 是一种基于原型的语言。

基于原型编程是面向对象编程的一种特定形式。在这种基于原型的编程模型中，不是通过声明静态的类，而是通过复制已经存在的原型对象来实现行为重用。这个模型一般被称作是 class-less，面向原型，或者是基于接口编程。

既然如此，基于原型模型其实并没有“类”的概念，这里所说的“类”是一种模拟，或者说是沿用了传统的面向对象编程的概念。

所以很快我们会发现，这种原型继承法和传统的类继承法并不一致。

或者说，原型继承法虽然有类继承法无法比拟的优点，也有其缺点，一个很大的缺陷就是前面所说的 prototype 的副作用。

要了解什么是“原型继承法”，先回顾一下上一节里 prototype 的特性，我们说，prototype 的最大特点是能够让对象实例共享原型对象的属性，因此如果把某个对象作为一个类型的原型，那么我们说这个类型的所有实例以这个对象为原型。这个时候，实际上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（回顾一下原型模式的第二个要求）。有意思的是，JavaScript 正是这样做的，例如：

例 21.15 原型继承法

```
<html>
<head>
  <title>Example-21.15 原型继承法</title>
</head>
<body>
<script>
<!--
  function dwn(s)
  {
    document.write(s + "<br/>");
  }
  //定义一个 Point 类型
  function Point(dimension)
  {

    this.dimension = dimension;
  }

  //定义一个 Point2D 类型，“继承” Point 类型
  function Point2D(x, y)
```

```

    {
        this.x = x;
        this.y = y;
    }
    Point2D.prototype.distance = function()
    {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
    Point2D.prototype = new Point(2);    //Point2D 继承了 Point

//定义一个 Point3D 类型，也继承 Point 类型
function Point3D(x, y, z)
{
    this.x = x;
    this.y = y;
    this.z = z;
}
Point3D.prototype = new Point(3);    //Point3D 也继承了 Point

//构造一个 Point2D 对象
var p1 = new Point2D(0,0);
//构造一个 Point3D 对象
var p2 = new Point3D(0,1,2);

dwn(p1.dimension);
dwn(p2.dimension);
dwn(p1 instanceof Point2D); //p1 是一个 Point2D
dwn(p1 instanceof Point);   //p1 也是一个 Point
dwn(p2 instanceof Point);   //p2 是一个 Point

-->
</script>
</body>
</html>

```

执行结果如图 21.11 所示：



图 21.11 原型继承法

在这个简单的例子里, Point2D 和 Point3D 都以 Point 为原型, Point 为 Point2D 和 Point3D 提供 dimension (维度) 属性。从面向对象的角度, 相当于 Point2D 和 Point3D 都继承了 Point, 有趣的是, p1 和 p2 虽然分别是 Point2D 和 Point3D 的实例, 但是 p1 instanceof Point 和 p2 instanceof Point 的值都是 true。

之前我们说过, 类型的原型可以构成一个原型链, 这样就能实现多个层次的继承, 继承链上的每一个对象都是实例的类型。下面是一个例子:


例 21.16 prototype 的多重继承

```
function Point()
{
    .....
}
//Point 继承 Object, 这个通常可以省略, 因为自定义类型的缺省原型为 Object
Point.prototype = new Object();
function Point2D()
{
    .....
}
//Point2D 继承 Point
Point2D.prototype = new Point();
function ColorPoint2D()
{
    .....
}
//ColorPoint2D 又继承 Point2D
ColorPoint2D.prototype = new Point2D();
```

同构造继承法相比, 原型继承法的优点是结构更加简单, 而且不需要每次构造都调用父类的构造函数 (尽管你仍然可以调用它), 并且不需要通过复制属性的方式就能快速实现继承。但是它的缺点也是很明显的, 首先它不方便直接支持多重继承, 因为一个类型只能有一个原型; 其次它不能很好地支持多参数和动态参数的父类构造, 因为在原型继承的阶段你还不能决定以什么参数来实例化父类对象; 第三是你被迫要在原型声明阶段实例化一个父类对象作为当前类型的原型, 有的时候父类对象是不能也不应该随便实例化的; 最后一个缺点是之前提到过的 prototype 的“副作用”。

既然 prototype 继承有那么多缺点, 那么我们是不是不应该使用它?

答案是否定的, 因为, 同类继承相比, 原型继承本来就是一个简化了的版本, 因此我们不应该要求它完全达到标准的类继承的效果, 实际上, 当你的父类是一个简单、抽象的模型或者一个接口的时候, 原型继承的表现在已知的 JavaScript 对象继承中是最好的, 甚至可以说, prototype 继承才是 JavaScript 文法上提供的真正意义上的继承机制。所以, 我们在使用 JavaScript 时, 能够采用原型继承的地方, 应当尽可能地采用这种继承方式。

 现在回过头来探讨前面关于“基于对象”和“面向对象”的话题。那么 JavaScript 究竟是不是一种面向对象语言呢? 我认为是。

面向对象不是只有类模型一种, prototype-based (基于原型) 是 class-based (基于类) 的简化版本, 是一种 class-less 的面向对象。对应地, prototype 继承是 class 继承的简化版本, 相对于 class 继承来说它简化了许多东西, 例如省略了多重继承、基类构造函数、忽略了引用属性的继承.....但不能因为它不支持这些特性, 就不承认它是一种完整的继承, 否则我们就在用 class-based 的眼光来看待 prototype-based, 实际上这可能是错误的。

其实 prototype-based 本来就是 class-based 的简化版, 因此给继承加一个限制, 要求父类必须是一个抽

象类或者接口，那么 prototype-based 就没有任何问题了。当然，也许这么做会使 OOP 的 reuse（重用）能力减弱（以 class-based 的眼光来看），但是这可以通过其他机制来弥补，比如结合其他类型的继承方式，再比如闭包。

是否为继承添加额外的特性，开发者可以自由选择，但是在不需要这些额外特性的时候，还是有理由尽量用 prototype-based 继承。

总而言之，prototype-based 认为语言本身可能不需要过分多的 reuse 能力，它牺牲了一些特型来保持语言的简洁，这没有错，prototype-based 虽然比 class-based 简单，但它依然是真正意义上的 object-based。

