

动态函数式语言精髓 与编程实践

(JavaScript 版)

周爱民 著

前言

1.1. 语言

我相信，在这个世界上无时不刻地在产生着新的语言。例如我刚才叫住我脚下的这只小猫，我喊了一声“嗨”，它就停下来望着我。如同我与猫一样，你可能正在用另外一种语言——声音的，或者符号的——与另外一种个体进行着交流。

无论这种个体是否是一个鲜活的生命，还是一堆电器元件，如果你们需要交流，那么唯一的方式就是创生一种语言，或者使用两种个体既有的语言之一进行交流。

我们一边在创生着与计算机交流的语言，一边也不得不用既有的语言与之交流。这就是现状。这与此时我跟脚下的猫，或者数千年前的铸剑师与一柄剑之间交流时的情状，是完全一样的。

语言是一种交流的工具，这约定了语言的“工具”本质，以及“交流”的功用。“工具”的选择只在于“功用”是否能达到，而不在于工具是什么。

在数千年之前，远古祭师手中的神杖就是他们与神交流的工具。祭师让世人相信他们敬畏的是神，而世人只需要相信那柄神杖。于是，假如祭师不小心丢掉了神杖，就可以堂而皇之地再做一根。甚至，他们可以随时将旧的换成更新或更旧的神杖，只要他们宣称这是一根更有利于通神的杖。对此，世人往往做出迷惑的表情，或者欢欣鼓舞的情状。今天，这种表情或情状一样地出现在大多数程序员的脸上，出现在他们听闻到新计算机语言被创生的时刻。

神杖换了，祭师还是祭师，世人还是会把头叩得山响。祭师掌握了与神交流的方法(如果真如同他们自己说的那样的话)，而世人只看见了神杖。

所以，泛义的工具是文明的基础，而确指的工具却是愚人的器物。

计算机语言有很多种分类方法，例如高级语言或者低级语言。其中一种分

类方法，就是“静态语言”和“动态语言”——事物就是如此，如果用一对绝对反义的词来分类，就相当于概含了事物的全体。当然，按照中国人中庸平和的观点，以及保守人士对未知可能性的假设，我们还可以设定一种中间态：半动态语言。你当然也可以叫它半静态语言，这个随便你。

所以，我们现在是在讨论一种很泛义的计算机语言工具。至少在眼下，他(在分类概念中)概含了计算机语言的二分之一。当然，限于我自身的能力，我只能讨论一种确指的工具，例如 JavaScript。但我希望你看到的是计算机编程方法的基础，而不是某种愚人的器物。JavaScript 的生命力可能足够顽强，我假定它比 C 还顽强，甚至比你的生命都顽强。但他只是愚人的器物，因此反过来说：它能不能长久地存在都并不重要，重要的是它能不能作为这“二分之一的泛义”来供我们讨论。

1.2. 分类法

新打开一副扑克牌，我们总看到它被整齐的排在那里，从 A 到 K 以及大小王。接下来，我们将它一分为二，然后交叉在一起；再分开，再交叉……完成了洗牌之后，我们便可以拿它新开一局了。但是你是否注意到在这个过程中：牌局的复杂性，其实不是由“分开”这个动作导致的，而是由“交叉”这个动作导致的。

所以分类法本身并不会导致复杂性。就如同一幅新牌只有四套 A~K，我们可以按十三牌面来分类，也可以按四种花色来分类。当你从牌盒里把他们拿出来的时候，无论他们是以哪种方式分类的，这幅牌都不混乱。混乱的起因，在于你交叉了这些分类。

同样的道理，如果世界上只有动态、静态两种语言，或者真有半动态语言而你又有明确的“分类法”，那么开发人员将会迎来清醒明朗的每一天：我们再也不需要花更多的时间却学习更多的古怪语言了。

然而，第一个问题便来自于分类本身。因为“非此即彼”的分类必然导致特性的缺失——如果没有这样“非此即彼”的标准，就不可能形成分类。而特性的缺失又正是开发人员所不能容忍的。我们一方面吃着碗里，一方面念着锅里。即使锅里漂起来的那片菜叶未见得有碗里的肉好吃，我们也一定要捞起来尝尝。而且大多数时候，由于我们吃肉吃腻了嘴，因此会觉得那片菜叶味道其实更好。

所以首先是我们的个性，决定了我们做不成绝对的素食者或肉食者。当然，更有一些人说我们的确需要一个新的东西来使得我们更加强健。但不幸的是，大多数提出这种需求的人，都在寻求杀死人狼的纯质银弹或者混合毒剂。无论如何，他们要么相信总有一种事物是完美武器，或者更多的特性放在一些就变成了魔力的来源。

我不偏向两种方法之任一。但是我显然看到了这样的结果，前者是我们在不断地创造并特化某种特性，后者是我们在不断地混合种种特性。

更进一步的说，前者在产生新的分类法以试图让武器变得完美，后者则通过混乱不同的分类法，以期望通过突变而产生奇迹。

二者相同之处，都在于需要更多的分类法。

函数式语言就是来源于另外的一种分类法。不过要说明的是，这种分类法是计算机语言的原力之一。基本上来说，这种分类法在电子计算机的实体出现以前，就已经诞生了。这种分类法的基础是“运算产生结果，还是运算影响结果”。前一种思想产生了函数式语言(如 Lisp)所在的“说明式语言”这一分类，后者则产生了我们现在常见的 C、C++等语言所在的“命令式语言”这一分类。

然而我们已经说过，人们需要更多的分类的目的，是要么找到类似银弹的完美武器，要么找到混合毒剂。所以一方面很多人宣称“函数式是语言的未来”，另一方面也有很多人把这种分类法与前一种分类法混在一起，于是变成了我们这本书所说的“动态函数式语言”——当然，毋庸置疑的是：还会有更多的混合法产生。因为保罗·格雷厄姆(Paul Graham)已经做过这样的总结：

二十年来，开发新编程语言的一个流行的秘诀是：取 C 语言的计算模式，逐渐地往上加 Lisp 模式的特性，例如运行时类型和无用单元收集。

保罗·格雷厄姆是硅谷著名的程序员之一，在 1995 年他和 Robert Morris 开发了第一个基于 Web 的应用程序 Via web，该项目在 1998 年被雅虎(Yahoo)收购。在 2002 年，他设计了一种垃圾邮件过滤器算法。他同时还是计算机程序语言 Arc 的设计者，写了多本关于程序语言以及创业方面的书籍。在 2005 年他和 3 位合伙人成立了 Y Combinator 创业投资基金公司，专注于早期阶段的种子投资。

然而这毕竟只是“创生一种新语言”的魔法。那么，到底有没有让我们在这浩如烟海的语言家族中，找到学习方法的魔法呢？

我的答案是：看清语言的本质，而不是试图学会一门语言。当然，这看起来非常概念化。甚至有人说我可能是从某本教材中抄来的，另外一些人又说我试图在这本书里宣讲类似于我那本《大道至简》里的老庄学说。

其实这很冤枉。我想表达的意思不过是：如果你想把一幅牌理顺，最好的法子，是回到他的分类法上，要么从 A 到 K 整理，要么按四个花色整理。毕竟，两种或者更多的分类法作用于同一事物，只会事物混淆而不是弄得更清楚。

因此，本书从语言特性出发，把动态与静态，函数式与非函数式的语言特性分列出来。先讲述每种特性，然后再讨论如何去使用(例如交叉)他们。

1.3. 漏掉了点儿什么

你会发现一个问题：无论是四种花色，还是 A 到 K 的牌面，我们都漏掉了两张王。是的，这正是问题之所在：这两种分类法是以特性为主的，而不是按照我们前面说的“绝对一分为二的方法”。因为如果用那样的方法，那么一副牌应该分为“王牌”和“非王牌”。

1.4. 特性

无论哪种语言(或其它工具)都有其独特的特性，以及借鉴自其它语言的特性。有些语言通体没有“独特特性”，只是另外一种语言的副本，这更多的时候是为了“满足一些人使用语言的习惯”。还有一些语言则基本上全是独特的特性，这可能导致语言本身不实用，但却是其它语言的思想库。

我们已经讨论过这一切的来源。

对于 JavaScript 来说，除了动态语言的基本特性之外，它还有着与其创生时代背景密切相关的一些语言特性。直到如今，JavaScript 的创建者还在小心翼翼地增补着它的语言特性。在特性集的设定方面，JavaScript 一直以来都是一个典范。

由于 JavaScript 轻量的、简洁的、直指语言本实的特性集设计，它成为解剖动态语言的有效工具。这个特性集包括：

1. 一套原型继承的、基于对象的语言特性和对象系统
2. 一套支持自动转换的弱类型系统
3. 语言、类型相关的基础函数/方法
4. 动态语言与函数式语言的基本特性

需要被强调的是，JavaScript 非常苛刻地保证这些特性是相应语言领域中的最小特性集（或称之为“语言原子”），这些特性在 JavaScript 中相互混合，通过交错与补充而构成了丰富的、属于 JavaScript 自身的语言特性。

本书的主要努力之一，就是分解出这些语言原子，并重现将它们混合在一起的过程与方法。通过从复杂性到单一语言特性的还原过程，让读者了解到语言的本实，以及“层出不穷的语言特性”背后的真相。

1.5. 合理性

本书之所以有趣，一个重要的原因是它讨论问题的角度，以及讨论这些问题的目标与其它书籍全然不同。我在这本书中试图以一种多范型语言（当然，我们已经知道它叫 JavaScript），来讨论不同语言范型之间的特性；用分解特性的方法来讨论它们为什么、以及如何被结合在一起，变成了一种语言。

因为讨论的角度与目标的不同，本书中你会看到一些全然不同的观点。你会发现这些观点可能在其它 JavaScript 书中，或者其它专论语言的书中都没有出现过。于是，你会质疑：这些观点合理吗？

一个创生这种语言的人都没有提出的观点，或者其它语言大师在同类范型的语言中都没有提出过的观点，的确是值得置疑的。我高兴看到这样的置疑，并欢迎读者带着疑问将本书读下去。但你不应当先给出一个结论说：这不合理，因为没人这样说过，甚至语言的创生者可能都没有这样想过。

要人们抛弃一种观点来接受另一种，的确非常艰难。但我所认为的合理性并不是一个人或者某些人所讲述的“论点”。声音越大、重复得越多的观点未见得一定合理，没被这样大声地宣讲或重复的观点未见得就不合理。我经常的问题是：存在的合理性。

如果一种事物“坚持不懈”地存在了十年或者一个时代，那么它必然有存在的理由，也有被人接受的理由。这些理由本身可能荒诞（例如是因为商业而非技术原因才有了 JavaScript 这个名字，而它原本是叫 Mocha），另一些也可能

很接近本质（例如 lisp 这种语言居然还活着）。而我则试图去揭示 JavaScript 语言“被创造成这样”却仍然还活着的一些本质原因。这些原因是否因为没有出自某个人或者某些人的观点而显得“不合理”，则正是我要忽略的。

因为如果我不忽略它，则我永远只能接受那些“看起来貌似合理”的观点，而这本书也就变成了重复之作。

1.6. 技巧

技巧是“技术的取巧之处”，所以根本上来说，技巧也是技术的一部分。很多人(也包括我)反对技巧的使用，是因为难于控制，并且容易破坏代码的可读性。

哪种情况下代码是需要“易于控制”和“可读性强”的呢？通常，我们认为在较大型的工程下需要“更好的控制代码”；在更多人共同开发的项目代码上要求“更好的可读性”。然而，反过来说，在一些更小型的、不需要更多人参与的项目中，“适度的”使用技巧是否就是可以接受的呢？

这取决于“需要、能够”维护这个代码的人对技巧的理解。这包括：

- 👉 技巧是否是语言特性支持的，还是仅特定版本所支持或根本就是 BUG；
- 👉 技巧是否是唯一可行的选择，有没有不需要技巧的实现；
- 👉 技巧是为达到目的、实现功能，而不是为了表现技巧而出现在代码中的。

即使如此，我仍然希望每一个技巧的使用都有说明，甚至示例。如果维护代码的人不能理解这个技巧，那么连代码本身都失去了价值，更何论技巧存在于这份代码中的意义呢？所以本书中的例子确要用到许多“技巧”，但我一方面希望读者能明白，这是语言/框架内核实现过程中必须的，另一方面也希望读者能从这些技巧中学习到它原本的技术/理论，以及活用的方法。

1.7. 本书讲的不是技巧

尽管我对技巧持以一种理解和宽容的态度，但我却要声明，我在本书中讲述的并不是技巧。

这样说并不是要为我的“出尔反尔”争回一点面子。事实上，对很多人来说，本书在讲述一个完全不同的语言类型。在这种类型的语言中，本书中讲述的一切，都只不过是“正常的方法”。然而在其它类型的一些语言中，这些看

起来就成了技巧。例如在 JavaScript 中要改变一个对象方法指向的代码非常容易，并且是语言本身赋予的能力；而在 Delphi/C++ 中，却成了“破坏面向对象设计”的非正常手段——以至于在编译级与系统级受到层层阻挠。

你最好能改变一下思维来看待本书中讲述的方法之精髓。无论它对你产生多大的冲击，你应该先想到的是这些方法的价值，而不是它对于“你所认为的传统”的挑战——事实上，这些方法，在另一些“同样传统”的语言类型中，已经存在了足够长久的时间，如同“方法”之与“对象”一样，原本就是那样“(至少看起来)自然而然”地存在于它所在的语言体系之中。

语言特性的价值依赖于环境而得彰显。横行的螃蟹看起来古怪，但据说那是为了适应一次地磁反转。螃蟹的成功在于适应了一次反转，失败(我们是说导致它这样难看)之处，也在于未能又一次反转回来。

1.8. 本书讲什么

你当然可以置疑：为什么要有这样的一本书？是的，这的确是一个很好的问题。

首先，这本书并不讲浏览器 (Browser，例如 Internet Explorer)。这可能令人沮丧。但的确如此。尽管在很多人看来，JavaScript 就是为浏览器而准备的一种轻量的语言，并认为它离开了 DOM、HTML、CSS 就没有意义。在同样的“看法”之下，国内、外的书籍一提及 JavaScript，无不讲述浏览器上如何开发，更多的是从“如何验证一个输入框值的有效性”讲起。

是的，最初我也是这样认为的。因为本书原来就是出自我在写《B 端开发》这本书的过程之中。事实上，《B 端开发》是一本讲述“在浏览器 (Browser) 上如何用 JavaScript”来开发的书。然而，《B 端开发》写到近百页就放下了，因为我觉得应该写一本专门讲 JavaScript 的书，这更重要。

所以，现在你将要看到的这本书就与浏览器无关。在本书中，我将会把 JavaScript 提升到与 Java、C# 或 Delphi 一样的高度，来讲述它的语言实现与扩展。由于在本书中的最后一部分内容中，讲述了名为“Qomo”的完整的 JavaScript 框架，因此如果你需要在浏览器上构建大型应用(例如基于 Ajax 的工程)，那么你可以从 Qomo 中得益良多。Qomo 可以成倍地提高你的开发工效，有利于你实现更多的、更有价值的应用特性。但是，嗯，本书不讲浏览器，不讲 WEB，也并不讲“通常概念下的” Ajax。

JavaScript 是一门语言，有思想的、有核心的、有灵魂的语言。如果你不意识到这一点，那么你可能永远都只能拿它来做那个“检测值是否有效的输入框”的代码。

本书讲述 JavaScript 的思想、核心、灵魂，以及如何去丰富它的血肉。主要包括三个部分：

- 👉 动态、函数式语言，以及其它语言特性在 JavaScript 的表现与应用；
- 👉 如何用动态函数式语言的特性来扩展 JavaScript 的语言特性与框架；
- 👉 如何将 JavaScript 引擎整合到其它高级语言的开发过程中。

另外，这本书里有很多示例的代码是大同小异的。但对于 JavaScript 这种语言来说，任何一点点看起来微不足道的变化，可能都源自对某些语言特性的深刻理解。另一方面，一段相同的代码，在很多时候也可以从不同的语言类型或特性来审视。

1.9. 本书的适读者

我试图给这本书找到一个适合的读者群体，但我发现很难。因为通常的定义是低级、中级与高级，然而不同的用户对自己的“等级”的定义标准并不一样。在这其中，有“十年学会编程”的谦逊者，也有“三天学会某某语言”的速成家。所以，我认为这样定位读者的方式是徒劳的。

如果你想知道自己是否适合读这本书，建议你先看一下目录，然后试读一二章节，可以选一些在你的知识库中看来很新鲜的，以及一些你自认为已经非常了解的。通过对比，你应该知道这本书会给你带来什么。

不过我需要强调一些东西。这本书不是一本让你“学会某某语言”的书，也不是一本让初学者“学会编程”的书。阅读本书，你至少应该有一点编程经验（例如半年至一年），而且要摒弃某些偏见（例如 C 语言天下无敌或 JavaScript 是新手玩具）。

最后，你至少要有一点耐心与时间。

第一部分 语言基础

JavaScript 是怎样的一种语言

- * JavaScript 的语法

- 通过语法给人们概念上的认识，表明“它是怎样”。基本效果是达到能编程和能阅读。

第二部分 语言特性及基本应用

- 从语言的分类学和各分类的进化出发，讲述 JS 的历史与语言特性的成因。

- * JavaScript 的非函数式语言特性

- * JavaScript 的函数式语言特性

- * JavaScript 的动态语言特性

第三部分 最佳实践

Qomo 项目简介

Qomo 的内核框架设计、实现及应用

一般性的动态函数式语言技巧

风格样张(排版)

但这个 typeof()中的括弧。

```
do
    statement
while (expression);
```

```
// 特殊代码片段 (一般是没有上下文关系的、单独的说明某种特定语法、语义或技巧)
foo(); // this throws an error
if (1)
    function foo() {}
```

```
// 大段代码示例
// 示例 2: 浏览器环境中使用的顶层对象是 window
<script language="JavaScript">
```

类型	直接量(v1)	包装类(v2)	特性			
			typeof <value>		<value> instanceof <class>	
			v1	v2	v1	v2
undefined	undefined		Y			

👉 赋值语句：使用“等号(=)”赋值运算符

👉 赋值语句：使用“等号(=)”赋值运算符

含义	详述	注
JavaScript 脚本语言	一种语言的统称，由 ECMAScript 262 规范。	*
浏览器 JavaScript	在 Netscape 和 Mozilla 系列中，浏览器 JavaScript 被称为 Client-Side JavaScript。这包括 DOM、BOM 模型等在内的对象体系。	

*注：JavaScript 的方言是指 ECMAScript 262 的某一种具体的实现。这种语言其实被广泛的用于浏览器环境、WEB 服务器、操作系统和应用软件。但不同的环境与语言的实现版本之间都存在较大的差异。

```
1 my_label: {
2     if (str && str.length < 10) {
3         // 错误一: 在标签化语句中用使用 break 而不带 label
4         break;
5     }
6     str = str.substr(str.length-10);
```

前言.....	3
1.1. 语言.....	3
1.2. 分类法.....	4
1.3. 漏掉了点儿什么.....	6
1.4. 特性.....	6
1.5. 合理性.....	7
1.6. 技巧.....	8
1.7. 本书讲的不是技巧.....	8
1.8. 本书讲什么.....	9
1.9. 本书的适读者.....	10
第一章 十年 JavaScript.....	24
1.1. 第一段在网页中的代码.....	24
1.1.1. 新鲜的玩意儿.....	24
1.1.2. 第一段在网页中的代码.....	25
1.1.3. 最初的价值.....	26
1.2. 用 JavaScript 来写浏览器上的应用.....	27
1.2.1. 我要做一个聊天室.....	27
1.2.2. Flash 的一席之地.....	29
1.2.3. RWC 与 RIA 之争.....	31
1.3. 没有框架与库的语言能怎样发展呢?	33
1.3.1. 做一个框架.....	33
1.3.2. 重写框架的语言层.....	36
1.3.3. 富浏览器端开发(RWC)与 Ajax.....	37
1.4. 为 JavaScript 正名.....	39
1.4.1. JavaScript.....	39
1.4.2. Core JavaScript.....	40
1.4.3. SpiderMonkey JavaScript.....	41
1.4.4. ECMAScript.....	41
1.4.5. JScript.....	41
1.5. [XXX] JavaScript 的应用环境.....	43
1.5.1. 宿主环境(host environment).....	43
1.5.2. 外壳程序(shell).....	46
1.5.3. 运行期环境(runtime).....	46
1.6. [XXX] 回顾.....	47
第二章 JavaScript 的语法.....	49
1.1. 语法综述.....	49
1.1.1. 识别语法错误与运行错误.....	50
1.2. JavaScript 的语法: 变量声明.....	51
1.2.1. 变量的数据类型.....	52
1.2.1.1. 值类型与引用类型.....	53
1.2.2. 变量声明.....	54

1.2.3.	变量声明中的一般性问题.....	55
1.2.3.1.	字符串直接量、转义符.....	56
1.2.3.2.	数值直接量.....	58
1.2.3.3.	函数直接量声明.....	59
1.2.3.4.	正则表达式的常见问题.....	59
1.2.3.5.	数组的常见问题.....	61
1.3.	JavaScript 的语法：表达式运算.....	62
1.3.1.	一般表达式运算.....	64
1.3.2.	逻辑运算.....	65
1.3.3.	字符串运算.....	66
1.3.4.	比较运算.....	66
1.3.4.1.	等值检测.....	66
1.3.4.2.	序列检测.....	68
1.3.5.	赋值运算.....	71
1.3.6.	函数调用.....	72
1.3.7.	特殊作用的运算符.....	73
1.3.8.	运算优先级.....	74
1.4.	JavaScript 的语法：语句.....	76
1.4.1.	表达式语句.....	78
1.4.1.1.	一般表达式语句.....	79
1.4.1.2.	赋值语句与变量声明语句.....	81
1.4.1.3.	函数调用语句.....	83
1.4.2.	分支语句.....	88
1.4.2.1.	条件分支语句(if 语句).....	88
1.4.2.2.	多重分支语句(switch 语句).....	89
1.4.3.	循环语句.....	93
1.4.4.	流程控制：一般子句.....	94
1.4.4.1.	标签声明.....	94
1.4.4.2.	break 子句.....	96
1.4.4.3.	continue 子句.....	98
1.4.4.4.	return 子句.....	100
1.4.5.	流程控制：异常.....	101
1.5.	面向对象编程的语法概要.....	103
1.5.1.	对象直接量声明与实例创建.....	104
1.5.1.1.	使用构造器创建对象实例.....	104
1.5.1.2.	对象直接量声明.....	107
1.5.2.	对象成员列举、存取和删除.....	109
1.5.3.	属性存取与方法调用.....	113
1.5.4.	对象及其成员的检查.....	114
1.5.5.	可列举性.....	118
1.5.6.	缺省对象的指定.....	120
1.6.	运算符的二义性.....	120

1.6.1.	加号“+”的二义性.....	121
1.6.2.	括号“()”的二义性.....	123
1.6.3.	冒号“:”与标签的二义性.....	126
1.6.4.	大括号“{ }”的二义性.....	127
1.6.5.	逗号“,”的二义性.....	131
1.6.6.	方括号“[]”的二义性.....	134
第三章	JavaScript 的非函数式语言特性.....	140
1.1.	概述.....	140
1.1.1.	命令式语言与结构化编程.....	140
1.1.2.	结构化的疑难.....	142
1.1.3.	“面向对象语言”是突破吗?	146
1.1.4.	更高层次的抽象: 接口.....	149
1.1.5.	再论语言的分类.....	151
1.1.6.	JavaScript 的命令式语言渊源.....	153
1.2.	基本语法的结构化含义.....	154
1.2.1.	基本逻辑与代码分块.....	155
1.2.2.	模块化的层次: 语法作用域.....	158
1.2.2.1.	主要的语法作用域及其效果.....	159
1.2.2.2.	语法作用域之间的相关性.....	163
1.2.3.	执行流程及其变更.....	165
1.2.3.1.	级别 2: “break <label>”等语法.....	166
1.2.3.2.	级别 3: return 子句.....	170
1.2.3.3.	级别 4: throw 语句.....	172
1.2.3.4.	执行流程变更的内涵.....	173
1.2.4.	模块化的效果: 变量作用域.....	176
1.2.4.1.	级别 1: 表达式.....	177
1.2.4.2.	级别 2: 语句.....	178
1.2.4.3.	级别 3: 函数(局部)	180
1.2.4.4.	级别 4: 全局.....	181
1.2.4.5.	变量作用域中的次序问题.....	183
1.2.4.6.	变量作用域与变量的生存周期.....	185
1.2.5.	语句的副作用.....	186
1.3.	[XXX]基本数据类型的结构化含义.....	188
1.4.	JavaScript 中的原型继承.....	189
1.4.1.	空对象(null)与空的对象.....	189
1.4.2.	原型继承的基本性质.....	191
1.4.3.	空的对象是所有对象的基础.....	191
1.4.4.	构造复制? 写时复制? 还是读遍历?	192
1.4.5.	构造过程: 从函数到构造器.....	194
1.4.6.	预定义属性与方法.....	196
1.4.7.	原型链的维护.....	197
1.4.7.1.	两个原型链.....	197

1.4.7.2.	constructor 属性的维护	198
1.4.7.3.	内部原型链的作用	203
1.4.8.	原型继承的实质	204
1.4.8.1.	原型修改	204
1.4.8.2.	原型继承	205
1.4.8.3.	原型继承的实质：从无到有	206
1.4.8.4.	如何理解“继承来的成员”？	207
1.5.	JavaScript 的对象系统	209
1.5.1.	封装	209
1.5.2.	多态	211
1.5.3.	事件	214
1.5.4.	类抄写？或原型继承？	216
1.5.4.1.	类抄写	217
1.5.4.2.	原型继承	220
1.5.4.3.	如何选择继承的方式	221
1.5.5.	JavaScript 中的对象（构造器）	222
1.5.6.	不能通过继承得到的效果	224
1.6.	[XXX]综述	226
第四章	JavaScript 的函数式语言特性	228
1.1.	概述	228
1.1.1.	从代码风格说起	228
1.1.2.	为什么常见的语言不赞同连续求值	229
1.1.3.	函数式语言的渊源	231
1.2.	函数式语言中的函数	233
1.2.1.	函数是运算元	233
1.2.2.	在函数内保存数据	234
1.2.3.	函数内的运算对函数外无副作用	236
1.3.	从运算式语言到函数式语言	236
1.3.1.	JavaScript 中的几种连续运算	237
1.3.1.1.	连续赋值	237
1.3.1.2.	三元表达式的连用	238
1.3.1.3.	一些运算连用	240
1.3.1.4.	函数与方法的调用	241
1.3.2.	运算式语言	242
1.3.2.1.	运算的实质，是值运算	243
1.3.2.2.	有趣的运算：在 Internet Explorer 和 J2EE 中	244
1.3.3.	如何消灭掉语句	248
1.3.3.1.	通过表达式消灭分支语句	248
1.3.3.2.	通过函数递归消灭循环语句	250
1.3.3.3.	其它可以被消灭的语句	251
1.4.	函数：对运算式语言的补充和组织	252
1.4.1.	函数是必要的补充	252

1.4.2.	函数是代码的组织形式.....	255
1.4.3.	当运算符等义于某个函数.....	256
1.4.3.1.	“函数” == “lambda”.....	256
1.4.3.2.	当运算符等义于某个函数.....	257
1.4.3.3.	JavaScript 语言中的函数式编程.....	259
1.5.	JavaScript 中的函数.....	260
1.5.1.	可变参数数，与值参数传递.....	261
1.5.2.	非惰性求值.....	265
1.5.3.	函数是第一型.....	268
1.5.4.	函数是一个值.....	269
1.5.5.	可遍历的调用栈.....	270
1.5.5.1.	callee：我是谁.....	272
1.5.5.2.	caller：谁呼(叫)我.....	273
1.6.	闭包.....	275
1.6.1.	什么是闭包.....	276
1.6.2.	什么是函数实例与函数引用.....	278
1.6.3.	(在被调用时，)每个函数实例至少拥有一个闭包.....	280
1.6.4.	函数闭包与调用对象.....	284
1.6.4.1.	“调用对象”的局部变量维护规则.....	285
1.6.4.2.	“全局对象”的变量维护规则.....	286
1.6.4.3.	函数闭包与“调用对象”的生存周期.....	286
1.6.4.4.	引用与泄漏.....	289
1.6.5.	函数实例拥有多个闭包的情况.....	293
1.6.6.	语句或语句块中的闭包问题.....	295
1.6.7.	闭包中的标识符(变量)特例.....	297
1.6.8.	函数对象的闭包及其效果.....	300
1.6.9.	闭包与可见性.....	302
1.6.9.1.	函数闭包带来的可见性效果.....	302
1.6.9.2.	对象闭包带来的可见性效果.....	305
1.6.9.3.	匿名函数的闭包与可见性效果.....	309
1.7.	综述.....	311
第五章	JavaScript 的动态语言特性.....	312
1.1.	概述.....	312
1.1.1.	动态数据类型的起源.....	312
1.1.2.	动态执行系统的起源.....	313
1.1.2.1.	编译系统、解释系统与编码.....	313
1.1.2.2.	动态执行.....	314
1.1.3.	脚本系统的起源.....	314
1.1.4.	脚本只是一种表面的表现形式.....	316
1.2.	动态执行(eval).....	318
1.2.1.	动态执行与闭包.....	318
1.2.1.1.	eval 代码专享闭包.....	320

1.2.1.2.	eval 可以使用全局闭包.....	321
1.2.1.3.	eval 使用当前函数的闭包.....	323
1.2.2.	动态执行过程中的语句、表达式与值.....	324
1.2.3.	奇特的、甚至是负面的影响.....	327
1.2.3.1.	变量作用域可能变化.....	327
1.2.3.2.	代码的不可编译性.....	327
1.3.	动态方法调用(call 与 apply).....	328
1.3.1.	动态方法调用中指定 this 对象.....	329
1.3.1.	丢失的 this 引用.....	331
1.3.2.	栈的可见与修改.....	332
1.3.3.	兼容性：低版本中的 call()与 apply().....	335
1.4.	重写.....	338
1.4.1.	原型重写.....	338
1.4.2.	构造器重写.....	340
1.4.2.1.	语法声明与语句含义不一致的问题.....	342
1.4.2.2.	对象检测的麻烦.....	345
1.4.2.3.	构造器的原型(prototype 属性)不受重写影响.....	348
1.4.2.4.	“内部对象系统” 不受影响.....	348
1.4.2.5.	让用户对象系统影响内部对象系统.....	349
1.4.2.6.	构造器重写对直接量声明的影响.....	350
1.4.2.7.	构造绑定.....	351
1.4.2.8.	内置构造器重写的概述.....	354
1.4.3.	对象成员的重写.....	355
1.4.3.1.	成员重写的检测与删除.....	356
1.4.4.	宿主对重写的限制.....	359
1.4.4.1.	重写与引用.....	359
1.4.4.2.	宿主环境的限制.....	361
1.4.5.	引擎对重写的限制.....	364
1.4.5.1.	this 的重写.....	364
1.4.5.2.	语句语法中的重写.....	365
1.4.5.3.	结构化异常处理中的重写.....	367
1.5.	包装类：面向对象的妥协.....	368
1.5.1.	显式包装元数据.....	369
1.5.2.	隐式包装的过程与检测方法.....	371
1.5.3.	包装值类型数据的必要性与问题.....	374
1.5.4.	其它直接量与相应的构造器.....	375
1.5.4.1.	函数特例.....	376
1.5.4.2.	正则表达式特例.....	376
1.6.	关联数组：对象与数组的动态特性.....	377
1.6.1.	关联数组是对象系统的基础.....	378
1.6.2.	用关联数组实现的索引数组.....	378
1.6.3.	干净的对象.....	382

1.7.	类型转换.....	386
1.7.1.	宿主环境下的特殊类型系统.....	387
1.7.2.	值运算：类型转换的基础.....	390
1.7.3.	隐式转换.....	391
1.7.3.1.	运算导致的类型转换.....	391
1.7.3.2.	语句(语义)导致的类型转换.....	393
1.7.4.	值类型之间的转换.....	393
1.7.4.1.	undefined 的转换.....	394
1.7.4.2.	number 的转换.....	394
1.7.4.3.	boolean 的转换.....	395
1.7.4.4.	string 的转换.....	396
1.7.4.5.	值类型数据的显式转换.....	397
1.7.5.	从引用到值：深入探究 valueOf()方法.....	398
1.7.6.	到字符串类型的显式转换.....	401
1.7.6.1.	重写 toString()方法.....	402
1.7.6.2.	从数值到字符串的显式转换.....	403
1.7.6.3.	其它类型的显式转换.....	404
1.7.6.4.	序列化.....	405
1.8.	综述.....	407
第六章	Qomo 框架的核心技术与实现.....	410
1.1.	Qomo 框架的技术发展与基本特性.....	410
1.1.1.	Qomo 框架的技术发展.....	410
1.1.1.1.	两种技术方案的选择.....	410
1.1.1.2.	WEUI 所奠定的基本风格.....	413
1.1.1.3.	Qomo 完善了整个内核框架.....	414
1.1.2.	Qomo 的体系结构.....	416
1.1.3.	Qomo 框架设计的基本原则.....	417
1.2.	基于类继承的对象系统.....	418
1.2.1.	Qomo 类继承的基本特性.....	418
1.2.2.	Qomo 类继承的语法.....	420
1.2.2.1.	基本的类类型声明与使用.....	420
1.2.2.2.	特性、默认读写器与限制符.....	421
1.2.2.3.	多投事件与继承父类方法.....	422
1.2.3.	Qomo 类继承系统的实现.....	424
1.2.3.1.	类继承系统的基本思想.....	424
1.2.3.2.	类继承系统的实现框架.....	426
1.2.3.3.	类数据块(CDB)与实例数据块(IDB).....	429
1.2.3.4.	细节 1：特性、读写器和存取限制符.....	434
1.2.3.5.	细节 2：调用父类方法.....	441
1.2.3.6.	细节 3：对多投事件系统的支持.....	447
1.2.3.7.	细节 4：实例构造周期的特殊性.....	450
1.2.4.	Qomo 类继承系统的高级话题.....	452

1.2.4.1.	类构造周期的完整特性与工具函数.....	453
1.2.4.2.	实例构造周期重写读写器(getter/setter).....	457
1.2.4.3.	类注册函数 Class()的特殊语法与应用.....	458
1.2.4.4.	Qomo 对象仍然是基于原型构造的.....	461
1.2.4.5.	Qomo 对象构造的第二种语法.....	462
1.2.4.6.	读写器声明的隐式效果.....	464
1.2.4.7.	通用读写器.....	465
1.2.4.8.	this.Create()的作用与重写.....	467
1.3.	多投事件系统.....	469
1.3.1.	多投事件系统的基本特性与语法.....	469
1.3.2.	多投事件系统的实现.....	471
1.3.3.	多投事件的中断与返回值.....	475
1.3.4.	多投事件系统的安全性.....	477
1.4.	接口系统.....	479
1.4.1.	基本概念与语法.....	480
1.4.1.1.	接口的声明与继承性.....	481
1.4.1.2.	接口的注册.....	482
1.4.1.3.	接口的查询.....	483
1.4.2.	接口实现.....	484
1.4.2.1.	类实现接口.....	484
1.4.2.2.	对象实现接口.....	485
1.4.2.3.	用户实现接口.....	486
1.4.3.	Qomo 接口系统的高级话题.....	488
1.4.3.1.	将接口注册到构造器、类与类类型.....	488
1.4.3.2.	函数作委托者的二义性问题.....	490
1.4.3.3.	接口的接口.....	491
1.4.3.4.	接口内聚(内部聚合).....	492
1.4.3.5.	接口与特性读写器.....	494
1.4.3.6.	为特定的对象注册接口.....	495
1.4.4.	接口委托.....	496
1.4.4.1.	接口委托的基本方法.....	497
1.4.4.2.	接口委托中的代理函数.....	499
1.4.4.3.	普通函数作委托者.....	500
1.4.4.4.	构造器函数作委托者.....	501
1.4.4.5.	类类型作委托者.....	503
1.4.4.6.	委托协议.....	505
1.4.4.7.	委托方法中的 this 引用.....	508
1.4.5.	Qomo 接口系统的实现.....	509
1.4.5.1.	接口系统的实现框架.....	509
1.4.5.2.	接口注册的实现.....	510
1.4.5.3.	接口查询的实现.....	512
1.4.5.4.	聚合与委托的实现.....	514

1.5.	命名空间.....	518
1.5.1.	Qomo 的命名空间的复杂性.....	518
1.5.2.	命名空间的使用.....	521
1.5.2.1.	空间创建.....	521
1.5.2.2.	将类注册到命名空间.....	524
1.5.2.3.	查询与转换.....	525
1.5.3.	命名空间的实现.....	525
1.6.	AOP.....	527
1.6.1.	基本概念与语法.....	528
1.6.1.1.	切面对象与切点.....	529
1.6.1.2.	将切面关联到被观察者.....	530
1.6.1.3.	切面行为.....	531
1.6.2.	高级切面特性.....	532
1.6.2.1.	定制连结点与定制切面.....	532
1.6.2.2.	切面的合并(merge)和联合(combine).....	535
1.6.2.3.	切面行为间的数据传递.....	537
1.6.2.4.	切面对目标系统的影响.....	538
1.6.2.5.	匿名函数的切面.....	539
1.6.2.6.	函数名对切面系统的影响.....	541
1.6.3.	Qomo 中切面系统的实现.....	542
1.7.	其它.....	544
1.7.1.	装载、内联与导入.....	544
1.7.1.1.	装载: \$import().....	545
1.7.1.2.	内联: \$inline().....	546
1.7.1.3.	导入: \$import2().....	547
1.7.2.	四种模板技术.....	548
1.7.2.1.	连接槽: joinSlot().....	549
1.7.2.2.	格式化字符串: format().....	550
1.7.2.3.	模板对象: Pattern().....	551
1.7.2.4.	模板类: TTemplet.....	554
1.7.3.	出错处理.....	556
1.7.4.	其它功能模块.....	557
1.7.4.1.	性能分析工具.....	558
1.7.4.2.	兼容与增强模块.....	561
1.7.4.3.	框架类.....	561
1.7.4.4.	集成: \$builder.....	564
第七章	一般性的动态函数式语言技巧.....	567
1.1.	消除代码的全局变量名占用.....	567
1.2.	一次性的构造器.....	568
1.3.	对象充当识别器.....	570
1.4.	识别 new 运算进行的构造器调用.....	572
1.5.	使用直接量及其包装类快速调用对象方法.....	574

1.6.	三天前是星期几?	576
1.7.	使用对象的值含义来构造复杂对象.....	577
1.8.	控制字符串替换过程的基本模式.....	580
1.9.	实现二叉树.....	581
1.10.	将函数封装为方法.....	584
1.11.	使用 with 语句来替代函数参数传递.....	587
1.12.	使用对象闭包来重置重写.....	588
1.13.	构造函数参数.....	590
1.14.	使用更复杂的表达式来消减 IF 语句.....	593
1.15.	利用钩子函数来扩展功能.....	596
1.16.	安全的字符串.....	599
附录一:	相关资料及参考章节.....	601
附录二:	术语表.....	603

第一部分 语言基础

“我们最初利用 JavaScript 的目的，是让客户端的应用不必从服务器重新加载页面即可回应用户的输入信息，并且提供一种功能强大的图形工具包给脚本编写者。”

（然而，JavaScript 的）部分技术被采纳为所有浏览器的标准，而其它技术则没有。导致的结果是“不成熟的标准化、碎片化以及开发商受挫”。

——JavaScript 之父，Mozilla 首席技术官 Brendan Eich

引自：英国《金融时报》“Google 搜索网速的答案”

第一章 十年 JavaScript

几乎每本讲 JavaScript 的书都会用很多的篇幅来讲 JavaScript 的源起与现状。本书也需要这样的一个开篇吗？

不。我虽然也想过这样，但我不打算让读者去读一些能够从 Wiki 中抄出来的文字，或者在不同的书籍中都可以看到的、千篇一律的文字。所以，我来写写我与 JavaScript 的故事。在这个过程中，你会看到一个开发者在每个阶段对 JavaScript 的认识，也可以知道这本书的由来。

当然，一个个人的历史，在一门语言的历史面前，甚至显得是那样的微不足道。因此除了故事性与可读性之外，对这一章的前三个小节，你也可以选择跳过去。

1.1. 第一段在网页中的代码

1.1.1. 新鲜的玩意儿

1996 年末，公司老板 P&J 找我去给他的一个朋友帮忙做一些网页。那时事实上还没有说要做成网站。在那个时代，中国可能还有 2/3 的 IT 人在玩一种叫“电子公告板”的东西(这与现在的 BBS 很不一样)，而另外的 1/3 可能就已经开始了互联网之旅，知道了主页(HomePage)、超链接(HyperLink)这样的一些东西。

我最开始做的网页只用于展示信息，是一个个单纯的、静态的网页，并通过一些超链接连接起来。当时的网页开发的环境并不好(象现在的 Dreamweaver 这种东西，那时只能是梦想)，因此我只能用记事本(notepad.exe)来写 HTML，当时显示这些 .htm 文件的浏览器，就是 Netscape Navigator 3。

我很快就遇到了麻烦，因为 P&J 的朋友说希望让来浏览网页的用户们能更多的事，例如搜索什么的。我笑着说：如果在电子公告板上，写段脚本就可以了；但在互联网上面，却要做很多的工作。

我事实上并不知道要做多少的工作。我随后查阅的资料表明：我们不但要

在网页中放一些表单让浏览者提交信息，还要在网站的服务器上写些代码来响应这些提交。我向那位先生摊开双手，说：“如果你真的想要这样做，那么我們可能需要三个月，或者更久。因为我还必须学习一些新鲜的玩意儿才行。”

那时的“触网者”们，对这些“新鲜的玩意儿”的了解还几乎是零。因此，这个想法很自然地被搁置了。而我在后来(1997年)被调到成都，终于有更多的机会接触 Internet 网络，而且浏览器环境已经换成了 Internet Explorer 4.0。

那是一个美好的时代。通过互联网络，大量的新东西被很快传递进来。我终于有机会了解一些新的技术名词，例如 CSS 和 JavaScript。HTML 4.01 的标准已经确定，浏览器的兼容性开始变得更好，Internet Explorer(简称 IE)也越来越有取代 Netscape Navigator(简称 NN)而一统天下的形势。除了这些，我还对在 Delphi 中进行 ISAPI CGI 和 ISAPI Filter 的开发技术也进行了深入的学习。

1.1.2. 第一段在网页中的代码

1998 年，我调回到河南郑州，成为一名专职程序员，任职于当时的一家反病毒软件公司。我的主要工作是用 Delphi 在 Windows 环境下的开发，而当时我的个人兴趣之一，就是“做一个个人网站”。那时大家都对“做主页”很感兴趣，我的老朋友傅贵()就专门写了一套代码，以方便普通互联网用户将自己的主页放到“个人空间”里。同时，他还为这些个人用户还提供了公共的 BBS 程序和其它的一些服务器端代码。但我并不满足这些，我满脑子想的是做一个“自己的网站”。我争取到了一台使用 IIS 4.0 的服务器，由于有 ISAPI CGI 这样的服务器端技术，因此一年多前的那个“如何让浏览者提交信息”的问题已经迎刃而解。而当时更先进的浏览器端开发技术也已经出现，例如 Java Applet。我当时便选择了一个 Java Applet 来做“网页菜单”。

但是，当时在 IE 中显示 Java Applet 之前，需要装载整个的 JVM()。这对于现在的 CPU 来说，已经不是什么大不了的负担了，但在当时这个过程却非常漫长。在这个“漫长的过程”中，网页显示一片空白，因此浏览者可能在看到一个“漂亮的菜单”之前就跑掉了。

为此我不得不象做 Windows 程序一样弄一个“闪屏窗口”放在前面。这个窗口只用于显示“Loading...”这样的文字(或图片)。而同时，我在网页中加入一个<APPLET>标签，使得 JVM 能偷偷地载入到浏览器中。然而，接下来的

问题是：这个过程怎么结束呢？

我当时能找到的所有 Java Applet 都没有“当 JVM 载入后自动链接到其它网页”的能力。但其中有一个可以支持一种状态查询，它能在一个名为 `isInited` 的属性中返回状态 `True` 或 `False`。

这时，我需要在浏览器中查询到这种状态，如果是 `True`，我就可以结束 Loading 过程，进入到真正的主页中去。由于 JVM 已经偷偷地载入过了，因此“漂亮的菜单”就能很快地显示出来。因为我得不到 Java Applet 的 java 源代码并重写这个 Applet 去切换网址，因此这个“访问 Java Applet 的属性”的功能就需要用一种在浏览器中的技术来实现了。

这时跳到我面前的东西，就是 JavaScript。我为此而写出的代码如下：

```
<script language="JavaScript">
function checkInited() {
    if (document.MsgApplet.isInited) {
        self.location.href = "mainpage.htm";
    }
}
setInterval("checkInited()", 50)
</script>
```

1.1.3. 最初的价值

JavaScript 最初被开发人员接受，其实是一种无可奈何的选择。

首先，网景公司（Netscape Communications Corporation）很早就意识到：网络需要一种集成的、统一的、客户端到服务端的解决方案。为此 Netscape 提出了 LiveWire 的概念^①，并设计了当时名为 LiveScript 的语言用来在服务器上创建类似于 CGI 的应用程序。而这时，网景公司也意识到他们的浏览器 Netscape Navigator 中需要一个脚本语言的支持，解决类似于“在向服务器提交数据之前进行验证”的问题。于是在 1996 年 3 月发布 NN 2.0 时，LiveScript 最早被作为一种“浏览器上的脚本语言”给推到网页制作人员的面前。同期发

^① LiveWire 是 Netscape 公司的一个通用的 Web 开发环境，仅仅支持 NetscapeEnterprise/FastTrackServer，而不支持其他的 Web 服务器。这种技术在服务器端通过内嵌于网页的 LiveScript 代码，使用名为 database、DbPool、Cursor 等的一组对象来存取 LiveWire Database。作为整套的解决方案，Netscape 在客户端网页上也提供 LiveScript 脚本语言的支持，除了访问 Array、String 等这些内置对象之外，也可以访问 window 等浏览器对象。LiveScript 后来发展为 JavaScript，而 LiveWire 架构也成为所有 Web 服务器提供 SP(Server Page)技术的蓝本。例如在 IIS 中的 ASP，以及更早期的 IDC(Internet Database Connect)。

布的“LiveWire Server Extension Engine (后来的 Netscape Enterprise Server 2.0)”也包括了一个该语言的服务器端版本^①。因此, JavaScript 最初其实是为了解决浏览器与服务器之间统一开发而被实现的一种语言。

这时 Sun 公司的 JAVA 语言大行其道。Netscape 决定在服务器端与 Sun 进行合作, 这种合作后来扩展到浏览器, 推出了名为 Java Applet 的“小应用”。而 Netscape 也借势将 LiveScript 改名为 JavaScript, 并在 1996 年 8 月发布的 NN3 中提供了支持。

微软在浏览器方面是一个后来者。因此, 他不得不在自己的浏览器中加入 JavaScript 的支持。但为了避免冲突, 微软使用了 JScript 这个名字。微软在 1996 年 8 月发布 IE3 时, 提供了相当于 NN3 的 JavaScript 脚本语言支持, 但同时也提供了自己的 VBScript。

在 IE 与 NN 进行那场著名的“浏览器大战”的时候, 没有人能够看到结局。因此要想做一个“可以看的网页”, 只能选择一个在两种浏览器上都能运行的脚本语言。这就使得 JavaScript 成为唯一可能正确的答案。当时, 几乎所有的书籍都向读者宣导“兼容浏览器是一件天大的事”。为了这种兼容, 一些书籍甚至要求网页制作人员最好不要用 JavaScript, “让所有的事, 在服务器上使用 Perl 或者 CGI 去做好了”。

然而随着 IE4.0 的推出以及 DHTML 带来的诱惑, 一切都发生了改变。

1.2. 用 JavaScript 来写浏览器上的应用

1.2.1. 我要做一个聊天室

大概是在 1998 年 12 月中旬, 我的个人网站完工了。

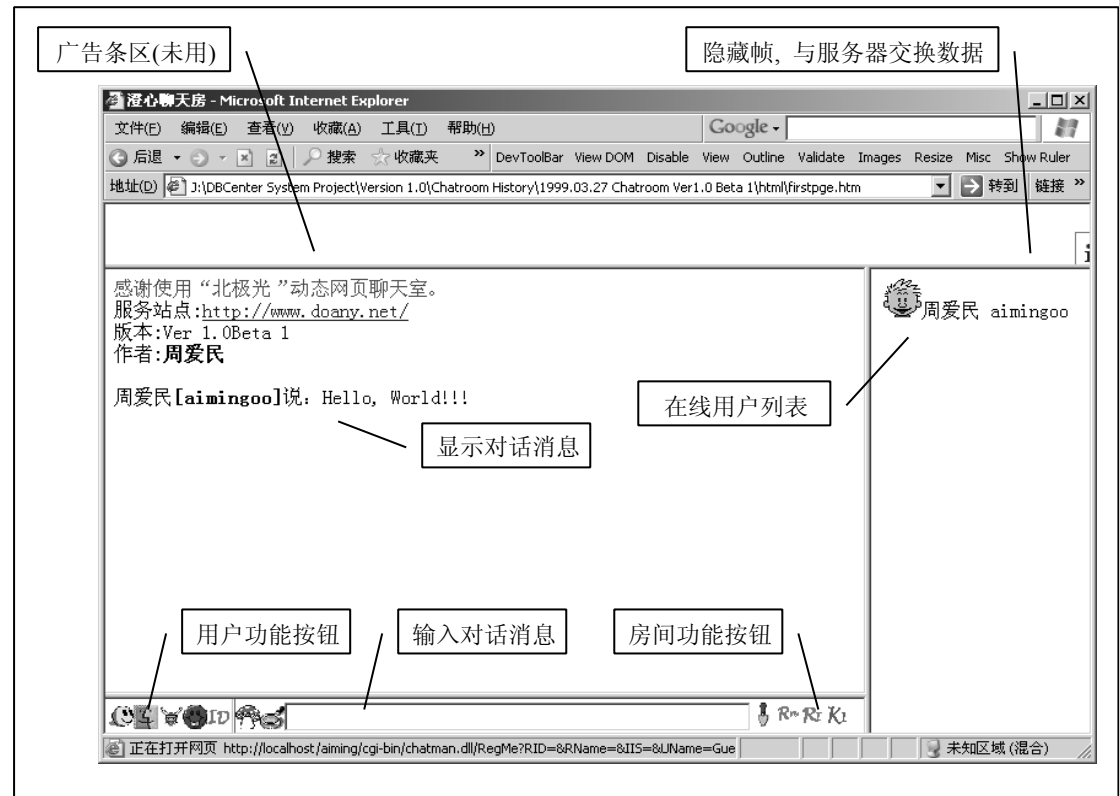
这是一个文学网站, 这个网站在浏览器上用到了 Java Applet 和 JavaScript, 并且为 IE4.0 的浏览器提供了一个称为“搜索助手”的浮动条(FlatBar), 用于快速地向服务器提交查询文章的请求。而服务器则使用了 Delphi 来开发的 ISAPI CGI, 运行于当时流行的 Windows NT 上的 IIS 系统。

接下来, 我冒出的想法是: 我要做一个聊天室。因为在我的个人网站中,

^① <http://wp.netscape.com/newsref/pr/newsrelease41.html>

包括论坛、BBS 等都有网站免费提供，唯独没有聊天室。

1999 年春节期间，我在四川的家中开始做这个聊天室并完成了原型系统 (我称为 beta0)，又一个月后，这个聊天室的 beta1 终于在互联网上架站运行。



这个聊天室的功能集设定为：

分类	功能	概要
用户	设定名字、昵称、肖像	用户功能按钮 1，对话框
	照片上传并显示给所有人	用户功能按钮 1，对话框
	更改名字、昵称、肖像	用户功能按钮 2~4，快速更换
消息	在当前房间发消息	普通聊天
	向指定用户发消息	用户功能按钮 6，私聊功能
	向所有房间发消息	用户功能按钮 6，通告功能
房间	转到指定房间	用户功能按钮 6，对话框
	创建新房间	房间功能按钮 1
	房间更名	房间功能按钮 2
	设定房间初始化串	房间功能按钮 3
界面	显示 / 不显示昵称	用户功能按钮 5，动态切换
工具	按名称查找房间	用户功能按钮 6，对话框
	按照 ID/名字查找用户	用户功能按钮 6，对话框
	踢人	房间功能按钮 4

在这个聊天室的右上角有一个“隐藏帧”，是用 FRAMESET 来实现的。这是最早期实现 WEB RPC(Remote Procedure Call)的方法，那时网页开发还不推荐使用 IFRAME，也没有后来风行的 Ajax。

从下面的状态栏上，我们也可以看到这个聊天室在调用服务器上的.DLL——这就是那个用 Delphi 写的 ISAPI CGI。当时还没有 PHP，而 ASP 也只是刚刚出现，并不成熟。

这个聊天室在浏览器上大量地使用了 JavaScript。一方面，它用于显示聊天信息、控制 CSS 显示和实现界面上的用户交互；另一方面，我用它实现了一个 Command Center，将浏览器中的行为编码成命令发给服务器的 ISAPI。这些命令被服务器转发给聊天室中的其它用户，这些用户的浏览器中的 JavaScript 代码能够解释这些命令并执行类似于“更名”、“更新列表”之类的功能——服务器上的 ISAPI 基本上只用于中转命令，因此效率非常高。

你可能已经注意到，这其实与现在的 Ajax 的思想如出一辙。

虽然这个聊天室在 beta0 时还尝试支持了 NN4，但 beta1 时就放弃了——因为 IE4 提供的 DHTML 模型已经可以动态更新网页了（使用 insertAdjacentHTML），而 NN4 仍只能调用 document.write 来修改页面。

1.2.2. Flash 的一席之地

我所在的公司也发现了互联网上的机会，成立了互联网事业部。我则趁机提出了一个庞大的计划，名为 JSVPS(JavaScripts Visual Programming System)。

JSVPS 在服务器端表现为 dataCenter 与 dataBaseCenter。前者用于类似于聊天室的即时数据交互，后者则用于类似于论坛中的非即时数据交互。在浏览器端，JSVPS 提出了开发网页编辑器和 JavaScript 组件库的设想。

这时微软的 IE4.x 已经从浏览器市场拿到了超过 80% 的市场份额，微软开始试图把 Java Applet 从它的浏览器中赶走。

微软依靠的是它在 IE 中加入的 ActiveX 技术，于是 Macromedia Flash 作为一个 ActiveX 插件挤了进来。Flash 在图形矢量表达能力和开发环境方面表现优异，使当时的 Java Applet 优势全失。当时，微软急于从桌面环境挤走 Java，以应对接下来在 .NET 与 Java 之间的语言大战。而 Flash 与 Dreamweaver 当时只是网页制作工具，因此微软并没有放在眼里，假手 Flash 赶走了 Java Applet。

Dreamweaver 系列的崛起，使得网页制作工具的市场变得几乎没有了悬念。主力放在 Java Applet 的工具，例如 HotDog 等等都纷纷下马；而纯代码编辑的工具，如国产的 CutePage 则被 Dreamweaver 慢慢地蚕食着市场。同样的原因，JSVPS 项目在浏览器端“开发网页编辑器”的设想也最终未能实施。而“JavaScript 组件库”也因为市场不明朗一直不能投入开发。

而在服务器端的 dataCenter 与 dataBaseCenter 都成功地投入了商用。此后，我在聊天室上花了更多的精力。到 2001 年下半年，它已经开始使用页签形式来管理多房间同时聊天，并加入语言过滤、表情、行为和用户界面定制等功能。而且，通过对核心代码的分离，聊天室已经衍生出“Web 即时通讯工具”和“网络会议室”这样的版本。



2002 年初，聊天室发布的最终版本 (ver 2.8) 的功能设定已经远远超出了现在网上所见的 Web 聊天室的功能集。上面的示例中，包括颜色选取器、本地历史记录、多房间管理、分屏过滤器、音乐、动作、表情库和 Outlook 样式的工具栏，以及中间层叠的窗体，都是由 DHTML 与 CSS 来动态实现的。这在后台驱动这一切的，就是 JavaScript。

我没有选择这时已经开始流行的 Flash，除了因为用 DHTML 做聊天室的

界面效果并不逊于 Flash，也因为在 RWC 与 RIA 的战争中，我选择了前者。

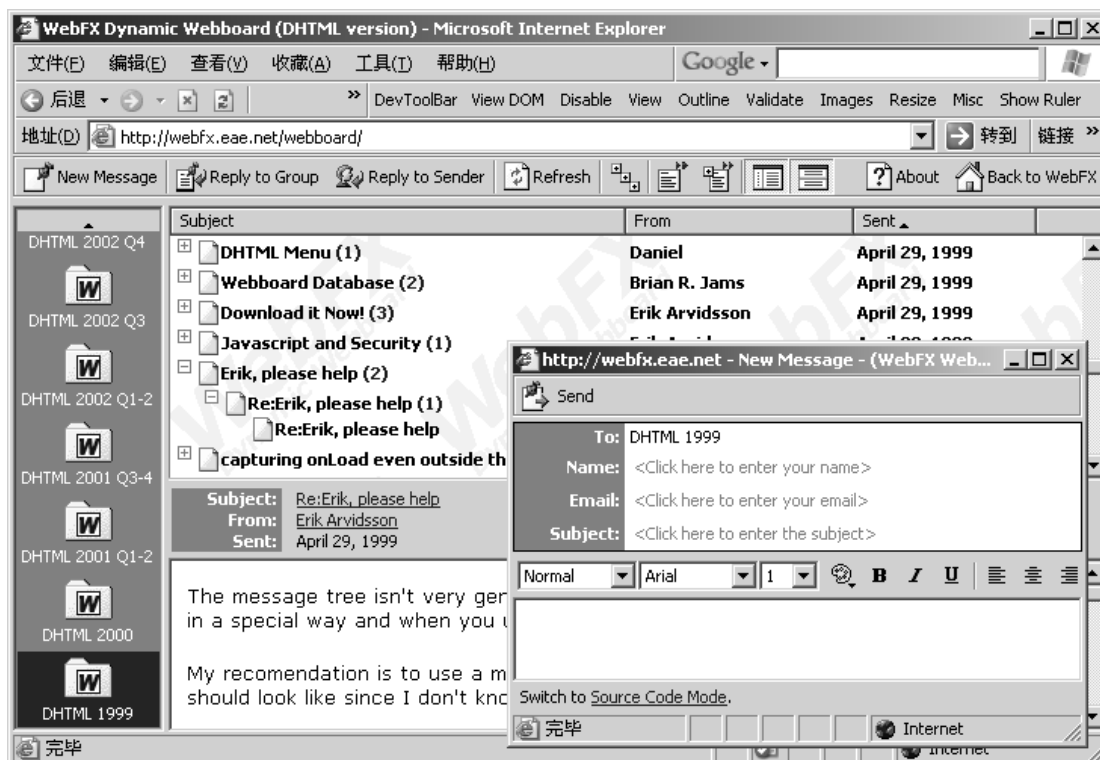
1.2.3. RWC 与 RIA 之争

追溯 RWC 的历史，就需要从“动态网页(DHTML)”说起。

在 1997 年 10 月发布的 IE4 中，微软提供了 JScript 3，这包括当时刚刚发布的 ECMAScript Edition 1，以及尚未发布的 JavaScript 1.3 的很多特性。最重要的是，微软颇有创见地将 CSS、HTML 与 JavaScript 技术集成起来，提出了 DHTML 开发模型(Dynamic HTML Model)，这使得几乎所有的网页都开始倾向于“动态(Dynamic)起来”。

在开始，人们还很小心地使用着脚本语言，但当微软用 IE4 在浏览器市场击败网景之后，很多人发现：没有必要为 10%的人去多写 90%的代码。因此，“兼容”和“标准”变得不再重要。于是 DHTML 成了网页开发的事实标准，以致于后来由 W3C 提出的 DOM(Document Object Model)在很长一个阶段中都没有产生任何影响。

这时，成熟的网页制作模式，使得一部分人热衷于创建更有表现能力和实用价值的网页，他们把这样的浏览器和页面叫做“Rich Web Client”，简称 RWC。Rich Web 的概念产于何时已经不可考，但 Erik Arvidsson 一定是这其中的先行者。他拥有一个知名的个人网站 WebFX(<http://webfx.eae.net/>)，从 1997 开始，他在 WebFX 上公布他关于浏览器上开发的体验的文章和代码。他可能是最早通过 JavaScript+DHTML 实现 menu、tree 以及 tooltip 的人。1998 年末，它就已经在个人网站上实现了一个著名的 WebFX Dynamic WebBoard：



这套界面完整地模仿了 Outlook，因而是在 Rich Web Client 上实现类 Windows 界面的经典之作。

而在这时盛行的 Flash 也需要一种脚本语言来表现动态的矢量图形。因此，Macromedia 很自然地在 Flash 3 中开始加入 JavaScript 脚本支持。随后 Macromedia 又以 JavaScript 作为底本完成了自己的 ActionScript，并加入到 Flash 5 中。随着 ActionScript 被浏览器端开发人员逐渐接受，这种语言也日渐成熟，于是 Macromedia 开始提出自己的对“浏览器端开发”的理解。这就是有名的 Rich Internet Application(RIA)。

这样一来，RIA 与 RWC 分争“富浏览器客户端应用 (Rich Web-client Application, RWA)”的市场的局面出现了——微软开始尝到自己种下的苦果：一方面他通过基于 ActiveX 技术的 Flash 赶走了 Java Applet，另一方面却又使得 Dreamweaver 和 Flash 日渐坐大。实在是前门拒虎，后门进狼。微软用丢失网页编辑器和网页矢量图形事实标准的代价，换取了在开发工具(例如 Visual Studio .NET)和语言标准(例如 CLS, Common Language Specification)方面的成功。而这个代价的直接表现之一，就是 RIA 对 RWC 的挑战。

RIA 的优势非常明显，在 Dreamweaver UltraDev 4.0 发布之后，Macromedia 成为网页编辑、开发类工具市场的领先者。而在服务器端，有基于 Server Page 思路的 ColdFusion、优秀的 J2EE 应用服务器 JRun 和面向 RIA 模式的 Flash 组件环境 Flex。这些构成了完整的 B/S 三层开发环境。然而似乎没有人能容忍 Macromedia 独享浏览器开发市场，并试图染指服务器端的局面，所以 RIA 没有得到足够的商业支持。另一方面，ActionScript 也离 JavaScript 越来越远，既不受传统网页开发者的青睐，而对以设计人员为主体的 flash 开发者来讲又设定了过高的门槛。

但 RWC 的状况则更加尴尬。因为 JavaScript 中尽管有非常丰富的、开放的网络资源，但却找不到一套兼容的、标准的开发库(象标准 C 一样的)，也找不到一套规范的对象模型(DOM 与 DHTML 纷争不断)，甚至连一个统一的代码环境都不存在(没有严格规范的 HOST 环境)。

RIA 热捧浏览器上的 Rich Application 市场的同时，自由的开发者们则在近乎疯狂地挖掘 CSS、HTML 和 DOM 中的宝藏，试图从中寻找到 RWC 的出路。支持这一切的，是 Java Script 1.3~1.5，以及在 W3C 规范下逐渐成熟 Web 开发基础标准。而在这整个的过程中，RWC 都只是一种没有实现的、与 RIA 的商业运作进行着持续抗争的理想而已。

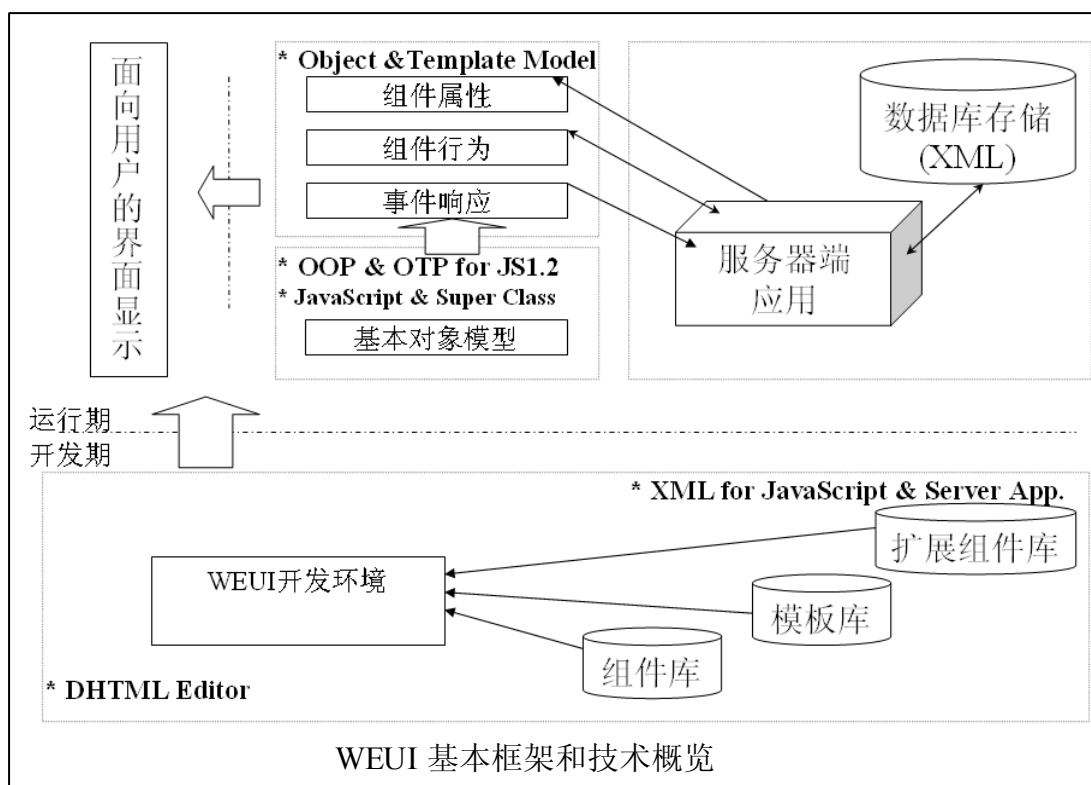
1.3. 没有框架与库的语言能怎样发展呢？

1.3.1. 做一个框架

聊天室接下来的发展几乎停滞了。我在 RWC 与 RIA 之争中选择了 RWC，但也同时面临了 RWC 的困境：我找不到一个统一的框架或底层环境。因此，聊天室如果再向下发展，也只能是在代码堆上堆砌代码而已。

于是，整个的 2003 年，我基本上都没有再碰过浏览器上的开发。2004 年初的时候，我到一家新的公司 (Jxsoft Corporation) 任职。这家公司的主要业务都是 B/S 架构上的开发，于是我提出“先做易做的 1/2”的思路，打算通过提高浏览器端开发能力，来加强公司在 B/S 架构开发中的竞争力。

于是我得到很丰富的资源，来主持一个名为 WEUI(Web Enterprise UI Component Framework)的项目的开发工作。这个项目的最初设想，跟 JSVPS 一样是个庞然大物(好象我总是喜欢这样构想，哈哈，下图中应该是 JS1.3)：



WEUI 包括了 B/S 两端的设计，甚至还有自己的一个开发环境。而真正做起来的时候，则是从 WEUI OOP Framework 开始的。这是因为 JavaScript 语言没有真正的“面向对象编程(OOP)”框架。

在我所收集的资料中，第一个提出 OOP JavaScript 概念的是 Brandon Myers，他为一个名为 Dynapi^①的开源项目工作中，提出了名为“SuperClass”的概念和原始代码。后来，在 2001 年 3 月，Bart Bizon 按照这个思路发起了一个名为 SuperClass 的开源项目，放在 SourceForge 上。这份代码维护到 ver 1.7b。半年后，Bart Bizon 放弃了 SuperClass 并重新发起 JSClass 项目，这成为 JavaScript 早期框架中的代表作品。

后来许多的 JavaScript OOP Framework 都不约而同地采用了与 SuperClass 类同的方法——使用“语法解释器”——来解决框架问题。然而前面提到过的实现了“类 Outlook 界面”的 Erik Arvidsson 则采用了另一种思路：使用 JavaScript 原生的(native)代码在执行期建立框架，并将这一方法用在了另一个同样著名的项目 Bindows 上。

对于中国的一部分的 JavaScript 爱好者来说，RWC 时代就开始于《程序员》2004 年第 5 期的一篇《王朝复辟还是浴火重生——The Return of Rich Client》。

^① Dynapi 是早期最负盛名的 JavaScript 开源项目中的一个，它创建得比 Bindows 项目更早，参与者也更多。

这篇文章讲的就是 Bindows。



Bindows 在浏览器上的不凡表现

Bindows 可能也是赶上了好时候，这年的 MS Teched 就有好几个专场来讲述智能客户端(Smart Client)。而“智能客户端”的基本思想就是跨平台的、弹性的富客户端(Rich Client)。因此“丰富的浏览器表现”立即成为“时新”的开发需求。以 Bindows 为代表的 RWC(Rich Web Client)才因此成为国内开发者和需求方共同关注的焦点。

WEUI v1.0 内核的研发工作大概就结束于此时。我在这个阶段中主要负责的就是 JavaScript OOP Language Core 的开发，并基本完成了对 JavaScript 语言在 OOP 方面的补充。而接下来，另外的两名开发人员^①则分别负责 Application Framework 与 Database Layer 的开发，他们的工作完成于 2004 年 8 月。紧接着 WEUI 就被应用到一个商业项目的前期开发中了——WEUI 很快显示出它在浏览器端的开发优势：它拥有完整的 OOP 框架与“基本够用”的组件库，为构建大型的浏览器端应用系统的可行性提供了实证。

WEUI 在开发环境和服务器端上没有得到投入。这与 JSVPS 有着基本相同

^① 他们分别是周鹏飞(leon)与周劲羽(yygw)。我们三人都姓周，实在是巧合。鹏飞现在是微软的软件工程师，而劲羽则领导了 Delphi 界非常有名的开源项目 CnPack & CnWizard 的开发，现居河南许昌。

的原因：没有需求。于是从 2004 年底开始，我就着手以 UI 组件库为主要目标的 WEUI v2.0 的开发，直到 2005 年 3 月。

1.3.2. 重写框架的语言层

Qomo 项目于 2005 年末启动，它自一开始便立意于继承和发展 WEUI 框架。为此我联系了 WEUI 原项目组以及产品所有的公司，并获得了基于该项目开源的授权。

从 WEUI 到 Qomo 转变之初，我只是试图整理一套有关 WEUI 的文档，并对 WEUI 内核中有关 OOP 的部分做一些修补。因此在这个阶段，我用了一段时间撰写公开文档来讲述 JavaScript 的基本技术，这包括一组名为“JavaScript 面向对象的支持”的文章。而这个过程正好需要我深入地分析 JavaScript 对象机制的原理，以及这种原理与 Qomo 项目中对 OOP 进行补充的技术手段之间的关系。然而这一个分析的过程，让我汗如雨下：在此以前，我一直在用一种基于 Delphi 的面向对象思想的方式，来理解 JavaScript 中的对象系统的实现。这种方式完全忽略了 JavaScript 的“原型继承”系统的特性，不但弃这种特性优点于不顾，而且很多实现还与它背道而驰。换言之，WEUI 中对于 OOP 的实现，不但不是对 JavaScript 的补充，反而是一种伤害。

于是，我决定重写 WEUI 框架的语言层。不过在做出这个决定时，我仍然没有意识到 WEUI 的内部其实还存在着非常多的问题——这其中既有设计方面的，也有实现方面的问题。但我已经决定在 WEUI 向 Qomo 转化的过程中，围绕这些（已显现或正潜藏着的）问题开始努力了。

Qomo Field Test 1.0 发布于 2006 年 2 月中旬，它其实只包括一个 \$import() 函数的实现，用于装载其它模块。两个月之后终于发布了 beta 1，已经包括了兼容层、命名空间和 OOP、AOP、IOP 三种程序设计框架基础。这时 Qomo 项目组已经发展到十余人，部分人员已经开始参与代码的编写和审查工作了。我得到了 Zhe 的有力支持^①，他几乎独立完成了兼容层框架以及其在 Mozilla、safari 等引擎上的兼容代码。很多开源界的，或者对 JavaScript 方面有丰富经验的朋友对 Qomo 提出了他们的建议，包括我后来的同事 hax^②等。这些过程贯穿于整个的 Qomo 开发过程之中。

在经历过 2 个 beta 之后，Qomo 赶在 2007 年 2 月前发布了 v1.0 final。这

^① Zhe 的全名是方哲，

^② hax 的全名是贺师俊，他领导着一个名为 PIES 的 JavaScript 开源项目。

个版本包括了 Builder 系统、性能分析与测试框架，以及公共类库。此外，该版本也对组件系统的基本框架做出了设计，并发布了 Qomo 的产品路线图，从而让 Qomo 成为一个正式可用的、具有持续发展潜力的框架系统。

回顾 WEUI 至 Qomo 的发展历程，后者不单单是前者的一个修改版本，而几乎是在相同概念模型上的、完全不同的技术实现。Qomo 摒弃了对特殊的或具体语言环境相关特性的依赖，更加深刻地反映了 JavaScript 语言自身的能力。不但在结构上与风格上更为规范，而且在代码的实用性上也有了更大的突破。即使不讨论这些（看起来有些象宣传词的）因素，仅以我个人而言，正是在 Qomo 项目的发展过程中，加深了我对 JavaScript 的函数式、动态语言特性的理解，也渐而渐之地丰富了本书的内容。

1.3.3. 富浏览器端开发(RWC)与 Ajax

事情很快发生了变化——起码，看起来时代已经变了。因为从 2005 年开始，几乎整个 B/S 开发界都在热情地追捧一个名词：Ajax。

Ajax 中的“j”就是指 JavaScript，它明确地指出这是一种基于 JavaScript 语言实现的技术框架。但事实上它很简单——如果你发现它的真相不过是“使用一个对象的方法而已”，那么你可能会不屑一顾。因为如果你在 C++、JAVA 或者 Delphi 中，有人提出一个名词/概念（例如 Biby），说“这是如何使用一个对象的技术”，那绝不可能得到如 Ajax 般的待遇。

然而就是这种“教你如何用一个对象”的技术在 2005 至 2006 年之间突然风行全球。Google 基于 Ajax 构建了 gmail；微软基于 Ajax 提出了 Atlas；Yahoo 发布了 YUI(Yahoo! User Interface)；IBM 则基于 Eclipse 中的 WTP(Web Tools Project)发布了 ATF (AJAX Toolkit Framework)……一夜之间，原本在技术上的对立或者竞争的公司都不约而同地站到了一起：他们不得不面对这种新的技术带来的巨大的网络机会。

事实上在 Ajax 的早期就有人关注到这种技术的本质不过是同步执行。而“同步执行”其实在 Ajax 出现之前就已经应用得很广泛：在 Internet Explorer 等浏览器上采用“内嵌帧(IFrame)”载入并执行代码；在 Netscape 等不支持 IFrame 的浏览器中采用“层(Layer)”来载入并执行代码。这其中就有 JSRS(JavaScript Remote Scripting)，它的第一个版本发布于 2000 年 8 月。

但是类似于 JSRS 的技术来实现的 HTTP-RPC 方案存在两个问题：

- ☞ IFRAME/LAYER 标签在浏览器中没有得到广泛的支持，也不为 W3C 标准所认可；
- ☞ HTTP-RPC 没有提出数据层的定义和传输层的确切实施方案，而是采用 B/S 两端应用自行约定协议。

然而这仍然只是表面现象。JSRS 一类的技术方案存在先天的不足：它仅是技术方案。JSRS 并不是应用框架，也没有任何商业化公司去推动这种技术。而 Ajax 一开始就是具有成熟商业应用模式的框架，而且许多公司快速地响应了这种技术并基于它创建了各自的“同步执行”的解决方案和编程模型。

所以真正使 Ajax 浮上水面的并不是“一个 XMLHttpRequest 对象的使用方法”，也并不因为它是“一种同步和异步载入远程代码与数据的技术”，而是框架和商业标准所带来的推动力量。

这时人们似乎已经忘却了 RWC。而 W3C 却回到了这个技术名词上，并在三个主要方面对 RWC 展开了标准化的工作：

- ☞ 复合文档格式 (Compound Document Formats, CDF)
- ☞ Web 标准应用程序接口 (Web APIs)
- ☞ WEB 标准应用程序格式 (Web Application Formats)

这其中，CDF 是对 Ajax 中的“x” (即 XML) 提出标准；而 Web APIs 则试图对“j” (即 JavaScript) 提出标准。所以事实上，无论业界如何渲染 Ajax 或者其它的技术模型或框架，WEB 上的技术发展方向，仍然会落足到“算法+结构”这样的模式上来。这种模式在浏览器(B 端)上的表现，后者是由 XML/XHTML 标准化来实现的，而前者就是由 JavaScript 语言来驱动的。

微软当然不会错过这样的机会。微软开始意识到 flash 已经成为“基于浏览器的操作平台”这一发展方向上不可忽视的障碍，因此一方面发展 Altas 项目，为 .NET Framework+ASP.NET+VS.NET 这个解决方案解决 RWC 开发中的现实问题，一方面启动被称为“flash 杀手”的 Silverlight 项目对抗 Adobe 在企业级、门户级富客户端开发中推广 RIA 思想。

现在，编程语言体系也开始发生根本性的动摇。Adobe 购得 Macromedia 之后，把基于 JavaScript 规范的 ActionScript 回馈给开源界，与 Mozilla 开始联手打造 JavaScript 2；SUN 在 Java6 的 JSR-223 中直接嵌入了来自 Mozilla 的 Rhino JavaScript 引擎，随后 Java 自己也开源了。在另一边，微软借助 .NET 虚拟执行环境在动态执行上天生的优势，全力推动 DLR(Dynamic Language Runtime)，其中包括了 Ruby、Python、JavaScript、VB 等多种具有动态的、函

数式特性的语言实现，这使得 .NET Framework 一路冲进了动态语言开发领域的角斗场。

1.4. 为 JavaScript 正名

至 2005 年，JavaScript 就已经被发明了十年了。然而十年之后，这门语言的发明者 Brendan Eich 还在向这个世界解释“JavaScript 不是 Java，也不是脚本化的 Java（Java Scription）”。

这实在是计算机语言史上最罕见的一件事了。因为如今 WEB 页面中约有超过 90% 的页面中同时包含了 JavaScript 与 HTML，而后者从一开始就被人们接受，前者却用了十年都未能向开发人员说清楚“自己是什么”。

Brendan Eich 在这份名为“JavaScript 这十年（JavaScript at Ten Years）^①”的演讲稿中，重述了这门语言的早期历史：最早被称为 Mocha(魔卡)，后来为了迎合 Netscape 的 Live 战略而更名为 LiveScript。最后，到了 1995 年晚期，为了迎合市场对 Java 语言的热情，正式地、也是遗憾地更名为 JavaScript，并随网景浏览器推出。

Brendan 在这篇演讲稿最末一行写道：“不要将语言的名称商业化 (Don't let Marketing name your language)”。一门被误会了十年的语言的名字之争，是不是就此结束了呢？

仍然不是。因为这十年来，JavaScript 的名字已经越来越乱，更多市场的因素困扰着这门语言——好象“借用 Java 之名”已经成了扔不掉的黑锅。

1.4.1. JavaScript

我们先说正式的、标准的名词：JavaScript。它实际是指两个东西：

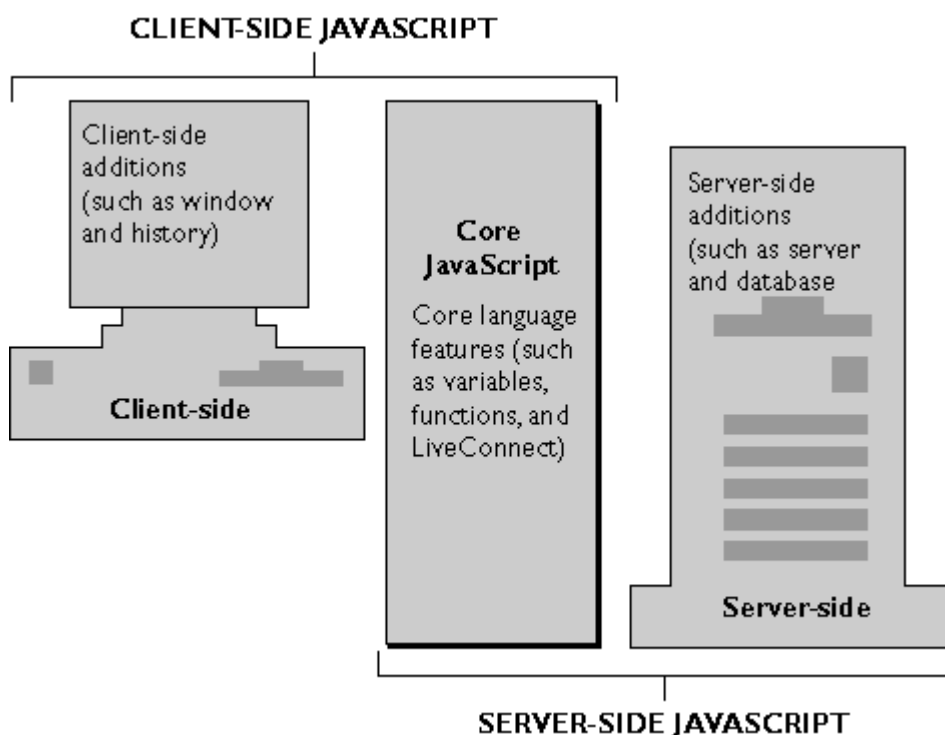
- 👉 一种语言的统称。该语言由 Brendan Eich 发明，最早用于 Netscape 浏览器。
- 👉 上述语言的一种规范化的实现。在 JavaScript 1.3 之前，网景公司将他们在 Netscape 浏览器上的该语言规范的具体实现直接称为 JavaScript，并以“Client-Side JavaScript”与“Server-Side JavaScript”区分它在浏览器 NN(Netscape Navigator)与 NWS(Netscape Web Server)上的实现——但后来他们改变了这个做法。

^① Brendan's keynote "JavaScript at Ten Years (Powerpoint)" for the ACM ICFP 2005 conference.
http://developer.mozilla.org/en/docs/JavaScript_Language_Resources

1.4.2. Core JavaScript

Core JavaScript 这个名词早在 1996 年(或更早之前)就被定义过，但它直到 1998 年 10 月由网景公司发布 JavaScript 1.3 时才被正式提出来。准确地说，它是指由网景公司和后来的开源组织 Mozilla，基于 Brendan Eich 最初版本的 JavaScript 引擎而发展出来的脚本引擎。是 JavaScript 规范的一个主要的实现者、继承者和发展者。

Core JavaScript 的定义如下：



在 JavaScript 1.3 发布时，Netscape 意识到他们不能仅仅以 Client/Server 来区分 JavaScript——因为市面已经出现了很多种 JavaScript。于是他们做了一些小小的改变：在发布手册时，分别发布“Core JavaScript Guide”和“Client-Side JavaScript Guide”。前者是指语言定义与语法规则，后者则是该语言的一种应用环境与应用方法。

所以事实上，自 1.3 版本开始，Core JavaScript 1.x 与 JavaScript 1.x 是等义的——换言之，我们现在常说的 JavaScript 1.x，就是指 Core JavaScript，而并不包括 Client-Side JavaScript。

需要注意一点的是，源于一些历史的因素，在 Core JavaScript 中，会有一

部分关于“LiveConnect 技术”的叙述及规范。这在其它(所有的)JavaScript 规范与实现中均是不具备的。

1.4.3. SpiderMonkey JavaScript

Brendan Eich 编写的 JavaScript 引擎最后由 Mozilla 贡献给了开源界，“SpiderMonkey”便是这个产品开发中的、开源项目的名称（code-name，项目代码名）。为了与我们通常讲述的 JavaScript 语言区分开来，我们使用 SpiderMonkey 来特指上述由 Netscape 实现的、Mozilla 及开源社区维护的引擎以及其规范。目前 SpiderMonkey JavaScript 已经发布了 1.7 版本的项目代码。

在本文此后的描述中，凡称及 SpiderMonkey JavaScript，将是特指于此；凡称及 JavaScript，将是泛指 JavaScript 这一种语言的实现。

1.4.4. ECMAScript

JavaScript 的语言规范被提交给 ECMA(标准化组织)去审定，并在 1997 年 6 月发布了名为 ECMAScript Edition 1 的规范，或称为 ECMA-262。四个月后，微软在 IE4.0 中发布了 JScript 3.0，宣称成为第一个遵循 ECMA 规范来实现的 JavaScript 脚本引擎。

而因为计划改写整个浏览器引擎的缘故，网景公司晚了整整一年才推出“完全遵循 ECMA 规范”的 JavaScript 1.3。请注意到这样一个问题：网景公司首先开发了 JavaScript 并提交 ECMA 标准化，但在市场的印象中，网景公司的 Core JavaScript 1.3 比微软的 JScript 3.0 “晚了一年”实现 ECMA 所定义的 JavaScript 规范。

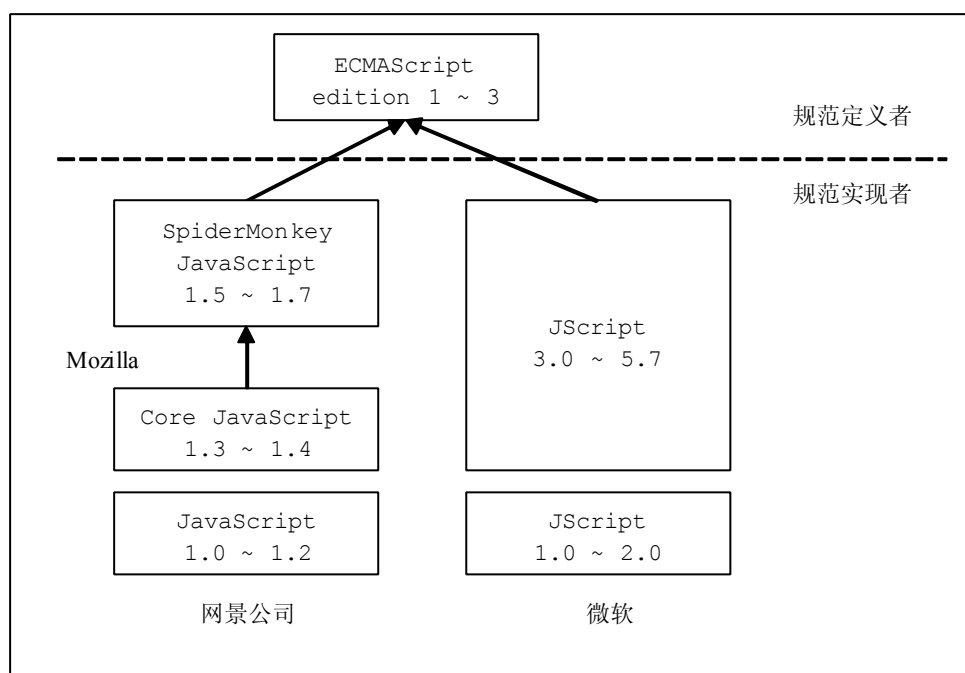
这直接导致了一个恶果：JScript 成为 JavaScript 语言的事实标准。

在本文此后的描述中，我们将基于 ECMAScript Editor 3 的规范来讲述 JavaScript。凡未特别指明的叙述中，所谓 JavaScript 即是指“一种符合 ECMAScript Editor 3 规范的 JavaScript 实现”。

1.4.5. JScript

微软在 1996 年按照也在 IE 中实现了一个与网景浏览器类似的脚本引擎，微软把它叫做 JScript 以示区别。结果 JScript 这个名字一直用到现在。一直到