### 21.3.2.3 实例继承法

构造继承法和原型继承法各有一个明显的缺点前面并没有具体提到。由于构造继承法没有办法继承类型的静态方法，因此它无法很好地继承 JavaScript 的核心对象（还记得什么是核心对象么？如果忘了，回顾一下第 7 章）。而原型继承法虽然可以继承静态方法，但是依然无法很好地继承核心对象中的不可枚举方法，下面举出一个例子：

例 21.17 构造继承的局限性

```
function MyDate()  
{  
    this.base = Date;  
    this.base.apply(this, arguments);  
}  
var date = new MyDate();  
alert(date.toGMTString);
```

//核心对象的某些方法不能被构造继承，原因是核心对象并不像我们自定义的一般对象那样

//在构造函数里进行赋值或初始化操作

上面的例子中，我们尝试用构造继承的方法来继承 Date 类型，但是却发现它不能很好地工作，date.toGMTString 的值为 undefined，这个方法并没有被成功继承。那么，既然用构造继承法不行，用原型继承法又如何呢？

例 21.18 原型继承的局限性

```
function MyDate()  
{  
}  
MyDate.prototype = new Date();  
var date = new MyDate();  
alert(date.toGMTString);
```

原型继承法的表现似乎不错，这一次终于获得了基类的方法，然而，令人吃惊的是，当你尝试调用 date 对象的 toString 或 toGMTString 方法时，Internet Explorer 抛出一个怪异的异常，说，“‘[object]’不是日期对象”。功败垂成，看来原型继承法还是不能解决核心对象的继承问题。

那么核心对象是不是就不能被继承呢？答案是否定的。下面要介绍的这种继承方法就是最好的继承核心对象的方法，不论是继承 Date 类型、String 类型还是 Array 类型或者其他什么核心类型，它都能够很好地工作。

先回顾一下第 7 章中曾经说过的一句话：“构造函数通常没有返回值，它们只是初始化由 this 值传递进来的对象，并且什么也不返回。如果函数有返回值，被返回的对象就成了 new 表达式的值”，这句话引出了一种新的继承方法，我们叫它“实例继承法”。下面这个例子给出了实例继承法继承 Date 对象的例子：

例 21.19 实例继承法

```
<html>  
<head>  
    <title>Example-21.19 实例继承法</title>  
</head>
```

```

<body>
<script>
<!--
function dwn(s)
{
    document.write(s + "<br/>");
}
function MyDate()
{
    var instance = new Date(); //instance 是一个新创建的日期对象
    instance.printDate = function(){
        document.write("<p> "+instance.toLocaleString()+"</p> ");
    } //对 instance 扩展 printDate()方法
    return instance; //将 instance 作为构造函数的返回值返回
}
var myDate = new MyDate();
dwn(myDate.toGMTString());
myDate.printDate();
-->
</script>
</body>
</html>

```

执行结果如图 21.12 所示:

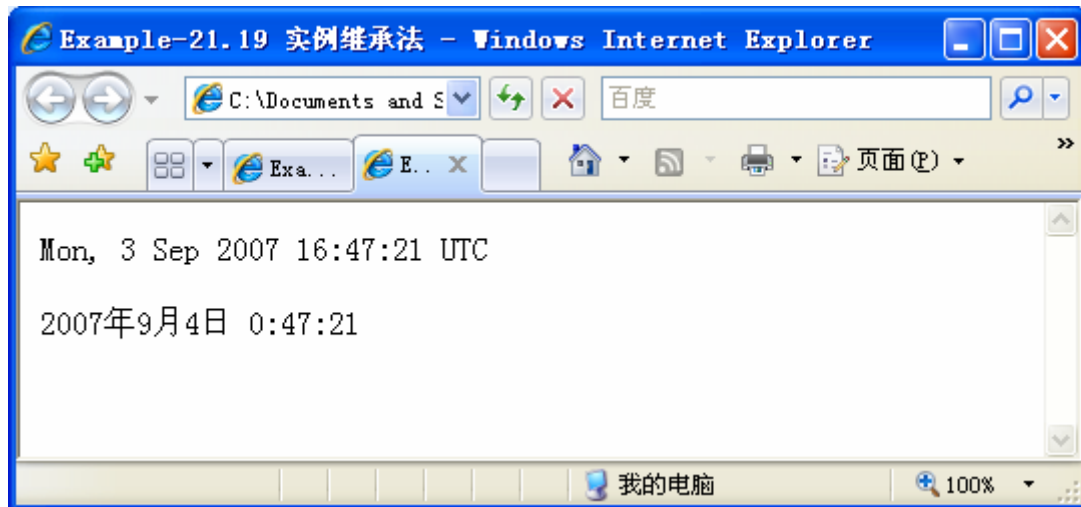


图 21.12 实例继承法

我们可以看到，这一次 MyDate 类型工作得很好，它确实继承了核心对象 Date 的方法。通常情况下要对 JavaScript 原生的核心对象或者 DOM 对象进行继承时，我们会采用这种继承方法。不过，它也有几个明显的缺点，首先，由于它需要在执行构造函数的时候构造基类的对象，而 JavaScript 的 new 运算与函数调用不同的是不能用 apply() 方法传递给它不确定的 arguments 集合，这样就会对那些可以接受不同类型和不同数量参数的类型的继承造成比较大的麻烦。

其次，从上面的例子可以看出，这种继承方式是通过在类型中构造对象并返回的办法来实现继承的，那样的话 new 运算的结果实际上是类型中构造的对象而不是类型本身创建的对象，alert(myDate instanceof MyDate); 的执行结果将会是 false，对象的构造函数将会是实际构造的对象的构造函数而不是类型本身的构造函数，尽管你可以通过赋值的办法修正它，但是你却无法修正 instanceof 表达式的结果，这不能不说是



一个很大的遗憾。

第三，这种方法一次只能返回一个对象，它和原型继承法一样不能支持多重继承。

所以，我们的结论是，构造继承法也不是一种真正的继承法，它也是一种模拟。构造继承法是目前所知的唯一一种可以较好地继承 JavaScript 核心对象的继承法。当你要继承 JavaScript 的核心对象或者 DOM 对象时，可以考虑采用这种方法。