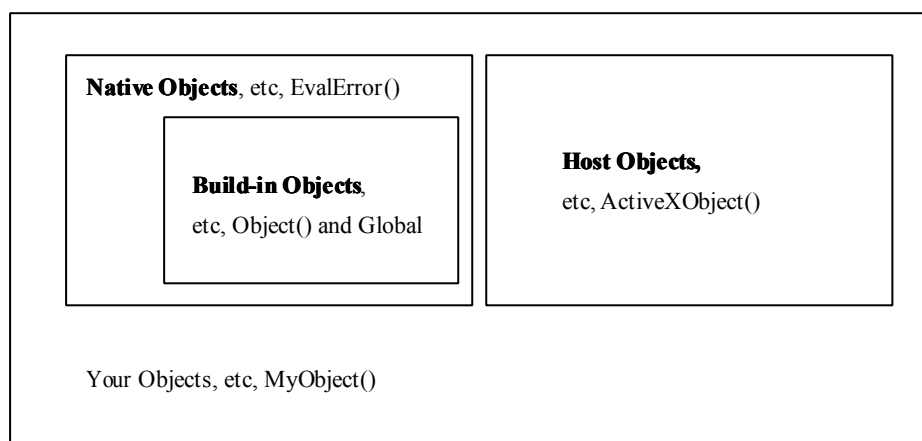
JScript 3.0 之后，JavaScript 语言的局面便显得明朗起来：



然而由于 JScript 成为 JavaScript 语言的事实标准，再有 Internet Explorer 浏览几乎占尽市场（如果我们现在不是在讨论 JavaScript，你也可以把这个因果颠倒过来），因此在 1999 年之后，WEB 页面上出现的脚本代码基本上都是基于 JScript 开发的，而 Core JavaScript 1.x 却变成了“（事实上的）被兼容者”。

直到 2005 年前后，源于 W3C、ECMA 对网页内容与脚本语言标准化的推动，以及 Mozilla Firefox 成功地返回浏览器市场，Web 开发人员开始注重所编写的脚本代码是否基于 JavaScript 的——亦即是 ECMAScript 的标准规范，这成为了新一轮语言之争的起点。

1.5. [XXX] JavaScript 的应用环境



1.5.1. 宿主环境(host environment)

我们通常希望一门语言是跨平台的。但事实上这非常难于做到，因为不同的平台提供的“可执行环境”不同。宿主环境就是为了隔离代码、语言与具体的平台而提出的一个设计。

虚拟机(Virtual Machine)是另一种隔离语言与平台环境的手段。现在 Java 与 .NET Framework 都以虚拟机的方式提供运行环境，前者提供 JVM(Java Virtual Machine)，后者则规范中间语言环境(CIL/IL, Common Intermediate Language)。

但 JavaScript 是一门设计得相对“原始”一点点的语言，他被创生的时候的最初目标仅仅是为 Netscape 提供一个在浏览器与服务器间都能统一使用的开发语言。简单的说，他原来只是想让 B/S 结构的开发人员用起来舒服那么一点点。但是，他一方面不能让浏览器上拥有一个巨大无比的运行期环境(例如象虚拟机那么大)，另一方面服务器端又需要一个较强大的环境。因此它就被设计成了需要“宿主环境”的语言。

然而，JavaScript 语言规范并没有对宿主环境提出明确的定义。比如说，它没有提出标准输入输出(std in, stdout)的要确切地实现在哪个对象中。为了弥补这个问题，RWC 在 WebAPIs 规范中首先就提出了“需要一个 Window 对象”的浏览器环境。这意味着在 RWC 或者浏览器端，是以 Window 对象以及其中的 Document 对象来提供输入输出。

这仍然不是全部的真相。因为“RWC 规范的宿主环境”，并不等同于“JavaScript 规范下的”宿主环境。为了让读者明白“没有规范”带来的后果，我们来对照一下几个常见的宿主环境下提供的 JavaScript 对象集与 ECMA Script 标准之间的差异。

(全局构造器)

ECMAScript 标准	Core JavaScript	JScript			WSH
Global					
Object					
Function					
Array					
String					
Boolean					
Number					
Math 对象					
Date					
RegExp					
Error					
Enumerator		*			
VArray		*			
ActiveXObject		*			
GetObject		*			
Java*.*					
Enumerator					IE
VArray					IE
ActiveXObject					IE
Debug	不是构造器				IE
Math	不是构造器				
Global	无构造器				
Arguments	无构造器				
Image	不是构造器(但可以用 new)				

除了对象集之外，不同的宿主环境提供的 JavaScript 语言的特性也并不一致。但这些语言特性应当归于运行期环境的范畴。

本书将不讨论与特定浏览器相关的任何细节。因此，我们事实上在说的的是一个公共语言环境，或者说公共的宿主环境的定义。作为一项基本输入输出的要求，本书设定宿主环境在全局应当支持如下方法：

方法	含义	注
----	----	---

alert(sMessage)	显示一个消息文本(字符串), 并等待用户一次响应。调用者将忽略响应的返回信息。	
confirm(sMessage)	(同 alert,)显示一个消息文本(字符串), 并在交互环境中等待用户一次响应。响应将返回布尔值 true/false, 调用者可以据此做后续操作。	
prompt(sMessage,sDefaultValue)	显示一个消息文本(字符串), 并在交互环境中等待用户一次输入和响应。如果用户取消响应, 则返回 undefined; 如果用户确认响应, 则返回字符串, 如果该字符串能被转换为数值, 则转换为数值返回。	
write(sText, ...)	输出一个段文本, 多个参数将被连接成单个字符串文本。	*
writeln(sText, ...)	(同 write,)输出一段文本, 多个参数将被连接成单个字符串文本。并在文本末尾追加一个换行符(\n)。	*

*注: write()与 writeln()在浏览器中是 document 对象的方法。因此, 为遵循这一惯例, 在本书的所有测试范例中并不直接使用这两个方法。但这里保留了它们, 以描述宿主环境的标准输入输出。

对于不同的宿主来说, 这些方法依赖于不同的对象层次的“顶层对象”。例如浏览器宿主依赖于 Window 对象, 而 WSH 宿主则依赖于 WScript 对象。但在本书中, 这些方法的调用将略去这个对象(实例), 不使用方法的全名。因此, 至少它看起来很象是 Global 对象上的方法(事实上, 大多数的宿主默认“顶层对象”不需要使用全名的约定)。例如:

```
// 示例 1: .NET Framework 使用 JScript 8.0, 顶层对象是 System
// (注: JScript.NET 中的脚本需要编译执行)
import System.Windows.Forms;
function alert(sMessage) {
    MessageBox.Show(sMessage);
}
alert('Hello, World!')

// 示例 2: 浏览器环境中使用的顶层对象是 window
alert('Hello, World!');

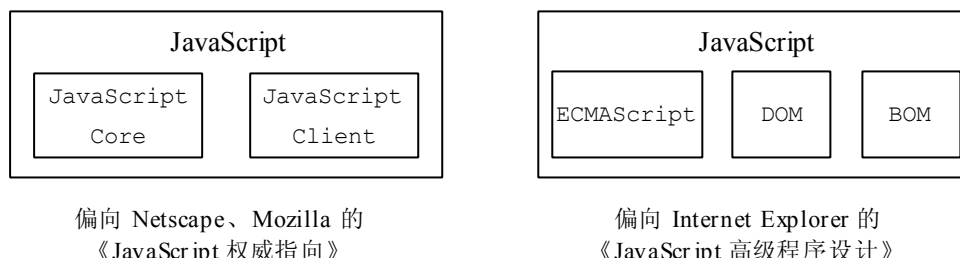
// 示例 3: WSH 环境中使用的顶层对象是 WScript, 但必须使用全名
function alert(sMessage) {
    WScript.Echo(sMessage);
}
alert('Hello, World!');
```

客户端 JavaScript (应用于 Netscape, mozilla 等)	JScript (应用于 IE)	JScript for WSH (应用于 Windows,IIS 等)
---	---------------------	--

1.5.2.外壳程序(shell)

1.5.3.运行期环境(runtime)

在不同的书籍中对 JavaScript 的阐释是不一致的。例如在《JavaScript 权威指南》中，它由“JavaScript 内核(Core)”和“客户端(Client) JavaScript”两部分构成；而在《JavaScript 高级程序设计》中，它被描述成由“核心(ECMAScript)”、“文档对象模型(DOM)”、“浏览器对象模型(BOM)”三个部分组成：



在本书中，(若非特别说明，)JavaScript 是指一种通用的、跨平台和跨环境的语言，并不特指某种特定的宿主环境或者运行环境。简单地说，它是指 ECMAScript 262 所描述的语言规范。

JavaScript 的运行期环境包括：

- 👉 一个对宿主的约定
- 👉 一个引擎内核
- 👉 一组对象和 API
- 👉 一些其它的规范

从这里来看，通常运行期环境其实就是由“脚本引擎(JavaScript Engines)”决定的。其中一些常见的引擎包括在 Windows 和 Internet Explorer 中的 jscript.dll，以及在 firefox 里面的 js3250.dll。基本上来说，目前开发界中常见的实现 ECMAScript 262-3 或 JavaScript1.5+ 的引擎包括：

引擎	应用	语言(*)	概述
SpiderMonkey	Netscape Navigator	C	
KJS	Konqueror, KDE	C++	
JavaScriptCore	Safari, Apple	C++	基于 KJS
NGS			
NJS		C	基于 NGS, 项目停止于 2001 年。
Rhino	for Mozilla	Java	Java
JScrip	Internet Explorer		
InScript	iCab	C++	
DMDScrip		D, C++	主要以兼容 JScript 为方向。
SEE			Simple ECMAScript Engine, 目前实现到 JavaScript 1.5, 一些编译开关可以使引擎兼容 JScript。
Resin		Java	Java
Narcissus	narrativejs	JavaScript	JavaScript 之父为验证该语言的自实现能力而写的一套代码. narrativejs 基于该项目实现了 JavaScript 上的解释器、编译器和语法扩展。
DMonkey		Delphi	对 JavaScript 2.0 有部分实现, 并扩展了大量用于 Win32 环境的内置对象。

(*) 开源或者提供源代码的开放项目。

其它一些实现 ECMAScript 262-4 或 JavaScript 2.0(JS2)的引擎有:

引擎	应用	语言(*)	概述
Epi metheus	Mozilla	C++	Mozilla 自 SpiderMonkey 之后的下一代引擎。在与 ECMAScript 262-4 在共同发展的过程中定义并实现着 JS2 特性。
JScript .NET	.NET Framework		微软自 JScript 5.6 自后推出的 JS2 引擎。
ActionScript 2.0	Flash		
Mono JScript	兼容 .NET 框架	C#	Mono Project 的一部分。
DotGNU JScript	兼容 .NET 框架	C#	DotGNU Project 的一部分。

其它一些实现 ECMAScript 262-4 或 JavaScript 2.0(JS2)的引擎有:

1.6. [XXX] 回顾

JavaScript 这个名词的多种含义包括:

含义	详述	注
----	----	---

JavaScript 脚本语言	一种语言的统称，由ECMAScript 262 规范。概含 Core JavaScript, JScript, ActionScript 等，而非特指其一。	
浏览器 JavaScript	在 Netscape 和 Mozilla 系列中，浏览器 JavaScript 被称为 Client-Side JavaScript。这包括 DOM、BOM 模型等在内的对象体系，但不确指具体脚本环境。这是目前 JavaScript 最为广泛的应用环境。	
Core JavaScript	仅指 Netscape/Mozilla 系列的浏览器环境中的 JavaScript。 Core JavaScript 是最常见含义之一，也是该语言的主要规范之一。 Core JavaScript 目前最新的实现版本为 1.7，试验版本为 2.0。	
JScript	仅指 Internet Explorer 系列的浏览器环境中的 JavaScript。 JScript 是使用最广泛的一种 JavaScript 脚本语言实现和主要规范之一。 JScript 目前最新的实现版本为 5.6。	
其它	在一些不多的场合中，被用来特指一种在 ECMAScript 262 标准下具体实现的“方言”。例如 Flash 中的 ActionScript, .NET 环境下的 JScript.NET，以及 Adobe Acrobat 中的 JavaScript 等。	*

*注：JavaScript 的方言是指 ECMAScript 262 的某一种具体的实现。这种语言其实被广泛的用于浏览器环境、WEB 服务器、操作系统和应用软件。但不同的环境与语言的实现版本之间都存在较大的差异。

第二章 JavaScript 的语法

当然可以从很多个方面来阐述“JavaScript 是怎样的一种语言”这个话题。但对于开发人员，最直接的感受总是来自于语言的语法与约定。

以对 JavaScript 的语法叙述来说，《JavaScript 权威指南》是最好的一本参考书。但我不能期望用户要读完那本厚厚的书才能阅读本书，因此我还是在这里讲述一下语法。

很多人、很多书会把浏览器、DOM 等等作为 JavaScript 的一部分进行讲述。然而在我看来，JavaScript 只是一种语言，由标识符、值、语句等等这样一些要素构成。而且本章（包括本书）面向的都是有一定开发经验的程序员，所以我仅讲述语法中的关键部分，并不打算讨论除此之外的一些细节。

由于本章是概述性质的，因此请留意每小节之前对内容的概括，以及汇总性的表格。它们可能是从另一个角度概述、汇总了相关的知识，因此或许出现与你正在阅读的书籍（或既有的知识）不一致的情况。但这些不一致，却是我们后面进一步讨论语言的基础。

1.1. 语法综述

一般来说，语言中的标识符可以分为两类，一类用于命名语法（的类型），一类用于命名值（的存储位置）。前者被称为“语法关键字”，后者则被称为“变量”和“常量”。

由此一来，就引入了一个概念：绑定。从标识符的角度来说，就分为语法关键字与语法（语义）逻辑的绑定，以及变量与它所存储值的位置的绑定。语法关键字对逻辑的绑定的结果，是作用域的限定；变量对位置的绑定的结果，则是变量生存周期的限定。

于是，我们看到了程序语言中“声明”的意义（而不是定义）：所谓声明，即是约定变量的生存周期和逻辑的作用域。

由于这里的“声明”已经概含了逻辑与数据（这相当于“程序”的全部），因此整个编程的过程，其实被解释成了“说明逻辑和数据”的过程：

- 👉 纯粹陈述“值”的过程，被称为变量和类型声明；
- 👉 纯粹陈述“逻辑”的过程，被称为语句（含流程控制子句）；

👉 陈述“值与（算法的）逻辑”的关系的过程，被称为表达式。

下表阐述主要标识符与其语义关系：

	标识符	子分类	JavaScript 示例(部分)
值 相 关	类型		(无显式类型声明)
	变量	直接量 对象	null undefined new Object()
与 逻 辑 和 值 都 相 关	表达式(*)	值运算 对象存取	v = 'this is a string.' obj.constructor
	逻辑语句(**)	顺序 分支 循环	v = 'this is a string.'; if (false) { // ... }
逻 辑 相 关	流程控制语句	标签声明 一般流程控制子句 异常	break; continue; return; try { // ... } watch (e) { // ... }
	其它	注释	(略)

(*) 表达式首先是与“值”相关的，但因为存在运算的先后顺序，因此它也有与“逻辑”相关的含义。

(**) 在 JavaScript 中，逻辑语句是有值的，因此它也是值相关的。这一点与其它多数语言都不一样。

1.1.1. 识别语法错误与运行错误

一般来说，JavaScript 引擎会在代码装入时先进行语法分析，如果语法分析通不过，整个脚本代码块都不执行；当语法分析通过时，才会执行这段脚本代码。若在执行过程中出错，那么在同一代码上下文中、出错点之后的代码将不再执行。

不同引擎处理这两类错误的提示的策略并不相同。例如在 Internet Explorer

的 JScript 脚本引擎环境中，两种错误的提示大多数时候看起来是一样的。要在不同的脚本引擎中简单地区别二种错误，较为通行的方法是在代码片断的最前面加上一行输出，例如使用 `alert()` 来显示一个信息^①。脚本引擎的出错提示在该行之前，则是语法分析期错。例如：

```
var head = 'alert("loaded.");';

// 示例 1: 声明函数的语法错误
var code = 'function func(){};';
eval(head + code);
```

如果在该行之后，则是执行期错。例如：

```
var head = 'alert("loaded.");';

// 示例 2: 执行时发现未定义 value 变量，触发运行期错误
var code = 'value++;';
eval(head + code);
```

我们在这里强调这两种错误提示，是因为本章主要讨论语法问题。在特定脚本引擎中，一段代码是否执行异常或是否语法分析错误，是需要通过上述的方法来区分的。但如果有该引擎下的调试器，或脚本宿主环境允许加载用户的错误处理代码，也是还有其它（较为复杂的）方法的。

1.2. JavaScript 的语法：变量声明

JavaScript 是弱类型语言。但所谓弱类型语言，只表明该语言在表达式运算中不强制校验运算元的数据类型，而并不表明该语言是否具有类型系统。所以有些书在讲述 JavaScript 时说它是“无类型语言(untype language)”，其实是不正确的，因此本小节将对 JavaScript 变量的数据类型做一个概述。

一般来说，JavaScript 的变量可以从作用域的角度分为全局变量与局部变量。为了便于本小节的叙述，读者可以简单地认为：所谓全局变量是指在函数外声明的变量，局部变量则是在函数或子函数内声明的变量。更详细的变量作

^① 事实上更好的做法是使用集成调试环境在这里设置一个断点。这些集成调试或集成开发环境，包括在 Microsoft JScript 中的 Script Debugger 或 Script Editor、Netscape 的 JavaScript Debugger、Mozilla 的 Firefox 插件、Adobe 的 Adobe ExtendScript Toolkit 等等。自从 1998 年以来，我用过上述所有的调试环境以在不同的宿主或引擎中开发脚本，然而直到现在都还有开发人员抱怨“JavaScript 无法单步调试”。这实是一件令人无可奈何的事情，因为本书的重点并不在于 step by step 地教会开发人员找到以及使用某种开发工具。

用域问题，将在“1.2.4 模块化的效果：变量作用域”章节中讲述。

1.2.1. 变量的数据类型

相对于 Pascal、C 等语言，JavaScript 没有明确的类型声明过程。所以事实上在 JavaScript 约定的保留字列表中，根本就没有 `type` 或 `define` 关键字。

JavaScript 识别 6 种数据类型，并在运算过程中自动处理语言类型的转换。这些类型包括：

类型	含义	说明
undefined	未定义	未声明的变量，或声明过但未赋值的变量的值，会是 <code>undefined</code> ；可以显式或隐式地给一个变量赋值为 <code>undefined</code> 。
number	数值	除赋值操作之外，只有数值与数值的运算结果是数值；一些函数 / 方法的返回值是数值。
string	字符串	不能直接读取或修改字符串中的单一字符。
boolean	布尔值	<code>true/false</code>
function	函数	JavaScript 中的函数存在多重含义。(1)
object	对象	基于原型继承的面向对象。(2)

任何一个变量或值的类型都可以(而且应当首先)使用 `typeof` 运算来得到。`typeof` 是 JavaScript 内部保留的一个关键字，与其它语言不一样的是，`typeof` 关键字可以象函数调用一样，在后面跟上一对括弧。例如：

```
// 示例 1: 取变量的类型
var str = 'this is a test.';
alert(typeof str);
// or
// alert(typeof(str));

// 示例 2: 对直接量取类型值
alert(typeof 'test!');
```

但这个 `typeof()` 中的括弧，只是产生了一种“使 `typeof` 看起来象一个函数”的假象。关于这个假象的由来，我随后会在“1.6 运算符的二义性”再讲——现在，你只需要知道，它的确可以这样使用就足够了。

`typeof` 运算总是以字符串形式返回上述 6 种类型值之一。如果不考虑 JavaScript 的中的面向对象编程，那么这个类型系统的确是足够简单的。

1.2.1.1. 值类型与引用类型

变量不但有数据类型之别，而且还有值类型与引用类型之别——这种分类方式主要约定了变量的使用方法。在 JavaScript 的六种类型中：

数据类型	值/引用类型	备注
undefined	值类型	无值。
number	值类型	
boolean	值类型	
string	值类型	???
function	引用类型	
object	引用类型	

在 JavaScript 中，“全等 (===) 运算符”用来对值类型/引用类型的实际数据进行比较和检查。按照约定：

- 👉 一般表达式运算的结果总是值(或 undefined)；
- 👉 函数/方法调用的结果可以返回值或者引用(或 undefined)；
- 👉 值与引用、值与值之间即使等值(==)，也不一定全等(===)；
- 👉 两个引用之间如果等值(==)，则一定全等(===)。

至少表面上来看，一个值应该与其自身“等值/全等”。但事实上，在 JavaScript 中这存在一个例外：一个 NaN 值，与自身并不等值，也不全等。

JavaScript 中的值类型与引用类型，同其它通用高级语言(或象汇编语言一样的低级语言)一样，表达的含义是数据在运算时的使用方式：参与运算的是其值亦或其引用。因此，在下面的示例中，两次调用函数 func()时，各自传入的数据采用了不同的方式：

```
var str = 'abcde';
var obj = new String(str);

function newToString() {
    return 'hello, world!';
}

function func(val) {
    val.toString = newToString;
}
```

// 示例 1: 传入值

```
func(str);
alert(str);

// 示例 2: 传入引用
func(obj);
alert(obj);
```

(从语义上来说，)由于在示例 1 中实际只传入了 `str` 的值，因此对它的 `toString` 属性的修改是无意义的；而在示例 2 中传入了一个对象的引用，因此对它的 `toString` 属性修改将会影响到后来的运算。

所以我们看到这两个示例返回的结果不一致。

关于“为什么可以修改一个值类型数据的属性”这个问题，以及 `str` 在传入 `func()` 后发生的一些细节，我们在“**错误！未找到引用源。错误！未找到引用源。**”中会进一步叙述。

1.2.2. 变量声明

JavaScript 中的变量声明有两种方法：

- 👉 显式声明
- 👉 隐式声明(即用即声明)

所谓显式声明，一般是指用关键字 `var` 进行的声明。例如：

```
// 声明变量 str 和 num
var str = 'test';
var num = 3 + 2 - 5;
```

显式声明也包括一些语句中使用 `var` 进行的显式声明，例如 `for` 语句：

```
// 声明变量 n
for (var n in Object) {
    // ...
}
// 声明变量 i, j, k
for (var i, j, k=0; k<100; k++) {
    // ...
}
```

显式声明的最后一种情况是函数中的函数名声明。例如：

```
// 声明函数 foo
function foo() {
```

```
str = 'test';  
}
```

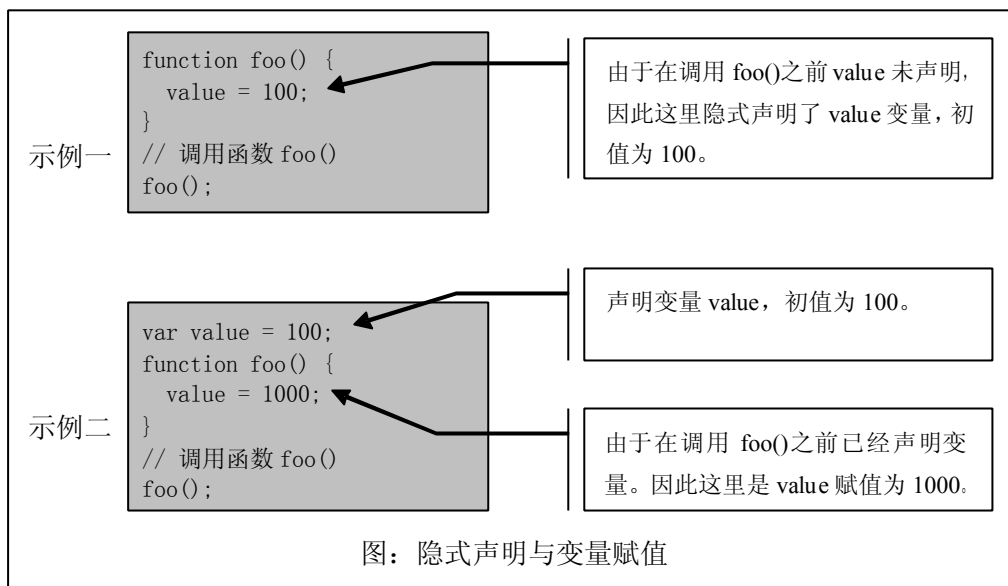
而隐式声明则不用关键字 `var` 声明。例如：

```
// 当 aVar 未被声明时，以下语句将隐式地声明它  
aVar = 100;
```

解释器总是将显式声明理解为“变量声明”。而对隐式声明则不一定：

- 👉 如果变量未被声明，则该语句是变量声明（并立即赋给值）。
- 👉 如果变量已经声明过，则该语句是变量赋值语句。

如下图所示：



1.2.3. 变量声明中的一般性问题

变量声明通常具有两个方面的特性，一是声明类型，二是声明初值。但 `JavaScript` 中没有类型声明的概念，因此变量声明仅限于说明一个量的初值。在声明中，等式右边既可以是表达式——这意味着将表达式运算的结果作为该变量的初值，也可以是更为强大和灵活的直接量声明。例如：

```
var str = 'test';      // 'test' 是直接量  
var num = 3 + 2 - 5;   // 3+2-5 是一个表达式
```

直接量类似于汇编语言中的立即值，是无需声明就可以立即使用的常值。本小节中主要讲述直接量声明的方法——当然直接量也可以作为表达式运算的运算元使用，这与其它语言中一致的。下表简要说明 `JavaScript` 中能进行直

接量声明的数据类型和对象：

	类型	直接量声明	包装对象	说明
基本类型	undefined	v = undefined	无	
	string	v = '...' v = "..."	String	参见：字符串直接量、转义符
	number	v = 1234	Number	参见：数值声明
	boolean	v = true v = false	Boolean	
	function	v = function() { ... }	Function	参见：函数直接量声明
对象	object	v = { ... } v = null	Object	
	array	v = [...]	Array	参见：正则表达式的常见问题
	regexp	v = / ... /...	RegExp	参见：数组的常见问题

本小节的部分问题参见：

- 👉 《JavaScript 权威指南》
- 👉 1.5.1.2 对象直接量声明
- 👉 1.4.2.1 语法声明与语句含义不一致的问题
- 👉 1.7.4 值类型之间的转换
- 👉 1.5 包装类：面向对象的妥协

1.2.3.1. 字符串直接量、转义符

你总是可以用一对双引号或一对单引号来表示字符串直接量。在早期 Netscape 的 JavaScript 中允许出现非 Unicode 的字符^①，但现在 ECMA 标准统一要求 JavaScript 中的字符串必须是 Unicode 字符序列。

转义符主要用于在字符串中包含控制字符，以及当前操作系统语言（以及字符集）不能直接输入的字符；也可以用于字符串中嵌套引号（不过你总是可以在单引号声明的字符串中直接使用双引号，或反过来在双引号中使用单引号）。转义符总是用一个反斜线字符“\”引导，包括如下转义序列：

转义符	含义	转义符	含义
\b	退格符	\'	单引号
\t	水平制表符	\"	双引号
\v	垂直制表符	\\	反斜线字符

^① 在早期 Netscape 的 JavaScript 中编程时，使用 `escape()` 可能会解出单字节的编码，而使用 `unescape()` 时则可能在字符串中包含相应的 ASCII 字符。即使是在 Internet Explorer 的早期版本（for Windows 9x）中，也可能由于 cookies 存取而出现在字符串中存在 ASCII 字符序列的情况。

\n	换行符	\0	字符 NUL (编码为 0 的字符)
\r	回车符	\xnn	ASCII 字符编码 nn 的字符 (*)
\f	换页符	\unnnn	UNICODE 字符编码为 nnnn 的字符

(*) 将被转换为 UNICODE 字符存储。

除了定义转义符之外，当反斜线字符“\”位于一行的末尾（其后立即是代码文本中的换行）时，也用于表示连续的字符串声明。这在声明大段文本块时很有用。例如：

```
1   var aTextBlock = '\
2   abcdefghijklmnopqrstuvwxyz\
3   \
4   123456789\
5   \
6   +-*/*';
```

注意第三行与第五行中各包括一个空格，因此输出时第二、四、六行将用一个空格分开。显示为：

```
abcdefghijklmnopqrstuvwxyz 123456789 +-*/*
```

在这种字符串表示法中也可以使用其它转义符——只要它们出现在文本行最后的这个“\”字符之前即可。而且与一般习惯不同的是，不能在这种表示的文本行末使用注释。

另外一个需要特别说明的是“\0”，它表示 NUL 字符。在某些语言中，NUL 被用于说明一种“以#0 字符结束的字符串”（这也是 Windows 操作系统所直接的一种字符串形式），这种情况下字符串是不能包括这个 NUL 字符串的。但在 JavaScript 中，这是允许存在的。这时，NUL 字符是一个真实存在于字符序列中的字符。下例说明 NUL 字符在 JavaScript 中的真实性：

```
// 或
// str = String.fromCharCode(0, 0, 0, 0, 0);
str = '\0\0\0\0\0';

// 显示字符串长"5", 表明 NUL 字符在字符串中是真实存在的
alert(str.length);
```

在 JavaScript 中也可以用一对不包含任意字符的单引号与双引号来表示一个空字符串（Null String），其长度值总是为 0。比较容易被忽视的是，空字符串与其它字符串一样也可以用作对象成员。例如：

```
obj = {
  '': 100
```

```
}  
// 显示该成员的值: 100  
alert(obj['']);
```

1.2.3.2. 数值直接量

数值直接量总是以一个数字字符，或一个点字符“.”，以及不多于一个的正值符号“+”或负值符号“-”开始。当以数字字符开始时，它有三条规则：

- 👉 如果以 0x 或 0X 开始，则表明是一个十六进制数值；
- 👉 如果仅以 0 开始，则表明是一个八进制数值；
- 👉 其它情况下，表明是一个十进制整数或浮点数。

当以点字符“.”开始时，它总是表明一个十进制浮点数。正值符号“+”、负值符号“-”总是可以出现在上述两种表示法的最前面。例如：

```
1234      // 十进制整数  
01234     // 八进制整数  
0x1234    // 十六进制整数  
-0X1234   // 负值的十六进制整数  
+100      // 正值的十进制整数
```

当一个直接量声明被识别为十六进制数值时，该直接量由 0..9 和 A..F 字符构成；被识别为八进制数值时，该直接量由 0..7 字符构成——如果在语法分析中发现此外其它字符，则出现语法分析错。

当一个直接量被识别为十进制整型数时，它内部的存放格式可能是浮点数，也可能是整型数，这取决于不同引擎的实现。因此不能指望 JavaScript 中的整型数会有较高的运算性能^①。但是你可以用位运算来替代算术运算，这时引擎总是以整型数的形式来运算的（即使运算元是一个浮点数）。

当一个直接量被识别为十进制时，它可以由 0..9，以及(不多于一个的)点字符“.”或字符 e、E 组成。当包括点字符“.”、字符 e、E 时，该直接量总被识别为浮点数（注意某些引擎会优化一些直接量的内部存储形式）。例如：

```
3.1415926  
12.345  
.1234  
.0e8  
1.02E30
```

^① 在 JScript 中，引擎对直接量会进行特别的分析并以最优的形式来存放它。例如值 100 与值 100.0 在 JScript 中其实都是以一个 LongInt 类型的整数值存来存放的，而 100.1 则以 Double 类型的浮点数来存放。

当使用带字符 e、E 的指数法表示时，也可以使用正、负符号来表示正、负整数指数。例如：

```
1.555E+30  
1.555E-30
```

1.2.3.3. 函数直接量声明

函数直接声明的语法是：

```
function functionName()  
    // ...  
}
```

当 `functionName` 是一个有效标识符时，表示声明一个具名函数；如果省略，则表示声明一个匿名 (anonymous) 函数。`functionName` 后使用一对不能省略的 “()” 来表示形式参数表。所谓形式参数，是指一个可以在函数体内部使用的、有效的标识符名。你可以声明零个至任意多个形式参数，即使你在函数体内部并不使用它。或者你也可以不声明形式参数，这时也可以在函数体中使用一个名为 `arguments` 的内部对象来存取调用中传出的实际参数。

函数体中可以有零至任意多行代码或内嵌的（子级的）函数。如果在函数体内出现显式变量声明，则视为函数体内部的局部变量；该函数的内嵌函数的名字，也作为局部变量。

在 JScript 中，所有在代码中出现的具名函数（直接量）声明，都将视为所在的语法作用域中的一个变量标识符。这对 SpiderMonkey JavaScript 来说存在一项限制：表达式中具名函数只识别为匿名函数而忽略它的函数名。下例所示的代码在 SpiderMonkey JavaScript 的任意位置都不是具名函数——而在 JScript 中将有一个具名函数声明^①：

```
(function foo()  
    // ...  
})();  
alert(foo);
```

1.2.3.4. 正则表达式的常见问题

正则表达式由普通字符（字符 a~z、0~9 等等）和元字符构成。JavaScript

^① 这个问题将在“1.4.2.1 语法声明与语句含义不一致的问题”中予以更详细地讨论。

实现了正则表达式的一个子集，这个子集包括如下元字符^①：

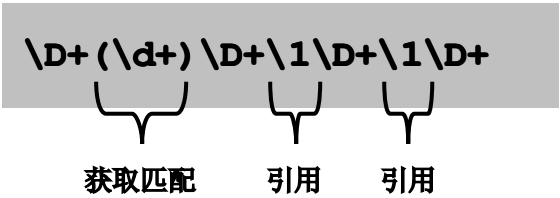
匹配对象	元字符	备注
字符子集	\d, \D, \s, \S, \w, \W	每个元字符只能匹配一个字符。若该元字符表示一个字符子集，则匹配子集中的任一字符。 在使用“自定义匹配字符子集”时，可以使用 'a-z' 格式来指定连续子集。
单个字符	\f, \n, \r, \t, \v, .	
一般性转义字符	\\, \[, \{ 等	
位置	^, \$, \b, \B	
控制字符	\cx	
十六进制 ASCII 字符	\xn	
八进制 ASCII 字符	\n, \nm, \nml	
十六进制 UNICODE 字符	\unnnn	
自定义匹配字符子集	[xyz], [^xyz]	
匹配方式	元字符或语法	
匹配分组	(), (?:), (?:=), (?!)	获取、非获取以及正、负向预查等。
匹配 x “或” 匹配 y	x y	
匹配次数设定	*, +, ?, {n}, {n,}, {n,m}	当 "?" 紧临 "匹配次数设定" 之后，表明非贪婪模式。缺省为贪婪模式。
非贪婪模式设定	?	
引用匹配	元字符	
引用一个已获取的匹配	\n	(*)

(*) 是指在一个正则表达式中复用已通过“匹配分组”获取的、文本中的子字符序列。它的指定格式与“八进制 ASCII 字符”是冲突的。当发生歧义时，优先理解为“获取匹配”；若没有足够的匹配数，则理解为“八进制 ASCII 字符”。

除了上表备注中说明的一些常见问题之外，不太常见的用法是在正则表达式中的“引用匹配”——我们可能会在 `String.replace()` 等方法中使用 `$xx` 来引用某个已获取的匹配，但那是在正则表达式之外进行的。而在正则表达式内的“引用匹配”是指如下这种情形：

```
// 有字符串如下格式
'abcd1234cdef.....1234.....1234....'
'abcd182349cdef.....182349.....182349....'
```

若试图表达上述重复出现的子匹配。那么可以使用这样的表达式：



就可以匹配上面列举的两个字符串，而不匹配下面这种：

^① 该表仅是 JavaScript 中正则表达式语法的一个概要，详细说明请参见 Microsoft JScript 手册或《JavaScript 权威指南》，以及《精通正则表达式》等资料或书籍。

```
// 不匹配的格式(注意数字不重复)
'abcd1234cdef.....1111.....11111111....'
```

与这种不常见的引用方式相反的另外一种情况，则是经常犯的错误。该错误源于正则表达式具有单独语法格式，而非一个字符串，因此当开发人员试图直接将正则表达式用在字符串中——例如用这样的字符串来创建对象时，就会出现一些意料之外的问题。最常见的情况是这样：

```
// 有正则表达式直接量如下
rx = /abcd\n\r/gi
```

开发人员试图将它修改成一个正则表达式对象的创建，于是直接复制了上述代码，修改如下：

```
// 使用字符串创建的正则表达式对象
rx = new RegExp('abcd\n\r', 'gi');
```

在开发人员的预期中，这两个正则应该是一样的。然而这正好忽略了一个问题：在字符串中“\”也是转义符，因此当使用转义后的字符串'abcd\n\r'来创建正则式对象时，就出现了错误。而在很多情况下，该错误既非语法错，也不导致运行错——它只是与开发人员的预期不一致而已，所以是被忽略的。而这，可能在代码中留下巨大的隐患。

解决该问题的方法是为字符串中的“\”增加转义，因此上例应被修改为：

```
rx = new RegExp('abcd\\n\\r', 'gi');
```

1.2.3.5. 数组的常见问题

JavaScript 可以用直接量声明一个数组，而且这个数组可以是异质的（数组元素的类型可以不同）、交错的（数组元素可以是不同维度的数组）。数组的交错性使它看起来象是“多维的”，但事实上不过是“数组的数组”这种嵌套特性。换成最直接的表达式方式，我们并不能用类似于：

```
arr = new Array(10, 10);
```

这样的方式来得到一个每维 10 个分量的二维数组（JavaScript 中上述语法会得到包含两个元素的一个一维数组），但可以用：

```
arr = [[1,2],[3,4]];
rx = new RegExp('abcd\\n\\r', 'gi');
```

这样的方法来得到一个交错的数组——数组的数组，尽管它的大小（以及表达的数学含义）也是 2*2 的，但在数据结构上的本质上并不具备某些多维数组的

特性^①。

从表面上来看，你也可以在 JavaScript 的数组中使用如下语法：

```
arr[1,2,3]
```

但这种语法并不返回某个多维数组指定为“1,2,3”的下标元素，而只是返回 arr[3]这个元素。因为 JavaScript 将“1,2,3”解释为连续运算，并返回最后一个表达式的值“3”（表达式运算的细节可以参考“1.3JavaScript 的语法：表达式运算”和“1.6.5 逗号“,”的二义性”）。如果你试图访问交错数组的某个下标分量，应该用类似如下的语法：

```
arr[1][2][3]
```

使用数组下标时也可以使用数值字符串，但这时在语义上却并不是对索引数组的下标存取，而是对关联数组中的“名-值”存取。这是因为 JavaScript 中的一个数组，既是一个以数组下标顺序存取的索引数组，也是一个可存取属性的关联数组。为了减少二者之间的差异，在将数组视为普通对象并用 for..in 语句列举时，可以列举到那些数值的索引下标。有关数组的这一特性，请参见如下章节：

- 👉 1.5.2 对象成员列举、存取和删除
- 👉 1.6 关联数组：对象与数组的动态特性
- 👉 1.9 实现二叉树

1.3.JavaScript 的语法：表达式运算

在 JavaScript 中，运算符大多数是特殊符号，但也有少量单词。例如我们在前面用来取数据类型的 typeof，其实就是一个运算符。相较与其它语言，JavaScript 在运算符上还有一种特殊性：许多语句/语法分隔符也同时是运算符——它们的含义当然是不同的，我的意思只是强调它们使用了相同的符号。下表列举这些单词形式的运算符，你应当避免把它们误解成语句：

运算符/符号		运算符含义	备注
单词形式的运算符	typeof	取变量或值的类型	参见： 1.3.7 特殊作用的运算符
	void	运算表达式并忽略值	
	new	创建指定类的对象实例	与面向对象相关。参见： 1.5 面向对象编程的语法概要
	in	检查对象属性	
	instanceof	检查变量是否是指定类的实例	

^① 《程序设计语言—实践之路》p374，从技术上说，连续布局才是真正的多维数组，而“行指针”只是指向数组的指针数组——亦即是数组分量是一个其它数组的引用（在本文中的被称为数组的数组、交错数组）。

	delete	删除实例属性	
--	--------	--------	--

表达式由运算符与运算元构成。运算元除了包括变量，还包括函数(或方法)的返回值，此外也包括直接量。

JavaScript 中可以存在没有运算符的表达式，这称为“单值表达式”。单值表达式有值的含义，表达式的结果即是该值。除单值表达式之外，一个 JavaScript 表达式中必然存在至少一个运算符。运算符可以有一至三个运算元，没有运算元的、孤立于代码上下文的运算符是不符合语法的。

通过对表达式的考察，我们发现 JavaScript 的表达式总是有结果值——一值类型或引用类型的数据，或者 `undefined`。单值表达式的结果是值本身，其它表达式的结果是运算的结果值（也因此必然有运算符）。

复合表达式由多个表达式连接构成。如前所述，由于每一个独立的表达式都总是有结果值，因此该值能作为“邻近”的表达式的运算元参与运算。有了这样的关系，我们就总是可以将无限个的表达式“邻近”的连接成复合表达式，该复合表达式的运算结果，也其它普通的表达式一样：值类型或引用类型的数据，或 `undefined`。

如同数学含义上的“运算”存在优先性（例如乘除法优先于加减法），复合表达式在“按从左至右的”邻近关系运算的同时，也受到运算符的优先级的影响。为了让这种运算次序可控，就有了强制优先级运算符。而有了缺省次序、优先级次序和强制优先级，则表达式就存在了算法的逻辑上的含义——这就是本章“1.1 语法综述”中说表达式既有值的含义，也有逻辑的含义的原因。

但是，表达式的本质目的，还是在于求值。尤其在 JavaScript 中，表达式的作用是在于求值的。关于这一点，我们将在后面的篇章中介绍。

通过对值的含义的考察，我们会注意到，所有 JavaScript 表达式的运算结果，要么产生一个“基本类型的值”，要么产生“对一个对象实例的引用”。根据运算元和结果值的不同，我们对运算符做一个简单分类：

分类	说明	运算符示例	运算元	目标类型	章节
计算运算	一般性的数值运算	+ - * / %	number	number	1.3.1
按位运算	数值的位运算	~ & ^ << >> >>>			
逻辑运算	布尔值运算	! &&	boolean	boolean	
	值逻辑运算	&&	(运算元)	(运算元)	1.3.2

字符运算	(仅有) 字符串连接	+	string	string	1.3.3
等值检测	检测两个值是否相等	== != === 等	*	boolean	1.3.4
赋值运算	一般赋值和复合赋值	= += 等	*	*	1.3.5
函数调用	(仅有) 函数调用	()	function	*	1.3.6
对象	对象创建、存取、检查等	. [] new	object	*	1.7(*)
其它	表达式运算、typeof 运算等	void typeof 等	(表达式等)	*	1.3.7

(*)有关对象的运算、语法等，我们在“1.7 面向对象编程的语法概要”章节中讲述。

1.3.1. 一般表达式运算

无论在何种语言中，一般表达式运算总是一个很大的分类。今后我们会讲到这种状况的成因，但现在，我们只需要概要地讲述它们。

JavaScript 中的一般表达式运算只操作两种运算数：数值和布尔值。这些运算的运算元与结果值总是同一类型的。对数据结构、存储系统或计算原理有一些基本了解的人都应该知道，这两种数据通常是可以被存储在基本的存储单元中，并参与 CPU 指令运算的^①。

除了“加减乘除”这类一般性的数值运算之外，这里说的“一般表达式运算”也包括数值的位运算。位运算操作中，JavaScript 强制运算目标为一个有符号的 32 位整型数^②：如果目标是非数值，那么会被强制转换为数值；如果目标是浮点数，那么会被取整；否则，将目标识别为有符号整型数。

在这里，我们强调一般表达式中的逻辑运算仅指“布尔值运算”，因此它的运算元和结果值都必然是布尔值。当然在使用中，这可能由编译器进行了类型转换，例如如下表达式：

```
!aVar
```

由于“逻辑否(!)”运算符强制运算元为 aVar，因此无论 aVar 是其它何种类型，都将被转换为 bool 值参与运算。JavaScript 中的类型转换是一种动态语言特性，因此有关它的细节请参考“1.7 类型转换”。

对于不同的硬件系统，数值和布尔值的表示法、表示范围都可能不同。但对于 JavaScript 这种运行在解释系统中的语言来说，他会通过一些约定来清除硬件差异的影响。关于这一点，请参见“1.2.3 变量声明中的一般性问题”。

^① 这并表明 JavaScript 采用这种方式存储或在运算中传送这些运算数，但这的确是 JavaScript 保留这两种值类型的原因之一。

^② 这里的转换、识别规则是非常复杂且与 JavaScript 版本相关的，因此本书只是（并不准确地）概述了几种可能性。有关转换的细节，请参考《JavaScript 权威指南》“5.8 逐位运算符”小节。

1.3.2. 逻辑运算

一般语言中，逻辑运算与布尔运算是等义的，其运算元与目标类型都是布尔值（true/false）。JavaScript 当然支持这种纯布尔运算，上一小节已经对此有过叙述。不但如此，JavaScript 还包括另外一种逻辑运算，它的表达式结果类型是不确定的。

只有“逻辑或(||)”和“逻辑与(&&)”两种运算能做这样的事。它们的使用方法与运算逻辑都与基本的布尔运算一致，例如：

```
var str = 'hello';
var obj = {};
x = str || obj;
y = str && obj;
```

这种运算的特别之处在于：运算符“||”与“&&”既不改变运算元的数据类型，也不强制运算结果的数据类型。除此之外，还有以下两条特性：

- ☞ 运算符会将运算元理解为布尔值，以进行布尔运算；
- ☞ 运算过程（与普通布尔运算一样）是支持布尔短路的。

由于支持布尔短路，因此在上例中“str || obj”表达式只处理第一个运算元就可以有结果，其结果值是 str——转换为布尔值时为 true，不过由于前面所述的“不强制运算结果的数据类型”，所以表达式的结果值仍是“str”。同样，若以“str && obj”为例，其返回结果值就会是“obj”了。

这种逻辑运算的结果一样可以用在任何需要判断布尔条件的地方，包括 if 或 while 语句，以及复合的布尔表达式中。例如：

```
(续上例)

// 用于语句
if (str || obj) {
    ...
}

// 用于复杂的布尔表达式
z = !str && !(str || obj);
```

由于表达式的运算元可以是值或其它表达式（包括函数调用等），因此连续的逻辑运算也可以用来替代语句。这也是一种被经常提及的方法，关于这一点，请参考如下章节：

- 👉 1.3.3.1 通过表达式消灭分支语句
- 👉 1.15 使用更复杂的表达式来消滅 IF 语句

1.3.3. 字符串运算

JavaScript 中的字符串有且只有一种“字符串运算”，就是字符串连接，该运算相应的运算符是加号“+”——但的确还有其它几种运算可以作用于字符串。符号“+”却还可以被用在其它两个地方：一元正值运算符和数值加法运算符。因此一个带有符号“+”表达式是否是“字符串连接”运算，取决于它在运算时存在几个操作数，以及每个操作数的类型。

字符串连接运算总是产生一个新的字符串，它在运算效果（结果值）上完全等同于调用字符串对象的 `concat()` 方法。

你不可以直接地修改字符串中的指定字符。

字符串其实还可以参与其它运算，例如比较、等值、赋值等运算。但在这里，我们把它放在相应的其它分类中讲述。

1.3.4. 比较运算

1.3.4.1. 等值检测

等值检测的目的是判断两个变量是否相同或相等。我们说相同与不相同，是指运算符“`===`”和“`!==`”的运算效果；说相等与不相等，是指运算符“`==`”和“`!=`”的运算效果。

具体来说，等值检测是指如下运算符的运算效果：

名称	运算值	说明
相等	<code>==</code>	比较两个表达式，看是否相等
不等	<code>!=</code>	比较两个表达式，看是否不相等
严格相等	<code>===</code>	比较两个表达式，看值是否相等并具有相同的数据类型
不严格相等	<code>!==</code>	比较两个表达式，看是否具有不相等的值或数据类型不同

对于等值检测来说，最简单和有效率的方法当然是比较两个变量引用（所指向的内存地址）。但这并不准确，因为我们显然会在两个不同的内存地址上存放同样的数据，例如两个相同的字符串。因此，比较引用虽然高效，但很多时候，我们却要比较两个变量的值。在下面的讨论中，我们会先忽略“比较引用”时的问题，侧重讲述值的比较。

我们先讨论等值检测中“相等”的问题。它遵循如下的运算规则：

类型	运算规则
两个值类型进行比较	转换成相同数据类型的值进行“数据等值”比较。
值类型与引用类型比较	将引用类型的数据转换为与值类型数据相同的数据，再进行“数据等值”比较。
两个引用类型比较	比较引用（的地址）。

运算规则中所谓“数据等值”，是仅指针对“值类型”的比较而言。表明比较变量所指向的存储单元中的数据（通常指“内存数据”）。

在三种值类型（数值、布尔值和字符串）中，数值和布尔值的“数据等值”检测开销都很小，但对字符串检测时就会存在非常大的开销。因为必须对字符串中的每一个字符进行比较，才能判断两个字符串是否相等。

下面的代码说明两个值类型的字符串检测：

```
var str1 = 'abc' + 'def';
alert(typeof str1); // 显示'string'

var str2 = 'abcd' + 'ef';
alert(typeof str2); // 显示'string'

// 下面的运算需要进行六次字符比较，才能得到结果值 true
alert(str1 == str2);
```

接下来我们讨论等值检测中“相同”的问题。它遵循如下的运算规则：

类型	运算规则
两个值类型进行比较	如同数据类型不同，则必然“不相同”； 数据类型相同时，进行“数值等值”比较。
值类型与引用类型比较	必然“不相同”。

两个引用类型比较	比较引用（的地址）。
----------	------------

所以“相同与否”的检测，仅对两个相同数据类型、值类型的数据有意义（其它情况下的比较值都是“不同”），这时所比较的方法，也是完全的“数据等值”的比较。换言之，下面的代码与上一个示例的所发生的运算，以及运算效果是完全一致的：

```
var str1 = 'abc' + 'def';
var str2 = 'abcd' + 'ef';

// 运算过程和结果完全等同于上一个示例
alert(str1 === str2);
```

通过上面的示例，我们成功地否定了——一个惯例性的说法：“===”运算是比较引用的，而“==”是比较值。因为在例2中，我们看到str1与str2是两个不同引用的变量，但它们是完全相等的。我们事实上也发现，在对两个引用类型的比较运算过程中，“==”与“===”并没有任何的不同。

引用类型的等值比较，将直接“比较引用（的地址）”。这听起来比较拗口，但实际的意义不过是说：如果不是同一个变量或其引用，则两个变量不相等，也不相同。

下面的例子说明这种情况：

```
var str = 'abcdef';
var obj1 = new String(str);
var obj2 = new String(str);

// 返回 false
alert(obj1 == obj2);
alert(obj1 === obj2);
```

我们看到，obj1与obj2是类型相同的，且值都是通过同一个直接量来创建的，但是，由于String()对象是引用类型，所以它们既不“相等”，也不“相同”。

1.3.4.2 序列检测

从数字概念来说，实数数轴上可比较的数字是无限的（正无穷到负无穷）。该数轴上的有序类型，只是该无限区间上的一些点。但对于具体的语言来说，由于数值的表达范围有限，所以数值的比较也是有限的。

如果一个数据的值能投射（例如通过类型转换）到该轴上的一点，则它可以参与在该轴所表达范围内的序列检测：亦即是比较其序列值的大小。在 JavaScript 中，这包括以下的数据类型（其中，Number 类型是实数数轴的抽象）：

可比较序列的类型	序列值
boolean	0~1
string	(*注 1)
number	NEGATIVE_INFINITY ~ POSITIVE_INFINITY(*注 2)

*注 1：在 JavaScript 中，“字符串”是有序类型的一种特例。一般语言中，“字符(char)”这种数据类型是有序的（字符#0~#255）。虽然 JavaScript 不存在“字符”类型，但它的字符串的每一个字符，都被作为单一字符来参与序列检测。

*注 2：负无穷~正无穷。值 NaN 没有序列值，任何值与 NaN 进行序列检测将得到 false。

不要以其它高级语言数据类型的分类中的“有序类型”来理解这里的序列。所谓有序类型，是指该类型的有限集合存在一种有序的排布。例如“字节”这种数据类型，即存在序数 0~255，而“布尔类型”则是“0~1”。这些高级语言中的“有序类型”并不包括实数。

序列检测的含义在于比较变量在序列中的大小，亦即是数学概念中的数轴上点的位置先后。所以运算符包括：

名称	运算值	说明
大于	>	比较两个表达式，看一个是否大于另一个
大于等于	>=	比较两个表达式，看一个是否大于等于另一个
小于	<	比较两个表达式，看是否一个小于另一个
小于等于	<=	比较两个表达式，看是否一个小于等于另一个

并遵循如下的运算规则：

类型	运算规则
两个值类型进行比较	直接比较数据在序列中的大小。
值类型与引用类型比较	将引用类型的数据转换为与值类型数据相同的数据，再进行“序列大小”比较。
两个引用类型比较	无意义，总是返回 false。(*注 1)

*注 1：其实，对引用类型进行序列检测运算仍然是可行的。但这与 valueOf() 运算的效果有关。关于这一点，我们将在“有关章节”中详细讲述。

下面的代码说明这个运算规则：

```
<script>
var o1 = {};
var o2 = {};
var str = '123';
var num = 1;
var b0 = false;
var b1 = true;
var ref = new String();

// 例 1：值类型的比较，考察布尔值与数值在序列中的大小
alert(b1 < num);    // 显示 false
alert(b1 <= num);   // 显示 true，表明 b1==num
alert(b1 > b0);     // 显示 true

// 例 2：值类型与引用类型的比较
// （空字符串被转换为 0 值）
alert(num > ref);   // 显示 true

// 例 3：两个对象（引用类型）比较时总是返回 false
alert(o1 > o2 || o1 < o2 || o1 == o2);
</script>
```

下面补充说明字符串的序列检测含义。

只有两个运算元都是字符串时，[上表](#)中所列的四个运算符才表示字符串序列检测。任意一个运算元非字符串时，将按数值来进行比较（也就是将字符串转换为数值参与运算）。下例说明这一点：

```
<script>
var s1 = 'abc';
var s2 = 'ab';
var s3 = '101';

var b = true;
var i = 100;

// 例 1：两个运算元为字符串，将比较每个字符的序列值。所以显示为 true。
alert( s1 > s2 );

// 例 2：当字符串与其它类型值比较时，将字符串转换为数值比较。所以显示为 true。
```

```

alert( s3 > i );

// 例 3: 在将字符串转换为数值时得到 NaN, 所以下面的三个比较都为 false.
// (注: 变量 b 中的布尔值 true, 转换为数值 1 参与运算)
alert( s1 > b || s1 < b || s1 == b );

// 例 4: 两个 NaN 的比较. NaN 不等值也不大于或小于自身, 所以下面的三个比较都为 false.
alert( s1 > NaN || s1 < NaN || s1 == NaN );
</script>

```

1.3.5. 赋值运算

JavaScript 里有两种赋值运算符, 如下表所示(v: variant, e: expression):

类型	示例	等价等式	备注
(一般) 赋值运算符	<code>v = e</code>		
带操作的赋值运算符 (复合赋值运算符)	<code>v += e</code>	<code>v = v + e</code>	字符串, 数值
	<code>v -= e</code>	<code>v = v - e</code>	数值
	<code>v *= e</code>	<code>v = v * e</code>	
	<code>v /= e</code>	<code>v = v / e</code>	
	<code>v %= e</code>	<code>v = v % e</code>	
	<code>v <<= e</code>	<code>v = v << e</code>	位运算
	<code>v >>= e</code>	<code>v = v >> e</code>	
	<code>v >>>= e</code>	<code>v = v >>> e</code>	
	<code>v &= e</code>	<code>v = v & e</code>	
	<code>v = e</code>	<code>v = v e</code>	
	<code>v ^= e</code>	<code>v = v ^ e</code>	

JavaScript 中, 赋值是一个运算, 而不是一个语句 (如何将它变成语句, 是下一小节的话题)。所以, 在赋值表达式中, 运算符左右都是运算元。当然, 按照“表达式”的理论来说, 表达式的运算元既可以是值 (也包括立即值), 也可以是引用。因此从语法上来说, 下面的代码是成立的:

```

// 下面的代码是两个运算元都是立即值的“赋值运算”表达式
100 = 1000;

```

在 JavaScript 中, 上面这行代码的确能通过语法检测。但是它会触发一个执行期错误。在 Internet Explorer 中的 JScript 引擎中, 错误信息是“不能给 '[number]' 赋值”; 在 Firefox(mozilla)的 JavaScript 引擎中, 错误信息则是“左侧无效赋值”。

这是由于左侧的运算元是直接量, 因此存储单元是不可写的。表达式运算

过程中，赋值的效果（修改存储单元中的值）无法完成，所以提示出错。

这里已经提及到了“赋值的效果是修改存储单元中的值”。而这，其实就是“赋值运算”的本质。

所谓“存储单元”，对于值类型数据来说是存放值数据的内存，对于引用类型数据来说，则是存放所引用地址的内存（即指向用数据的指针）。所以赋值运算在值类型来讲是复制数据，而对于引用类型来讲，则只是复制一个地址。

这里存在一个特例：值类型的字符串是一个大的、不确定长度的连续数据块，这导致复制数据的开销很大，所以 JavaScript 中将字符串的赋值也变成了复制（连续数据块起始处的）地址。由此引入了三条字符串处理的限制：

- 👉 不能直接修改字符串中的字符；
- 👉 字符串连接运算必然导致写复制，这将产生新的字符串；
- 👉 不能改变字符串的长短，例如修改 `length` 属性是无意义的。

赋值运算符除了等号“=”之外，还有一类“复合赋值运算符”。这类运算符由一个一般表达式运算符与一个赋值运算符复合构成。在表格 X X 中的列的复合运算符中，除第一个“+=”能用于字符串之外，其它的都只能用于数值类型——如果将它们使用于非数值类型，则运算中会出现隐式的类型转换。

1.3.6. 函数调用

JavaScript 中只有一种方法来完成函数调用，即在函数后直接跟函数调用运算符“()”。这个运算符被解释成两个含义：

- 👉 使函数得以执行；并且，在函数执行时，
- 👉 从左至右运算并传入“()”内的参数序列。

这里所说的“函数”，包括普通的、类型值（即 `typeof` 值）为“function”的函数，也包括类型值为“object”的、创建自 `Function` 类的函数（对象实例）。也就是说，函数调用运算符作用于以下两个变量的效果，是一致的：

```
var func_1 = function() { };
var func_2 = new Function('');

// 调用函数 1
func_1();
```



```
// 调用函数 2
func_2();
```

如果该运算符之前既非上述两种之一（函数直接量与 `func_1` 是类同的），也非它们的引用，则函数调用运算将会出错，这会触发一个运行期异常。

1.3.7. 特殊作用的运算符

有些运算符不直接产生运算效果，而是用于影响运算效果，这一类的运算符的操作对象通常是“表达式”，而非“表达式的值”。另外的一些运算符不直接针对变量的值运算，而是针对变量运算（例如 `typeof`）。在这里所说的特殊作用的运算符，主要是指这两类，包括：

目标	运算符	作用	备注
运算元	<code>typeof</code>	返回表示运算元 数据类型的字符串	
表达式	<code>void</code>	避免表达式返回值	使表达式总是返回值 <code>undefined</code>
	<code>? :</code>	根据条件执行两个表达式 中的一个	也称“三元（三目）条件运算符”
	<code>()</code>	表达式分组 和调整运算次序	也称“优先级运算符”
	<code>,</code>	表达式 顺序地 连续执行	也称“多重求值”或“逗号运算符”

注意这里的 `typeof` 是运算符，而不是语句的语法关键字。之所以说它是特殊作用的运算符，是因为在其它运算符中，变量是以其值参与运算。例如下面的表达式中，变量 `N` 是以值 13 参与运算的：

```
var N = 13;
alert( N * 2 );
```

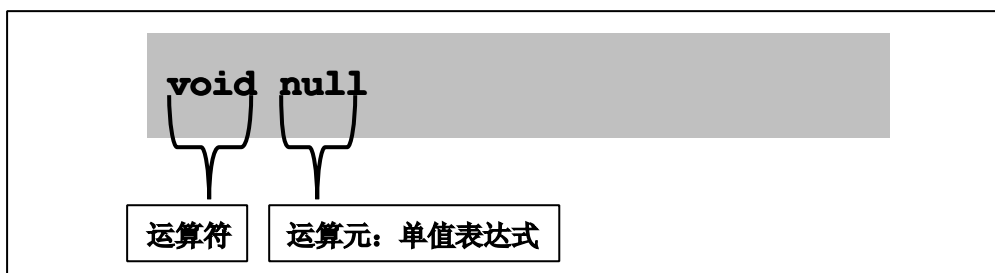
而 `typeof` 运算符并不访问变量的值，而是取值的类型信息。例如，下面的代码显示值的类型，而非变量 `N` 所在的存储单元中的值。

```
// 续上例
alert( typeof N );
```

其它的四个运算符则是面向表达式进行运算。其特殊性在于：普通运算符对表达式求值以获得结果，而这四个运算符不（直接）对表达式进行求值运算，而是通过操作表达式去影响运算结果。例如 `void` 运算符，它在它之后的表达式的影响就是：避免产生结果。

尽管它们是以“表达式”目标，但语义上它们仍然是运算符。因此它们与运算对象（表达式）结合的结果仍然是表达式（而非语句）。例如下面的代码

是表达式：



但运算符的后面不能是语句（很显然，这是语法规则）。因此，下面的代码是不合法的：

```
// 用{}表示的复合语句不能作为 void 的运算对象
void {
  // ...
}
```

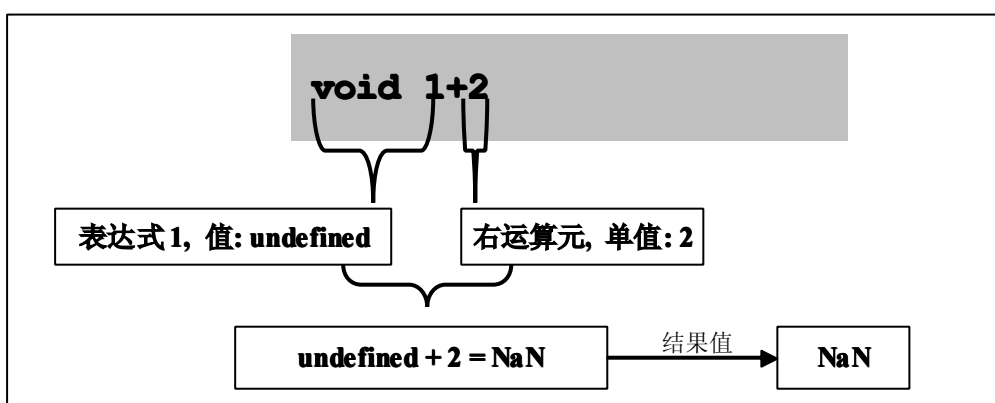
1.3.8. 运算优先级

上面这个例子存在一个问题。既然 **void** 运算的对象是表达式，且 JavaScript 允许单值表达式，那么这样的代码中：

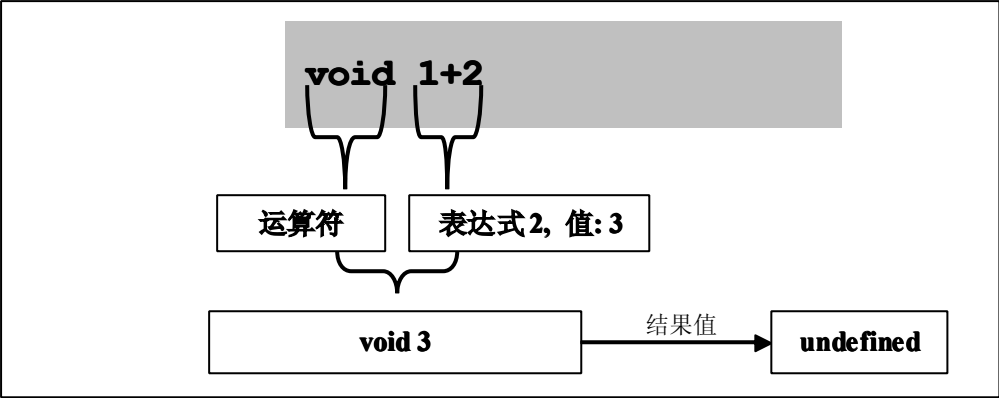
```
void 1+2
```

void 运算的对象到底是 "1" 这个单值表达式，还是 "1+2" 这个算术表达式呢？

如果是第一种情况，那么结果会是：



如果是第二种情况，则结果会是：



这就涉及到这两个例子中表达式 1 的运算符"void"和表达式 2 中的运算符"+"的优先级的问题了：谁的运算优先级更高，则先以该运算符来构成表达式并完成运算。在 JavaScript 中，该优先级顺序如下表(序数越小，越优先运算)：

	运算符	描述
1	. [] ()	对象成员存取、数组下标、函数调用等
2	++ -- ~ ! delete new typeof void	一元运算符等
3	* / %	乘法、除法、取模
4	+ - +	加法、减法、字符串连接
5	<< >> >>>	移位
6	< <= > >= instanceof	序列检测、instanceof
7	== != === !==	等值检测
8	&	按位与
9	^	按位异或
10		按位或
11	&&	逻辑与
12		逻辑或
13	? :	条件
14	= oP=	赋值、运算赋值
15	,	多重求值

通过该优先级表可知：由于"void"运算符高于"+"运算符值，因此应该以上述的第一种情况进行运算，其结果值为 NaN。

因此如果我们希望是以第二种情况进行运算（事实上在我写下这个例子之前，我认为是以这种情况运算的）。这时，我们就需要使用强制运算符"(")"来改变执行（优先级别的）顺序。下面的代码实现第二种情况的运算效果：

```
void (1 + 2)
```

正如我们留意到的：在“void”与“()”参与的这几个运算中，运算符并

不对值进行运算，而是对表达式（包括单值表达式）的运算效果进行影响。所以我们一再强调，“void”与“()”等等这些运算符的运算对象，是“表达式”而非变量 / 值运算元。

类同的，我们看到下面两个运算符("?:"和",")也都用于影响表达式运算的效果(小节“1.6.4”将对示例 2 做更多的说明)：

```
/**
 * 示例 1: 运算符"?:"用于条件化地运算表达式
 */

// 显示表达式 100+20 的值
alert( true ? 100+20 : 100-20 )

/**
 * 示例 2: 三个表达式连续运算求值，返回最后一个表达式的值
 */

// 显示最后表达式的值"value: 240"
var i = 100;
alert( (i+=20, i*=2, 'value: '+i) );
```

1.4. JavaScript 的语法：语句

整个的 JavaScript 代码，是由语句构成的。语句表明执行过程中的流程、限定与约定。可以是单行语句，或者由一对大括号“{}”括起来的复合语句——在语法描述中，复合语句可以整体作为一个单行语句处理。

下面两个原则，有助于你了解 JavaScript 的“语句”的定义：

- 👉 语句由语法分隔符“; (分号)”来分隔(注意，它不是运算符)。
- 👉 （除空语句和语句中的控制子句之外，）语句存在返回值，该值由执行中的最后一个子句/表达式的值决定。

当语句位于以下几种情况之一时，也可以省略分号。

- 1、一个文本行或整个文本文件的末尾，或
- 2、在语法分隔符之前（如复合语句的大括号“}”），或
- 3、在复合语句的大括号“}”之后。

关于这三点，JavaScript 的创始者的原始意图，是为了更好的容错。当然，一部分原因，也在于这符合其它一些语言的惯例。

在前面的章节中，我们已经讲述过的语句有：

- 👉 赋值语句：使用“等号(=)”赋值运算符
- 👉 变量声明语句：使用 **var** 关键字开始一个变量声明
- 👉 标签声明语句：使用“**identifier: statement**”的语句开始一个标签声明

除了标签声明，下面我们还将补充更多的有关赋值与变量声明语句的内容。此外，还有其它的一些语句，包括：

类型	子类型	语法*	备注
声明语句	变量声明语句	var <i>variable1</i> [= <i>v1</i>] [, <i>variable2</i> [= <i>v2</i>], ...] ;	
	标签声明语句	<i>labelname</i> : <i>statements</i> ;	
表达式语句	变量赋值语句	<i>variable</i> = <i>value</i> ;	
	函数调用语句	<i>foo</i> () ;	
	属性赋值语句	<i>object.property</i> = <i>value</i> ;	
	方法调用语句	<i>object.method</i> () ;	
分支语句	条件分支语句	if (<i>condition</i>) <i>statement1</i> [else <i>statement2</i>] ;	
	多重分支语句	switch (<i>expression</i>) { case <i>label</i> : <i>statementlist</i> case <i>label</i> : <i>statementlist</i> ... default : <i>statementlist</i> };	
循环语句	for	for ([var] <i>initialization</i> ; <i>test</i> ; <i>increment</i>) <i>statements</i> ;	**
	for .. in	for ([var] <i>variable</i> in < <i>object</i> <i>Array</i> >) <i>statements</i> ;	**
	while	while (<i>expression</i>) <i>statements</i> ;	
	do .. while	do <i>statement</i> while (<i>expression</i>) ;	
控制结构	继续执行子句	continue [<i>label</i>] ;	
	中断执行子句	break [<i>label</i>] ;	
	函数返回子句	return [<i>expression</i>] ;	
	异常触发语句	throw <i>exception</i> ;	

	异常捕获与处理	<pre> try { <i>tryStatements</i> } catch (<i>exception</i>) { <i>catchStatements</i> } finally { <i>finallyStatements</i> }; </pre>	
其它	空语句	;	
	with 语句	<pre> with (<i>object</i>) <i>statements</i>; </pre>	

*注：语法描述中加粗的为语法标识符/关键字；加方括号为可选的语法部分；加尖括号的为必选的语法部分；“|”表示所列项中选一。

****注：**这里的 `var` 关键节的用法是非标准的。

1.4.1. 表达式语句

为什么会存在“表达式语句”这样的概念呢？因为我们对编程语言术语的“表达式”的约定是：由运算数和操作构成，并运算产生结构的语法结构。那么很显然，下面的代码就是一个表达式：

```
1+2+3
```

接下来，另外的一项约定是：程序是由语句构成的，语句是则由“；(分号)”分隔的句子或命令。那么如果在表达式后面加上一个“；”分隔符，JavaScript 又如何理解呢？

```
1+2+3;
```

这就被称为“表达式语句”。它表明“只有表达式，而没有其它语法元素的语句”。

在 JavaScript 中，许多语法与语义其实最终都是由“表达式语句”来实现的。例如赋值、函数调用，以及我们在其它高级语言中常见的“调用对象方法”。

我们前面说过，语句的返回值由最后的一个子句或表达式的值决定。因此，“表达式语句”的值，就是该表达式运算的结果——即使不返回值(原格说来并没有“不返回值”，而是返回 `undefined` 值)，也可以参与后续运算。

1.4.1.1. 一般表达式语句

对于一个表达式，你可以“计算值，并参与运算”，例如：

```
v * (1+2+3)
```

由于 $1+2+3$ 是表达式，而括号“ $()$ ”作为运算符时，是指强制运算并求值。那么“ $(1+2+3)$ ”就是计算值，而后参与和变量“ v ”的求乘积运算了。

或者，你也可以计算但不返回值，例如：

```
void 1+2+3
```

这里用到了 `void` 运算符，它的含义就是使后面的表达式运算，但忽略值。`void` 也是运算符，因此上述代码其实经过了三次运算(先两次求和，后忽略返回值)，其结果值是 `undefined`。同样的道理，下面的表达式就经过了四次运算(先强制运算，再两次求和，最后忽略返回值)：

```
void (1+2+3)
```

再或者，你也可以用 `eval()` 函数来执行一行字符串(如果它是表达式)，例如：

```
eval('1+2+3')
```

这种用法在一些语法分析，或者动态执行语句时会有实用价值。由于 `eval()` 实际上也是一个运算，所以这项表达式其实完成了三次运算(`eval` 函数调用、两次求和)并返回值。

所以上述几种情况，其实都是在运算表达式。但你也可以在表达式末尾加一个“`;`”号，表明这里完成了一个表达式语句：你运算这个表达式语句，并且返回语句的值。例如：

```
v * (1+2+3);  
void 1+2+3;  
1+2+3;
```

事实上，JavaScript 承认单值表达式。尽管可能这个表达式不能表明任何确定的含义，但某些时候它的确有用。所以同样也存在单值语句。例如：

```
2;
```

由于“2”是一个单值，也可以理解为一个单值表达式，因此“2;”就变成了单值表达式语句。这个语句在远程读取数据时可能会有用，因为你不能确保读到本机的是一个单独的数，还是一个数组。因此下面的代码可能给你提供了处理的机会：

```
// 远程数据
// returnValue = '2;';

// 取远程数据到本地
var remoteMetaData = ajax.Get(your_url);
var remoteData = eval(remoteMetaData);

switch (typeof remoteData) {
    // ..
}
```

我们发现上面的代码中，即使远程数据是“2;” (亦即是 `returnValue`)，程序仍然能够正常的处理。因为“2;”被理解为语句并有效的执行了。

我们也顺便讲一下空语句。上面这个例子中，如果单值表达式“2”也没有了，只剩下了一个分号“;”又会如何呢？

在 **JavaScript** 中，它就被理解为空语句。我们前面说到过，语句也是有值的，可以用来运算。因此，空语句就返回值 `undefined`——再次强调一下：完全没值的语句或表达式，在 **JavaScript** 中并不存在。

空语句的另一种应用情况是写空循环，或者空分支。例如：

```
var value = 100;

// 使用空语句的空循环
while (value--); // <- 这里存在一个空的循环

if (value > 0); // <- 这里有一个空的 then 分支
else {
    // ...
}
```

使用空语句这样的代码时，一定要添加准确的注释，否则代码回顾(**review**)时将无法清晰的理解使用这个技术的意图。

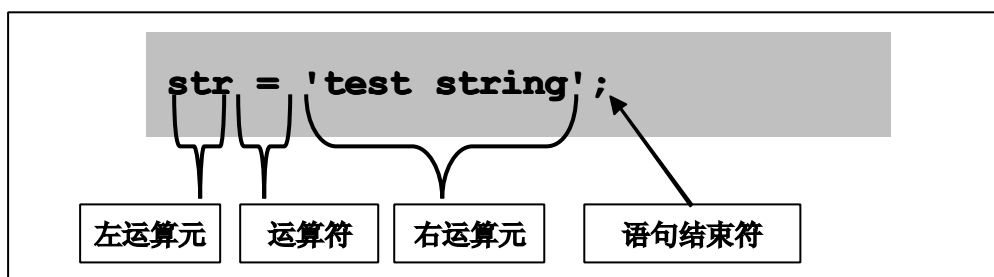
最后要强调一点：`eval()`函数总是执行语句。即使传入下面这样的代码：

```
eval('1+2+3') // 是试图“运算表达式”并返回表达式结果吗？
```

在语法逻辑上，它也是被当成语句处理的——因为还存在另一条规则：换行符和文本结束符的前面可以没有分号“;”。所以，上面这行代码仍然是执行语句并返回语句的结果值。

1.4.1.2. 赋值语句与变量声明语句

赋值语句也是典型的表达式语句，例如下面这个赋值表达式：



它一方面可以继续参与运算，例如：

```
str2 = 'this is a ' + (str = 'test string')
```

另一方面，也可以直接加上一个语句结束符“;”号，以表明这是一个“表达式语句”：

```
str = 'test string';
```

所以在 JavaScript 中，“赋值语句”其实是“赋值表达式运算”的一种效果。这与其它语言对“赋值语句”的理解并不一致。

JavaScript 的另一个语言特性使得赋值语句具有了“隐式变量声明”的功能。这个特性是它的动态语言特性之一：变量即用即声明。

也就是说，一个变量（标识符）在赋值前未被声明，则脚本 如果从来未被声明过，则它在第一次使用时将会被声明。

变量声明语句中的 `var` 是语句语法符号，而不是运算符。下面的代码的效果，是在语法解释期和执行期分两次来实现的。

```
var i = 100;
```

JavaScript 中可以使用显式声明和隐式声明。语法区别在于前者使用 `var` 关键字，而后者不用；语义上的区别在于前者在函数中使用时，表示声明局部变量，而后者声明的结果总是全局变量。

显式声明时可以为变量赋一个初值。如果不赋初值，则该变量默认值为 `undefined`。隐式声明与此不同的是，它需要一个“赋值表达式语句”来实现声明。好的，这里我们已经出现了概念上的混乱。因为一方面有“显式声明时赋的初值”，另一方面有“隐式声明时使用的赋值表达式”。

然而在概念原本是无比清楚的：前一种情况是语句的语法；后一种是表达式带来的隐式效果。

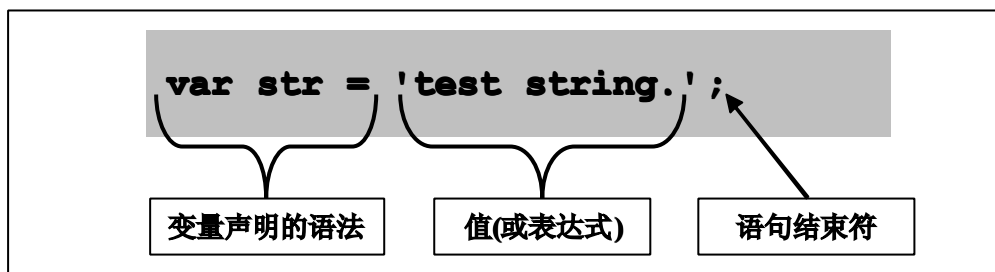
显式声明语句需要使用一个语法关键字 `var`，该语句的语法是：

```
var variable1 [ = value1 ] [, variable2 [ = value2], ...]
```

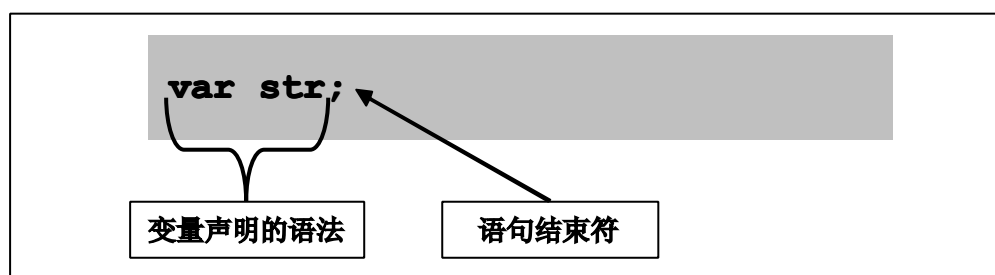
所以我们下面这个语句中的“=(等号)”，其实是语法分隔符，而非运算符：

```
var str = 'test string.';
```

如下图所示：



出于语法的设定，我们显然也可以不使用“=”这个语法分隔符(或标识符)来指明初始赋值。那么此时该语句就可以是：



所以，它与变量声明语句的区别在于：

👉 “变量声明语句”是一个有关键字的语句，并不是表达式语句；

👉 “赋值语句”是一个表达式语句。

我们知道，在赋值表达式中，至少要包括一个赋值运算符。根据赋值运算符的规则，其左侧一定是一个变量或对象属性，右侧可以是值或求值的表达式。因此，赋值表达式的语法如下：

语法：变量 赋值运算符 值(或求值表达式)

示例：str = 'this is ' + 'sample.'

我们在语法说明中，为什么使用中文的“赋值运算符”，而不是象其它语言一样直接使用“等号(=)”呢？这是因为变量声明语句与赋值表达式存在根本的不同，因此下面的语法并不成立：

// 错误的语法

```
var str += 'test!';
```

这是因为这里的“=”是语句的语法分隔符，而不是“赋值运算符”，因此并不能替代成“+=”(或其它复合赋值运算符)。

最后的一点说明，是使用 **var** 来声明变量的语法，还可以在如下两种情况下使用：

// 1. 在 for 循环中声明变量

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

// 2. 在 for .. in 循环中声明变量

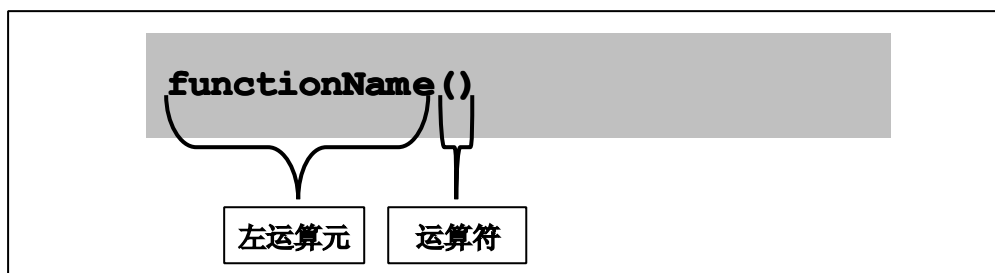
```
for (var prop in Object.prototype) {  
    // ...  
}
```

这两种情况下它相当于一个语句的子句。由于 JavaScript 的变量作用域只能达到函数一级(而非语句块一级)，因此这里声明变量，与在 for 或 for..in 语句之外声明变量没有什么区别。

1.4.1.3. 函数调用语句

JavaScript 中，函数本身是一个变量/值，因此函数调用其实是一个表达式。

如下图所示：



所以，下面的代码就成了函数调用语句，它也是一个表达式语句：

```
functionName();
```

JavaScript 中，具名函数可以使用上述的方法直接调用，匿名函数可以通过引用变量调用，但没有引用的匿名函数怎么调用呢？下面的例子说明这三种情况：

// 示例 1. 具名函数直接调用

```
function foo() {  
}  
foo();
```

// 示例 2. 匿名函数通过引用来调用

```
fooRef = function() {  
}  
fooRef();
```

// 示例 3. 没有引用的匿名函数的调用方法 (1)

```
(function() {  
    // ...  
})();
```

// 示例 4. 没有引用的匿名函数的调用方法 (2)

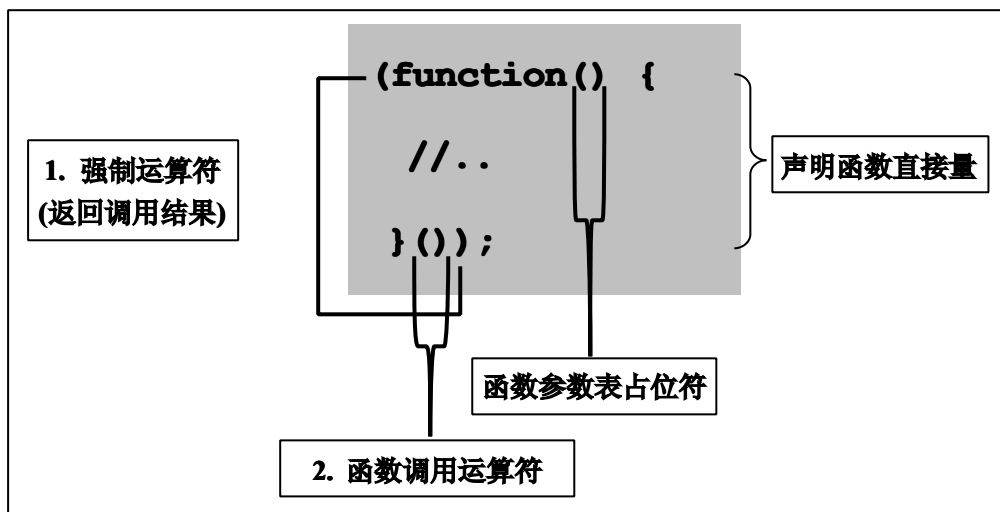
```
(function() {  
    // ...  
})();
```

// 示例 5. 没有引用的匿名函数的调用方法 (3)

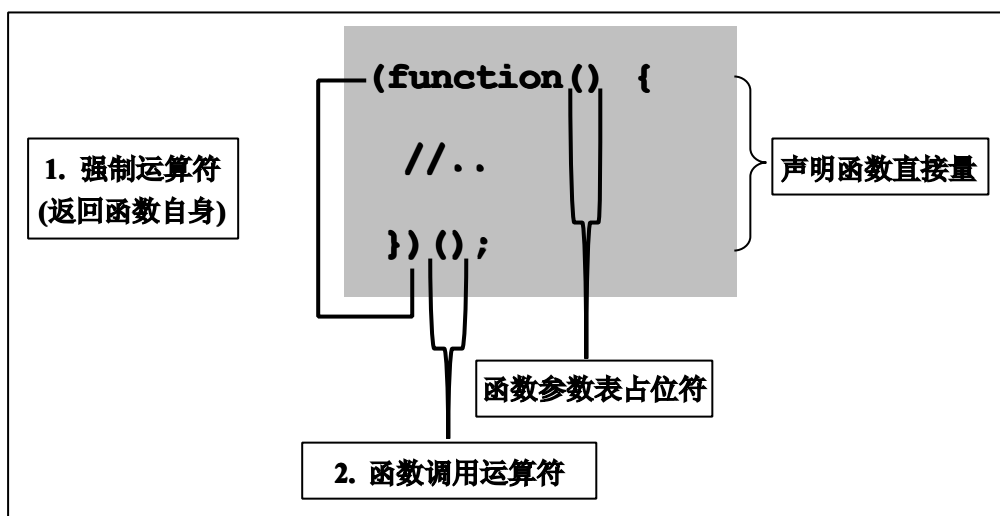
```
void function() {  
    // ...  
}();
```

示例 1、2 的用法比较常见。而例 3、4、5 虽不太多见，但各有其用。

其中，例 3 与例 4 都用于“调用函数并返回值”。两种表达式都有三对括号，但含义各不相同，如下图所示（例 3）：



下图是对例 4 的说明：



我们看到例 3 与例 4 基本一致。但事实上两种表达式的运算过程略有不同：例 3 中是用强制运算符使函数调用运算得以执行，例 4 中则用强制运算符运算“函数直接量声明”这个表达式，并返回一个函数自身(或引用)，然后通过函数调用运算符“()”来操作函数引用。

换言之，“函数调用运算符()”在例 3 中的作用于匿名函数本身，在例 4 中却作于用于一个运算的结果值。

最后的示例 5，则用于“调用函数并忽略返回值”。运算符 `void` 用于使其后的函数表达式执行运算。

那么如果不使用 `void` 与“`()`”这两个运算符，而直接使用下面的代码，能否使函数表达式语句得到执行呢？

```
// 示例 6. 直接使用函数调用运算符"()"调用
function() {
    // ...
}()

// 示例 7. 使用语句结束符";"来执行语句
function() {
    // ...
}();
```

示例 6、7 看起来是正确的——起码用我们以前提到的所有知识来看，这两个示例中的代码均能被理解。但是事实上它们都不可执行。究其原因，则是因为它们无法通过脚本引擎的语法检测——在语法检测阶段，脚本引擎会认为下面的代码：

```
function() {
}

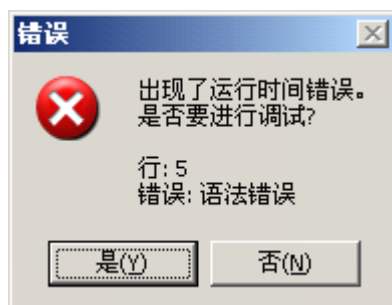
// 或
function foo() {
}
```

是函数声明——因此事实上这里使用具名函数和匿名函数都是通不过语法检测的，而后面的讨论也因此对这二者都有效——这就会使得在示例 6 和示例 7 的代码中，函数后面后面的一对括号没有语法意义。这样一来，它们的代码无疑被语法解析成了：

```
// 示意：对示例 6 的语法解释
function() {
    // ...
};
();

// 示意：对示例 7 的语法解释
// (略)
```

也就是说“`function () {}`”作为完整的语法结构被解释，因此相当于其后已经存在语句结束符。而“`();`”则被独立成一行进行语法解释，显示这是错误的语法，因而会出现下面的错误提示：