在第 22 章我们讨论闭包的时候，将会给出一个比较复杂的 ListArray 的例子，它采用的就是构造继承法，继承 Array 对象。

#### 21.3.2.4 拷贝继承法

顾名思义，拷贝继承法就是通过对象属性的拷贝来实现继承，早期的 Prototype 和其他一些框架在特定的情况下就用到了这种继承方法。下面是一个拷贝继承的例子：

例 21.20 拷贝继承法

```
Function.prototype.extends = function(obj)
{
    for(var each in obj)
    {
        this.prototype[each] = obj[each];
        //对对象的属性进行一对一的复制，但是它又慢又容易引起问题
        //所以这种“继承”方式一般不推荐使用
    }
}
var Point2D = function(){
    .....
}
Point2D.extends(new Point())
{
    .....
}
```

从上面的例子中可以看出，拷贝继承法实际上是通过反射机制拷贝基类对象的所有可枚举属性和方法来模拟“继承”，因为可以拷贝任意数量的对象，因此它可以模拟多继承，又因为反射可以枚举对象的静态属性和方法，所以它同构造继承法相比的优点是可以继承父类的静态方法。但是由于是反射机制，因此拷贝继承法不能继承非枚举类方法，例如父类中重载的 toString() 方法，另外，拷贝继承法也有几个明显的缺点，首先是通过反射机制来复制对象属性效率上非常低下。其次它也要构造对象，通常也不能很好地支持灵活的可变参数。第三，如果父类的静态属性中包含引用类型，它和原型继承法一样导致副作用。第四，当前类型如果有静态属性，这些属性可能会被父类的动态属性所覆盖。最后这种可支持多重继承的方式并不能清晰地描述出父类与子类的相关性。

#### 21.3.2.5 几种继承法的比较

我们通过下表总结一下上面各种继承方法的优缺点：

表 21.1 比较几种继承方法的优劣

比较项	构造继承	原型继承	实例继承	拷贝继承
静态属性继承	N	Y	Y	Y
内置对象继承	N	部分	Y	Y
多参多重继承	Y	N	Y	N
执行效率	高	高	高	低
多继承	Y	N	N	Y
instanceof	false	true	false	false

### 21.3.2.6 混合继承

混合继承是将两种或者两种以上的继承同时使用，其中最常见的是构造继承和原型继承混合使用，这样能够解决构造函数多参多重继承的问题。例如：

例 21.21 混合继承法

```
<html>
<head>
  <title>Example-21.21 混合继承法</title>
</head>
<body>
<script>
<!--
function Point2D(x, y)
{
  this.x = x;
  this.y = y;
}
function ColorPoint2D(x, y, c)
{
  Point2D.call(this, x, y);
  //这里是构造继承，调用了父类的构造函数
  this.color = c;
}
ColorPoint2D.prototype = new Point2D();
//这里用了原型继承，让 ColorPoint2D 以 Point2D 对象为原型
-->
</script>
</body>
</html>
```

另外，在模拟多继承的时候，原型继承和部分条件下的拷贝继承的同时使用也较常见。

## 第二十二章 闭包与函数式编程

在 JavaScript 里，“闭包”是一个神奇的东西。借着闭包的力量，我们将跨过面向对象的领域，来攀登一座新的高峰。保罗格雷厄姆曾经说过，我认为目前为止只有两种真正干净利落，始终如一的编程模式：C 语言模式和 Lisp 语言模式。此二者就象两座高地，在它们中间是尤如沼泽的低地。在这里 C 语言代表着过程式语言的精髓，它目前所知的高层境界是面向对象。而称为 Lisp 的语言，则以另一种形式的无与伦比的美，成为与过程化对等的存在，即，我们将要介绍的函数式编程。

### 22.1 动态语言与闭包

程序语言中的闭包（closure）概念不是由 JavaScript 最先提出的，从 smalltalk 开始，闭包就成了编程语言的一个重要概念。几乎所有的知名动态语言（如 Perl、python、ruby 等）都支持闭包，JavaScript 也不例外。

闭包（closure）的确是个精确但又很难解释的电脑名词。因此在理解它之前，必须先解释下面一些简单概念。

## 22.1.1 动态语言

所谓动态程序设计语言（Dynamic Programming Language），准确地说，是指程序在运行时可以改变其结构：新的函数可以被引进，已有的函数可以被删除等在结构上的变化。相反，非动态语言在编译（或解释）时，程序结构已经被确定，在执行过程中不能再发生改变。

JavaScript 是一个典型的动态语言。除此之外如 Ruby、Python 等也都属于动态语言，而 C、C++ 等语言则不属于动态语言。



一些人习惯上将编译型语言认为是非动态语言，而解释型语言认为是动态语言，实际上这是完全错误的概念，动态语言的概念与语言是编译还是解释没有关系，一些解释型的语言也可以是静态语言，编译型语言确实不易设计为动态语言，但也仍然可以通过良好的设计和使用技巧达到“动态”的效果。

在这里还需要区分一下另外一对容易和上面概念混淆的概念，即动态类型语言（Dynamically Typed Language）和静态类型语言（Static Typed Language）。

所谓动态类型语言是指在执行期间才去发现数据类型的语言，静态类型语言与之相反，如 JavaScript、VBScript 和 Perl 都是典型的动态类型语言。很多人常常将动态类型语言和动态语言混为一谈，显然从上面的描述看来，它们是两个完全不同的概念。虽然，大多数动态语言都是动态类型语言，但动态语言本身并不要求一定是动态类型的，而动态类型语言也不一定是动态语言。

## 22.1.2 语法域和执行域

所谓语法域，是指定义某个程序段落时的区域，所谓执行域则是指实际调用某个程序段落时所影响的区域。

在非动态语言中，语法域和执行域范围基本上是一致的，执行域通常只能访问它自身语法域的范围和少量向它开放的语法域，而不能访问它外层或者与它关联的执行域。而在动态语言中，执行域的范围通常大得多。



非动态语言，如 C++ 的函数在调用时（执行域上）只能访问自身语法域上允许访问的环境，如全局变量和函数、所在对象的属性和方法以及自身的参数和临时变量，这和定义函数时的许可范围一致。动态语言如 JavaScript 的函数不但能够访问语法域上的这些范围，还能够访问它外层环境中的执行域范围，例如：

例 22.1 动态语言的执行域

```
<html>
<head>
  <title>Example-22.1 动态语言的执行域</title>
</head>
<body>
<script>
<!--
//产生随机数的函数
function RandomAlert()
{
    var x = Math.random()
    return function()
```

```

    {
        alert(x);
    }
}
var a = RandomAlert();
    //闭包的执行域随函数调用而创建
var b = RandomAlert();
a(); //调用 a, 打印出产生的随机数
b(); //调用 b, 打印出产生的随机数
//一般情况下, a 和 b 得到的数值不同
-->
</script>
</body>
</html>