因此这个“语法错误”事实上是针对于“`();`”，而不是针对于前面的函数声明的。为了证明这一点，我们改写代码如下：

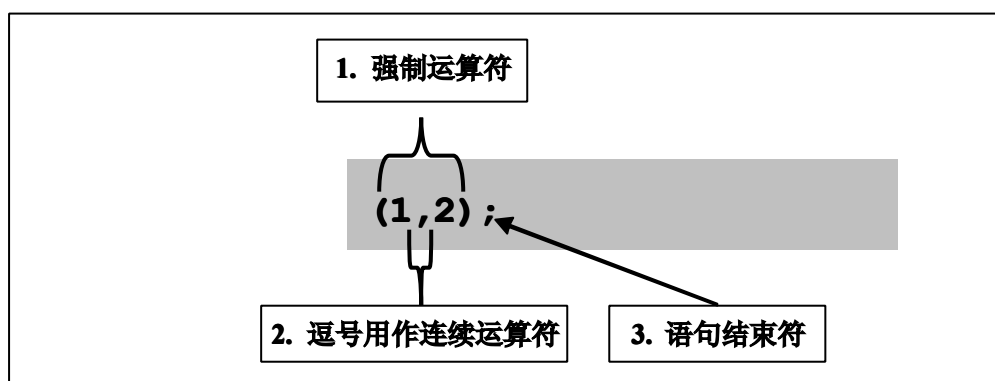
// 改写示例 6 的代码以通过语法解释

```
function() {  
    // ...  
}(1,2)
```

这样一来你会发现语法检测通过了。因为语句被语法解释成了：

```
function() {  
    // ...  
};  
(1,2);
```

而最后这行代码被解释成：



其中的“1”和“2”被解释成了两个单值表达式——当然也可以是“`(1)`”这样的一个单值表达式。因此语法上就通得过了。但重要的是，由于这段代码被解释成了一个函数直接量声明和一个表达式语句，因此它事实上并不能起到“执行函数并传入参数”的作用。简单地说，你不能指望用下面的代码来声明，并

同时执行函数：

```
function foo() {  
    // ...  
}(1,2);
```

如果你真的想在声明的时候执行一下该函数，那么请参考前本小节开始的示例 3、4、5，用一个 `void` 或括号 “`()`” 运算将函数声明变成 “（直接量的）单值表达式”：

```
// 示例：声明时立即执行该函数 (也可以用于匿名函数声明)  
void function foo() {  
    // ...  
}(1,2);
```

1.4.2. 分支语句

基本上来说，JavaScript 的 `if` 语句跟 C 风格下的 `if` 语句并没有什么不同，所以事实上 `if` 分支并没有什么可赘述的地方 (同样的原因，《JavaScript 权威指南》一书只好花了两页的篇幅来教导读者如何排布 `if` 语句的缩进格式)。同样，JavaScript 的 `switch` 分支语句也同于其它 C 风格的语言——但是，不幸的是，C 语言与 Pascal 等其它一些语言的多重分支语句之间，却存在着较大的差异。

因此在下面的小节，我会略为阐述一下 `if` 和 `switch` 语句的基本语法。但对于 `switch` 语句，我会详细地叙述它的一些独特之处。

1.4.2.1. 条件分支语句(if 语句)

`if` 语句语法描述如下：

```
if (condition)  
    statement1  
[else  
    statement2];
```

当 `condition` 条件成立 (值为 `true`) 时执行语句 `statement1`；否则执行 `statement2`。语法描述中的方括号表明 `else` 子句可以省略——这时，如果 `condition` 条件不成立 (值为 `false`)，则什么也不做。

由于 `statement1`、`statement2` 在语法上表明是 “语句”，因此事实上它们即

可以是单行语句，也可以是复合语句——本小节开始的部分，我们已经说过“在语法描述中，复合语句可以整体作为一个单行语句处理”。

这表明下列代表中的大括号“{}”是复合语句的语法符号，而并非(象一些人想见的那样)是 if 语句的语法元素：

```
// 代码风格 1: if 语句中使用复合语句带来的效果
if ( condition ) {
    // ...
}
else {
    // ...
}
```

同样，我们也应该了解，if .. else if .. 这样的格式，并非是“一种语法的变种”。只不是 else 子句中的“statement2”是一个新的、单行的 if 语句而已：

```
// 代码风格 2: 在 else 子句中，使用单行 if 语句带来的效果
if ( condition1 ) {
    // ...
}
else if ( condition2 ) {
    // ..
}
```

1.4.2.2. 多重分支语句(switch 语句)

无论是解释 switch 语句，还是使用 switch 语句，都非常容易令人迷惑。其中的主要原因之一，则在于 switch 语句中的 break 子句的使用。

在 Pascal 风格的语言系统中，switch 是没有 break 这样的子句的。这虽然导致有些控制逻辑的代码会变得复杂，但也使得代码的结构化程度得以提高。但 C 风格的语言系统中，由于存在了 break 子句，使得多重分支的流程出现了“例外”，因此提供了灵活性，却也产生了类似于 GOTO 语句的副作用。

我们比较一下两种风格的 switch 语句(在 Pascal 中称为 case)：

```
(**
 * pascal 语言风格的多重分支语句
 *)
var i,j : integer;
// ... (略去有 i,j 初值或相关运算的代码)
```

```
/**
 * C 语言风格(JavaScript)的多重分支语句
 */
var i, j;
// ... (略去有 i,j 初值或相关运算的代码)
```

```

case ( i ) of
  100: begin

  end;

  200: begin

  end;

  else begin

  end;

end;

```

```

switch ( i ) {
  case 100: {
    break;
  }

  case 200: {
    break;
  }

  default: {
    break;
  }
}

```

可见除了一些关键字和语法符号的差异之外，二者并没有不同。但是，pascal 中没 break 语句，因此 begin..end 之间总是一个完整的语法块。关键字 end 起到了“结束该语句块(及其逻辑流程)”的作用。而对应的，在 C 语言中，你必须使用 break 子句来中止这个语句块的逻辑流程。

case 分支中的大括号“{ }”在这里只起到了标识一个复合语句的作用。而 switch 语句的“一批语句”是由该语句的语法：

```

switch (condition) {
  statements
}

```

来标识的（case 只用于标识这批语句的某个入口点），因此这时就不必用大括号来局部的开始和结束了(因为这没有逻辑意义)。所以一般情况下，可以省却大括号而直接书写多行代码。例如：

```

switch ( i ) {
  case 100:
    j++;
    i += j;
    break;

  case 200:
    // ...
}

```

但是，在 pascal 风格的代码中，由于 begin..end 必须是一个完整的语法块，

因此下面的代码必然会出现：

```
(**  
* pascal 语法风格中没有 break 语句导致的问题  
*)  
case ( i ) of  
  100: begin  
    i := i + 1;  
    j := j + 1;  
  end;  
  
  200: begin  
    j := j + 1;  
  end;  
  
  // ...  
end;
```

也就是说，在 i 值为 100 和 200 的处理中，可能有一些相同的代码行需要重复使用。然而由于 **begin..end** 决定了完整的代码块，因此只能出现冗余的代码。除非在该这个多重分支语句前面/后面写条件分支代码(或新的多重分支)。例如：

```
(**  
* 解决方案一  
*)  
case ( i ) of  
  100, 200: begin  
    if (i = 100) i := i + 1;  
    j := j + 1;  
  end;  
  
  // ...  
end;  
  
(**  
* 解决方案二  
*)  
case ( i ) of  
  100, 200: begin  
    j := j + 1;  
  end;
```

```
// ...  
end;  
if (i = 100) i := i + 1;
```

但 C 语言在代码风格上解决这个问题，这使得两个分支复用同一个代码块成为可能。上面的代码在 C 风格的语言中的解决方案如下：

```
/**  
 * C 风格语言 (JavaScript) 的解决方案  
 */  
switch ( i ) {  
    case 100 : i++;  
    case 200 : j++;  
}
```

我们只需要不在语句 “i++” 后面写 **break** 子句，那么逻辑流程就会 “漏” 到下一行的 “case 200” 这个分支，从而达到了 “复用多个分支的代码” 的效果。

因为 **break** 改变了控制的流程，对使用一对大括号 “{ }” 表示的结构化的代码块造成了语法伤害。因此，包括在经典的 C 或 C++ 语法材料中，这种 “漏掉 **break** 子句” 的技巧都是被有争议地、谨慎地进行描述的，而不会 “推荐使用” —— 甚至要求不要省略最后一个分支后的 **break**，以避免将来加入新的分支时遗忘掉一个 **break** 子句。

即使在使用这种技巧的代码中，也被明确要求 “对算法或省却 **break** 的原因做出备注”。所以，仍以上面的代码为例，一个较为良好的风格应当是：

```
/**  
 * C 风格语言 (JavaScript) 的解决方案  
 */  
switch ( i ) {  
    case 100 : i++;    // defer break;  
    case 200 : j++;    // break omitted for end.  
}
```

其中，第一个备注明确指出在这里的算法要求延迟(**defer**)进行 **break**；第二个注释则说明由于结束而省略 **break**。

由于 **break** 是一个在循环和多重分支中都可以使用的子句，所以还存在更

多的细节没有描述。关于这些细节，请阅读“1.4.4.2 break 子句”。

1.4.3. 循环语句

JavaScript 的循环语句相对丰富——尽管您使用其中一个就可以替代其它循环语句。一般来讲，以下三种循环结构是开发人员所熟知的：

```
// for 循环 (增量循环)
for ([var ] initialization; test; increment)
    statements;

// while 循环
while (expression)
    statements;

// do..while 循环
do
    statement
while (expression);
```

通常 while 与 do..while 中的循环条件（*expression*）都应当是有意义的，仅有在少数情况下，它被置为 true 以表示无限循环（当然在循环体内应使用 break 子句来中断）。例如：

```
while (true) ...

//或
do
    ...
while (true);
```

在很多情况下，上述用法仍可称为良好的结构化设计^①，但是在循环体中的空语句却是不宜使用的。例如：

```
var i = 10;
while (alert(i), i--);
```

这样的代码除了展示 while 条件中可以使用连续运算（的技巧）之外毫无用处，它完全等价于：

```
// 方法一
var i = 10;
while (i) {
```

^① 但我不建议使用“for(;;)...”来表示无限循环，这种用法大概只是出炫耀的目的而被创造出来的。

```
    alert(i);
    i--;
}

// 方法二
var i = 10;
do {
    alert(i);
}
while (--i)
```

与此类同的问题也会出现在 for 语句中：很多人习惯于将循环体塞在 for 语句的表达式中。例如：

```
for (var i = 10; i<10; alert(i), i--);
```

这种用法对读代码的人来说会是一种灾难。因此，建议 for、while 与 do..while 等循环语句中只放置与循环条件相关的表达式运算（例如同时处理多数组时使用的多个下标控制变量）。

除了上述三种循环之外，JavaScript 也支持一种用于对象成员列举的循环语句 for..in：

```
for ([var ]variable in <object | Array>)
    statements;
```

需要注意的是，从语法上来看它能够处理数组中的元素，但其实这不过是 JavaScript 将数组作为对象处理时的一种表面现象^①。关于 for..in 语句的使用细节，我们将在“1.5.2 对象成员列举、存取和删除”小节中专门讲述。

1.4.4. 流程控制：一般子句

1.4.4.1. 标签声明

JavaScript 中的标签就是一个标识符，它可以与变量重名而互不影响。它是另一种独立的语法元素(既不是变量，也不是类型)，作用是指示“标签化语句(labeled statement)”。

^① Core JavaScript 1.5 中约定了一种 for each 语句语法，用以从一个对象中列举所有属性的值，它直到 SpiderMonkey JavaScript 1.6 以上版本才开始支持。但这不包含在 ECMA 规范中，也不为 JScript 等其它 JavaScript 引擎所支持。（仅对数组的成员列举来说，）相比之下更值得称赞的一项扩展是 JavaScript 1.6 中的设计：为 Array 对象原型扩充 forEach() 方法。在 SpiderMonkey JavaScript 1.7 中，由于提出了“destructuring assignment”的概念，因此允许在 for..in 中同时列举对象的属性和属性的值。

标签的声明很简单：一个标识符后面跟一个冒号“:”。你可以在任何一个语句前面加上这样的标签，以使得该语句被“标签化(labeled)”。例如：

```
this_is_a_label:
    myFunc();
```

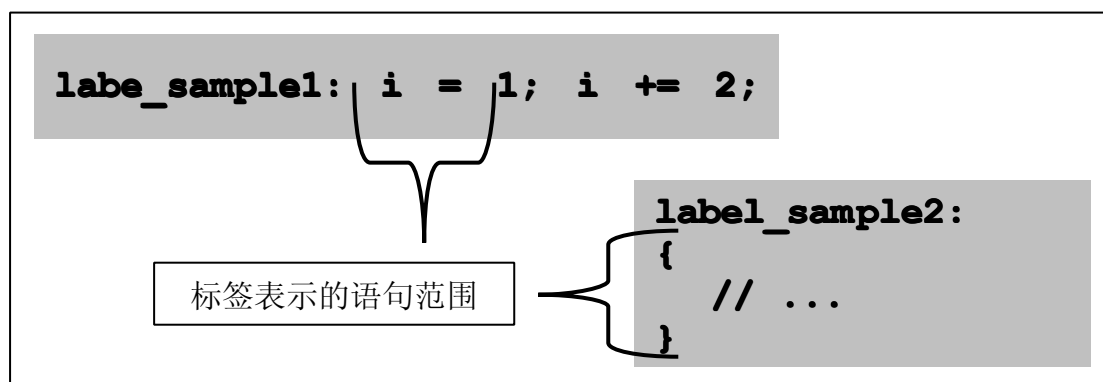
除了单一的语句之外，标签也可以作用于由大括号表示的一批语句(后面会叫它复合语句)。例如：

```
label_statements: {
    var i = 1;
    while (i < 10) i++;
}
```

但是标签不能作用于注释语句。因此在下例中，标签实际作用后注释语句后面的一个语句(即 if 条件语句)：

```
my_label_2:
/*
    hello, world;
*/
if (true) {
    alert('hello, is a test!');
}
```

标签表示一个具有“语句块”含义的范围(但不是后文中提到的上下文环境、闭包或者变量作用域等概念上的范围)。这个语句块的范围是指单一语句的开始到结束位置(分号或回车)，或者成批语句的开始到结束位置(一对大括号)。下图说明这个范围：



在 JavaScript 中，标签只能被 `break` 语句与 `continue` 语句所引用。前者表明停止语句执行，并跳转到 `break` 的所指示的范围之外；后者表明停止当前循环，并跳转到 `break` 所指示的范围的开始。

我们已经被无数次地忠告“不要使用 GOTO 语句”，然而还是有一些语言保留了 GOTO 语句。事实上 JavaScript 语言也将这个关键字作为保留的语法关键字——但至今仍未启用。在除去了 GOTO 语句之外，程序系统中仍然会存在一些语句来改变程序的程序执行流程。例如本小节中将提及到的 `break` 和 `continue`，以及 `return` 语句。

但是这些在语言中顽强地存活下来的语句，都不再象 GOTO 语句一样可以“无条件地任意跳转”。事实上它们总是受限于语句的上下文环境。例如 `break` 只能用于 `for`、`while` 等循环语句、`switch` 分支语句和标签化语句的内部，而 `return` 只能用于函数内部。因此，一些讲述 JavaScript 语言语法的书籍中会称它们为“子句”。本书在这里也采用这种说法，表明它们“是上下文受限的”这一事实。

1.4.4.2. `break` 子句

通常，我们在 `for`、`while` 等循环中使用 `break`，表明停止一个最内层的循环；或在 `switch` 语句中跳出 `switch` 语句。例如：

```
/**
 * 在 for 循环中使用 break 的简单示例
 * ( 使 i=50, j=50 不被处理 )
 */
for (var i=0; i<100; i++) {
    //...
    for (var j=0; j<100; j++) {
        if (i==50 && j==50) break;
        // ...
    }
}

/**
 * 在 switch 中使用 break 的简单示例
 */
var chr = prompt('please input a char', 'A');
switch (chr) {
    case null: break;
    case 'A':
    case 'B': break;
    default:
        chr = 'X';
}
```



```
    break;
}
alert(chr);
```

一部分开发人员并不理解 **default** 分支中的 **break** 有什么价值，因为在他们看来，这里使用 **break** 跳出 **switch**，与 **default** 分支直接运行到结束代码并没有什么区别。下例则说明 **break** 在 **default** 分支中的使用价值：

```
// (参见上例, ...)
default:
    if (!isNaN(parseInt(chr))) break;
    chr = chr.toUpperCase();
}
```

缺省情况下，**break** 子句作用于循环语句的最内层，或者整个 **switch** 语句，因此它不必特别地指定中断语句的范围。但 **break** 子句也具有有一种扩展的语法，以指示它所作用的范围。该范围用声明过的标签来表示。例如：

```
break my_label;
```

这使得 **break** 子句不但可以使用在循环与条件分支内部，也可使用在标签化语句(labeled statement)的内部。如下例：

```
/**
 * 显示输入字符串的末 10 个字符
 */
var str = prompt('please input a string', '12345678910');

my_label: {
    if (str && str.length < 10) {
        break my_label;
    }
    str = str.substr(str.length-10);
}

// other process...
alert(str);
```

这种情况下，**break** 子句后的这个 **my_label** 不能省略——尽管 **break** 位于 **my_label** 所表示的语句范围之内。因此，以下三种用法都将触发语法编译期的触发脚本异常：

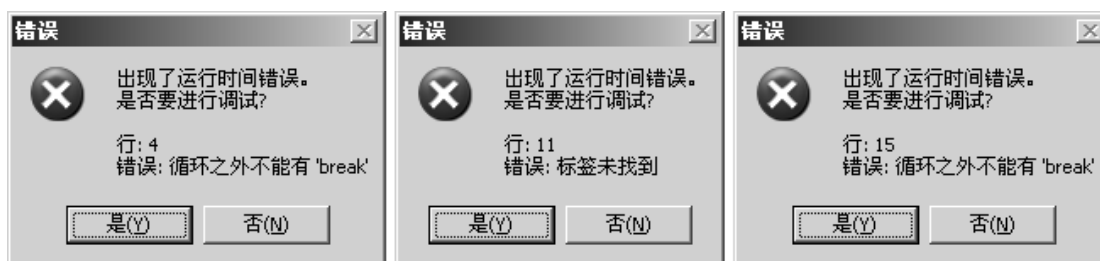
```
7    my_label: {
8        if (str && str.length < 10) {
```

```

9      // 错误一：在标签化语句中用使用 break 而不带 lable
10     break;
11   }
12   str = str.substr(str.length-10);
13 }
14
15 if (true) {
16   // 错误二：在标签化语句的范围之外引用该标签
17   break my_label;
18 }
19 else {
20   // 错误三：在有效的范围 (标签化语句、循环和 switch 分支) 之外使用 break;
21   break;
22 }

```

三种错误情况下的异常信息如下图所示：



应该注意到：错误一与错误三的异常信息是一样的。由于语法上我们是可以“在循环之外使用 break 子句的”，因此事实上这个提示信息所指的是“unlabeled break”——即在脚本引擎内部，认为“不带标签的 break”仅能用于循环(和 switch 分支)的内部。

1.4.4.3. continue 子句

continue 仅对循环语句有意义，因此它只能作用于 for、for..in、while 和 do..while 这些语句的内部。在缺省情况下，它表明停止当前循环并跳转到下一次循环迭代开始执运行。例如：

```

// 声明一个 "工人(Worker)" 类
function Worker() {
  this.headSize = 10;
  this.lossHat = false;
  this.hat = null;
  this.name = 'anonymous';
}

```

```

}

// 声明工人使用的"帽子(Hat)"类
function Hat() {
    this.size = 12;
}

// 有三个工位(或更多)
works = new Array(3);

// 这里有一段业务逻辑, 例如给每个工人发帽子, 或者工作中丢掉帽子
// works[2] = new Worker();

// 现在检查每个工人的帽子, 给没帽子的工人补发一个
for (var i=0; i<works.length; i++) {
    if (!works[i]) continue;
    if (!works[i].lossHat) continue;

    works[i].hat = new Hat(works[i].name);
    works[i].lossHat = false;
}

```

我们看到 `continue` 使得代码的结构变得简单了。我们可以用 `continue` 尽早地清理掉一些分支, 这样在循环体的尾部, 就只剩下符合条件的数据了。于是, 我们就可以在这里很轻松地写出“干净”的业务代码。

`continue` 后面也可以带一个标签, 这时它表明从循环体内部中止, 并继续到标签指示处开始执行。

如果这个标签指示的语句不是一个循环语句, 即使是标签作用于的一批语句中包括循环, `JavaScript` 引擎也认为是一个语法错误。也就是说, `continue` 后面的标签, 只能是对单个的循环语句有意义。例如:

```

// (... , 接上面的代码)

// 建立一个(空的)仓库
library = {};
library.hats = function(sex) {
    return [];
}

```

```

breakToHere : {
  for (var i=0; i < works.length; i++) {
    if (!works[i]) continue;
    if (!works[i].lossHat) continue;

    var oldHats = library.hats(works.sex);
    for (var j=0; j < oldHats.length; j++) {
      if (oldHats[j] && (oldHats[j].size > works[i].headSize)) {
        works[i].hat = oldHats[j];
        works[i].lossHat = false;
        delete oldHats[j];

        continue breakToHere;
      }
    }

    works.hat = new Hat(works.name);
    works.lossHat = false;
  }
}

```

这段代码是可以执行的，但是如果你在 `breakToHere` 后面加一对大括号(如下面的代码那样)，脚本解释引擎就会认为出错了：

```

breakToHere : {
  for (var i=0; i < works.length; i++) {
    // ...
  }
}

```

因为 `continue` 不允许跳转到“当前循环/外层循环语句的起始”之外的其它任何地方。

1.4.4.4. return 子句

`return` 子句只能用在函数之内。同一个函数之内允许存在多个 `return`。当函数被调用时，代码执行到第一个 `return` 子句则将退出该函数，并返回 `return` 子句所指定的值；当 `return` 子句没有指定返回值时，该函数返回 `undefined`。例如：

```

7 function test( tag ) {
8   if (!tag) {

```

```

9      return;
10   }
11
12   return tag.toString();
13 }
14 var v1 = test('');
15 var v2 = test(1234);

```

这里的“第一个 `return` 子句”是指逻辑含义上的、第一个被执行到的 `return` 子句，而不是物理位置上的。例如在上面的代码中，第 8 行以空字符串为入口参数调用 `test()`，因此将从第 3 行的 `return` 返回；而第 9 行调用 `test()` 时，则从第 6 行的 `return` 返回。

当使用 `void` 运算调用函数时，`return` 子句指定的返回值将被忽略。这使得下面的代码返回 `undefined`。

```

function test2( tag ) {
    return true;
}
var v3 = void test2();

```

最后一行代码使得 `v3` 赋值为 `undefined`。但这是 `void` 运算的结果，而并不是说 `return` 子句没有返回值。两者看起来效果一致，但本质却不同。

当执行函数的逻辑过程中没有遇到 `return` 子句时，函数将会执行到最后一条语句(末尾的大括号处)，并返回 `undefined` 值。

1.4.5. 流程控制：异常

异常是比前面所提到的其它子句复杂许多的一种流程控制逻辑。异常与一般子句存在着本质的不同：一般子句作于用语句块的内部，并且是编程人员可预知、可控制的一种流程控制逻辑；而异常正好反过来，它作用于一个语句块的全局，处理该语句块中不可以预知、不可控制的流程逻辑。

结构化异常处理的语法结构如下：

```

try {
    tryStatements
}
catch (exception) {
    catchStatements
}

```

```
}  
finally {  
    finallyStatements  
};
```

该处理机制被分为三个部分（上述语法只说明了其中两个部分），包括：

- 👉 触发异常：使用 `throw` 语句可以在任意位置触发异常，或由引擎在执行过程中内部地触发异常；
- 👉 捕获异常：使用 `try...catch` 语句可以（在形式上表明）捕获一个代码块中可能发生的异常；
- 👉 忽略异常：使用 `try...finally` 语句可以忽略指定代码块中发生的异常，并保证 `finally` 语句块中的代码总是被执行。

在上述语法中，`finally{...}` 块是可选的，但如果存在同一级别的 `catch() {...}` 块，则它 `finally` 块必须位于 `catch` 块之后。在执行上，`catch` 块是先于 `finally` 块的。但如果在 `finally` 执行中存在一个未被处理的异常——例如在 `finally` 之前没有 `catch` 处理块，或在 `catch`、`finally` 块处理中又触发了异常，那么这个异常会被抛出到更外一层的 `try...catch/finally` 中处理。

`finally{...}` 语句块的一个重要之处在于它“总是在 `try/catch` 块退出之前被执行”。这一过程中常常被忽略的情况包括^①：

```
// 在(函数内部的)try块中使用 return 时，finally 块中的代码仍是在 return 子语之前执行的  
try {  
    // ...  
    return;  
}  
finally {  
    ...  
}  
  
// 在(标签化语句的)try块中使用 break 时，finally 块中的代码仍是在 break 子句之前执行的  
aLabel:  
try {  
    // ...  
    break aLabel;  
}  
finally {  
    ...  
}
```

^① 类似情况还包括 `continue` 子句。在《JavaScript 权威指南》中包含了一个这样的例子，读者请自行参考。

最后我们讨论 `throw` 语句。这个语句既可以作用于上述语法的 `try {...}` 块，也可以作用于 `catch` 与 `finally` 块。无论是在哪个位置使用，它总是表明触发一个异常并终止其后的代码执行。`throw` 语句后应当是一个错误对象^①：`Error()` 构造器的实例，或通过“`catch (exception)`”子句中捕获到的异常 `exception`。

有趣的是，如果 `throw` 语句位于一个 `finally{...}` 语句块中，那么在它之后的语句也不能被执行——这意味着 `finally{...}` 语句中的代码“不一定”能被完整地执行。同样的道理，即使不是使用 `throw` 语句显式地触发异常，在 `finally{...}` 块中出现的任何执行期异常也可能中止其后的代码执行。

因此，对于开发者来说，应尽可能保证 `finally{...}` 语句块中的代码都能安全的、无异常的执行。如果不能确信这一点，那么应当将那些不安全的代码移入到 `try{...}` 块中。

1.5. 面向对象编程的语法概要

JavaScript 中，面向对象框架看起来有一套自己的语法规则，但其实很多规则都是演化自此前所述的“声明、(表达式)运算、语句”这一基本体系。例如对象属性的存取与方法调用，就是一种简单的表达式语句。

在本书的第四章（“JavaScript 的非函数式语言特性”）中我们还将更加详细地详述面向对象编程，因此在接下来几个小节中，我们将仅仅概述在 JavaScript 中编程的基本语法。这些为面向对象设计的语法元素包括：

类型	语法元素	语法	含义	注
(直接量)	{ ... }	<pre>{ propertyName_1: expression_1, ... propertyName_n: expression_n }</pre>	(一般)对象直接量	*
	[...]	<pre>[element_1, ... element_n]</pre>	数组对象直接量	
	/ ... / ...	<pre>/ expression pattern / flags</pre>	正则表达式对象直接量	**

^① 可以任意创建一个对象并由 `throw` 抛出，而该对象也总是能被 `catch` 捕获的。仔细考查 `Error()` 对象可知：它相对于 `Object()` 并没有什么特殊性，是一个简单的构造器而已。

运算符	new	new <i>constructor</i> [(<i>arguments</i>)]	创建指定类的对象实例	
	in	<i>propertyName</i> in <i>object</i>	检查对象属性	
	instanceof	<i>objectInstance</i> instanceof <i>constructor</i>	检查变量是否是 指定类的实例	
	delete	delete <i>expression</i>	删除实例属性	
	.	<i>object</i> . <i>Identifier</i>	存取对象成员	
	[]	<i>object</i> [<i>string_expression</i>]	(属性、方法)	
语句	for ... in	for ([var] <i>variable</i> in < <i>object</i> <i>Array</i> >) <i>statements</i> ;	列举对象成员	
	with	with (<i>object</i>) <i>statements</i> ;	设定语句默认对象	

* 各表达式和语法元素可以写在同一行，或者不同的行。此处使用这样的代码格式只为了清晰地展示语法。

** 正则表达的立即值必须写在同一行。此处使用这样的代码格式，是为了突出两个“/”的语法分隔符(它们与斜体字符混在一起时难于识别)。

1.5.1. 对象直接量声明与实例创建

一般情况下，你可以使用直接量声明对象，或者用 **new** 关键字创建新的对象实例。此外，你也可以通过宿主程序来添加自己的构造器，以便 JavaScript 中使用 **new** 关键字来创建。更加特殊的方法是，你可以直接在宿主环境中创建对象实例，某些引擎的脚本代码中要持有并使用它——不过这已经超过了本小节讲述的范围，因此将放在本书的第三部分来讲述。

1.5.1.1. 使用构造器创建对象实例

在 JavaScript 中，有两个方法来“初始化对象实例”。一种是通过用户代码，利用这个 **this** 引用来初始化；另一种则是通过构造原型实例来初始化。我们这里只讲前者。后一种方法，我们将放到第四章的“JavaScript 的面向对象语言特性”中去讲。

通过 **new** 运算，你可以使用构造器来产生对象实例。这里的构造器，可以是你自己写的 JavaScript 代码(构造函数)，也可以是 JavaScript 内置的(或宿主程序扩展的)构造器。下面的示例简要说明构造器创建实例的方法：

```
// 可以被对象方法引用的外部函数
```



```

function getValue() {
    return this.value;
}

// 构造器 (函数)
function MyObject() {
    this.name = 'Object1';
    this.value = 123;
    this.getName = function() {
        return this.name;
    }
    this.getValue = getValue;
}

// 使用 new 运算符，实现实例创建
var aObject = new MyObject();

```

实例创建时需要使用一个函数作为“构造器(constructor)”。构造器其实就是一个普通的函数，如上面的函数 `MyObject()`。不过在该函数执行过程中，JavaScript 将传入 `new` 运算所产生的实例，以该实例作为函数上下文环境中的 `this` 对象引用。这样一来，在构造器函数内部，就可以通过“修改或添加 `this` 对象引用的成员”来完成对象构造阶段的“初始化对象实例”的工作——就象我们在这个例子中声明的构造器 `MyObject()` 一样。

`new` 运算的语法规则为：

```
obj = new constructor[(arguments)];
```

当参数表为空时，与没有参数表是一致的。因此下面两行代码是等义的：

```

// 示例 1：下面两行代码等义
obj = new constructor;
obj = new constructor();

```

但是，我们不能认为 `constructor` 后面的括号是函数调用的括号。因为如果是函数调用的括号，那么下面的代码就应该是合理的了(但事实上，这行代码对于构造器来说是错误的用法)：

```

// 示例 2：错误的代码
obj = new (constructor());

```

所以，不能错误地认为：`new` 运算是“产生对象实例，并调用 `constructor` 函数”——尽管看起来很象是这样。

但是，如果我们的确不打算让 `new` 后面的函数作为构造器，而只是作为函数使用，那么我们的确可以使用下面的代码来做一些特殊的功能：

```
function foo() {  
    var data = this; // <<- 这里暂存了 this，你当然也可以不暂存它。  
    return {};  
    // or  
    // return new Object();  
}  
  
// 示例 3：将 foo() 视为普通函数  
obj = new foo();
```

在这个例子中，最终 `obj` 也会被赋值为一个对象实例。但这个实例并不是 `new` 运算产生的，而是 `foo()` 函数中返回的——注意，使用这种方法的时候，只能通过 `return` 返回一个引用类型的直接量或对象实例，但不能是值类型的直接量（例如不能是 `true`、`'abc'` 之类）。当用户试图返回这种值类型数据时，脚本引擎会忽略掉它们，则仍然使用原来的 `this` 引用——而 `foo()` 函数内部保留的 `new` 运算产生的实例，也可以拿来其它的作用（例如保存私有数据）。

将 `foo()` 视为普通函数的方法，虽然带来了一些特殊效果，但是也破坏了对象的继承链。关于这种技术对继承链产生的影响，在“1.4 JavaScript”中讲述。

在构造器中“暂存的 `this` 实例的一个引用”是一个有用的技巧，——尽管你也可以不暂存它——[关于这个技巧的使用，请参见第 X X 章。](#)

最后，我们回到“将 `new` 右边的运算元视为构造器处理”的情况上来。由于“构造器”在 JavaScript 是由函数（直接量或对象）来承担的，因此如果改变运算顺序，事实上也可以使用“返回函数的运算”来得到构造器。例如：

```
function getClass(name) {  
    switch (name) {  
        case 'string': return String;  
        case 'array': return Array;  
        default: return Object;  
    }  
}  
  
// 示例 4：用表达式(运算)作为 new 的运算元  
obj = new (getClass());
```

在示例 3、4 中，我们没有将 `foo()` 和 `getClass()` 作为构造器来使用，但我们

却用到了 `new` 运算的特性——并且，令人惊奇地产生了一些效果。

这里已经涉及到 JavaScript 的函数式特性。因此，更多的内容我们将放到“第四章 JavaScript 的函数式语言特性”中讲述。

1.5.1.2. 对象直接量声明

对象直接量声明比用构造函数来得简单方便。它的基本语法是：

```
obj = { propertyName: expression[, ...] }
```

示例如下：

```
// 示例：一些已声明过的变量或标识符
function getValue() {
    // ...
}

// 对象直接量声明
var aObject = {
    'name': 'Object1',
    value: 123,
    getName: function() {
        return this.name;
    }
    getValue: getValue
}
```

这里的名字(`propertyName`)可以用字符串来表示，也可以只是一个标识符。因为这是语句语法，因此我们通常都用标识符，只有在特殊的情况下，才使用它的字符串格式。

这些“特殊的情况”通常是指：

- 👉 使用的标识符不满足 JavaScript 对标识符的规则；
- 👉 特殊的、强调的属性名。

例如你可以用“`abc.def`”来做属性名，但这并不是一个合法的标识符；也可以用数字“1”来做属性名，同样它也不是合法的标识符。这时，就可以象下面这样声明：

```
obj = {
    'abc.def': 123,
    '1': 456
}
```

```
};
```

对于“名字：值”对的右边，可以是任何类型的立即值，也可以任何表达式运算的结果。因此下面的声明也是合法的：

```
// 示例 1：嵌套的对象立即值声明
```

```
obj = {  
  'obj2': {  
    name: 'MyObject2',  
    value: 1234  
  }  
};
```

```
// 示例 2：使用函数(表达式)的返回值
```

```
function getValue() {  
  return 100;  
}  
  
obj = {  
  name: 'MyObject3',  
  value: getValue()  
}
```

如果一对大括号“{ }”中间没有任何“名称：值”对，那么将得到一个“空的对象”——在某些时候，我会称其为“干净的对象”，关于这一点，请参阅“ ”。

空字符串和点号也可以作为属性名——尽管可能没有什么实用性。此外，JavaScript 对数字属性名还有一些特殊使用的情况。在后面的章节中会独立出来再讲。

某些类的对象实例也可以使用直接量的方式声明。具体来说，这包括数组(Array)、正则表达式(RegExp)。另外，空对象也以直接量“null”的形式存在(当然，你也可以把它看成常量，亦或者语法关键字)。下面的代码简要的说明这三种直接量的声明：

```
// 示例 1：数组对象
```

```
var arrayObject = [1, 'abcd', true, undefined];
```

```
// 示例 2：正则表达式对象
```

```
var regexpObject = /^a?/gi;
```

```
// 示例 3: 空对象
var nullObject = null;
```

为什么不把字符串、布尔值、数字的直接量也放在本小节中来说明呢？因此这三种直接量声明的变量，其类型 (`typeof`) 是基类型。它们表现得象一个相应的类实例，是因为使用了“包装对象”的技术。

1.5.2. 对象成员列举、存取和删除

可以用下面的方法列举成员的显式成员列表：

```
var obj = {
    // ..
}

// 示例：列举成员的显式成员
for (var n in obj) {
    // 在变量 n 中保存有成员名
    alert('name: ' + n + ', value: ' + obj[n]);
}
```

你可以列举到每一个显式的成员名，但你并不能确知该对象的设计者是打算让这个成员作为一个方法 (`method`)，或者属性 (`property`)，亦或者是事件句柄 (`event`)。在 JavaScript 中，任何类型的值都可以成为对象属性，因此这也包括“函数类型的值”；而对象方法和事件句柄的类型信息，也都是“函数类型(或者函数对象实例)”。因此，你根本没有(任何的)办法来辨识它们——也就是说，在 JavaScript 中，你不可能从成员的类型上准确了解对象/类的设计者的原始意图。

我们强调可以列举的是“显式的成员名”，其原因在于 JavaScript 约定了一些隐含的成员名称（例如所有对象都应该具有的“`toString`”方法），这些名称在 `for .. in` 语句中不会被列举出来。对于一些引擎来说，这些隐含的成员名称总是不被列举(例如在 JScript 引擎中)；而对于另一些引擎中，只要覆盖、重写这个名称，它就可以被列举出来了(例如在 SpiderMonkey JavaScript 引擎中)。因此在跨引擎的设计中，我们并不能依赖于 `for .. in` 语句来获取对象成员名称。

在一些书籍中，“隐含的成员名称”被称为“内置成员”或“预定义成员(方法或属性)”。

现在，我们通过一些方法得到了成员的名字(或者我们在设计代码时，便已经自知了)。有了名字，我们就可以存取它。这有两种方法：

```
var obj = {  
  value: 1234,  
  method: function() { }  
}  
  
// 方法 1: 使用对象成员存取运算符 “.”  
var aValue = obj.value;  
  
// 方法 2: 使用对象成员存取运算符 “[ ]”  
var aValue = obj['value'];
```

“.” 和 “[]” 都是对象成员存取运算符，所不同的是：前者右边的运算元必须是一个标识符，后者中间的运算元可以是变量、直接量或表达式。“.” 是 JavaScript 中唯一一个用 “标识符” 作为运算元的运算符。

由于 “.” 号要求运算元是标识符，因此对一些不满足标识符命名规则的属性，就不可以使用 “.” 号。例如我们前面提到过的 “abcd.def”、“1” 和 “.” 这些属性名。这种情况下就只能使用 “[]” 运算符，例如：

```
var obj = {  
  'abcd.def': 1234,  
  '1': 4567,  
  '.': 7890  
}  
  
// 示例：需要使用 [] 运算符的一些情况  
alert(obj['abcd.def']);  
alert(obj['1']);  
alert(obj['.']);
```

在某些情况下，你可能需要删除一个对象的属性，以使它不能被 for .. in 语句列举。这时你可以使用 delete 运算符。

delete 可以删除很多东西(包括对象成员和数组元素)，但它不能删除：

- 👉 用 var 声明的变量；
- 👉 直接继承自原型的成员。

举例来说，你可以这样使用 delete 运算符：

```
var obj = {
```

```

    method: function() { },
    prop: 1234
}
global_value = 'abcd';
array_value = [0,1,2,3,4];

function testFunc() {
    value2 = 1234;
    delete value2;
}
// 调用 testFunc() 函数，函数内部的 delete 用法也是正确的
testFunc();

// 以下四种用法都是正确的
delete obj.method;
delete obj['prop'];
delete array_value[2];
delete global_value;

```

delete 不能删除继承自原型的成员。但如果你修改了这个成员的值，你仍然可以删除它，这将使它恢复到原型的值。关于这一点的真相，是 **delete** 运算事实上是删除了实例的成员表中的值——你可以通过阅读“1.4JavaScript 中的原型继承”的内容来了解成员继承与值的获取。下面的例子只描述这种现象的表面：

```

function MyObject() {
    this.name = "instance's name";
}
MyObject.prototype.name = "prototype's name";

// 创建后，在构造器中 name 成员被置为值"instance's name"
var obj = new MyObject();
alert( obj.name );

// 删除该成员
delete obj.name;
// 显示 true，成员名仍然存在
alert( 'name' in obj );
// 恢复到原型的值"prototype's name"
alert( obj.name );

```

这个例子也说明：**delete** 不能通过实例来删除原型的、以及父代类原型的

成员（本例中的 `name` 成员是在原型中定义的）。如果真的需要删除该属性，你只能对原型实例进行操作——当然，由于这是原型，所以它会直接影响到这个类构造的所有实例。下面的例子说明这一点：

```
function MyObject() {
    // ...
}

// 在原型中声明属性
MyObject.prototype.value = 100;

// 创建实例
var obj1 = new MyObject();
var obj2 = new MyObject();

// 示例 1：下面的代码并不会使 obj1.value 被删除掉
delete obj1.value;
alert(obj1.value);

// 示例 2：
// 下面的代码可以删除掉 obj1.value。但是，
// 由于是对原型进行操作，所以也会使 obj2.value 被删除。
delete obj1.constructor.prototype.value;
alert(obj1.value);
alert(obj2.value);
```

手册中提及到 `delete` 仅在删除一个不能删除的成员时，才会返回 `false`。其它情况下——例如删除不存在的成员，或者继承自父代类 / 原型的成员——都将返回 `true`。请注意下面的示例：

```
function MyObject() {
}

MyObject.prototype.value = 100;

// 该成员继承自原型，且未被重写，删除返回 true。
// 由于 delete 操作不对原型产生影响，因此 obj1.value 的值未变化。
var obj1 = new MyObject();
alert( delete obj1.value );
alert( obj1.value );

// 尝试删除 Object.prototype，该成员禁止删除，返回 false
alert( delete Object.prototype );
```


这种因“不能删除成员”而返回 `false` 的情况并不多。我所知道的不能删除的成员大概有 `Function` 对象的 `length`、`prototype`、`arguments` 等极少数的几个——而且这还依赖不同的脚本引擎，例如在 `Mozilla` 中 `arguments` 就是可以删除的了。大多数情况下成员都能够被删除。甚至包括全局对象 `Global` 的成员，例如：

```
alert( delete isNaN );
```

但用 `delete` 操作来删除宿主对象的成员时，也可能存在其它的问题。例如下面这个例子中，属性 `aValue` 就删除不掉，而在取值时又触发异常^①：

```
// code in Internet Explorer 5~7.x
aValue = 3;

// 显示 true
alert('aValue' in window);
delete aValue;

// 条件仍然为真，然而用 alert() 显示值时却出现异常
if ('aValue' in window) {
    alert(aValue);
}
```

尽管并没有资料提及 `delete` 运算会导致异常（上例中是取值而非删除操作导致异常），但这种情况的确会发现。如果你试图删除宿主（例如浏览器中的 `windows`）的成员，那么这种可怕的灾难就会发生了：

```
// code in Internet Explorer 5~7.x
window.prop = 'my custom property';
delete window.prop;
```

这时发生的异常可能是“对象不支持该操作”，表明宿主不提供删除成员的能力——不过不同的宿主的处理方案也不一致，例如 `firefox` 浏览器就可以正常删除。

1.5.3. 属性存取与方法调用

很多语言中的“面向对象系统”是由编译器，或者指定的语句/语法来实现的。例如 `Delphi`、`C++`、`Java` 等都是如此。`Delphi` 从 `Pascal` 过渡而来，`C++` 则源自 `C` 语言，为了实现“面向对象”，他们都各自扩展了自己的语法。

^① 这里疑为 `JScript` 的一个 `BUG`。

然而，JavaScript 并不这样，它是通过“运算”来实现面向对象特性的典型。例如我们在这一小节要说到属性存取与方法调用。

如果我们已经创建了一个对象实例 `obj`，那么我们可以用上一节所提到的两种方法之一来存取对象属性：

```
// 方法 1：使用对象成员存取运算符 “.”  
var aValue = obj.value;  
  
// 方法 2：使用对象成员存取运算符 “[ ]”  
var avalue = obj['value'];
```

我们已经说过，“.”与 “[]”是两个运算符。在此基础上，JavaScript 并没有做任何语法扩展就实现了方法调用：

```
// 方法 1：基于运算符 “.”  
obj.method();  
  
// 方法 2：基于运算符 “[ ]”  
obj['method']();
```

所以，事实上 JavaScript 中的方法调用，就是指“取得对象的成员，并执行函数调用运算”。具体到语法的实现方法上，只需要使得“.”和 “[]”运算的优先级高于“()”即可。

关于方法调用与函数调用，还涉及到 `this` 对象的维护。这一点我们可以放到“第五章 `this` 对象的维护”中讲述。

在 JScript 环境中还存在一种特例：如果对象一个 `ActiveObject` 对象实例，那么该对象的方法“也许”能不使用“()”即得到调用。如下例所示：

```
var excel = new ActiveXObject("Excel.Application");  
alert('enter to close excel...');  
excel.Exit;
```

1.5.4. 对象及其成员的检查

我们可以用“`for .. in`”语句来列举对象成员，但如果要检查一个对象是否具有某个成员，就不用这么麻烦了。JavaScript 使用 `in` 运算来得到这个结果：

```
// 示例：用 in 运算检测属性
```

```
'value' in obj
```

这种运算最经常出现在引擎环境兼容的代码中，例如：

```
// 示例：为不同的脚本引擎初始化一个 XMLHttpRequest 对象
if ('XMLHttpRequest' in window) {
    // for ie7.0+, or mozilla and compat browser
    return new XMLHttpRequest();
}
else if ('ActiveXObject' in window) {
    // for ie 4.0 .. 6.0
    return new ActiveXObject('Microsoft.XMLHTTP');
}
else {
    throw new Error('can't init ajax system... ');
}
```

但这样的检查并不一定可靠。因为更早版本的 JavaScript 引擎可能根本没有实现“in”运算。因此在 JavaScript v1.3 之前，我们被建议用下面的代码来做相同的事：

```
// 示例：兼容更早版本的对象检查代码
if (window.XMLHttpRequest) {
    // for ie7.0+, or mozilla and compat browser
    return new XMLHttpRequest();
}
else if (window.ActiveXObject) {
    // for ie 4.0 .. 6.0
    return new ActiveXObject('Microsoft.XMLHTTP');
}
else {
    throw new Error('can't init ajax system... ');
}
```

按照前面讲述过的知识，“window.XMLHttpRequest”这个运算用于取 XMLHttpRequest 成员。如果该属性存在，则返回该属性的值。接下来，这个值被类型转换为 if 语句所需的布尔值“true”。所以当对这些属性做取值运算时，在 if 语句中的效果正好“相当于”使用 in 运算：

```
// '.'运算的表达式返回一个值，被类型转换为值 true
if (window.XMLHttpRequest) { ... }

// 'in'运算的表达式返回值 true
```

```
if ('XMLHttpRequest' in window) { ... }
```

同样，按照 JavaScript 语言的约定，取一个“不存在的属性”的值并不会导致异常，而是返回“**undefined**”。而“**undefined**”可以被类型转换为 if 语句所需的布尔值“**false**”。所以当对一个“不存在的属性”做取值运算时，在 if 语句中的效果也正好相当于使用 in 运算。

但是，应该注意到这里有一个“隐含的类型强制转换”。正是因为这个强制转换，所以下面所有情况都可能导致表达式结果为 **false**：

```
var obj = {};  
  
function _in(obj, prop) {  
    if (obj[prop]) return true;  
    return false;  
}  
  
// 检测不存在的属性  
alert( _in(obj, 'myProp') );  
  
// 检测某些有值的属性，仍会返回 false  
var propertyNames = [0, '', [], false, undefined, null];  
for (var i=0; i<propertyNames.length; i++) {  
    alert( _in(obj, propertyNames[i]) );  
}
```

我们看到，一个属性即使存在，如果他的值为 **propertyNames** 中任一的一个，都将导致 **_in()** 检测返回 **false**。这显然没有达到我们的目的。因此，在 JavaScript 另外推荐一种方案，以在旧版本中检测属性是否存在：

```
// 示例：兼容更早版本的对象检查代码  
if (typeof(window.XMLHttpRequest) != 'undefined') {  
    // for ie7.0+, or mozilla and compat browser  
    return new XMLHttpRequest();  
}  
else if (typeof(window.ActiveXObject) != 'undefined') {  
    // for ie 4.0 .. 6.0  
    return new ActiveXObject('Microsoft.XMLHTTP');  
}  
else {  
    throw new Error('can't init ajax system... ')
```

```
}
```

由于前面说过“取不存在的属性将返回 `undefined`”，因此用 `typeof` 运算来检测该值，一定是“`undefined`”字符串。这样看起来是有达到目的了，但事实上还是存在问题。如下例：

```
var obj = {
  'aValue': undefined
};

// 示例：使用 typeof 运算存在的问题
if (typeof(obj.aValue) !== 'undefined') {
  ...
}
```

这种情况下，`aValue` 属性是存在的，但我不能通过 `typeof` 运算来检测它是否存在。正是由于这个缘故，在 WEB 浏览器中，DOM 的约定是“如果一个属性没有初值，则应该置为 `null`”，这就是因为早期的 JavaScript 不能有效地通过 `undefined` 来检测属性是否存在。同样的道理，JavaScript 规范在较后期的版本中，便要求引擎应该实现 `in` 运算，用以更有效地检测属性。

如果我们需要检测“对象是否具有某个属性”，这应当使用 `in` 运算符；而另一些情况下，还需要检测“对象是否是一个类的实例”，这时就应该使用 `instanceof` 运算符了。如下例所示：

```
// 示例：用 instanceof 运算检测实例类别
obj instanceof MyObject
```

在 JavaScript 中没有严格意义上的“类类型”，因此实例只能通过它的构造器来检测。也就是说，上面的代码会在下面的条件下成立：

```
// 声明构造器
function MyObject() {
  // ...
}

// 实例创建
var obj = new MyObject();

// 显示 true
alert(obj instanceof MyObject);
```

`instanceof` 将会检测类的继承关系。因此一个子类的实例，在对父类做 `instanceof` 运算时，仍会得到 `true`。如下例：

```
function MyObjectEx() {
    // ...
}
MyObjectEx.prototype = new MyObject();

// 实例创建
var obj2 = new MyObjectEx();

// 检测构造类，显示 true
alert(obj2 instanceof MyObjectEx);

// 检测父类，显示 true
alert(obj2 instanceof MyObject);
```

1.5.5. 可列举性

对象成员是否能被列举，称为成员的可列举性。由于 JavaScript 的成员可以被概括为属性（方法调用被实现为“存取对象属性并作为函数执行‘()’运算”），因此“成员是否能被列举”也就可以表述为“属性是否能被列举”。这也是在 JavaScript 中可以看到 `propertyIsEnumerable()` 方法，而不会有（也不必有）检测方法是否可列举的操作。

当某个对象成员不存在，或它不可列举时，则对该成员调用 `propertyIsEnumerable()` 方法将返回 `false`——比较常见的情况是，JavaScript 对象的内置成员是不能被列举的。例如：

```
var obj = new Object();

// 不存在'aCustomMember'，显示 false
alert( obj.propertyIsEnumerable('aCustomMember') );

// 内置成员不能被列举
alert( obj.propertyIsEnumerable('constructor') );
```

这种情况下，你可以用“`in`”运算检测到该成员，但不能用“`for..in`”语句来列举它。

《JavaScript 权威指南》指出了 ECMAScript 对 `propertyIsEnumerable()` 的一

个规范错误。按照规范，该方法是不检测对象的原型链的——但如果规范更合理一些，那么该方法应该检测原型链的。为什么呢？因为事实上原型链上的（父代类的）成员也是可以被“for..in”语句列举的，但它的 `propertyIsEnumerable()` 却是 `false`。如下例所示：

```
// 定义原型链
function MyObject() {
}
function MyObjectEx() {
}
MyObjectEx.prototype = new MyObject();

// 'aCustomMember' 是原型链上的（父代类的）成员
MyObject.prototype.aCustomMember = 'MyObject';

// 显示 false, 因为“继承来的成员”不能被列举
var obj1 = new MyObjectEx();
alert( obj1.propertyIsEnumerable('aCustomMember') );

// 列举 obj1 时, 将包括'aCustomMember'
for (var propName in obj1) {
    alert( propName );
}
```

但是，既然规范是这样要求的，那么也必须“按照错误的法子来实现”——这是所谓标准的强制性。结果是，在脚本引擎中 `propertyIsEnumerable()` 被实现为“只检测对象的非（自原型链继承而来的）继承属性”。

为了说明这句比较坚涩的话，下面为读者提供一个 Internet Explorer 5.0x 版本上实现的 `propertyIsEnumerable()` 方法——IE5.x 中的 JScript 5 引擎未实现该方法。读者可以看看这个方法的实现逻辑：

```
// code from Qomo Project.
Object.prototype.propertyIsEnumerable = function(proName) {
    for (var i in this) {
        if (i==proName) break;
    }
    if (i != proName) return false;

    // if clone from prototype, return false.
    return this[proName] !== this.constructor.prototype[proName];
}
```

```
}
```

在 JScript 与 JavaScript 中，对可枚举性中的“继承自……”这个描述的理解也是不同的。这导致 `propertyIsEnumerable()` 的行为，以及（包括内置成员在内的）可见性的处理并不一致。关于这一点，我们在[下一章](#)会再次讲到。

1.5.6. 缺省对象的指定

JavaScript 提供 `with` 语句，以使得开发人员可以在一个代码块中显式地指定缺省对象。如果不指定缺省对象，则默认使用宿主程序在全局指定的缺省对象（在浏览器环境中的缺省对象是 `window`）。

一个代码块中，用户代码要么是在访问一个对象成员，要么是在访问对象（或值）。二者有非常明显的区别：是否使用对象成员存取运算符。因此，脚本引擎可以容易地区分下面的代码的含义：

```
1 // 示例 1: 存取对象成员
2 var obj = new Object();
3 obj.value = 100;
4
5 // 示例 2: 访问(全局的)对象或值
6 value = 1000;
```

`with` 语句可以改变上述第 6 行代码的语义，让它存取 `obj` 对象的成员（而不是全局变量）。例如：

```
7 // (续上例)
8 with (obj) {
9     value *= 2;
10 }
11
12 // 显示 200
13 alert(obj.value)
14 // 显示 1000
15 alert(value);
```

1.6. 运算符的二义性

在 JavaScript 中，很多运算符是具有二义性的。因为这些二义性的存在，

一方面使 JavaScript 显得更灵活，另一面也使一些 JavaScript 的代码显得更难理解。

严格来说，我们这里说的二义性并不是学术含义上的。JavaScript 会通过一套语法规则、优先级算法或默认的系统机制来处理这些“存在二义的代码”，使代码在运行时存有某一确定的含义。但我在这里要说的是，这些代码在开发人员的理解中是存有二义的，或是不能那么清晰、直观地理解代码的意图。

我们接下来讲述这些由运算符二义性带来的、不清晰的语法。这些语法之所以存在，不是因为我在玩弄技巧，而（在某些时候）只是出于想使语法更清晰。但如果原本只是修改代码过程中的误操作，而 JavaScript 引擎未能检出错误，继而使得系统中容存了“不可知”的隐患，那就很可怕了。

本书不打算讨论正则表达式中的符号与运算符之间的二义性问题。基于这个前提，下表基于对运算符的考察，列出存在二义性的语法元素：

	运算符/符号	运算符含义	其它含义	备注
具有二义性的运算符	,	连续运算符	参数分隔符 对象 / 数组声明分隔符	1.6.5
	+	增值运算符 正值运算符 连接运算符		1.6.1
	()	函数调用运算符	强制运算（优先级） 参数声明	1.6.2
	?:	条件运算符	: 号有声明标签的含义 : 号有声明 switch 分支的含义 : 号有声明对象成员的含义	1.6.3
	[]	数组下标	数组直接量声明 对象成员存取	1.6.6
其它	{ }		函数直接量（代码部分的）声明 对象直接量声明 复合语句	1.6.4
	;		空语句 语句分隔符	(*)

*注：空语句可以视作语句分隔符使用中的特例，因此本书不讨论这个二义性的细节。

1.6.1. 加号“+”的二义性

首先指出，本小节讨论的有关加号“+”的二义性问题，部分内容同样适

用于运算符“+”。

JavaScript 中，(单个的)加号有三种作用。它可以表示字符串连接，例如：

```
var str = 'hello ' + 'world!';
```

或表示数字取正值的一元运算符，例如：

```
var n = 10;  
var n2 = +n;
```

或数值表达式的求和运算，例如：

```
var n = 100;  
var n2 = n + 1;
```

三种表示法里，字符串连接与数字求和是容易出现二义性的。因为 JavaScript 中对这两种运算的处理将依赖于数据类型，而无法从运算符上进行判读。我们单独地看一个表达式：

```
a = a + b;
```

是根本无法知道它真实的含义是在求和，亦或是在做字符串连接——JavaScript 引擎做语法分析时，也无法确知。

加号“+”带来的主要问题与另一条规则有关。这条规则是“如果表达式中存在字符串，则优先按字符串连接进行运算”。例如：

```
var v1 = '123';  
var v2 = 456;  
  
// 显示结果值为字符串'123456'  
alert( v1 + v2 );
```

这会在一些宿主中出现问题。例如浏览器中，由于 DOM 模型的许多值看起来是数字，但实际上却是字符串。因此试图做“和”运算，却变成了“字符串连接”运算。下面的例子说明这个问题：

```
<img id="testPic" style="border: 1 solid red">
```

我们看到这个 id 为 testPic 的 IMG 元素(element)有一个宽度为 1 的边框——我们省略了默认的单位 px(pixel，像素点)。但是如果你试图用下面的代码来加宽它的边框，你会得到错误(一些浏览器忽略这个值，另一些则弹出异常，还有一些浏览器则可能会崩溃)：

```
var el = document.getElementById('testPic');
```

```
el.style.borderWidth += 10;
```

因为事实上在 DOM 模型里，borderWidth 是一个包括了单位的字符串值，因此这里的值会是"1px"。JavaScript 本身并不会出错，它会完成类似下面的运算，并使得值赋给 borderWidth：

```
el.style.borderWidth = '1px' + 10; // 值为 '1px10'
```

这时，浏览器的 DOM 模型无法解释“1px10”的含义，因此出错了。当你再次读 borderWidth 值时，它将仍是值 1px——那么，我们怎么证明上述的运算过程呢？下面的代码将表明 JavaScript 运算的结果是 1px10，但赋值到 borderWidth 时，由于 DOM 忽略掉这个错误的值，因此 borderWidth 没有发生实际的修改：

```
alert( el.style.borderWidth = '1px' + 10 ); // 值为 '1px10'
```

这个问题追其根源，一方面在于我们允许了省略单位的样式表写法，另一方面也在于脚本引擎不能根据运算符来确定这里的操作是数值运算还是字符串连接。后来 W3C 推动 XHTML 规范，试图从第一方面来避免这个问题，但对开发界的影响仍旧有限。因此，浏览器的开发商提供的手册中，都会尽可能地写明每一个属性的数据类型，以避免开发人员写出上面这样的代码。

最正确的写法是：

```
var el = document.getElementById('testPic');

// 1. 取原有的单位
var value = parseInt(el.style.borderWidth);
var unit = el.style.borderWidth.substr(value.toString().length);
// 2. 运算结果并附加单位
el.style.borderWidth = value + 10 + unit;

// 如果你确知属性采用了缺省单位 px，并试图仍然省略单位值，
// 那么你可以用下面这种方法（我并不推荐这样）：
// el.style.borderWidth = parseInt(el.style.borderWidth) + 10;
```

1.6.2. 括号“()”的二义性

我们来看下面语句中的括号“()”应该是什么含义呢：

```
var str = typeof(123);
var str = ('please input a string', 1000);
```

第一个 `typeof`，看起来好象 `typeof` 被当成了函数使用——这是不是说明 `typeof` 是一个“内部函数”呢？而第二行代码，你相信它会成功执行并使 `str` “意外地”被赋值为 1000 吗？

括号首先可以作为语句中的词法元素。例如函数声明中的“虚拟参数表”。例如：

```
// 声明函数时，括号用作参数表。
function foo(v1, v2) {
    //...
}
```

第二种情况，就是括号只作为“传值参数表（这有别于函数声明中的‘虚拟参数表’）”——注意这里它并不表达函数调用的含义。目前为止，这只出现在 `new` 关键字的使用中：`new` 关键字用于创建一个对象实例并负责调用该构造器函数，如果存在一对括号“`()`”指示的参数表，则在调用构造器函数时传入该参数表。例如：

```
// 构造对象时，用于传入初始化参数
var myArray = new Array('abc', 1, true);
```

——关于这一点的具体分析，我们已经在“1.5.1.1 使用构造器创建对象实例”中讲述过了。

接下来，它也可以在 `for`、`if`、`while` 和 `do..while` 等语句中用来作为限定表达式的词法元素：

```
// for 语句 (for..in 语句类同)
for ( initialize; test; increment )
    statement

// if 语句
if ( expression )
    statement

// while 语句
while ( expression )
    statement
```

```
// do..while 语句
do
    statement
while ( expression )
```

在充当 if、while 和 do..while 的语句中的词法元素时，括号会有“将表达式结果转换为布尔值”的副作用(参见“”)。除此之外，在语句中的括号，并不产生类似于“运算”这样的附加效果。

第四种情况，是括号“()”用于强制表达式运算。这种情况下，基本含义是我们通常说的强制运算优先级。但事实上，不管有没有优先级的问题，括号总是会强制其内部的代码作为表达式运算。例如我们在这一小节开始列举的例子：

```
var str1 = typeof(123);
var str2 = ('please input a string', 1000);
```

在第一行代码中，“()”强制 123 作为单值表达式运算，当然运算结果还是 123，于是再进行 typeof 运算。所以这里的一对括号起到了强制运算的作用。同样的道理，第二行代码里的一对括号也起到相同的作用，它强制两个单值表达式做连续运算，由于连续运算符“,”的返回值是最后一个表达式的值，因此这个结果返回了 1000。所以我们要意味到，上面的第一行代码并没有调用函数的意思，而第二行代码将使 str2 被赋值为 1000。

最后的这种情况最为常见：作为函数/方法调用运算符。例如：

```
// 有 (), 表明函数调用。
foo();

// 没有 (), 则该语句只是一个变量的返回。
foo;
```

在函数调用过程中，我们需要一再强调：括号“()”的作用是运算符。我们也因此得出推论，当“()”作为运算符时，它只是作用于表达式的运算，而不可能作用于语句。所以，你只能将位于：

```
function foo() {
    return( 1 + 2 );
}
```

这个函数内的、return 之后的括号理解成“强制表达式优先级”，而不是理解成“把 return 当成函数或运算符使用”。

所以从代码格式化的角度上来说，在下面两种书写方法中，第二种才是正确的：

```
// 第一种，象函数调用一样，return 后无空格
return(1 + 2);

// 第二种，return 后置一空格
return (1 + 2);
```

基于同样的理由，无论“`break (my_label)`”看起来如何合理，也会被引擎识别为语法错误。因为 `my_label` 只是一个标签，而不是（可以交“`()`”运算符处理的）运算元。标签与运算元属于两个各自独立的、可重复(而不发生覆盖)的标识符系统。

1.6.3. 冒号“:”与标签的二义性

冒号有三种语法作用：声明直接量对象的成员和声明标签，以及在 `switch` 语句中声明一个分支；冒号还具有一个运算符的含义：在“`? :`”三元表达式中，表示条件为 `false` 时的表达式分支。下面的例子说明这几种情况：

```
// 例 1：用于声明直接量对象的成员
```

```
var obj = {
  value: 100,
  func: function() {
    //...
  }
}
```

```
// 例 2：用于声明标签
```

```
myLabel: {
  // ...
}
```

```
// 例 3：声明 case 分支
```

```
switch (obj) {
  case ... : break;
  default: {
  }
}
```

```
// 例 4: 用于条件(三元)表达式
alert(prompt('y/n?', 'y')== 'y' ? 'yes'
      : 'no');
```

其中，由于三元表达式中的问号“?”没有二义性，因此冒号“:”作为运算符的情况是比较容易识别的；`case` 和 `default` 分支能被作为标签语句的特例来解释（参见《JavaScript 权威指南》）。因此，冒号的二义性问题集中在标签声明与对象成员的问题上。这在下一小节中，我们将给出一个实例来详细说明它。

1.6.4. 大括号“{}”的二义性

大括号有四种作用，四种都是语言的词法符号。第一种比较常见，表示“复合语句”。例如：

```
// 例 1: 表示标签后的复合语句
myLabel : {
    //...
}

// 例 2: 在其它语句中表示复合语句
if ( continon_is_true ) {
    // ...
}
else {
    // ...
}
```

我们在前面曾经说过，在复合语句末尾的大括号之前，语句的“;”号可以省略。我们又说过，有一类“表达式语句”。因此，下面这行代码，就值得回味了：

```
// 例 3: 复合语句中的表达式语句
{
    1,2,3
}
```

这其实是一个只有一条语句的复合语句。这条语句是：

```
1,2,3;
```

语句中的逗号“,”是连续运算符。由于外面有一对大括号，所以我们省略了语句末尾的一个分号。

例 3 中的格式，与大括号的另一种用法就很接近了。这就是对象直接量声明。例如：

```
// 例 4：声明对象直接量
var obj = {
  v1: 1,
  v2: 2,
  v3: 3
}
```

由于上面是一条赋值语句，所以其中的直接量声明部分其实是一个表达式：

```
{
  v1: 1,
  v2: 2,
  v3: 3
}
```

注意这几行代码，它们的整体才是一个表达式——一个直接量的单值表达式。

按照前面的解释，我们可以有两种方法将这个单值表达式变成一个语句。如下：

```
// 方法 1：使用分号的表示法
{
  v1: 1,
  v2: 2,
  v3: 3
};

// 方法 2：使用复合语句的表示法
{
  {
    v1: 1,
    v2: 2,
    v3: 3
  }
}
```

在第二种方法里，两对大括号的含义就完全不同。从语法解析的角度上讲，二者的区别在于，对象立即值声明时，大括号的内部会有一个：

```
propertyName: expression
```


的语法格式，而大括号用做复合语句时没有这项限制。

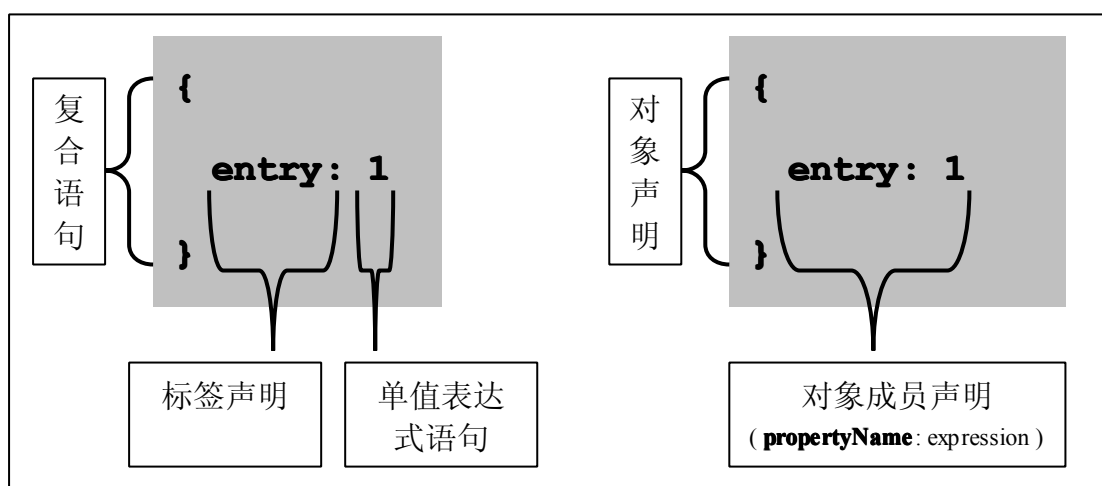
好了，接下来我们要问的问题，就会混淆这两种大括号了。我们看下面这段代码：

```
// 示例大括号的语法歧义
if (true) {
  entry: 1
}
```

在这段代码中，由于在 if 后面的语句可能是以下三者之一：

- 👉 一个单行语句
- 👉 一个表达式(语句)
- 👉 一个由大括号包含起来的复合语句

那我们这里是应该把这一对大括号理解为一个对象声明的语法符号呢，还是表示复合语句的语法符号？对照一下下面两种解释：



左边这种情况中，if 语句后面是一个复合语句；右边这种情况中，if 语句后面被理解成了一个由“对象直接量”构成单值表达式语句。所以仅从语法上来看，我们无法正确地识别这两种情况。

JavaScript 对此作出的解释是“语句优先”——因为语句的语法分析在时序上先于代码执行。所以表达式被作为次级元素来理解。因此我们输出语句的值，会是按左边的方式解析执行的结果值 1：

```
// 显示上面的示例语句的值
var code = 'if (true) { entry: 1 }';
var value = eval(code);
alert( value ); // 显示值 1
```

如果用户代码需要用右边的方式来解析执行，那么应该用一对括号 “()” 来强制表明 “表达式运算”。如下：

```
if (true) ({  
    entry: 1  
});
```

这样一来，返回的结果值就是一个对象直接量。下面的代码用于验证这一结果：

```
// 使用括号 “( )” 强制表达式运算的结果  
var code = 'if (true) ({ entry: 1 })';  
var value = eval(code);  
alert( value ); // 显示值"[object Object]"
```

大括号的第三种用法，是被用于函数直接量声明时的语法符号：

```
foo = function() {  
    //...  
}
```

但由于存在 “function ()” 这样的语法关键字作为前缀，因此基本上是不会混淆的。当然，你得留意阅读一下小节 “1.4.1.3 函数调用语句”，理解一下下面这段导致语法分析失败的代码。需要先强调的是，这里的语法歧义是括号 “()” 运算符导致的，而不是大括号 “{ }” 的问题：

```
function foo() {  
    //...  
}(1,2);
```

最后，大括号也是结构化异常处理的语法符号：

```
try {  
    // 代码块 1  
}  
catch (exception) {  
    // 代码块 2  
}  
finally {  
    // 代码块 3  
}
```

需要强调的是，这里的大括号并不是复合语句的语法符号，而是结构化异常处

理中用来分隔代码块的符号。因为，如果它是复合语句的语法标识符，那么它必然是可以用单行语句来替代的。然而你会发现下面的语句会出现语法分析错误：

```
// 示例：遗漏了 try 后面的语法符号，因此下面的代码导致语法分析错误
try
    i=100;
catch (e) { /* 略 */ }
```

1.6.5. 逗号 “,” 的二义性

我们先看看下面两行语句的不同：

```
// 例 1
a = (1, 2, 3);

// 例 2
a = 1, 2, 3;
```

接下来，你能预想下面这个语句的结果吗？

```
// 例 3
a = [1, 2, (3,4,5), 6]
```

这种用法产生的混乱，是因为逗号 “,” 既可以是语法分隔符，又可以是运算符所导致的。

在上面的例 1、例 2 中，逗号都被作为“连续运算符”在使用。例 1 中的括号是强制运算符，因此它的效果是运算表达式：

```
(1, 2, 3)
```

并将结果值赋值给变量 **a**。由于该表达式是三个（直接量的）单值表达式连续运算，其结果值是最后一个表达式，亦即是数值 3。因此例 1 的效果是“变量 **a** 赋值为 3”。而例 2 则因为没有括号来强制优先级，因此按默认优先级会先完成赋值运算，因此效果是“变量 **a** 赋值为 1”。

我们在前面提到过“语句是有值的”。对于例 1、2 来说，尽管例 1、2 在变量赋值的效果上并不一样，但语句的值却都是 3。这是因为例 2 在完成赋值运算“**a** = 1”之后，还仍将继续执行连续运算，而最后一个表达式就是直接量 3。下面的代码考察这两行代码的语句返回值：

```
// 显示数值 3
alert( eval('a = (1, 2, 3);') );
// 显示数值 3
```

```
alert( eval('a = 1, 2, 3;') );
```

同样的，我们也可以知道例 3 的结果是使变量 a 赋值为：

```
[1, 2, 5, 6]
```

这是因为表达式“(3,4,5)”将会被先运算并返回结果值 5，作为数组声明时的一个元素。例 3 中要强调的是，逗号在这里分别有“数组声明时的语法分隔符”和“连续运算符”两种作用。

我们再回顾一下例 2。该示例所展示的问题容易出现在变量声明中。例如我们可能会写出下面这样的代码：

```
// 例 4  
var a = 1, 2, 3;
```

然而这个代码却并不会象例 2 一样“正常地执行”。因为例 4 中，逗号被解释成了语句 var 声明时用来分隔多个变量的语法分隔符，而不是连续运算符。而语句 var 要求用“,”号来分隔的是多个标识符，而数值 2 和数值 3 显然都不是合法的标识符，因此例 4 的代码会在语法解释期就提示出错。

例 4 的代码出自某个真实的项目。最早这个项目使用如下的代码并可以正常运行：

```
function loadFromRemote() {  
    // 获取服务器端的数据片断并返回成字符串，例如返回：  
    // "a = 1, 2, 3;"  
}  
  
function getRemoteData() {  
    var ctx = loadFromRemote();  
    return eval(ctx);  
}
```

但在一次改写后，结果出错了。修改后的代码是这样：

```
function getRemoteData_2(name) {  
    var ctx = loadFromRemote();  
    eval( 'var ' + ctx );  
    return eval(name);  
}
```

这次改写的原因在于想获取某个指定名称 name 的变量值，但又不想因为 eval() 执行而破坏全局变量，因此在代码前加“var”，使语句变成局部变量声明和赋

值。然而正如例 4 所示的，在某些情况下，这行代码会失效（在语法解释期就失败）。而这个错误并不总是出现，因此浪费了大量的时间来调试。

根源在于我们将原来的代码从“表达式执行”变成了“变量声明语句”。由于代码已经从语义上发生了根本性的改变，因此并不能预期的效果来执行。事实更合理的改写方法是：

```
function getRemoteData_2(name) {  
    var ctx = loadFromRemote();  
    eval('var ' + name);  
    eval(ctx);  
    return eval(name);  
}
```

这与前面的区别在于：我们将变量声明与变量赋值分开，这样保证了“eval(ctx)”这行代码与原来的语义完全相同，而只是限定了它的变量作用域。这不但与我们改写这个函数的本意就一致，而且也避免了 var 声明语句带来的“语法检测”的副作用。

另外的一个教训是：var 声明会使连续运算表达式变为连续声明语句。

存在同样混乱的问题的，还有在“1.3.7 特殊作用的运算符”中列举过的示例：

```
// 显示最后表达式的值"value: 240"  
var i = 100;  
alert( (i+=20, i*=2, 'value: '+i) );
```

在这个示例的 alert() 函数调用中，还存在一对括号：用来表示强制运算。因为 alert() 是函数，所以它会把下面的代码：

```
alert( i+=20, i*=2, 'value: '+i );
```

理解为三个参数传入。又由于函数参数表的传值顺序是从左至右的，因此这行代码被理解为：

```
alert( 120, 240, 'value: 240' );
```

但 alert() 这个函数自身只会处理一个参数，因此最终显示的结果会是值“120”——需要补充的是，这行代码执行完毕之后，变量“i”的值已经变成 240 了。

为了让 JavaScript 理解这里的逗号“,”是连续运算符，而不是函数参数表的语法分隔符，我们在这里加入了一对强制(表达式)运算的括号：

```
alert( (i+=20, i*=2, 'value: '+i) );
```

这样，由于外层的括号是函数调用符，因此它将内部的

```
(i+=20, i*=2, 'value: '+i)
```

理解为一个参数，并且是一个需要求值的表达式。因此这里的括号就顺理成章地理解为了“强制运算符”，下面的代码：

```
i+=20
i*=2
'value: '+i
```

也被作为这个强制运算符的一个运算元：由","连结起来的一个表达式。直到绕了这样大的一个圈子，逗号才被合理地解释为“连续运算符”。

1.6.6. 方括号“[]”的二义性

下面的代码会有语法错误吗？

```
/**
 * 方括号的二义性示例 (一)
 */
a = [ [1] [1] ];
```

是的，很奇怪，这个语句并没有语法错误。尽管我们几乎不理解这行代码的含义，但 JavaScript 解释器可以理解，它会使得 a 被赋值为[undefined]。也就是说，右边部分作为表达式，可以被运算出一个结果：只有一个元素的数组，该元素为 undefined。

这个例子其实是在工程中有实际意义的。它最早出现在我的一个项目中，因为我用如下的方式来声明一个二维表——这种用数组来实现本地数据表的方法其实很常见，[sourceforge](#) 上还专门有用这种思路实现的 JSDB 和 JSQL：

```
var table = [
  [ ... ],
  [ ... ],
  [ ... ]
];
```

但是我在一次“copy/paste”时，漏掉了一个逗号。因此变成了下面这个样子：

```
/**
 * 方括号的二义性示例 (二)
 */
```

```
var table = [
  ['A', 1, 2, 3]    // <-- 这里漏掉了一个逗号
  ['B', 3, 4, 5],
  ['C', 5, 6, 7]
];
```

然而这在 JavaScript 看来却是正常的语法，因此没有在语法解释时给出警告，而是在运行时产生了意想不到的效果——上面的代码被 JavaScript 引擎解释成了如下的数组声明：

```
var table = [
  undefined,
  ['C', 5, 6, 7]
];
```

出现这个问题的原因，首先在于方括号既可以用于声明数组的直接量，又可以是存取数组下标的运算符。对于最开始的例子：

```
a = [ [1] [1] ];
```

来说，它相当于在执行下面的代码：

```
arr = [1];
a = [ arr[1] ];
```

由于 JavaScript 中直接量可以参与运算，因此第一个 “[1]” 被理解成了一个数组的直接量，它只有一个元素，即 “arr[0] = 1”。接下来，由于它是对象，所以 arr[1] 就被理解为取下标为 1 的元素——很显然，这个元素还没有声明。因此 “[1][1]” 的运算结果就是 undefined，而 a = [[1][1]] 就变成了：

```
a = [ undefined ];
```

根据这个分析，我们可以推论出下面的一些结果：

```
a = [ [][100] ];    // 第一个数组为空数组，第二个数为任意数值，都将得到 [ undefined ]
a = [ [1,2,3][2] ]; // 第一个数组有三个元素，因此 arr[2] 是存在的，故而得到 [ 3 ]
a = [ [][] ];      // 第一个数组为空，是正常的；但第二个作为下标运算时缺少索引，故语法错。
```

我们也知道方括号不但可以作为数组下标运算符，也可以作为对象成员的存取运算符。因此下面的代码也可以得以运行：

```
a = [ []['length'] ]; // 第一个数组为空数组，因此将返回它的长度。结果得到 [ 0 ]
```

这可能看起来是个小麻烦，但下面的例子就可能真的会出现在你的代码中了：

```
/**
 * 方括号的二义性示例 (三)
 */
array_properties = [
  ['pop'],
  ['push']
  ['length']
  // more ...
];
```

无论出于什么原因，你可能忘掉了 “[push]” 后面的那个逗号，然而你将得到如下的一个数组(问题是，看起来它还能继续参与运算)：

```
arr_properties = [
  ['pop'],
  0
];
```

不过这样的二义性仍然不够复杂，因为我们还是无法解释在“示例（二）”中为什么会出现一个 **undefined**。而“示例（二）”之复杂，就在于它集中呈现了下面三个语法二义性带来的恶果：

- 👉 方括号可以被理解为数组声明，或下标存取；
- 👉 方括号还可以被理解为对象成员存取；
- 👉 逗号可以被理解为语法分隔符，或连续运算符。

我们再来看这个例子：

```
/**
 * 方括号的二义性示例 (二)
 */
var table = [
  ['A', 1, 2, 3]    // <-- 这里漏掉了一个逗号
  ['B', 3, 4, 5],
  ['C', 5, 6, 7]
];
```

它的第二行并没有被理解成一个数组，也没有直接地被理解成数组元素的存取。相反，它理解成了四个表达式连续运算。因为从语法上来说，由于 [‘A’, 1, 2, 3] 是一个数组对象，因此后面的方括号 “[]” 会被理解为“对象属性存取运算符”。那么规则就变成了这样：

- 👉 如果其中运算的结果是整数，则用于做下标存取；

👉 如果其中运算的结果是字符串，则用于对象成员存取。

由于在这里 `['B', 3, 4, 5]` 的作用是运算取值，因此 `'B', 3, 4, 5` 被当成了四个“各由一个直接量构成的”表达式。而这里的“,”号，就不再是数组声明时的语法分隔符，而是连续运算符。

在上一小节中，我们说过“,”号作为连续运算符时，是返回最后一个表达式的值。于是，`'B', 3, 4, 5` 作为表达式，就得到了 5 这个值。然后，JavaScript 会据此把下面的代码：

```
var table = [  
  ['A', 1, 2, 3]    // <-- 这里漏掉了一个逗号  
  ['B', 3, 4, 5],  
  ['C', 5, 6, 7]  
];
```

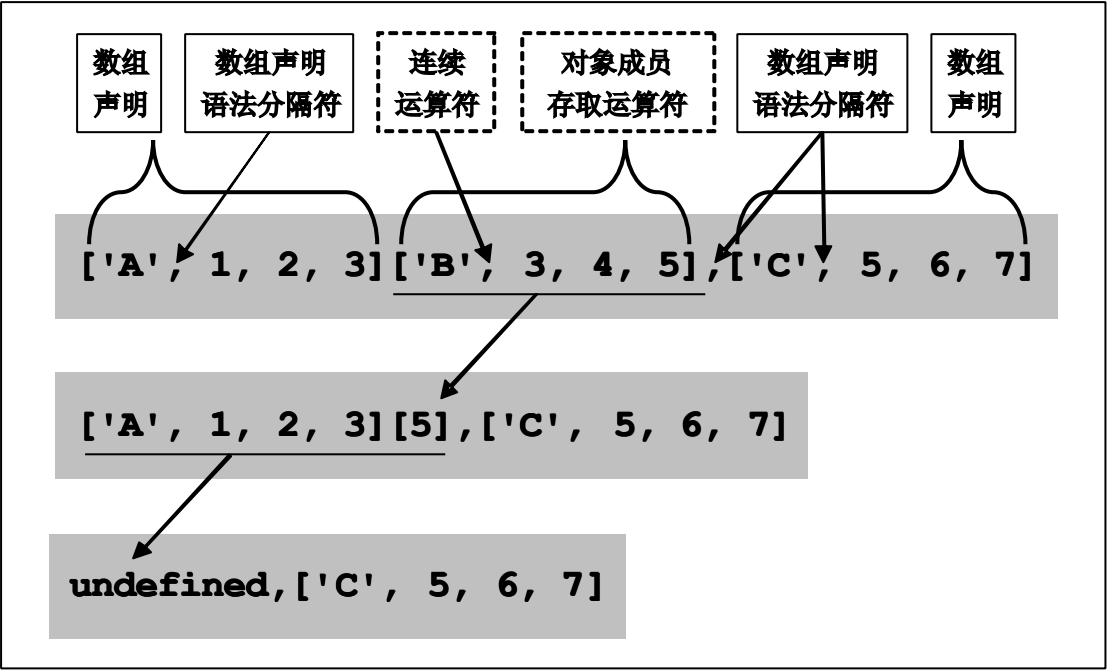
理解成：

```
var table = [  
  ['A', 1, 2, 3][5],  
  ['C', 5, 6, 7]  
];
```

而 `['A', 1, 2, 3]` 这个数组没有第五个元素，于是这里的声明结果变成了：

```
var table = [  
  undefined,  
  ['C', 5, 6, 7]  
];
```

下面这张图，更加清晰地展示了这个声明与运算交叠在一起的过程：



用同样的方法，我们就不难解释下面的代码了：

	用户声明	引擎的理解
例一	<pre>var table = [['A', 1, 2, 3] // <-- 这里漏掉了一个逗号 ['B', 3, 4, 0], // 理解为取下标 0 ['C', 5, 6, 7]];</pre>	<pre>var table = ['A', ['C', 5, 6, 7]];</pre>
例二	<pre>var table = [['A', 1, 2, 3] // <-- 这里漏掉了一个逗号 ['B', 'length'], // 理解为取属性 'length' ['C', 5, 6, 7]];</pre>	<pre>var table = [4, ['C', 5, 6, 7]];</pre>

第二部分 语言特性 及基本应用

过程式风格仍与能够解释（“运行”）程序的计算机最为接近。计算机的显著特性是内存，其中各个单元可以单独更新，它与程序设计语言中的变量正好对应。

面向对象风格是基于过程式风格的。它是后者的一个变种，差别并不太大。尽管过程现在称为“方法”，调用一个过程现在表述为“发送一条消息”。

我认为较之于过程式程序设计，函数式和逻辑式程序设计更有机会成为“逢源”的风格。

—— Pascal 之父，1984 年图灵奖获得者 Niklaus Wirth

引自《程序设计语言概念》

第三章 JavaScript 的非函数式语言特性

术语“命令式”(imperative)来自于命令和动作，这种计算模型就是基于基础机器的一系列动作。

——《程序设计语言概念和结构》，Ravi Sethi。

1.1. 概述

源于对计算过程的认识的不同而产生了不同的计算模型，基于这些计算模型进行的分类，是计算机语言的主要分类方式之一。在这种分类法中，一般将语言分为四大类：命令式语言、函数式语言、逻辑式语言和面向对象程序设计语言。

本小节将首先讨论程序的本质，并从这个本质出发，以另一种的分类法对程序语言做出分类：命令式语言和说明式语言。本章则基于该分类法讨论 JavaScript 的非函数式语言特性，其内容组织如下：

分类	子类	章节
命令式	冯·诺依曼（结构化编程）	本章第二节～第五节
	面向对象（面向对象编程）	本章第六节～第十节
说明式	函数式	第五章
	(其它)	

1.1.1. 命令式语言与结构化编程

“命令式”这个词事实上过于学术化。简单的说，我们常见的编程语言，从“低级的”汇编语言到“高级的”C++，以及我们常用的 Basic、Pascal 之类都是命令式语言。

命令式语言的演化分为“结构化编程”和“面向对象编程”两个阶段。无论是从语言定义还是从数据抽象的发展来看，面向对象编程都是结构化编程的自然延伸。

结构化程序设计语言中，对结构的解释包括两个部分：程序的控制结构和数据结构。所谓控制结构，即是顺序、分支和循环这三种基本程序逻辑；所谓数据结构，包括基本数据结构和复合数据结构，且复合数据结构必然由基本数

据结构按复合规则构成。

整个命令式语言的发展历程，都与“冯·诺依曼”计算机系统存在直接关系。这种计算机系统以“存储”和“处理”为核心，而在编程语言中，前者被抽象成“内存”，后者被抽象成“运算(指令或语句)”。

所以命令式语言的核心就在于“通过运算去改变内存(中的数据)”——我们应该注意到：软件程序与硬件系统在本质上就存在如此亲密的关系。

由于命令式语言的实质是面向存储的编程，所以这一类语言比其它语言更加关注存储的方式。在程序设计的经典法则“程序=算法+结构”中，命令式语言是首先关注“结构”的，亦即是“数据结构(或类型系统)”。下表说明在 x86 系列上的“数据结构”上的简单抽象：

自然语义	机器系统	编程系统	语言/类型系统
基本数据单元	16/32/64 位系统	位、字节、字、双字	bit、byte、word、dword, ...
连续数据块	连续存储块	数组、字符串、结构体 ^①	array, string, struct, ...
有关系的数据片断	存储地址	指针、结构、引用	pointer, tree, ...