```

## 22.1.2 JavaScript 的闭包

在程序语言中, 所谓**闭包**, 是指语法域位于某个特定的区域, 具有持续参照 (读写) 位于该区域内自身范围之外的执行域上的非持久型变量值能力的段落。这些外部执行域的非持久型变量神奇地保留它们在闭包最初定义 (或创建) 时的值 (深连结)。

从上面的概念可以看出, 闭包通常是在动态语言中才有的概念, 它是某些可以访问外部执行域的段落。JavaScript 中的闭包, 是通过定义在函数体内部的 `function` 来实现的。



例 22.1 就是典型的闭包应用, `RandomAlert()` 函数的返回值是一个闭包, `a()`, `b()` 分别访问了闭包两次被创建时对应的外层 `RandomAlert()` 函数的执行域上的局部变量 `x` 的值。

闭包这个概念我们之前已经多次提到过, 但是一直没有解释清楚。相信你即使看了本节前面两段的解释, 仍然还是会觉得有一点困惑。闭包和函数的概念到底有什么相同点和不同点, 相信这是大多数读到这里的读者心中最大的疑惑。其实, 闭包和函数的关系, 应当类似于一种动态和静态、结构和实例的关系, 下面再通过一个例子来简单说明:

例 22.2 闭包的本质

```

<html>
<head>
    <title>Example-22.2 闭包的本质</title>
</head>
<body>
<script>
<!--
//A 是一个普通的函数
function A(a)
{
    return a;
}
//B 是一个带函数返回值的函数
function B(b)
{

```

```

    return function () {
        return b;
    }
}
var x = A(10);
    //因为 A 除了返回 a 外什么也没做，执行 A()函数后，调用堆栈被销毁
    //没有产生闭包，或者说在调用“瞬间”产生了闭包，然后马上被销毁
var y = B(20);
    //因为 B 返回了一个匿名函数引用，它访问到 B()被调用时产生的环境
    //因此这里产生了一个“闭包结构” (closure 或者 function instance)
    //在它的环境中，b = 20，因此 y()的返回结果是 20
var z = B(30);
    //同样，这里产生了第二个“闭包结构”
    //在它的环境中，b = 30，因此 z()的返回结果是 30
alert(x); //得到 10
alert(y()); //得到 20
alert(z()); //得到 30
-->
</script>
</body>
</html>

```

我们说例 22.2 中，y()和 z()的结果不同，因为两次 B()创造的闭包被执行时访问的是不同的 b 值，它正好是分别的调用 B()时 b 被初始化的值。这里最奇怪的地方在于，当 y()和 z()被调用时，B()函数调用已经结束了。如果你有 C++、Java 或者其他什么编程语言的知识，也许你的潜意识里会认为当 B()调用结束时，局部变量 b 的值已经被销毁，但结果却是令人惊讶的，由于被返回的闭包里引用了 B()调用域上的 b 值，所以它并没有随着 B()调用的结束而被销毁。

类似的还有之前我们见到过的例 22.1 和例 4.6，在这里我们再次列出例 4.6:

```

function dice(count, side) //count 定义骰子的数量，side 定义每个骰子的面数
{
    var ench = Math.floor(Math.random() * 6); //+0~+5 的骰子随机变数修正
    //这里返回一个闭包，该闭包的作用是对指定的面数和修正值的骰子进行“投掷”
    return function()
    {
        var score = 0;
        for(var i = 0; i < count; i++)
        {
            score += Math.floor(Math.random() * side) + 1;
        }
        return score + ench;
    }
}
var d1 = dice(2,6); //生成一组 2d6+n 的骰子，其中的 n 为 0~5 的随机数
var d2 = dice(1,20); //生成一颗 20 面的骰子，带有 0~5 的随机点数修正

```

例 4.6 中，d1、d2 引用的闭包都使用了外部环境中的局部变量 ench 和 side 的值，而这两个局部变量是在 dice()方法才被初始化的，在 dice()调用结束后，它们并没有被销毁。当你调用 d1()和 d2()时，你将会引用到 d1 和 d2 在获取闭包时分别创建的 side 和 ench 值。



我通常认为闭包是一种引用结构，至少在 JavaScript 中是这样的。JavaScript 中的闭包 (closure)，也可以理解为一种“函数实例引用” (function instance referer)。

## 22.2 闭包的特点与形式

闭包，作为一种特殊的结构，有其自身的特点和各种形式。

### 22.2.1 闭包的内在：自治的领域

闭包的“闭”是指闭包的内部环境对外部不可见，也就是说闭包具有控制外部域的能力但是又能防止外部域对闭包的反向控制。换句话说，闭包的领域是对外封闭的。

闭包的这一个特点不用过多解释，因为 JavaScript 闭包是通过 function 实现的，所以它天然具有基本的函数特征，在闭包内声明的变量，闭包外的任何环境中都无法访问的，除非闭包向外部环境提供了访问它们的接口。例如：

例 22.3 闭包的封闭性

```
<html>
<head>
  <title>Example-22.3 闭包的封闭性</title>
</head>
<body>
<script>
<!--
  function dwn(s)
  {
    document.write(s + "<br/>");
  }
  //我们说匿名函数调用产生一个“瞬时”的闭包
  //因此当调用结束后，私有变量无法访问，并且如果没有外部引用存在
  //内部对象就会被销毁
  //而如果返回了函数，或者被全局引用，则“闭包”被保留了下来
  //闭包中的内容被“有选择”地开放出来
  (function(){
    //封闭的私有域
    var innerX = 10, innerY = 20;

    //开放的公共域
    outerObj = {x : innerX, y : innerY};
  })();

  try{
    dwn(innerX); //内部数据无法访问
  }
  catch(ex){
    dwn("内部数据无法访问");
```

```

    }
    dwn(outerObj.x); //通过外部接口访问
-->
</script>
</body>
</html>

```

执行结果如图 22.1 所示:

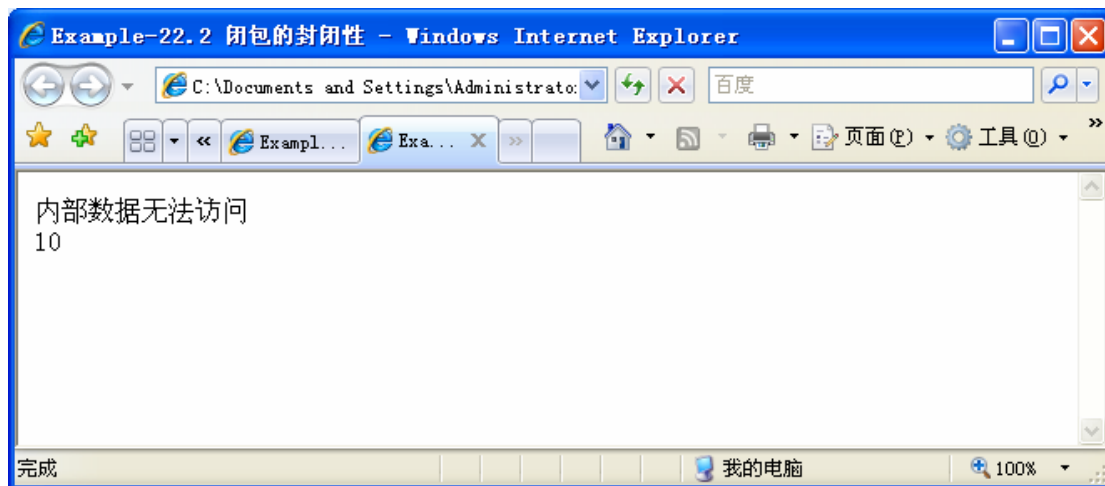



图 22.1 闭包的封闭性

## 22.2.2 访问外部环境

我们说，闭包可以访问外部环境，前面我们已经见过闭包读外部环境的例子，事实上闭包不但可以读外部环境，还可以写外部环境。

 严格来说，外部环境既包括闭包外部的语法域也包括闭包外部的执行域。但是闭包对语法域环境的访问和普通函数一致，因此我们这里主要强调的是闭包对执行域环境的访问。

下面是一个用闭包写外部环境的例子：

例 22.4 闭包改变外部环境

```

<html>
<head>
  <title>Example-22.4 闭包改变外部环境</title>
</head>
<body>
<script>
<!--
//定义一个计数器生成函数，生成某种类型的计数器
function counter(iden, addi)
{
  //闭包“外部”，函数 counter “内部”的参数 iden 的值在闭包被调用的时候会被改变
  return function(){
    //改变 iden 的值
    iden = iden+addi;
    return iden;
  };
}

```


```

    }
}
//产生一个从 0 开始计数，每次计数值加 1 的计数器
var c1 = counter(0, 1);
//产生一个从 10 开始计数，每次计数值减 1 的计数器
var c2 = counter(10, -1);
for(var i = 0; i < 10; i++){
    //循环计数
    c1();
}
for(var i = 0; i < 10; i++){
    //循环计数
    c2();
}

-->
</script>
</body>
</html>

```

我们说 `c1` 和 `c2` 通过调用 `counter` 构造了两个不同的计数器它们的初值分别是 0 和 10，步长分别是 1 和 -1，在调用闭包时，我们用步长改变计数器值 `iden`，使得计数器的值按照给定的步长递增。

 上面的例子用面向对象的思想也能够实现，但是用闭包从形式上要比用对象简洁一些，后面我们会看到，实际上我们在上面的例子中用了另外一种和面向对象等同的抽象思想，即函数式（functional）思想。

有趣的是，外部环境的读写和闭包出现在函数体内的顺序没有关系，例如：

```

function createClosure(){
    var x = 10;
    return function()
    {
        return x;
    }
}

```

和

```

function createClosure(){
    function a()
    {
        return x;
    }
    var x = 10;
    return a;
}

```

的结果是一样的。



## 22.2.3 闭包和面向对象

我们说, JavaScript 的对象中的私有属性其实就是环境中的非持久型变量, 而在构造函数内用 `this.foo = function(){...}` 形式定义的方法其实也是闭包的一种创建形式, 只是它提供的是一种开放了“外部接口”的闭包:

例 22.5 闭包和面向对象

```
<html>
<head>
  <title>Example-22.5 闭包和面向对象</title>
</head>
<body>
<script>
<!--
  function dwn(s)
  {
    document.write(s + "<br/>");
  }
  //定义一个 Foo 类型
  function Foo(a)
  {
    function _pC() //私有的函数
    {
      return a;
    }

    //公有的函数, 通过它产生的闭包可以访问对象内部的私有方法_pC()
    this.bar = function(){
      dwn("foo" + _pC() + "!");
    }
  }
  var obj = new Foo("bar");
  obj.bar(); //显示 Foo bar!
-->
</script>
</body>
</html>
```

执行结果如图 21.22 所示:

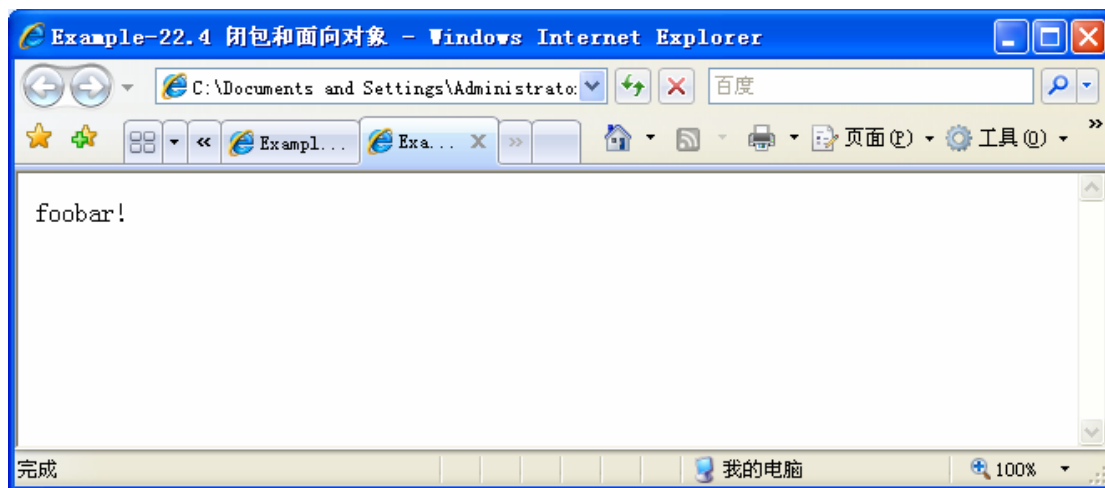


图 22.2 闭包和面向对象

## 22.2.4 其他形式的闭包


JavaScript 的闭包不仅仅只有以上几种简单的形式，它还有其他更加“诡异”的形式，例如：

例 22.6 闭包的其他形式

```
<html>
<head>
  <title>Example-22.6 闭包的其他形式</title>
</head>
<body>
<script>
<!--
//测试函数，异步计数
function test()
{
  for (var i = 0; i < 5; i++)
  {
    //如果没有这个闭包，不能正确得到 0,1,2,3,4 的结果
    //因为 setTimeout 是在循环结束后才被“异步”调用的
    (function(j){
      setTimeout(function(){alert(j)}, 100);
    })(i);
  }
}
test();
-->
</script>
</body>
</html>
```

这个例子我们曾经见到过，`function(j){setTimeout(function(){alert(j)}, 100);}`是一个闭包，它访问 `test()` 的调用环境，而 `function(){alert(j)}` 也是一个闭包，它又访问由外部闭包提供的环境。这样的闭包使用法经常被用在异步的环境中，用来将特定的引用“绑定”给闭包。例如，下面的用法通过闭包环境绑定修正了事件注册时的“this”指针：

```
button1.onclick =  
    (function(owner){return function(){button1_click.apply(owner,arguments)}})(button1);
```

 回顾一下第 21 章中关于利用闭包修正 this 指针的讨论，这是闭包的一个非常重要的作用。

## 22.3 闭包使用的注意事项

我们说，闭包的最大特点是可以访问外部环境的执行域，而这些执行域相对于语法域来说是灵活多变的，这有可能为程序引入额外的复杂度。另外由于执行域被闭包引用，所以返回了闭包的执行域在函数调用结束后，并没有马上被销毁。如果你在程序在执行过程中产生了大量的闭包，而又忘记或者及时销毁它们，就有可能导致程序内存的剧增。

在一些特定情况下需要注意限制闭包的使用。

### 22.3.1 不适合使用闭包的场合

首先，如果你用来返回闭包的函数是一个非常庞大的函数，而你需要的只是访问这个环境中的一小部分属性，那么你就应该充分衡量一下这么使用的利弊，因为被这个很小的闭包所引用会使得整个调用对象耗费的存储空间不能被及时销毁。


其次，除非你很确定闭包引用被调用时真正访问到的外部执行环境是什么样子，否则最好不要轻易使用闭包。尤其是嵌套使用闭包，因为这样做虽然可能使得程序代码量大大减少，但是极大地增加了程序的逻辑复杂度，因为你如果不能很明确地确定闭包使用时的外部环境是什么样子的，这就意味着当你的程序出现异常情况的时候，排查和修复将会变成一项非常复杂的工作。

## 22.4 函数式编程

函数式编程是一种和面向对象编程对等的程序设计思想，在某些偏于数学形式的模型中，函数式编程拥有比面向对象编程更大的优势。与面向对象相比，函数式编程天生简洁直接，并且有更高的效率。而且函数式编程和面向对象也并不矛盾，它们的结合有利于我们改善系统的代码和优化结构。

### 22.4.1 什么是函数式编程

什么是函数式编程？如果你这么直白地询问，会发现它竟是一个不太容易解释的概念。许多在程序设计领域有着多年经验的老手，也无法很明白地说清楚函数式编程到底在研究些什么。函数式编程对于熟悉过程式程序设计的程序员来说的确是一个陌生的领域，闭包（closure）、延续（continuation）、和柯里化（currying）这些概念看起来是这么的陌生，同我们熟悉的 if、else、while 没有任何的相似之处。尽管函数式编程有着过程式无法比拟的优美的数学原型，但它又是那么的高深莫测，似乎只有拿着博士学位的人才玩得转它。

 这一节有点难，但它并不是掌握 JavaScript 所必需的技能，如果你不想用 JavaScript 来完成那些用 Lisp 来完成活儿，或者不想学函数式编程这种深奥的技巧，你完全可以跳过它们，进入下一章的旅程。

那么回到这个问题，什么是函数式编程？答案很长……

#### 22.4.1.1 函数是第一型

这句话本身该如何理解？什么才是真正的“第一型”？我们看下面的数学概念：

二元方程式  $F(x, y) = 0$ ,  $x, y$  是变量, 把它写成  $y = f(x)$ ,  $x$  是参数,  $y$  是返回值,  $f$  是由  $x$  到  $y$  的映射关系, 被称为函数。如果又有,  $G(x, y, z) = 0$ , 或者记为  $z = g(x, y)$ ,  $g$  是  $x, y$  到  $z$  的映射关系, 也是函数。如果  $g$  的参数  $x, y$  又满足前面的关系  $y = f(x)$ , 那么得到  $z = g(x, y) = g(x, f(x))$ , 这里有两重含义, 一是  $f(x)$  是  $x$  上的函数, 又是函数  $g$  的参数, 二是  $g$  是一个比  $f$  更高阶的函数。

这样我们就用  $z = g(x, f(x))$  来表示方程  $F(x, y) = 0$  和  $G(x, y, z) = 0$  的关联解, 它是一个迭代的函数。我们也可以用另一种形式来表示  $g$ , 记  $z = g(x, y, f)$ , 这样我们将函数  $g$  一般化为一个高阶函数。同前面相比, 后面这种表示方式的好处是, 它是一种更加泛化的模型, 例如  $T(x, y) = 0$  和  $G(x, y, z) = 0$  的关联解, 我们也可以用同样的形式来表示 (只要令  $f=t$ )。在这种支持把问题的解转换成高阶函数迭代的语言体系中, 函数就被称为“第一型”。

JavaScript 中的函数显然是“第一型”。下面就是一个典型的例子:

```
Array.prototype.each = function(closure)
{
    return this.length ? [closure(this.slice(0, 1)).concat(this.slice(1).each(closure)) : []];
}
```

这真是段神奇的魔法代码, 它充分发挥了函数式的魅力, 在整个代码中只有函数 (function) 和符号 (Symbol)。它形式简洁并且威力无穷。

`[1,2,3,4].each(function(x){return x * 2})` 得到 `[2,4,6,8]`, 而 `[1,2,3,4].each(function(x){return x-1})` 得到 `[0,1,2,3]`。