命令式语言在运算上也是基于上述的“存储结构”来进行算法设计。例如表检索，那么通常认为是在一个“连续数据块”中找到指定的、一个“基本数据单元”中的值。例如：

```
/**
 * programming language: javascript
 * params:
 *   - key, a value. etc, type of byte
 *   - table, a array. etc, type of byteArray.
 */
function SearchInTable(key , table) {
  for (var i=0; i<table.length; i++) {
    if (table[i] == key) return true;
  }
  return false;
}
```

基于上例的基本需求和数据结构的设定，推论出“有序表检索效率更高”，

^① C 语言中的“结构”类型在 Pascal 中称为“记录(record)”。为了避免与本章中所述的“算法+(数据)结构”中的结构混淆，在后文中，编程语言中的“结构”称为“结构体”。而“结构”一词，通常用来表达概念上的“数据结构(或类型系统)”。

并进一步提出有表排序的相关算法(例如冒泡排序),设计出“二分法查找”等有序表检索算法。再后来,算法从“对原始数据排序”进化到“对数据映射排序”,从而有了更快速的“hash 排序”与“hash 检索”。海量数据处理的原始模型才由此逐渐形成。而所有这些算法的原始基础,仍旧是上表中对“数据表现形式”的设定。

象 Frederick P. Brooks, Jr. 这样的先驱们,很早就意识到“程序=算法+结构”的价值。Brooks 就在《人月神话》中指出“数据的表现形式是编程的根本”。所以,正是大师们在“结构体”上的不懈努力,成就了 C/Pascal 这样的结构化编程语言^①、windows、linux/unix 这些伟大的操作系统,以及 ORACLE、MSSQL 这些数据库系统。

然而,从基于 x86 系统的汇编语言,到代表近三十年来“高级语言”发展史的 C、Pascal、Basic,以及在关系数据库方面独领风骚的 SQL……所有这些在通用软件开发领域耳熟能详的编程语言,都困守在“冯·诺依曼”体系之中,无数的经典语言与编程大师谨遵“程序=算法+结构”这句断言,而从未在本质上出现过任何的突破。

在另一种分类体系中,SQL 被归类为“第四代程序设计语言(4GL, Fourth-Generation Language)”。在该分类体系中,还包括机器语言(1GL)、汇编语言(2GL)和高级语言(3GL)和图形化程序设计语言(5GL)。这是一种较为笼统的以语言演化的次序、功用及实现方式来分类的方法。

1.1.2. 结构化的疑难

在命令式语言发展上的所有努力,最终都必然面临的问题是“如何抽象数据存储”。我们知道,在结构化编程时代,解决这个问题的是“结构体”。但是一方面,结构体在数据表达上过度的弹性带来了编程设计中的不规范,因此事实上在结构化编程时代,除了关系性数据库之外,并没有什么一致的、规范化的编程模型出现。另一方面,结构体是根本上是面向机器世界的“存储描述”,因此它的抽象层次明显过低。

^① 结构化程序设计中的“结构”并不是语言概念中的“结构体”,二者并没有必然的关系。结构化分析方法的要点是根据数据的处理过程,自顶向下地分解系统模块。这个过程被称为结构化。它的产物是模块(module)、过程(procedure)等之间的交互与接口,而不是一个具体的数据结构。从软件开发过程来讲,编程语言中的数据类型(包括结构体等),来自于分析阶段的数据建模。

抽象层次过低带来的问题至少包括三个方面。

其一，结构体直接与实体相关并直接呈现在使用者的面前，因此开发人员必须面临数据的具体含义与关系。

在命令式语言中，变量(数据)的作用域首先按冯·诺依曼体系分为数据域与代码域。然后根据编译器的约定，分为局部域、单元域与全局域。一些编译器也约定了“块”级别的作用域，例如 C 语言中的线程锁机制。

然而，结构体本身并不具有隐藏数据域的特性。它只是忠实的反应程序系统与实际应用环境的映射关系。例如一个对房间的描述：

```
(**
 * programming language: pascal
 *)
TRoom = record
  bed: integer;
  desk: integer;
  chair: integer;
  lamp: integer;
  window: integer;
  people: integer;
  // reserved : array [0..300] of byte;
end;
```

我们假设将这个 TRoom 这个结构体应用于一个实际系统中：对于 CAD 系统来说，people 成员显然是多余的；而对于实境系统(例如导游)来说，people 又是主要的成员，其它的则可能由另一个封闭的子系统处理。因此，很直接的问题是，对于更复杂的系统来说，需要更多的、更复杂的“实体与成员”的包含或封装关系，数据对于不同的子系统、结构体和逻辑代码来说，应该存在不同的可见性。

在结构化时代，处理这个问题的方法，是在 SDK 中约定“带下划线(_)前缀的成员是保留的”，或者直接隐匿掉这些成员的名字，并从文档中彻底清除它们(如上例中的 reserved 成员)。这些做法，除了激发程序员们探索不止的欲望，并最终写出《某某系统未公开文档技术大全》之类的著作之外，别无它用。

其二，结构体的抽象更面向于数据存储形式的表达和算法实现的方式，脱离了具体使用环境与算法的结构缺乏通用性

这其实是一个非常致命的问题。因为大多数情况下，结构一旦设定，算法也就确定了。例如对 Zip 文件的文件头的描述：

```
/**
 * programming language: pascal
 */
TCommonFileHeader = packed record
    VersionNeededToExtract: WORD;      // 2 bytes
    GeneralPurposeBitFlag: WORD;       // 2 bytes
    CompressionMethod: WORD;          // 2 bytes
    LastModFileTimeDate: DWORD;        // 4 bytes
    Crc32: DWORD;                     // 4 bytes
    CompressedSize: DWORD;             // 4 bytes
    UncompressedSize: DWORD;           // 4 bytes
    FilenameLength: WORD;              // 2 bytes
    ExtraFieldLength: WORD;            // 2 bytes
end;

TLocalFile = packed record
    LocalFileHeaderSignature: DWORD;   // 4 bytes (0x04034b50)
    CommonFileHeader: TCommonFileHeader; // 26 bytes
    filename: AnsiString;              // variable size
    extrafield: AnsiString;            // variable size
    CompressedData: AnsiString;        // variable size
end;
```

这个结构体的设计中，TLocalFile 是作为文件头被写入 .zip 文件的每一个子文件的压缩数据的头部的，其中前 30 个字节可以作为一个完整的数据块直接保存。但是，TCommonFileHeader 的设计中，Crc32 和 CompressedSize 这两个成员，却需要在完成数据压缩之后才能写入。也就是说，在做 .zip 压缩文件时，要在添加完一个文件的压缩数据后，将文件读写指针移回到这个位置来重写这两个值。

结构的设计就决定了算法的实现。这已然是很明显的事。现在所有的 .zip 文件都以这种方式标识着子文件，因此我们已经没有任何办法来修改算法，使结构被重用到新的算法，或者使其它算法被应用到这个旧的结构。

结构体的设计直接面向存储，过低的抽象层次使得重用性大大地降低。程序、系统和开发人员被约束在结构的设计与调整之上，而不是关注于现实系统

的实现之上。

其三，僵化的类型与僵化的逻辑并存，影响了业务逻辑的表达。

现实生活中，人们并不关心“关注对象”的类型，而只关注于其具体的逻辑。例如人们在饥饿时只关注“吃”，并不关注于吃的是什么。

在一个子系统的逻辑产生的时候，子系统事实上只关注于逻辑作用于一个该作用的对象，而并不关注这个对象的形式(如类型)。例如财务人员面对手中的一堆票据，他只关心这些票据的总金额是多少，因此“求总计”的子系统最直接的实现方法，就应当类似于“财务人员手执一个计算器(或算盘)”：计算系统内部如何处理小数与整数，那是另外的一套法则去保障，而最好不要直接地与原始数据(票据)关联起来。

泛型运算解决的正是这样的问题。在一个强类型系统中，泛型系统象一台计算器或算盘一样，用独立的逻辑(例如 C 语言中模板在编译时生成代码)去应付各种数据类型上的运算法则。而在业务逻辑层面，开发人员只需要将来自输入的原始数据(例如票据)累加即可。

```
/**
 * programming language: C
 * 示例 1: 处理确定类型值的累加函数
 */
long add_values(long a, long b) {
    return (a + b);
}

/**
 * programming language: C
 * 示例 2: 处理不同类型值的累加函数，通过模板(泛型)来解决强类型问题的示例，
 */
#include <iostream.h>

template <class type1, class type2> type1 add_values(type1 a, type2 b) {
    return (a + b);
}

long add_values(long a, int b);
double add_values(double a, long b);

/**
```

```
* call demo
*   v1, v2, v3 模拟输入的可变类型的原始数据
*/
void main(void) {
    long v1 = 1200L;
    int v2 = 1100;
    double v3 = 100.0 / 3;

    cout << "Value: " << add_values(v3, add_values(v1, v2)) << endl;
}
```

强类型与泛型出现的真正原因，仍然是因为“结构体”是面向存储进行的数据抽象。只有抽象层次更高一些，抽象不会影响到存储本身时，这个矛盾才会被真正解决。

1.1.3. “面向对象语言”是突破吗？

在上一节中，我有意地将“结构化的疑难”归结为由“抽象层次过低”所引发的三点，而忽略了“结构化”带来的其它问题。

而这三点，就正是“面向对象”所解决的问题：

- 👉 开发人员必须面临数据的具体含义与关系
- 👉 脱离了具体使用环境与算法的结构缺乏通用性
- 👉 类型与逻辑僵化从而影响了业务逻辑的表达

首先，“面向对象”提出通过更加细化的可见性设定，实现更好的数据封装性及数据域管理。这些可见性标识包括：

可见性	含义	备注
published	已发布，面向特殊系统（例如 IDE）的。	*
public	公开，不限制访问的。	
protected internal	内部保护，访问限于该成员所属的类或从该类派生来的类型。	**
protected	保护，访问限于此程序。	
internal	内部，访问限于此程序或从该成员所属的类派生的类型。	**
private	私有，访问限于该成员所属的类型。	

(*) 在 delphi 中，该可见性仅面向可视化组件库、RTTI 和 IDE。

(**) 部分语言未实现。

通过指定更确定含义的可见性，设计良好的类/对象层次可以极大程度上避免不相干的子系统了解到更多的结构(面向对象系统中的“对象”)细节。

接下来，“面向对象”中的继承被用来解决结构体的通用性问题。如果一个结构所声明的“成员 p”既可以是 A 对象的成员，又可以是 B 对象的成员，并且“成员 p”对两个(或更多)对象中的含义在抽象概念上存在类似，那么就可以在 A 和 B 之上声明一个父类 O，A 和 B 从父类 O 中继承“成员 p”。这样 A、B 具有各自子系统所需的特性，而父类 O 就可以在多个子系统中复用。

最后，解决“强类型”与业务逻辑表达之间的冲突的重任，就落在了“面向对象”系统的“多态性”上。对于任意子系统来说，由于子类 A 与子类 B 都具有父类 O 的特性，因此任意能作用于父类 O 的行为都必然可以作用于 A、B 两个子类。所以，在类型系统检查的过程中，一旦明确“父类行为的抽象”，那么子类如何设计，都不会影响到父类的行为(业务逻辑)。简单的说，如果一个“对象结构”相关的逻辑是确定的，那么这个结构无论如何衍生，逻辑仍旧是确定的。

下面的代码简单说明面向对象系统的这三种特性：

```
// 下面处理成处理器模式!!!!

/**
 * programming language: Delphi
 * (1): 基类及其表达的运算逻辑
 */
type
  TCalcObject = class(TObject)
  end;

  TCalcMachine = class(TObject)
  public
    function calc(obj: TCalcObject): value;
  end;

/**
```

```
* (2): TCalcObject 的子类, 表达各自子系统对数据的理解
*/
type
    TCalc

/**
* (3): 由 1,2 所决定的算法逻辑
*/
```

我们看到,“对象”无疑是比“结构体”更高层次的数据抽象(结构)。它所基于的基础,正是“结构确定,则算法确定”。在这样的前提下,按照“面向对象”的理论,无论怎样进行类衍生,都不会影响到“已经确定的类设计”。

因此结构、数据与逻辑被绑在一起,从而形成了对象/类声明。它包含了数据实体、实体关系以及与实体相关的运算。简而言之,对象不但封装了更多的局部逻辑,还潜在地描述了它如何对整个体系架构与业务逻辑的进行支撑。

但是,我们在这里应该注意:对象只是更高层次的数据抽象。它所基于的,仍旧是对结构的认可。它并不是以对算法的认可为前提的。正是因为它并没有突破“结构影响算法”的边界,所以我们才在面向对象系统中看到一种状况:如果对象的继承树设计得不合理,那么在这个对象系统上的应用开发将会束手束脚——接下来,对继承体系的重构又会影响到业务逻辑(算法)的实现。

对象的继承树是对数据抽象的表达。“结构化”的抽象是实体到结构体的直接映射;“面向对象”的抽象则是实体到类、衍生关系到“类树”的映射。面向对象系统在数据抽象上比结构化系统要复杂,因此更高级而又更难深入。

但同时,由于继承关系是现实系统中非常泛化的一种关系,也是人类社会中的一种普遍关系,因此能够为开发人员理解并应用。这是面向对象系统可以得到发展的根源。

1.1.4. 更高层次的抽象：接口

接口(Interface)这个词在早期开发中使用得很广泛。例如通常说的 API，就是“应用程序接口(Applications Programming Interface)”；HCI，是“人机接口(Human-Computer Interface)。而 pascal 语言中，模块对外部系统的声明也称为接口，并有单独的关键字来标识它，甚至出现过单独的格式文件(.INT)来描述这些接口——在 C 语言中，与此相同的文件被称为头文件(.H)。

但这里要说的不是这些接口。

如果我们将对象系统理解为三个元素的复合体：

- 👉 数据：对象封装了数据体以及数据的存储逻辑；
- 👉 行为：对象向外表现了数据上可以进行的运算与运算逻辑；
- 👉 关系：对象系统设定了一些交互关系，例如观察者模式中的“观察”与“被观察”关系。

那么我们会发现这个对象系统所表达的含义又过于的确定了。也就是说，我们又回到了原来的话题上：数据系统与业务系统耦合度还是过高。

这个问题的根源还是抽象程度过低：我们确定了一个被运算的目标(对象)的结构与行为，其实在一定程度上也就限制了它的抽象性。而接口概念则更加符合我们对“自然系统”的定义：系统提供能力，我们使用系统的能力，而不是关注能力的来源与获取方法。

还是回到开始那个例子：我们需要一个计算系统来求和。但是我们为什么要关注这个计算系统是继承是怎样的一个基础类型呢？有了“基类”的概念后，我们就将在不同的子系统之间挖开了道道沟渠——我们无法让一个 C++ 语言的对象用在 Java 中，也无法让一个继承自 TManualCalc 的对象与继承自 TRobotCalc 的对象互换——如果你一开始设计它们为不同的基类的话。

如果我们需要计算，那么我们其实只关心计算系统能否接受 calc 方法，方法的入口有一些计算元，然后返回计算结果即可。至于这个系统是人工在处理，还是计算机在处理，我们并不是真的那么需要关注。

接口(Interface)提出的观点就是：只暴露数据体的逻辑行为，而不暴露它的数据特性——这里用数据体而不是“对象”，是因为 Interface 并不关注实现者的数据结构特性(尽管具体的语言中，它是与对象或其它的数据类型相关的)。在有了接口的观念之后，我们会发现系统间的关系变得无比清晰明朗：用或者不用。

接口首次从系统或模块中剥离了“数据”的概念，进而把与数据有关的关系也清理了出去——例如引用(对象间的引用是面向对象体系的灾难之源)。因此，接口是一种更高层次的抽象。它是目标系统与计算机系统的功能特性的投影：如果二者的投影一致，则必然是一个能够互换或互证的系统。

接口的高度抽象带来了很多的附加价值。其中之一，就是体系的可描述性。例如某个部署在服务器上的 Web Services，可能是一套由 python 开发极为复杂的系统，但对于外部的接口来说，可能只是一个 Interface：

```
ISearch = Interface
    procedure search;
end;
```

虽然不同的子系统可能对这个接口有自己的描述，例如 Delphi：

```
(**
 * programming language: delphi
 *)
ISearch = Interface
    function search(anything: IKeys): ICollection;
end;
```

但是在这个抽象的系统之外，作为使用者——我，其实只需要从网页中输入一个字符串，至于：

- 👉 系统如何处理
- 👉 是在本地亦或还是远程
- 👉 目标系统是人工的，还是机器的
- 👉 如果是人工的，是一个人，还是一群人
- 👉 或者既不是人工，也不是机器的，而是一群猴子
- 👉

等等这样的一些问题，则是我不需要考虑的。即便有人告诉我说：在远在地球之外的星系，一群猴子在处理这个系统，因而产生了我需要得到的搜索结果，

那么我也会无视——因为我只关心我是否搜索到了我想要的东西。

这就是 Web Services。Web Services 的基础之一，就是更加泛义化的 Interface。而把除了“有没有猴子参与搜索工作”这样有明显答案的问题之外的、所有类似 Interface、Python、目标系统和海量检索等等这些虚头八脑^①的概念深藏在背后，因而成就了一代帝国的软件公司，就是 Google。

——Google 的首页，就是这样的一个 Interface。

1.1.5. 再论语言的分类

到现在为止，我们已经对“语言”进行了好几次的分类。

其中，我们在前言里用“对立与中庸的方法”设定了语言可以分为“动态语言”、“静态语言”与“半动态语言”。在本章开始部分，我们又从“计算范型的角度”将语言分成了命令式语言、函数式语言、逻辑式语言和面向对象程序设计语言四大类。

值得一提的是，第二种分类方法是教科书上的经典分类法。

然而我们在这里还要再讨论一下分类。因为前面讨论的过程中隐含着一个推论：既然命令式语言的实质是面向存储的编程，而面向对象解决的也只是“更高层次的抽象数据存储”的问题，那么面向对象是否也是一种命令式语言呢？

回到编程的经典法则：程序=算法+结构，我们前面就说命令式语言关注于后者，其本质是基于结构的运算，因此可以毫无疑问的说，“面向对象编程”也是一种命令式语言。这两点予以佐证：

- 👉 在语源上，面向对象是命令式语言的直接继承者。例如作为典型代表的 C++ 与 JAVA，在经典教材中，称前者为“结合命令式和面向对象特性的语言”，后者为“基于命令式的面向对象语言”。
- 👉 在实现时，“对象”仍然是基于 x86 的连续存储的概念进行的结构设计。事实上，对象尽管是更高的数据抽象，但仍旧不能摆脱结构对算法的限制。例如 GoF 模式(即是设计，也是实现)，便是在这种限制下的产物。

^① 这个词是我到上海之后学到的第一个“无来由的、奇怪的”词语。意思大概就是莫名其妙、难于解释或者很学术、很象牙的那些东西。按照我在每本书中留下一个彩蛋的做法，我想问一个问题：上海人说“蛮好”，到底是“满好”呢，还是“蛮好”？

进一步地说，“从(经典法则所述的)程序本质出发”进行语言分类，则可以将语言分为“说明式”和“命令式”，前者描述“基于算法的实现”，后则关注“基于结构的运算”。在《程序设计语言——实践之路》中描述了这样的分类体系（加暗部分）：

层次分类			子类	语言示例	注
计算机语言	程序设计语言	说明式	函数式	Lisp/Schemem, ML, Haskell	*
			数据流式	Id, Val	
			逻辑式	Prolog, VisiCalc	
		命令式	冯·诺依曼	Fortran, Pascal, Basic, C, ...	**
			面向对象	Smalltalk, Eiffel, C++, Java, ...	
	数据设计语言	标志语言	...	HTML, XML	
	模型设计语言	建模语言	...	UML	

(*) 说明式语言的几种子分类的区别主要在于“说明”所陈述的主体的不同。例如函数式主要陈述运算规则，数据流式主要陈述数值计算，逻辑式则主要陈述推理过程等等。

(**) 一般概念下的命令式语言，或称为结构化程序设计语言。

运算式语言(expression language)

并没有太多人注意到一种事实：“面向接口的编程方法”已经悄悄地出现了。例如前面提到过的 Web Services，无疑就是基于面向接口编程思想的。而且，面向接口的编程语言(IOPL，Interface Oriented Programming Languages)也已经出现。L. Robert Varney 在 2003 年提交过一份有关 IOP 的研究报告，并在 ARC(一种 Lisp 的方言)中实现过一个语言原型。在最近(2005.12)，Christopher Diggins^①又尝试性地对 IOPL 做过一个定义。Konrad Anton 也在 java 环境中提出了一个 IOPL 语言的实现方案(2006.02)。与此同时，IOP 作为一个概念，更多地出现在 SOA/SOP(Services Oriented Architect/Programming)的实现或阐释中。

然而在上面这种分类体系下，我们也会看到一个问题：接口关注于行为的描述，而不是结构的描述。接口基于的原则并不是“结构确定，则算法确定”，而是“在公司的规约描述下的（算法的）功能，是确定的”。同样，正是因为接口突破了“结构影响算法”的边界，我们才看到接口弥补了 OOP 的不足(例如对象继承树的设计可能不合理)，变成了现代 OOP 编程语言中不可或缺的一

^① 《C++ Cookbook》一书的作者，O'Reilly 2005 年出版。是一种支持 IOP 的 Heron 语言的创建者。

个部分。面向接口的编程，就此成为对面向对象编程方法的一种突破。

这种突破表现在：IOPL 并不是一个命令式语言，因为它缺乏“基于结构”这样的基本特性；IOPL 更象是一种说明式语言，因为它更加面向对算法的描述——例如用接口来描述的 GoF 模式，实际是不单单是陈述架构，也陈述了实现算法。

我们看到，一种在“命令式”的、面向对象编程的实践过程中创建出来的“面向接口编程(IOP)”，却是“说明式”的。这一方面表明 IOP 在 OOP 中实现并应用存在一些思想方法的障碍，另一方面也体现了语言的不同分类之间相互衍生和促进的事实。

同样的，JavaScript 也是语言不同分类间相互衍生的产物^①：它也同时是一种说明式语言和命令式语言。它在两个分类上的表现，分别是“函数式特性”与“命令式(面向对象和过程)特性”。

1.1.6. JavaScript 的命令式语言渊源

1992 年前后，Nombas 公司开发了一种名叫 Cmm(C-minus-minus)的嵌入式脚本语言，希望在保持与 C 和 C++高度兼容的同时，能足够强大，以替代宏操作。这个语言被捆绑在一个名为 CEnvi 的共享软件中。在 Netscape Navigator 发布了早期版本并受到普遍的追捧之后，Nombas 发布了一个可以嵌入网页中的 CEnvi 版本，成为第一个在万维网(WWW, World Wide Web)上使用的客户端脚本语言。

这时出现的 JavaScript，其设计灵感不可避免地受到这种“能嵌入在 Web 页上的脚本语言”的影响。而另一方面，JavaScript 在语法等方面又借鉴自 ANSI C。所以在语源上，JavaScript 直接来自于这两种语言：Cmm 和 ANSI C。其中，Cmm 是经由 C++演化过来的、具备面向对象特征的语言；而 ANSI C，也就是标准 C 语言，是一种“命令式程序设计语言”。

因此，在“非函数式”语言特性方面，JavaScript 语言就不可避免的带有命令式语言和面向对象语言的特性^②。

^① 这种具有交叉分类特性的语言，通常被称为“多范型语言”。例如 Heron 就被称为“命令式多范型编程语言(imperative multi-paradigm programming language)”。而 JavaScript 则支持三种编程范型：函数式、命令式和(基于原型的)面向对象。

^② 值得一提的是，JavaScript 在语言设计上并没有从 Java 身上得到什么灵感。因此事实上这两种语言缺乏最基本的相似性。

(JavaScript 的面象对象渊源)

Around 1992, a company called Nombas began developing an embedded scripting language called C-minus-minus (Cmm for short). The idea behind Cmm was simple: a scripting language powerful enough to replace macros, but still similar enough to C (and C++) that developers could learn it quickly. This scripting language was packaged in a shareware product called CEnvi, which first exposed the power of such languages to developers. Nombas eventually changed the name Cmm to ScriptEase.

ScriptEase is now the driving force behind Nombas products. When the popularity of Netscape Navigator started peaking, Nombas developed a version of CEnvi that could be embedded into Web pages. These early experiments were called Espresso Pages, and they represented the first client-side scripting language used on the World Wide Web. Little did Nombas know that its ideas would become an important foundation for the Internet.

As Web surfing gained popularity, a gradual demand for client-side scripting languages developed. At the time, most Internet users were connecting over a 28.8 kbps modem even though Web pages were growing in size and complexity. Adding to users' pain was the large number of round-trips to the server required for simple form validation. Imagine filling out a form, clicking the Submit button, waiting 30 seconds for processing, and then being met with a message telling you that you forgot to complete a required field. Netscape, at that time on the cutting edge of technological innovation, began seriously considering the development of a client-side scripting language to handle simple processing.

When JavaScript first appeared in 1995, its main purpose was to handle some of the input validation that had previously been left to server-side languages such as Perl. Prior to that time, a round trip to the server was needed to determine if a required field had been left blank or an entered value was invalid. Netscape Navigator sought to change that with the introduction of JavaScript.

The capability to handle some basic validation on the client was an exciting new feature at a time when use of telephone modems (operating at 28.8 kbps) was widespread. Such slow speeds turned every trip to the server into an exercise in patience.

http://p2p.wrox.com/topic.asp?TOPIC_ID=46392

1.2. 基本语法的结构化含义

JavaScript 相对来说是比较易用的，而且它的学习曲线也很平缓。这得益

于它与其它通用语言基本一致的一个特性：基本语法由语句、表达式和变量构成。其中，语句用于展现程序逻辑结构上的组织和流程，表达式用于陈述和实现算法，变量用于定义和处理存储。

本节主要讨论 JavaScript 语句在“展现程序逻辑结构上的组织和流程”方面的能力，也就是所谓的“代码分块”。同时，也讨论由这种结构带来的信息隐藏的效果。

1.2.1. 基本逻辑与代码分块

JavaScript 编写的代码基本上仍以语句为主要表达方式，顺序、分支和循环这三种基本逻辑通常都是以“语句”的形式呈现。例如：

```
if (...) ... else ...;
```

基本逻辑语句是保证程序执行的基本要素，而复合语句则使这种结构能在组织大型程序的同时，能够以模块化形式清晰地呈现出来。例如上面的语句以复合语句的形式呈现的结果为：

```
if (...) {  
    // ...  
}  
else {  
    //...  
}
```

由基本逻辑组合而成的代码块，又被以函数(function)的形式组合成更大的程序片断。例如：

```
function foo1() {  
    // 代码行  
    // 代码行  
}  
  
function foo2() {  
    // 代码行  
    // 代码行  
}
```

最后，更多的函数、语句与代码块则构成了 JavaScript 文件^①。但是 JavaScript 对逻辑和模块结构的解释到此戛然而止。因为 JavaScript 中没有单元

^① 一般以.js 为扩展文件名。

和程序的概念：例如它没有 `unit/uses` 和 `program/project` 关键字。

JavaScript 将所有的代码块（哪怕是放置在 .js 文件中的函数或代码块）都视为有顺序关系的代码片断。JavaScript 在逻辑上的责任只是连续的执行它们，以至于对它们何时、以及如何装载到 JavaScript 引擎中都没有做出约定。

以在浏览器中的 JavaScript 为例。载入 JavaScript 脚本的方式是由浏览器中的 HTML 页面来决定的。下面的代码：

```
<!-- 代码块 1 -->
<script src="file1.js"></script>

<!-- 代码块 2 -->
<script src="file2.js"></script>

<script>
// 代码块 3
</script>
```

事实上表明代码块 1~3 是连续装载并执行。这与把 `file.js` 与 `file2.js` 中的代码块取出，并拼合到代码块 3 没有区别。而且这个“拼合”也只有次序关系，也没有引用关系。一个与此不同的例子是：C 语言中的 `include`（或 Pascal 中的 `uses`）就存在交叉引用的关系。

所以，上面这段代码可以被浏览器解释成：

```
<script>
load_and_exec('file1.js');
load_and_exec('file2.js');

// 代码块 3
</script>
```

不同脚本宿主对“单元（unit）”以上级别的模块的解释与处理都不相同。例如 `rhino` 就提供 `load()` 函数来讲 `shell` 能够装载 .js 文件，`Simple ECMAScript Engine` 则实现为 `SEE.load()`，而 `WScript` 中对此却没有现成的实现。

同样，脚本宿主对“程序（program）”的解释与处理也不相同。这表现在对“程序入口”和“入口参数”的处理上。

首先 JavaScript 没有约定任何“程序入口”。与此相对的，C 语言约定 `main()`

为程序入口，而 `pascal` 约定以主程序文件中以 “`end.`” 结束的代码块为程序入口。`JavaScript` 没有这类似的约定，脚本引擎对载入的每一个代码先进行语法分析，而后从第一条语句开始执行——即使这条语句看起来并不合理，或者引用了一个从未声明过的变量。例如：

```
var test = function() { };  
100  
yest();
```

这三行代码中，第一行是一个声明语句，第二行是一个单值表达式语句，第三行则是一行函数调用。其中，第二行可能是代码拼写时少输入了一个分号，也可能是开发人员有意使用其后的回车换行符来替代了分号；第三行则可能是真的调用一个 `yest()` 函数，也可能是试图调用前面声明的 `test()` 函数，但输入时拼写错误。`JavaScript` 没有程序入口，也没有对上面三行代码进行任何预期的分析的机制。因此它并不能识别第二行代码其实没有任何价值，也不能在语法检测时判断第三行代码是否是正确的函数调用。

接下来，`JavaScript` 也没有约定在装卸一个 `.js` 文件 / 模块时，如何处理入口参数。也许你会说：没有程序入口，当然不需要入口参数。这看起来合理，但实际使用时却并不是这样。

`WScript.exe/CScript.exe` 是一个常见的 `Windows` 下的脚本宿主，它是一套被称为 `Windows Script` 中的一个 `Shell`。它允许用如下的命令行装入一个脚本文件：

```
C:>WScript <filename> [option...] [arguments...]
```

这里的 `arguments` 就是在执行指定脚本时传入的入口参数。该参数能在脚本文件中访问 “`WScript.Arguments`” 对象来存取，例如：

```
/**  
 * filename: argu_ws.js  
 * -----  
 * load from dos prompt:  
 *   WScript argu_ws.js "hello world!!!" 100  
 */  
function alert(str) {  
    WScript.echo(str);  
}  
  
// 显示参数数量: 2  
alert(WScript.Arguments.Length)  
// 显示参数 0: hello world!!!
```

```
alert(WScript.Arguments(0));  
// 显示参数 1: 100  
alert(WScript.Arguments(1));
```

脚本引擎 **rhino** 也允许在启动 **shell** 时传入参数。但是读取参数时，则要求访问一个全局的 **arguments** 对象。例如：

```
/**  
 * filename: argu_rh.js  
 * -----  
 * load from dos prompt:  
 *   java -jar js.jar argu_rh.js "hello world!!!" 100  
 */  
function alert(str) {  
    print(str);  
}  
  
// 显示参数数量: 2  
alert(arguments.length)  
// 显示参数 0: hello world!!!  
alert(arguments[0]);  
// 显示参数 1: 100  
alert(arguments[1]);
```

由于脚本引擎没有约定 **function** 以上级别的代码块的处理方式，因此不同的宿主程序采用自己的方式来处理，因此“应用程序”或“单元模块”这些结构化元素在 **JavaScript 1.x** 中是需要自行实现的。

JavaScript 中保留了 **package** 关键字（但目前还没有启用它）。在本书的第二部分中，将提供一种方法来实现“单元模块”和“命名空间”这种级别上的结构化方法，也将讨论类似 **package** 方式的模块化机制。

1.2.2. 模块化的层次：语法作用域

上一小节讨论了 **JavaScript** 通过语句来构造代码的组织结构的问题，这种结构的基本形式是“代码分块”。而代码分块带来的语法效果，是信息隐藏。

一般说来，所谓信息隐藏指的是变量或成员的可见性问题。而这个可见性的区间，则依赖于语法的陈述。这被称为作用域，包括语法作用域和变量作用域两个部分，这两个部分是一个语言中模块化层次的全部体现。

在 JavaScript 中的语法作用域有四种，大部分被严格限制在“语句 / 批语句”作用域内：

序号	作用域	示例	说明	备注
1	表达式	参见：1.3.7 特殊作用的运算符		(***)
2	语句	var ...;	语法关键字表示特定行为动作。被省略部分是动作的目标，而不能是复合语句。	(****)
		if () ... else ...; for () ...; do ... while (); while () ...; with () ...;	被省略的部分既可以是单行语句，也可以是用 { } 标识的复合语句。	
	批语句	switch () {...}; try {...} catch() {...} finally {...};	虽然这些语法关键字本身仍然是语句，但它们用 { } 来限制一段代码块，作为它的语法作用域。	
3	函数	function () {...};	以函数所声明的代码块为作用域。	
4	全局	依赖于 HOST 的实现		

注*：注意表格中（尤其是在“批语句”作用域中）的语句结束符“;”的使用。它表明何处是一个语句，而其它的则是表达式或代码块。

注**：这些语句中，使用()表示的语法部分是“表达式”级别的作用域。它们只起到表达式示例的作用（或类似的副作用），请参考“[语句的副作用：表达式求值](#)”小节。

注***：JavaScript 中有少量的语法作用域被限制为“表达式”：一些作用于“其它表达式”而非运算值的运算符。它们被列举在“[”](#)的表格中。

注****：类似的还有 break、continue 和 throw，但它们是不应当例举在本表格中的。

1.2.2.1. 主要的语法作用域及其效果

表面上来看，“批语句”与“语句”作用域的不同，在于前者的语法中有一对大括号。你可能会说：一般的语句也可以有大括号呀。是的，例如 with 语句或 if 语句等等，它们看起来也是在限制一批语句，但事实上它仍是限制一个“复合语句”的。例如：

```
16 // 例 1：单一语句
17 label_1:
18     var value_1 = 100;
19     var value_2 = 1000;
20
```

```

21 // 例 2: 复合语句
22 label_2: {
23     var value_3 = 100;
24     var value_4 = 1000;
25 }

```

由于在例 1 中 `label_1` 只作用于单一语句，因此第 4 行代码事实上没有被任何标签限定。而相应的，`label_2` 则将代码行 7~10 声明为一个标签化语句。所以，无论标签语句限定的是单行语句还是复合语句，它的语法域都只是“语句”。

相比下，“批语句作用域”就略有差异。虽然它们也用一对大括号“{ }”来声明被影响的代码，但是一定不能将这个大括号理解为“复合语句的语法标识符”。简单地区别二者的方法，是尝试将“大括号表示的语句”替换为“单行语句”或“空语句”。例如下面的代码一和代码二可以相互置换，因此 `if` 语句中的大括号被用作表示“复合语句”：

```

// 代码一
if (true) i += 2;

// 代码二
if (true) {
    i += 2;
}

```

而下面的示例中，大括号是语法结构的一部分，替换后导致出错。因此 `try` 语句中的大括号是用来表示“批语句”这样一个作用域：

```

// 代码一：结构化异常处理的语法
try {
    i += 2;
}
catch (e) {
    // ...
}

// 代码二：导致语法分析错
try
    i += 2;      // <-- 错误 1: 试图替换为单行语句
catch (e) ;     // <-- 错误 2: 试图替换为空语句

```

除了上面例举的 `try...语句` 之外，同样也具有“批语句语法作用域”的是 `switch` 语句。很显然，我们可以写出：


```
switch ( true ) {  
}
```

这样的没有任何分支的代码，却不能省略大括号使它变成：

```
switch ( true ) ;
```

这会导致语法分析错误。但 `switch` 中存在的 `case` 语句使我们对该语句的语法产生了很多的误解。下面我们来解释 `switch` 语句的语法作用域效果。在此之前，先陈述两个观点：

- 👉 `case` 分支使 `switch` 内的代码形成了局部的分块，但这只是开发人员的主观行为，而不是语法效果。或者说，它是开发人员视觉上的“代码块”，而不是 JavaScript 语法概念上的某种作用域。
- 👉 `break` 语句的含义是跳出 `switch` 语句，而非跳出某个 `case` 分支；

首先，我们一般习惯性的将 `case` 分支的代码写成如下格式：

```
var b = true;  
switch ( b ) {  
  case true: {  
    // ...  
  }  
  case false: {  
    // ...  
  }  
}
```

这看起来使得 `switch()` 内部形成了“代码分块”的效果，所以我们会习惯性的将 `case` 后面的一个代码块理解为该分支影响的一个部分。这一点也颇符合 `pascal` 语言对“分支”的理解：将分支视作一个独立的代码块。

但是在 JavaScript 中，事实上 `case` 子句所影响的只是其后的一个语句。准确的说，它只是标识其后的一行代码的“起始位置”。对于整个 `switch` 语句来说，下面的代码是它的“语法作用域”所影响的范围：

```
switch ( b ) {  
  i := i + 1;  
  j := j + 1;  
  j := j + 1;  
  // ...  
}
```

而 `case` 语句只是在这个范围内做了一些标识，以提示一个“起始位置”，而没

有语法作用域所必须的“结束位置”的含义。因此构成了下面的语法：

```
switch ( b ) {  
  case true: i := i + 1;    // 标识该行的起始位置  
             j := j + 1;  
  case false: j := j + 1;   // 标识该行的起始位置  
             // ...  
}
```

因此我们说在 JavaScript 的语法概念上，case 分支后是没有“作用域”这样的东西的，有的只是一行代码。因此开发人员习惯将代码写成：

```
switch ( b ) {  
  case true: {  
    i := i + 1;  
    j := j + 1;  
  }  
  case false: {  
    j := j + 1;  
    // ...  
  }  
}
```

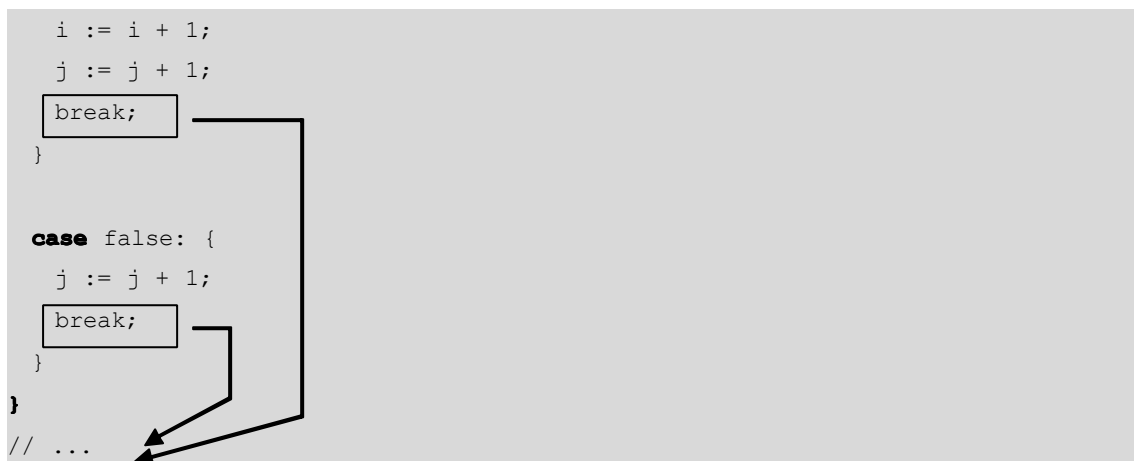
这种格式，只是开发人员的主观行为。同样的原因，写成如下：

```
switch ( b ) {  
  case true: {  
    i := i + 1;  
  }  
  j := j + 1;      // <-- 这里与上面的代码不在同一个“视觉效果”的‘块’内。  
  
  case false: {  
    j := j + 1;  
    // ...  
  }  
}
```

的格式，对于 JavaScript 来说，也与前一种写法没有任何语义的差别。

接下来我们讲述第二个观点，也就是 break 子句的效果。由于 case 子句没有“作用域”的含义，所指示的也仅是其后一个语句的起始位置，因此 break 子句事实上也并不对 case 子句起到语法效果。换言之，下面的代码：

```
switch ( b ) {  
  case true: {
```

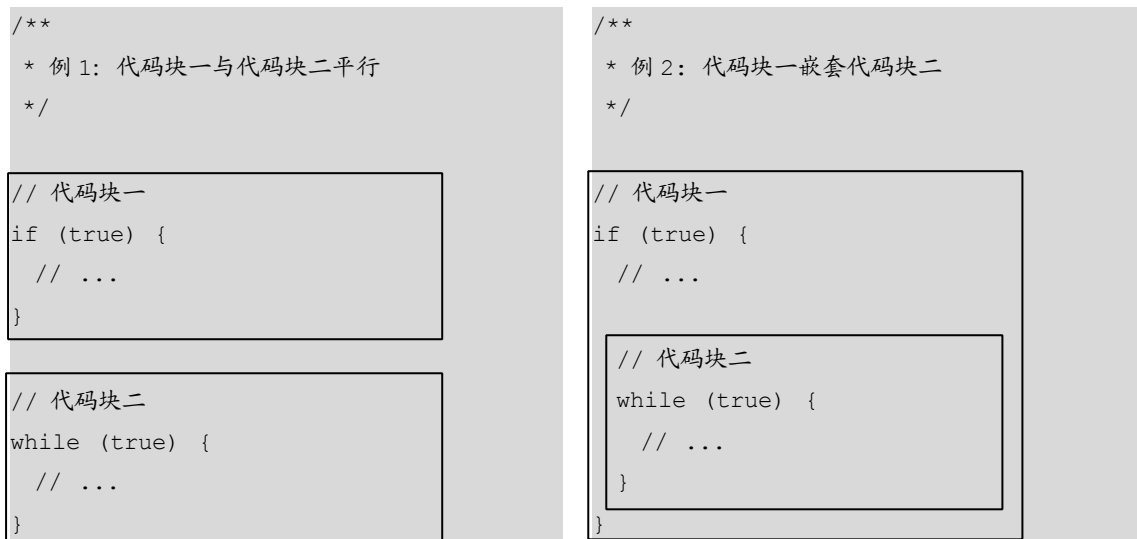


中有两个 `break`。而两个 `break` 子句的语义都是跳出 `switch` 所限制的“语法作用域”，而不是它们各自所在的 `case` 分支的“视觉效果上的‘代码块’”。

最后补充一点，在《JavaScript 权威指南》中说 `case` 分支是标签化语句的特例。这种说法有貌似合理之处：二者都对其后的一行语句起作用，并且标识该语句所在的位置。但是，在考虑 `break` 子句的效果的情况下，我们不能将 `case` 分支与标签等同起来，因为 `break` 可以作用于标签化语句，但却并不能作用于 `case` 分支。

1.2.2.2. 语法作用域之间的相关性

结构化语言中，代码块的作用域是互不相交的。在这些作用域（及其形成的代码块之间）只存在平行或嵌套两种相关性。例如：



结构化语言正是通过代码块这种的“互不相交”特性来保证逻辑上的独立，消除代码块之间的耦合。但是，也如同上例所示，在“嵌套”这种相关性中，代码块二与代码块一的语法作用域存在重叠。结构化语言必须描述是两个代码块之间的相互作用关系。这种关系，就是通过前面所说的“语法作用域的级别”来控制的。具体说就是：

- 👉 相同级别的语法作用域可以相互嵌套；
- 👉 高级别的语法作用域 **能够** 包含低级别的语法作用域；
- 👉 低级别的语法作用域 **不能** 包含高级别的语法作用域。由于不存在包含关系，因此语言实现时，一般处理所语法上的违例，或者理解为“平行”的关系。

第一个规则的应用是常见的。上例的例 2 中，由于 if 语句与 for 语句同是“语句”这个级别的语法作用域（等级 2），因此“if 语句包含了 for 语句的语法作用域”。除此之外，嵌套函数也是一个非常典型、常见的例子：

```
function foo1() {  
    // ...  
    function foo2() {  
        // ...  
        function foo3() {  
            // ...  
        }  
    }  
}
```

第二个规则在我们写函数时就已经很经常使用了：

```
function foo() {  
    // ...  
    if (true) {
```

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

但是，对于第三个规则，我们就需要更加详细的分析一下。在下面的例子中，例 3 与例 4 完全等效。因为语句无法“包含”比它等级更高的“函数”语法作用域：

```
/**  
 * 例 3: 该例在效果上与例 4 等效  
 */  
  
// 代码块一  
if (true) {  
    // ...  
  
    // 代码块二  
    function foo() {  
        // ...  
    }  
}
```

```
/**  
 * 例 4  
 */  
  
// 代码块一  
if (true) {  
    // ...  
}  
  
// 代码块二  
function foo() {  
    // ...  
}
```