函数式和面向对象本质都是“道法自然”。如果说, 面向对象是一种真实世界的模拟的话, 那么函数式就是数学世界的模拟, 从某种意义上说, 它的抽象程度比面向对象更高, 因为数学系统本来就具有自然界所无法比拟的抽象性。

#### 22.4.1.2 闭包与函数式编程:

闭包, 在前面的章节中我们已经解释过了, 它对于函数式编程非常重要。它最大的特点是不需要通过传递变量 (符号) 的方式就可以从内层直接访问外层的环境, 这为多重嵌套下的函数式程序带来了极大的便利性, 例如下面这段代码:

```
JavaScript:(function outerFun(x){return function innerFun(y){return x * y}})(2)(3);//innerFun 访问外层的 x
```

#### 22.4.1.3 科里化 (Currying)

什么是 Currying? 它是一个有趣的概念。还是从数学开始: 我们说, 考虑一个三维空间方程  $F(x, y, z) = 0$ , 如果我们限定  $z = 0$ , 于是得到  $F(x, y, 0) = 0$  记为  $F'(x, y)$ 。这里  $F'$  显然是一个新的方程式, 它代表三维空间曲线  $F(x, y, z)$  在  $z = 0$  平面上的两维投影。记  $y = f(x, z)$ , 令  $z = 0$ , 得到  $y = f(x, 0)$ , 记为  $y = f'(x)$ , 我们说函数  $f'$  是  $f$  的一个 Currying 解。

下面给出了 JavaScript 的 Currying 的例子:

例 22.7 Currying (科里化)

```
<html>
<head>
    <title>Example-22.7 Currying</title>
</head>
<body>
<script>
<!--
//这是一个计算 x+y 的函数, 但是它和常规函数的不同之处在于
//它是被 Currying 的
function add(x, y)
{
```

```

//当 x,y 都有值得时候, 计算并返回 x+y 的值
if(x!=null && y!=null) return x + y;
//否则, 若 x 有值 y 没有值
else if(x!=null && y==null) return function(y)
{
    //返回一个等待 y 参数进行后续计算的闭包
    return x + y;
}
//若 x 没有值 y 有值
else if(x==null && y!=null) return function(x)
{
    //返回一个等待 x 参数进行后续计算的闭包
    return x + y;
}
}
//计算 add(3,4)的值, 得到 3+4 的结果 7
var a = add(3, 4);
//计算 add(2)的值, 得到一个相当于求 2+y 的函数
var b = add(2);
//继续传入 y 的值 10, 得到 2+10 的结果 12
var c = b(10);
-->
</script>
</body>
</html>

```

上面的例子中, `b=add(2)`得到的是一个 `add()`的 Currying 函数, 它是当 `x = 2` 时, 关于参数 `y` 的函数, 注意到上面也用到了闭包的特性。

有趣的是, 我们可以给出任意函数一般化 Currying 的形式, 例如:

```

function Foo(x, y, z, w)
{
    var args = arguments;

    //如果函数的形参个数小于实参个数
    if(Foo.length < args.length)
        //返回一个闭包
        return function()
        {
            //这个闭包用之前已经输入的参数和此次输入的参数构成参数调用 Foo 函数自身
            return
                args.callee.apply(Array.apply([], args)
                    .concat(Array.apply([], arguments)));
        }
    else
        //否则对函数求值
        return x + y - z * w;
}

```

### 22.4.1.4 延迟求值和延续

惰性（或延迟）求值是一项有趣的技术，考虑下面的代码片断：

```
var s1 = somewhatLongOperation1();
var s2 = somewhatLongOperation2();
var s3 = concatenate(s1, s2);
```

在一个命令式语言中求值顺序是确定的，因为每个函数都有可能会变更或依赖于外部状态，所以就必须要有序的执行这些函数：首先是

`somewhatLongOperation1`，然后 `somewhatLongOperation2`，最后 `concatenate`，在函数式语言里就不尽然了。

只要确保没有函数修改或依赖于全局变量，`somewhatLongOperation1` 和 `somewhatLongOperation2` 可以被并行执行。但是如果不想同时运行这两个函数，还有必要保证有序的执行他们呢？答案是不。我们只在其他函数依赖于 `s1` 和 `s2` 时才需要执行这两个函数。我们甚至在 `concatenate` 调用之前都不必执行他们——可以把他们的求值延迟到 `concatenate` 函数内实际用到他们的位置。如果用一个带有条件分支的函数替换 `concatenate` 并且只用了两个参数中的一个，另一个参数就永远没有必要被求值。在函数式语言中，不确保一切都（完全）按顺序执行，因为函数式只在必要时才会对其求值。

例如，在 JavaScript 中，我们可能这么写：

```
function concatenate(s1, s2)
{
    if(cond1) s1();
    s2();
    .....
}
var s3 = concatenate(somewhatLongOperation1,somewhatLongOperation2);
```

假如 `cond1` 的条件不满足，那么 `somewhatLongOperation1` 就不需要被执行，这样从一定程度上强化了程序逻辑的优化潜力。

一个更为有趣的话题是，函数式编程可以定义无穷数据结构，对严格语言来说实现这个要复杂的多。考虑一个 Fibonacci 数列，显然我们无法在有限的时间内计算出或在有限的内存里保存一个无穷列表。在严格语言如 Java 中，只能定义一个能返回 Fibonacci 数列中特定成员的 Fibonacci 函数，在函数式语言中，我们对其进一步抽象并定义一个关于 Fibonacci 数的无穷列表，因为作为一个惰性的语言，只有列表中实际被用到的部分才会被求值。这使得可以抽象出很多问题并从一个更高的层次重新审视他们。（例如，我们可以在一个无穷列表上使用表处理函数）。

下面是一个例子：

例 22.8 Fibonacci 无穷数列

```
<html>
<head>
  <title>Example-22.8 Fibonacci 无穷数列</title>
</head>
<body>
<script>
<!--
//这个函数我们在例 6.1（2）已经见过了，在这里再次举出来，以体验 functional 的魅力
function dwn(s)
{
  document.write(s + "<br/>");
}
```

```

//“无穷”的斐波纳契数据结构
function Fib(n, x, y)
{
    //这里借参数 x,y 来保留前面的计算结果，即斐波数当前数列到 n 的最后两个数值
    //在实际调用中通常并不用到 x、y 这两个参数
    var a = x || 1;
    var b = y || 1;
    if(n == 0) b = a;

    var t;

    //计算斐波数的算法
    for(var i = 2; i <= n + 1; i++)
    {
        t = b;
        b = a + b;
        a = t;
    }

    var ret = function(n, x, y){
        //构造一个闭包，这个闭包本身包含一个以新起点计算 Fib 值的函数
        x = x || a;
        y = y || b;
        return Fib(n, x, y);
    }

    //重写 valueOf 和 toString，这样在表达式中可以直接对返回的斐波函数自动求值
    //在第五部分我们还会详细讨论到这种用法
    ret.valueOf = ret.toString = function()
    {
        return a;
    }
    return ret;
}
var f6 = Fib(6);    //奥妙在这里，f6 是一个新起点的斐波数列函数
dwn(f6);
dwn(f6(2));
-->
</script>
</body>
</html>

```

执行结果如图 22.3 所示：

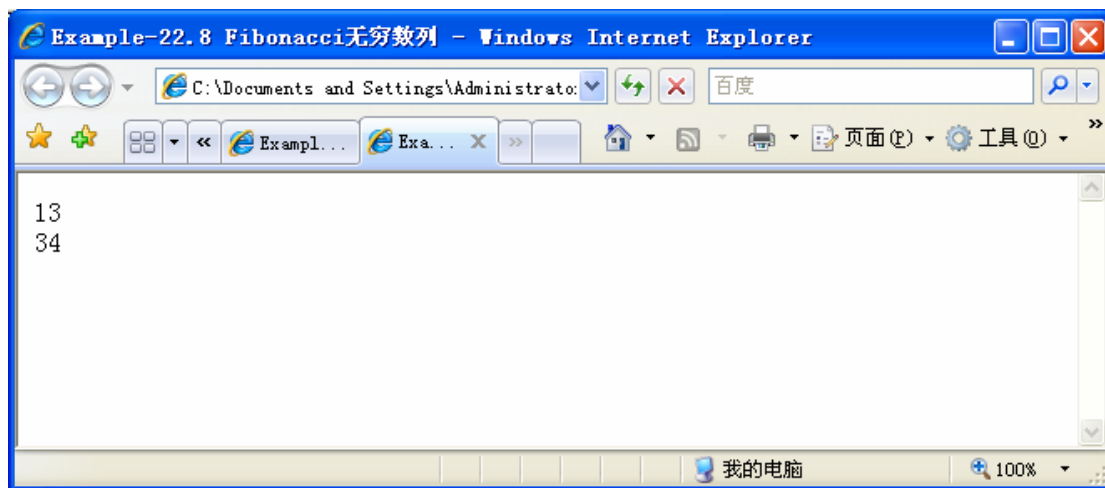


图 22.3 无穷数据结构

上面这个函数的好处是，求出  $fn$  之后，要计算  $fm$  只需要计算  $fn(m-n)$  就行了，而且它几乎不需要额外的存储空间。

“延续”（Continuations）是为了解决延迟求值带来的一个不小的副作用。我们说，在精确的函数式程序结构中，延迟求值的结果让我们很难描述函数 `somewhatLongOperation1` 和 `somewhatLongOperation2` 之间的依赖关系。如果 `somewhatLongOperation1` 必须先于 `somewhatLongOperation2` 被执行那么我们要么无法控制这种必然性（可能会导致潜在的程序逻辑错误），要么会用额外的约定破坏函数式的完备性，幸运的是，函数式的形式可以描述这种依赖关系：

```
var s3 = concatenate(somewhatLongOperation1(somewhatLongOperation2));
```

这，就是“延续”的含义。

或许，对于 Continuations，我们应该找到一个更加合适的中文词汇来翻译，不过其实它的含义并不复杂。我们说，在函数式模型中，子系统  $s1$  和  $s2$  没有一种固定的次序关系，而是取决于实际的调用，那么如何来描述系统中的依赖关系呢？答案很简单，当你不能确定  $s1$  的输出是否在  $s2$  之前时，要想把  $s1$  的输出作为  $s2$  的输入，那么你可以把  $s1$  系统本身作为  $s2$  的输入。

#### 22.4.2 函数式编程、公式化与数学模型

同面向对象的“道法自然”相比，函数式更贴近于数学，它是数学王国的代言人。而数学本身，是对自然界的一种“强力的抽象”，所以，一般我们认为，函数式编程表现出比面向对象更强的“抽象性”。

我们说数学是一种“先验”科学，它对自然界的抽象是“与生俱来”的，目前已知的任何自然规律，都近乎完美地服从于数学定律。有意思的是，古往今来，数学定律的发现，往往要先于自然规律地发现。这样看起来似乎违背原离，不像是数学替自然规律说话，倒有点像是自然规律依附于数学王国了，不过，这正是数学魅力的所在。

我们说函数式是公式化的语言，它具有明显的数学特征。例如，在前面的例子中，我们已经见到过，JavaScript 里可以这么定义抛物线方程（族）：

例 22.9 抛物线方程

```
<html>
<head>
  <title>Example-22.9 抛物线方程</title>
</head>
<body>
<script>
<!--
function parabola(a, b, c) //构造抛物线方程
```



```

    {
        return function(x)
        {
            return a * x * x + b * x + c;
        }
    }

var p1 = parabola(2,3,4); //抛物线  $y = 2*x^2 + 3*x + 4$ 
alert(p1(15));
-->
</script>
</body>
</html>

```

仔细研究它，你会发现，这种函数定义方式，同数学语言的描述方式几乎完全一致！

这种数学形式上的一致性，在传统的过程式语言中，几乎是无法想象的。如果不利用 JavaScript 的函数式特性，要定义和调用抛物线方程，只能以下面这种丑陋的方式：

```

function parabola(a, b, c, x)
{
    return a * x * x + b * x + c;
}
var y = parabola(2, 3, 4, 15);
如果用面向对象来表示，则问题又有一点点差别：
function Parabola(a, b, c)
{
    this.evaluate = function(x)
    {
        return a * x * x + b * x + c;
    }
}
var p1 = new Parabola(2,3,4); //抛物线  $y = 2*x^2 + 3*x + 4$ 
alert(p1.evaluate());

```

面向对象把抛物线当作了“对象”，从自然界的角度来讲，这没有什么问题，然而从数学的角度来讲，它把问题复杂化了。抛物线本来就是一个方程（函数），不需要再定义成一个对象，然后用蹩脚的 `evaluate()` 来进行求值。