1.2.3. 执行流程及其变更

“术语‘命令式’(imperative)来自于命令和动作，这种计算模型就是基于基础机器的一系列动作。”这句话很好地阐述冯诺依曼体系上的编程语言能得到运算效果的本质：顺序执行。

我们前面在对语言进行语法作用域的分析，其目的也正是要说明“代码分块（或模块化）”的最终目的还是顺序执行。假设我们能将复杂的.js 代码“微缩”一下，你就会发现，所有的代码行其实是这样写的：

```
1  var i = 100;  
2  i += 100;  
3  if () ...;  
4  function foo() { ... }  
5  foo();  
6  while () ...;  
7  {...};
```

的确，无论多么复杂的代码或代码块，其实最终都只不过是一行行的语句（如同上面这 7 行语句一样）。而解释引擎（或计算机系统）只需要去解释这

些语句，分解它的语法结构、表达式和变量，然后完成最终的运算即可。

但接下来命令式语言就出现了问题：无论如何对代码分块，程序执行总会存在“例外”。一旦我们需要分块，但又要在分块中处理“例外”，那么我们就需要一些语法，改变程序的“顺序执行”的流程。

最自然的想法当然是“GOTO”，但 GOTO 语句带来的灾难与它解决的问题一样的多。于是，更进一步的想法是：如果我们对上面的代码做足够的抽象（例如分析它们的语法作用域），并对每一个抽象设计一些类似 GOTO 功能的语句，那么我们必然得到足够的灵活性，而又避免了 GOTO 的滥用。

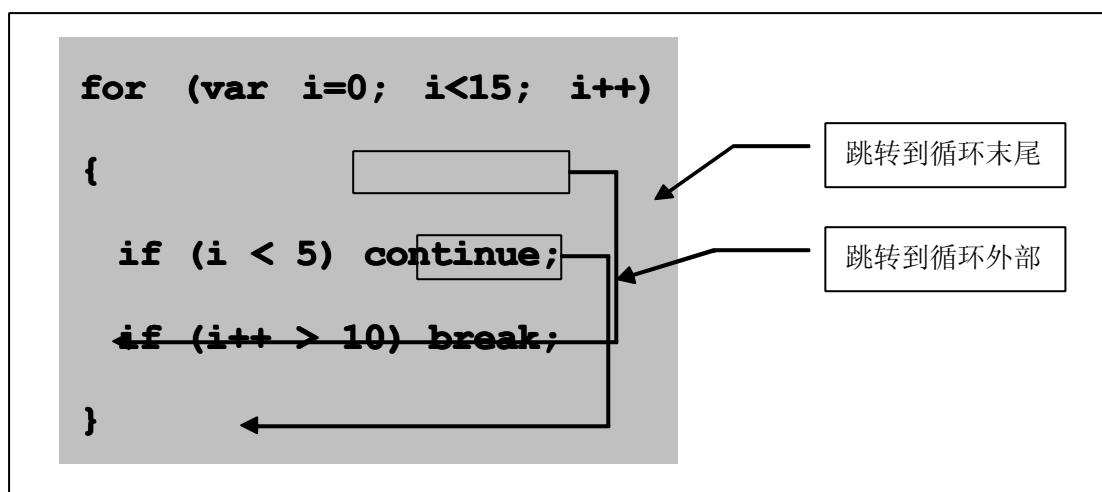
简单的说，就是“为每个语法作用域设计类似 GOTO 的语句”，以改变代码在该语法作用域中的流程。这些专用的 GOTO 语句——我们今后称之为流程变更语句——包括：

级别(*)	作用域	示例	说明
2	语句	continue [label];	对循环语句构成影响。
		break ;	
		break [label];	对标签化语句构成影响。
	批语句	break ;	对多重分支构成影响。
3	函数	return [...];	对函数构成影响。
4	全局（或文件单元）	throw ...;	对全局代码构成影响。

注*：这里使用了与前表对应的序号，但具有级别含义。

1.2.3.1. 级别 2：“break <label>”等语法

在循环中，我们可以用 continue 和 break 来改变循环流程。这种效果如下图所示：



`continue` 与 `break` 对 `for` 语句（或其它循环语句）产生效果，而这些语句的作用域是“语句语法作用域”，所以 `continue` 与 `break` 在上表中被列在第二级，作用于“语句”这个作用域。

`break` 作用于标签化语句时，也是处于第二级的。至于标签化语句为什么是“语句语法作用域”的问题，我们已经在上一小节讨论过了。

在上表中，我们说明 `for` 循环和多重分支语句中的 `break` 中，在示例中都只写“`break;`”，而没有写“`break [label]`”。这是为什么呢？

接下来，我们就详细讨论 `break` 子句的语法效果：在本质上来说，“`break [label]`”是针对标签语句的，而并不是针对循环语句的一个语法。下面我们籍由几个示例的演进关系来说明这一点：

```
1 // 示例 1
2 var i = 0;
3 my_label : {
4     i++;
5     break my_label;
6     i = 0;
7 }
8 alert( i );
```

在示例 1 中，当行 5 的 `break` 执行时，执行流程跳出 `my_label` 标签所指示的作用域。接下来，我们看下面的代码：

```
1 // 示例 2
2 var i = 0;
3 my_label : {
```

```

4      i++;
5      while (true) {
6          break my_label;
7      }
8      i = 0;
9  }
10 alert( i );

```

在这个示例 2 中，我们看到，**break** 的效果仍然是作用于 **my_label** 的语法作用域的，而并不是作用于 **while** 的作用域。所以，尽管你可以在循环体内使用“**break [label]**”子句，但该子句在语法上应是“标签化语句”的子句，而不是循环语句的子句。更确切的说，循环语句的 **break** 子句只有一种语法：

```
break;
```

同样，标签化语句的 **break** 子句也只有一种语法：

```
break <label>; // 在这里标签名是不可省略的
```

后者仅能用于标签化语句。而在示例 2 中，它能从 **while** 循环的“作用域”中跳出的原因，应当解释为“流程变更语句可以跨越相同等级作用域”。

我们也应该留意到 **switch** 中的 **break** 子句。我们说过，**switch** 中的 **break** 的作用是结束整个 **switch** 而非某个 **case** 分支——我们甚至强调，**case** 分支并没有语法上的“作用域”的意义。那么，**break** 子句在 **switch** 的语法效果是否也和上面一致呢？

是的，我们把 **switch** 套在上面的例子中来测试一下：

```

11 // 示例 3
12 var i = 0;
13 my_label : {
14     i++;
15     switch (true) {
16         case false:
17             break;
18         case true:
19             break my_label;
20     }
21     i = 0;
22 }
23 alert( i );

```


在这个例子中，第一个 `break` 作用于 `switch`，因此跳出到第 11 行；而第二个 `break` 作用于标签化语句，因此跳出到第 13 行。这里的结论与前面的一致：

- 👉 “`break;`” 是循环和多重分支语句的 `break` 子句的语法；
- 👉 “`break <label>`” 是标签化语句的 `break` 子句的语法；
- 👉 示例 3 中第 9 行的 `break` 语句穿过了两个语法结构（的作用域）而跳转到第 13 行，因此它具有“跨越相同等级语法作用域”的特性。

既然循环和多重分支都不需要 “`break <label>`” 这种语法，那么这个语法为什么还存在呢？其实，“`break`” 本身是“中断一个语法作用域”的概念是没错的，这个语义用在循环和分支中也没有问题。只是，重要的是，我们需要一个在“跨越语句级别的作用域”的语法结构。更明确的说，因为我们需要一个“语句级的流程变更语句”，所以 “`break <label>`” 的语法被保留下来了。

无论你是多么不愿意承认，“`break <label>`” 这个语法并不属于循环和多重分支，它不过是被“变着法子保留下来的”一个 `GOTO` 语句。这一点在 `pascal/delphi` 中可以得到印证：`pascal/delphi` 明确声明保留了 `goto` 语句，并且它只用于一个明确声明的标签语句，也只有一种语法格式。例如：

```
(**
 * pascal/delphi 语法中的 goto
 *)
procedure test();
label my_label;
begin
    //...
    if ( you_want_jump_jump ) then
        goto my_label;
    //...
    //...

my_label:
    //...
end;
```

而在 `pascal/delphi` 中，循环语句中的 `break` 就没有带标签的语法——如果你需要那样的功能，那么你能做的，就是定义一个标签，然后 “`goto <label>`”。

所以说，“`break <label>`” 语法其实就是 “`goto <label>`”。而且它的作用，也如同 `GOTO` 语句一样：跨越语句和批语句（这一级别）的作用域。

在处理多重循环时，`continue` 子句还存在一种 “`continue [label]`” 的语法。但需要留意，我们在 “1.4.4.2 `break` 子句” 小节就强调过这种 `label` 必须指定在一个循环语句的开始，即使将该循环语句套在一个（只有一个语句的）复合语句中，也是不合法的。所以可以说：“`continue [label]`” 是解决多重循环时的便利手段。事实上，以语法严格著称的 `pascal` 就没有这样的 “便利手段”，而只能添加一个标志变量或者使用 `GOTO` 语句来达到类似的效果。

```
<script>
i = 0;
lbl : {
  i ++;
  try {
    switch (i) {
    case 1:
      i = 0;
      i ++;
      break lbl;
      while (true) {
        break lbl;
      }
      i ++;
    }
  }
  finally {
    i++;
  }
  i++;
}
alert( i );
</script>
```

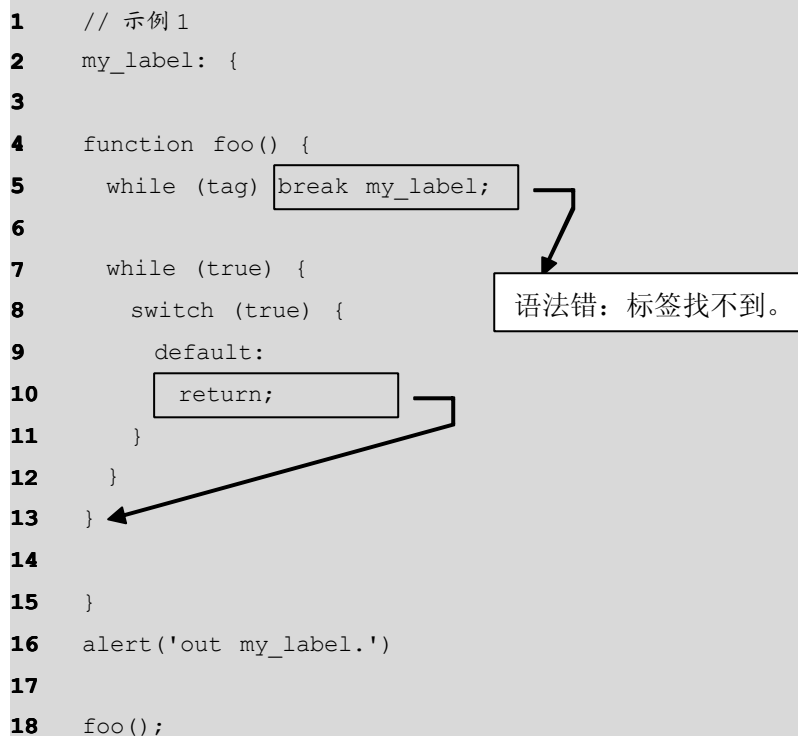
1.2.3.2. 级别 3: `return` 子句

下面我们来讨论另一个语法效果：

- 👉 `break <label>` 不能跨越函数的语法作用域；
- 👉 `return` 子句可以跨越任何语句的作用域而退出函数。

以下面的代码为例：

```
1 // 示例 1
2 my_label: {
3
4 function foo() {
5   while (tag) break my_label;
6
7   while (true) {
8     switch (true) {
9       default:
10        return;
11      }
12    }
13  }
14
15 }
16 alert('out my_label.')
17
18 foo();
```

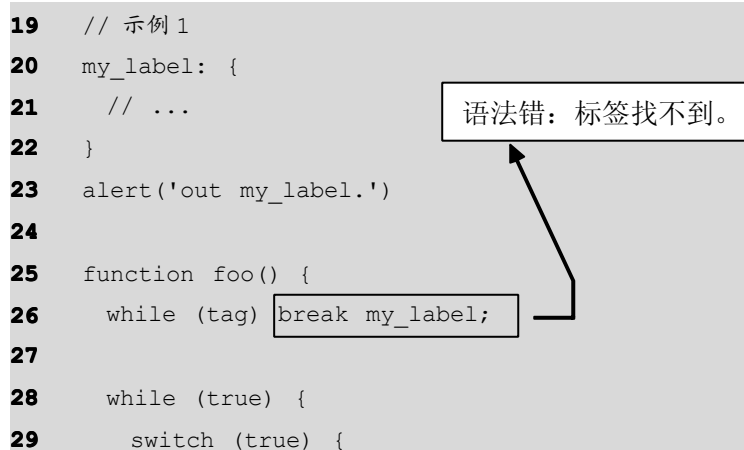


在这个例子中，第 5 行试图跳转到第 15 行之后，但语法上这不成立，提示为：“标签找不到”。这意味着标签 `my_label` 不在函数 `foo()` 可见的语法作用域内。

但在上面的代码中，从形式上来看，函数 `foo()` 的确是在 `my_label` 的语法作用域之内的。那么，为什么 `break` 不能作用于 `my_label` 呢？

使用“1.2.2.2 语法作用域之间的相关性”所说的规则，这个关于 `my_label` 标签的例子其实相当于：

```
19 // 示例 1
20 my_label: {
21   // ...
22 }
23 alert('out my_label.')
24
25 function foo() {
26   while (tag) break my_label;
27
28   while (true) {
29     switch (true) {
```



```

30     default:
31         return;
32     }
33 }
34 }
35 foo();

```

——这就是 `foo()` 函数中找不到 `my_label` 标签的根源了：他们事实上处在不同的语法作用域中（也就是说他们事实上是相并行语法作用域的）。

通过对作用域的等级和相互关系的分析，我们陈述下面的观点：

- 👉 流程变更语句可以“穿过”相同级别的任何语法作用域；
- 👉 高级别的流程变更语句，可以“穿过”嵌套其内的任何低级别的语法作用域。反之则不成立。

第一个观点用于解释在上一小节中“`break <label>`”语法其实就是“`goto <label>`”，而且“`break <label>`”可以在它的语法作用域内，穿过循环、分支、异常捕获（批语句）和复合语句等“语句级别”的作用域。

第二个观点用于解释在任何语句中，可以用 `return` 退出函数；反过来，也可以解释上例中，`break` 子句找不到标签的原因。

1.2.3.3. 级别4：throw 语句

首先应该注意到“`try...catch..finally`”语法结构是的语法作用域是“语句”，而 `throw` 则是“全局”。接下来，我们也要注意，我们称呼 `throw` 时，是用的“语句”，而不是“子句”。

`throw` 语句并不依赖“`try...catch..finally`”而存在：你随时可以 `throw` 一个异常。因此，它并不是一个子句。尽管 `throw` 的效果与“`try...catch..finally`”有一些关系：`throw` 抛出的异常可以由“`try...catch...`”捕获。但是“捕获”`try` 语句的行为，而 `throw` 所做的只是向全局范围内抛出一个异常，至于该异常被（哪个作用域范围内的）语法结构捕获和处理，并不是 `throw` 的责任。所以，从这个角度来看，`throw` 也是一个流程变量的语句，其作用域是脚本引擎全局。

前面推论出来的有关作用域和流程变更的观点，也完全适用于 `throw` 语句。例如 `throw` 语句可以“穿过”函数的语句作用域：

```
function test() {
```

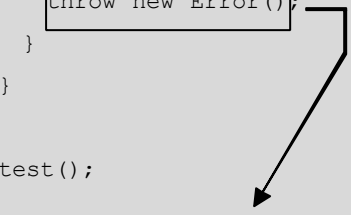
```

while (true) do {
    throw new Error();
}
}

test();

// 系统全局作用域或包括 test() 函数
// 调用的语句块。

```



在调用 `test()` 函数之后，我们看到 `throw` 语句起到了 `return` 语句的效果（尽管这个效果并不漂亮），所以它退出了循环语句的和函数的作用域，并使流程回到全局作用域中——直到它遇到一个异常捕获语句，或者交由全局的或宿主主的 `onerror` 事件处理程序，或者最终交给系统的异常处理程序，以显示一个错误提示框。

1.2.3.4. 执行流程变更的内涵

我们几乎是凭空地推论出来了一个“等级”的概念和“跨越相同等级”的语义。但我们随后用这个理论来解释了一些语法现象，在这个过程中，我们看到：

- 👉 语法作用域是互不相交的。正是作用域互不相交的特性构造了代码结构化的层次，并消除了一些错误隐患。
- 👉 语法作用域间可以存在平行或包含关系。高级别可以嵌套低级别的语法作用域，反之则不成立。
- 👉 高级别的流程变更子句（或语句）可以跨越低级别的作用域，反之则不行。

我们论述过“`break <label>`”实质上是“`goto <label>`”——我们的目的是要把这个语法从循环与分支的作用域问题中分离出来。但事实上 JavaScript 与 Java 一样，对“`break <label>`”是有使用限制的。在 pascal 语法中，`goto` 语句可以转向任何一个语句的起点位置（该位置只有标识起点的作用，而没有标识终点的作用）。但在 JavaScript/Java 中，标签化语句是有“语句”级的语法作用域的，而“`break <label>`”的效果是仅影响该作用域，因此它并不是那种“无所不能的 GOTO”。“`break <label>`”对循环与分支的影响，可以解释为同等级语法作用域下的 GOTO 效果。

我们前面也论述过语法作用域不相交的特性。因此我们不可能创造出标签语句与其它语句“交叉”的代码结构来。在这样的前提下，“`break <label>`”与

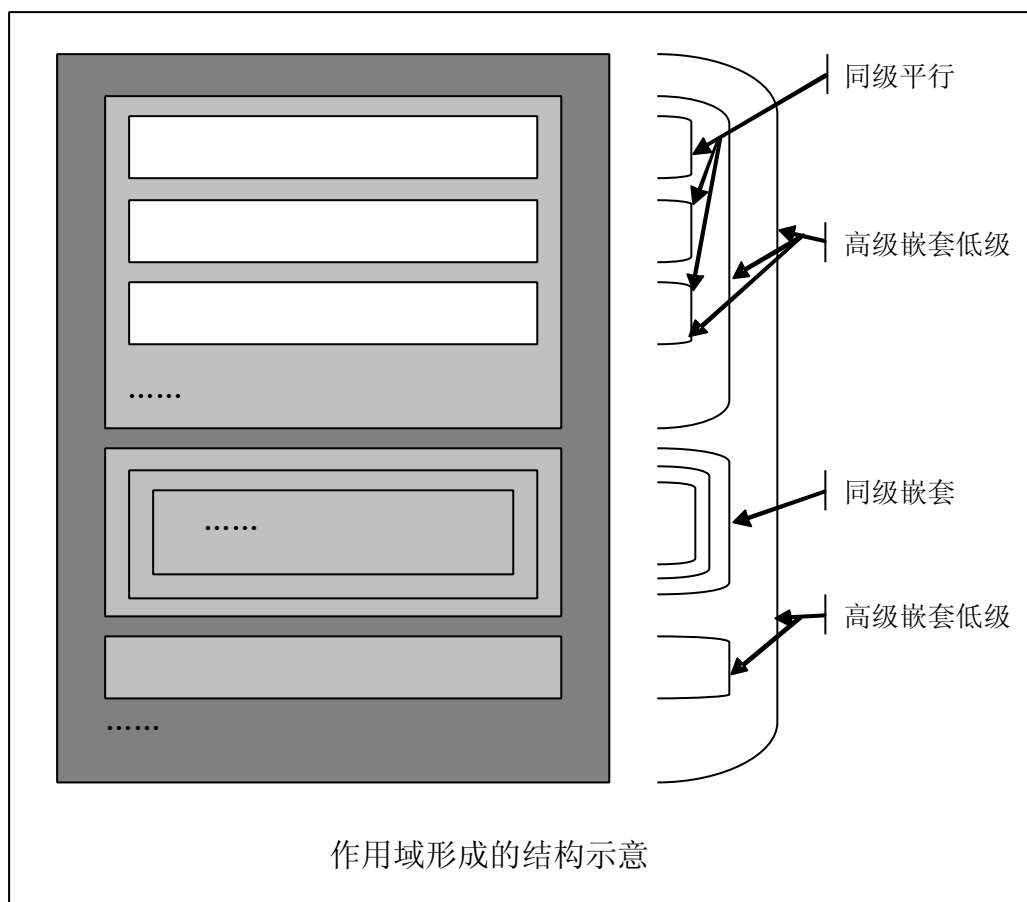
“标签具有作用域”共同产生的效果就成了：“**break <label>**”只能跳转出一个作用域，而不能跳入一个作用域。这成功地避免了 pascal 中类似如下的语法：

```
(**
 * 示例：pascal 语法中 goto 跳入循环结构
 *)
label my_label;
//...

while (i < 100) do
begin
    goto my_label;
end;

while (i>100) do
begin
    // ...
my_label:
    writeln('hello');
end;
```

从与语言和语法无关的角度来看待上述的事实，我们发现，语法作用域在结构化中的本质是将代码表现为下图所示的形式：



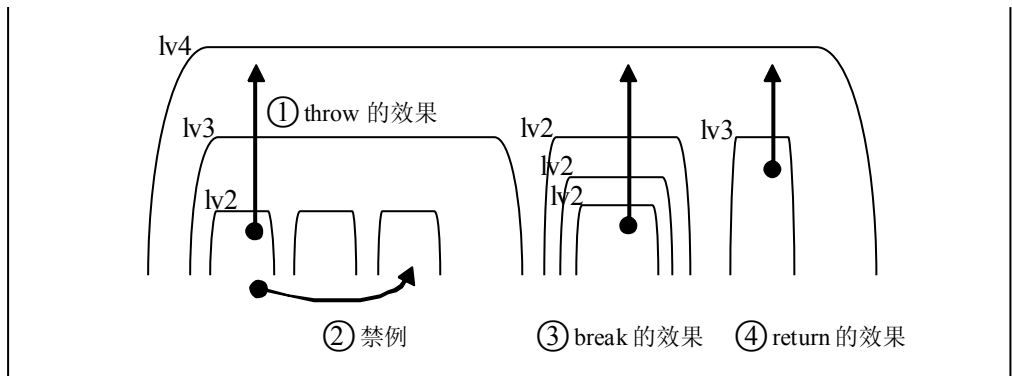
这种形式使得代码清晰，并且能表达结构化分析阶段对系统自顶向下逐层精化时所展现的逻辑组织。但应当注意到这种结构也使得执行流程变得僵化，缺乏一些灵活性。

Bohm 和 Jacopini 在理论上早已证明过类似这样的灵活性是不必须的，而 E.W.Dijkstra 则更进一步的指出灵活性对系统带来的危害。然而在既存的语言中（即使已经声称“消灭了 GOTO 语句”的 Java/JavaScript），依然保留有造成“流程变更”这种事实的语句或语法。

通过前面的分析，我们看到在这些新的语言实现中，程序执行的流程变更，本质上已经转义为作用域（及其等级）的变更。而这正是这些语言在保障“结构化编程”的清晰风格的情况下，能够具有充分（且安全）流程控制的灵活性的根源。这里要进一步加强的是：高级别的流程控制语句，对低级别的语句的作用域会产生“突破”——这正是流程控制的关键，也是结构化编程严谨而不失灵活性的关键。

其中的要诀，在于让流程变更子句（或语句）的设计要覆盖不同级别的作用域，以获得最大的灵活性。但并不必覆盖所有的语句或语法结构——那将导

致浪费和纵容。例如针对上面的作用域示意，流程控制的设计仅仅是：



注：设计 2 在 java/javascript 中是禁例，但在 pascal 中的 goto 却可以产生这样的语法效果。

接下来，还有三个问题：

- 1. "语句作用域"中，为什么 var 和 if 条件语句没有“专用的 GOTO”？
- 2. 为什么 try...catch... 语句属于代码块作用域，而与之相关的 throw 却属于全局作用域的“专用的 GOTO”？
- 3. return 如何越过同级的 funtion 语法作用域。

1.2.4. 模块化的效果：变量作用域

变量作用域又叫变量的可见性。一般的讲述程序设计语言的书籍中，都是不讨论语句（和其它语法结构）的作用域，而直接讨论变量的可见性的。这样会使我们少了一个观察程序的视角。在我看来，语法作用域讨论的代码的组织结构上的抽象，讨论的是“圈地”的问题；而变量的作用域完成对信息的隐蔽，也就是处理“割据”问题。前者是形式上的规范，所者是实际的占有^①。

又例如我们在“1.5.6 缺省对象的指定”章中给出的示例：

```
16 // (续“1.5.6 缺省对象的指定”示例)
17 with (obj) {           // <-- 依赖语法作用域的限定
18     function foo2() {
```

^① 从实现的方式来看，一些书籍中称纯粹的语法作用域实现为“静态作用域”，而与代码执行期效果相关的变量作用域（意即变量作用域与静态作用域不一致性时）则称为“动态作用域”。本书重在讨论实现的意义而非方法，所以使用在表现形式（而非实现方式）上的“语法作用域”、“变量作用域”来命名它们。


```

19     function foo() {
20         value *= 2;    // <-- 依赖变量作用域的存取
21     }
22     foo();
23 }
24 foo2();
25 }

```

with 语句不能限定到 23 行对 value 值的存取，其真实的原因在于 with 语句的作用域是“语法作用域”，限定对象为“语句”；而行 23 是一个代码块执行过程中对变量的存取，它受限于“变量作用域”^①。

二者的区别，在于前者是语法分析阶段对代码块组织结构的理解，后者是代码执行阶段对变量存储的理解。

圈地与割据并不是相等的，例如我们可以将一个区域划分为九块，却只有六块有人占领。事实上程序设计语言中就存在这种情况。以 JavaScript 为例，我们前面划分出了它的四个语法作用域：表达式、语句、函数和全局。但在变量作用域上，它并没有“语句”这个级别。

1.2.4.1. 级别 1：表达式

直接量是一种表达式级别的量（数据），但是我们通常会把它归类到“常量”这个范围中去。由于有了“常量”的概念，因此我们无法讨论某个“直接量”的可见性是在这里，或是在哪里：因为哪里都可以使用常量。

因此在“表达式级别上的变量作用域”的话题上，我们放弃讨论以“常量”形式存在的一些直接量。例如下面这段代码中，我们不讨论字符串“ABCDEF”和布尔值 true 的“变量作用域的问题”：

```

// 由于不讨论下面的两个 true 与两个"ABCDEF"的变量作用域问题，
// 因此它们是否是同一个地址上的值，并不重要。又或者它们何时被销毁，也无关紧要。
if (true) {
    i = 'ABCDEF';
}
else {
    i = 'ABCDEF' + true;
}

```

^① 这里的代码除了涉及两种作用域之外，还涉及到“（标识符的）可见性”问题。在 JavaScript 中，标识符（在上下文中）的可见性是与函数式语言的闭包相关的概念，因此将被放在章节“1.6 闭包”中讲述。

但哪种情况下某个量会存在“有价值的”表达式级别的变量作用域呢？这通常是匿名函数和对象直接量了。例如下面的代码：

```
// 下面的代码中，后面的一个匿名函数被声明出来立即就被执行了。而在该行代码之外（的任
// 何地方），你不可能引用到它。缘于引擎对内存回收的实现的差异，我们也无法保证在其它
// 任何代码"试图"用到它时，该函数是否还存在。
foo = (100 + function() {
    return 100;
})();
```

接下来你可能会说这个地方的函数是直接量，也可以被理解为常量。是的，在这个例子中的确会有这样的疑问。但下面的两个例子呢：

```
// 例 1：创建后立即使用的匿名的函数对象
foo = (100 + new Function('return 100;') ());

// 例 2：创建后立即使用的未命名对象
foo = 'Hello, ' + new Date();
```

在这两个例子中，新对象在被创建出来之后就被立即使用。在该语句中（使用到该对象的某一个）表达式执行完之后，你就再也访问不到它了。

值得一提的是，在 JavaScript 中表达式级别的变量都是匿名的^①。这通常各种直接量（包括匿名函数），此外也包括一些函数中返回的值或引用。

1.2.4.2. 级别 2：语句

我们说某个变量存在语句级别的作用域，是指它（我这里是指包括对象和直接量等的一个可运算对象）被创建出来之后，在脱离了创生它的一个（单个或连续的）表达式之后，仍然可以在（且仅在）所在语句的作用域中被访问。

例如在一些语言中，我们会看到类似于这样的语法约束：

```
// 语法规则说明：下面的变量是循环中的变量，在循环结束后不能访问
for (var i=0; i<10; i++) {
    // ...
}

// 基于上述的规则，下面的代码显示变量 i 不存在
```

^① 在 SpiderMonkey JavaScript 1.7 中出现了一个 let 关键字，它可以声明一批变量并使这些变量仅作用于其后的一个表达式。这可能是唯一能命名变量，且又能使变量作用于“表达式级别的变量作用域”的语法。需要强调的是，let 关键字可以引导表达式、语句以及语句中的语法元素，这些被统称为“let block scope”。

```
alert( i );
```

这个例子在 JavaScript 中会显示结果值 11。而在 C#或 Java 中，上面测试代码会在编译期就通不过，在最后一行提示变量 i 未声明。因此我们称在 C#或 Java 语言存在一种“语句”级别的变量作用域。当然，因为语言的差异，示例代码应该写成：

```
/**
 * 上述示例代码的 C 版本
 */
for (int i=0; i<10; i++) {
    // ...
}

// (对于 C#、Java 以及某些 C/C++编译器来说,) 下一行代码不能访问到变量 i
printf( i );
```

此外，在我们前面说过复合语句的语法作用域也是语句级别的。在 C++ 语言中，就存在一种“块锁（或局部锁）”，用来写类似多线程并行的代码。例如：

```
{
    CLocal_Lock Lock(&m_cs);
    // 相当于如下的 JavaScript 代码: Lock = new CLocal_Lock(&m_cs);
    // 其中 CLocal_Lock 是一个用户实现的类.

    // ...
}

// 上面创建的锁将不能在代码块之外访问
```

这是在复合语句所表示的“语句语法作用域”内的一个变量。在 C++中，被称为自动变量。但这个例子中，“锁”的效果（线程同步）并不是我们所说的语法效果：是类所设计的一个功能，而不是语法中的通用模式。

因此我们在本书中讨论的 JavaScript 并没有“语句”级别的变量作用域^①——它没有提供类似上述示例的语法效果。

^① 在上一小节中讲到的在 SpiderMonkey JavaScript 1.7 中出现的 let 关键字，也可以声明一个命名变量，并使之作用于“语句级别的变量作用域”。这种语法只能应用在 for 和 for..in 循环中，并且除了作用域的不同之外，与 var 的效果是一致的。

1.2.4.3. 级别 3：函数（局部）

我们多数时候到的是函数级别的变量作用域。具有这种作用域的变量也称为局部变量，以跟后面的“全局变量作用域”相对。

下面声明的三个变量(i1,i2,i3)的作用域在函数 foo()之内。离开这个函数，这些变量都不可见了：

```
function foo() {  
    var i1 = 10;  
    for (var i2=0; i2<100; i2++) {  
        // ...  
    }  
  
    var i3 = 100;  
}
```

在这个示例中我们有意地使用了一个 for 循环。如同我们在上一小节所说的，由于 JavaScript 不存在语句级别的变量作用域，因此 for 循环中 i2 的变量作用域也就“逸出”到函数中，变成（函数的）局部变量了。当然，如果没有函数这个作用域在限制，那么它会变成一个全局变量——我的意思是，如果你在一个全局的代码块中写上面这个循环，那么 i2 就会变成全局变量。没有语句级别的变量作用域，其实也是 JavaScript 的全局命名污染的诸多灾难之源中的一个。

函数可以被嵌套，又存在“函数”级别的变量作用域。因此我们事实上可以看到：在多重嵌套的函数中，内一级的变量不会“逸出”到外一级。如下例所示：

```
/**  
 * 嵌套函数中的变量作用域的效果  
 */  
function foo() {  
    var i1 = 10;  
    // 代码段 1  
  
    function foo2() {  
        var i2 = 100;  
        // 代码段 2
```

```
function foo3() {  
    var i3 = 1000;  
    // 代码段 3  
}  
}  
}
```

在这里，代码段 1 不能访问到代码段 2 和 3 中的变量 `i2` 和 `i3`，是因为他们处在各自的（函数级的）变量作用域中。相对于 `foo()` 来说，`foo2()` 和 `foo3()` 都是局部，而局部不能访问全局的信息，是信息分层和信息屏蔽的重要原则。

但同样的嵌套结构，在循环(或其它语句)中却可以访问内层的变量。我们前面也解释过：在语句这个级别的“语法作用域”上，没有相应的“变量作用域”。从这里我们也得到了两点事实：

- 👉 语法作用域不等于变量作用域

- 👉 变量的可见性受限于它所在的语法结构的（语法）作用域

对于第二条有一点补充：如果语言没有实现相应的变量作用域，那么该变量的可见性会逸出到同级的其它结构中去。

1.2.4.4. 级别 4：全局

在大多数人来看，JavaScript 中的变量作用域只有两个：局部变量和全局变量。这两个作用域的划分也很简单：在函数内，或是在函数外。

这的确是个简单的法子。

我们前面所说，JavaScript 的语法作用域有四级，而函数语法作用域是第三级，可以嵌套自身的和低级的语法作用域。因此，在这些低级的语法作用域被“函数”掩蔽之后，的确只剩下函数的“内”、“外”之分。函数内的叫局部，函数外的叫全局，因此也是说得通的。

在浏览器开发中，JavaScript 的全局作用域使它拥有充分的便捷特性。但也正是全局作用域使它成为大型开发中的灾难。当我们需要在浏览器或 JavaScript 环境中开发一些大型项目时，“随时随地”都可以从函数内泄露到全局的命名，以及随在函数内对全局变量可能发生的修改，总在威胁着项目的品质。

从大型项目的角度来看，项目整体的品质比代码局部的灵活性更为重要。

因此 JavaScript 以下两项特性是不利于大型项目的开发的：

- 👉 在全局范围内任意声明变量，尤其是在全局范围内使用 `for` 循环的同时声明变量；
- 👉 在函数内部不使用 `var` 声明而直接对一个变量名赋值时，该变量会隐式的在全局声明，或读写到一个全局的变量。

全部变量的作用域可以被解释为 JavaScript 中的 Global 对象的成员，也可以被解释成宿主对象(例如浏览器中的 `window`)的成员，也可以被解释成单独处理的全局数据。这取决于不同的宿主程序或脚本程序的处理。在 Internet Explorer 浏览器中的 JavaScript 引擎对这一点的处理是存在弊病的。例如下面的代码：

```
<!-- 以下代码在 IE5,6,7 上测试 -->
<script>
value = 100;

// 全局变量在 window 的成员列表中
alert('value' in window, '<BR>');

// 显示 true, 表示成功清除该全局变量。但事实上它仍在 window 的成员列表中。
alert(delete value);

// 由于条件为真，所以下面的代码试图存取 value，但这将导致异常
if ('value' in window) {
    document.writeln('bug test: ', value);
}
</script>
```

这个问题出在浏览器将全局变量处理为宿主对象(`window`)的成员，而在 `delete` 运算时又没有同时处理它。在 `firefox` 浏览器中，就没有这个错误，我相信它的开发者们维护好了 `window` 这个对象。

但接下来仍然有问题。因为 JavaScript 规定 `delete` 不能够删除使用 `var` 显式声明的变量。因此将上述代码中的：

```
value = 100;
```

改成：

```
var value = 100;
```

则 `delete` 运算就会返回 `false` 了。而 `value` 也无法从 `window` 的成员列表中被清除。

因此尽管在全局变量作用域用使用 `var` 声明的变量，以及函数作用域中不使用 `var` 而产生的变量，二者都是全局变量。但它们的处理规则仍然是不尽相同的。

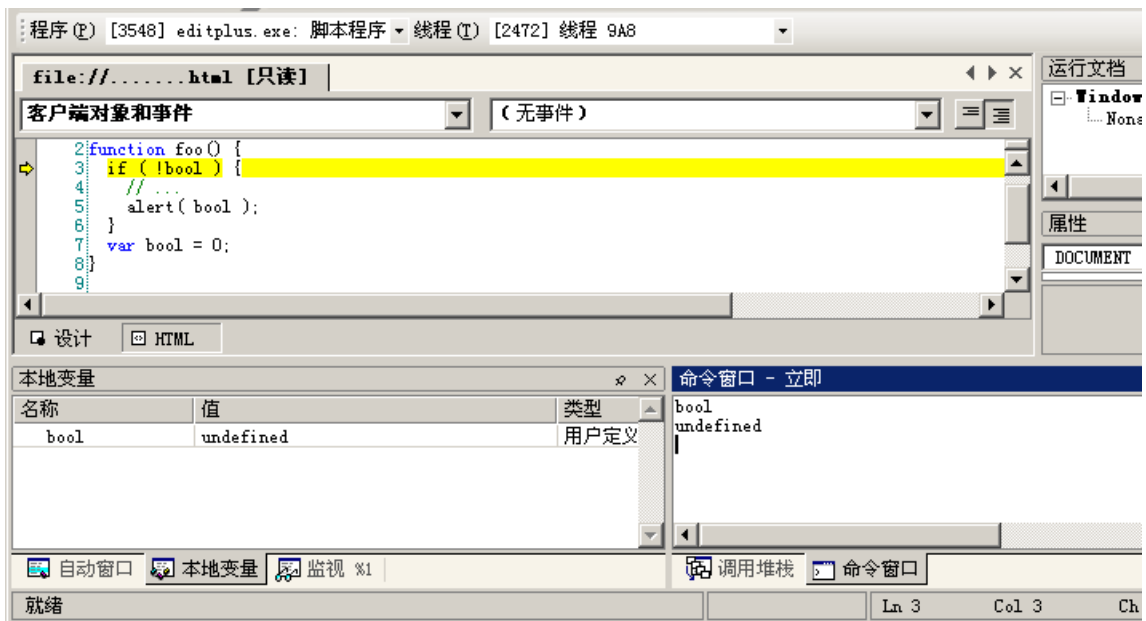
1.2.4.5. 变量作用域中的次序问题

一些语言认为，即使是在同一个语法作用域中，变量也只能在隐式或显式地声明之后才能被访问（才具有可见性）。C 语言就具有这种特性，例如：

```
1  function foo() {  
2      if ( !bool ) {  
3          // ...  
4          alert( bool );  
5      }  
6      var bool = true;  
7  }
```

在 C 语言中，这个函数是不能通过语法检测的。但在 JavaScript 中，这个函数可以通过语法检测，而且也存在执行意义。这是因为在 JavaScript 中，语法解释与执行分成两个阶段，而变量声明是在语法解释阶段处理的，因此在 `foo()` 函数执行之前，这个变量就已经在它的语法作用域在存在了。

如果在上述代码的第 2 行中设置一个断点，当你的代码停在该断点时，你会调试器中看到如下图所示的信息：



这时，在本地变量查看窗口中，变量 `bool` 是存在的，而且有一个值为 `undefined`。我们在命令窗口中访问一下变量 `bool`，也能得到这个值。由于 `!bool` 值为 `true`，所以 `foo()` 函数进入 `if` 分支，并显示值 `'undefined'`。

所以在 JavaScript 中，显式声明的变量在函数和全局作用域中，是没有次序限制的。你可以先声明再使用，也可以先（在某些语句或表达式中）使用它，最后再显式地声明。因此显式声明总是早在代码执行之前，就被引擎理解了的。

上面的这个例子可能不够说明问题，因为通过 `var` 语句来声明的变量，在语法解释期中的值总是 `undefined`。但在函数的显式声明中，就不是这样了。例如：

```
TMyClass = Class(TObject, _ConstructorName(foo));

function foo() {
    // ...
    this.Class = TMyClass;
}
```

在这个例子中，由于 `foo` 是一个函数声明的标识符，所以在执行期的任意位置都可以直接使用。在顺序上，使用 `foo` 这个标识可以早于该标识符的声明，这使得 JavaScript 不必使用专门的语法来处理前置声明（例如 pascal 语法中的 `forward`）。同样的原因，在 JavaScript 中，对象可以轻松地持有它自身，或者它的类——只要它们在可视的局部或全局范围内，被显式地声明过。

1.2.4.6. 变量作用域与变量的生存周期

也许你会问，我们为什么不直接说变量的作用域就是变量的生存周期呢？其实，我们把变量的作用域说成与它的可见性一致是合理的，因为二者都是同一个角度来看同一个问题。然而，生存周期却是从另一个角度——实现——来看待作用域的问题的。

变量的生存周期是指它何时被创建和何时被释放。在 JavaScript 中，一个变量的创建是在：

- 👉 在引擎做语法分析，发现 `var` 声明时；或
- 👉 在引擎做代码执行，发现试图写(例如赋值)一个未被创建的变量时。

而它的释放，则是在：

- 👉 引擎执行到函数结束 / 退出操作时，将清除函数内的未被引用的变量；
- 👉 引擎执行到全局的代码块终结或引擎卸载和重载入时，将清除全局的变量和数据的引用。

所以 JavaScript 中的变量生存周期只有两个：函数内的局部执行期间，和函数外引擎的全局执行期间。这是由 JavaScript 的引擎在实现“函数”这个机制的时候所采用的方法所决定的，是实际实现中的一种选择，而不是语法规义上的约定。

变量作用域讨论的是“在形式上这个变量能在哪个范围内存取到”，变量的生存周期讨论的是“在实现中什么时候创建和释放一个变量”。正是由于二者并不完全重叠，才会使 JavaScript 出现上一小节所示例的问题，导致应用中出现下面这样的代码：

```
1  function foo() {  
2      if ( !bool ) {  
3          // ...  
4          alert( bool );  
5      }  
6      var bool = true;  
7  }
```

这个例子中 `bool` 变量是在函数还未开始执行时已经被引擎创建好了，因此它的变量生存周期早于它在代码中被声明的位置“第 6 行”。然而，如果你将第 6 行中的“`var`”声明去掉，则它的生存周期便是从第 6 行开始的了。从根本

上来说，在这个变化过程中它的变量作域从来没有改变过。

1.2.5. 语句的副作用

既然我们认为语句的基本作用在于陈述逻辑和组织结构（即代码分块），那下面的一些特性，就应该被视为它的副作用：

- 👉 语句的值
- 👉 变量声明，包括函数直接量声明
- 👉 语句导致的表达式运算

在 JavaScript 中，“语句具有值”既是其解释执行的一种自然结果^①，也是源于语言本身的函数式特性，以及动态执行过程中 `eval()` 对语法解释的需求。对于后面这两点，读者可以参阅下列章节：

- 👉 1.3.2 运算式语言
- 👉 1.3.3 如何消灭掉语句
- 👉 1.2.2 动态执行过程中的语句、表达式与值

“变量声明”也是语句的副作用之一。这里强调的并不是直接使用 `var` 来声明的变量，而是 `for` 和 `for..in` 语法中使用 `var` 关键字来声明的变量。因为 `for/for..in` 语句本身只陈述循环逻辑和循环过程中的代码分块结构，因此“是否使用一个本地的变量来迭代”就不是该语法本身必须的效果了。

对于具名的函数声明来说，抛开语法解析导致的问题不讲，一个具名函数是完全等同于将一个匿名函数赋值给一个已经声明的变量标识符的。然而正是这“被抛开的问题”成了困惑的根源。因为这带来了一个“是否在表达式的执行过程（而非语句的解析过程）中理解标识符”的问题。即使将这个问题局限在语法解析期，也会有“在语法作用域中是否需要理解多层次的标识符声明”的问题。这两个问题造成了 JScript、SpiderMonkey JavaScript 以及 KJS 等主要的 JavaScript 引擎在动态执行、函数闭包等方面的明显差异^②，关于这方面的一些细节，读者可以参阅：

- 👉 1.6 闭包
- 👉 1.2.2 动态执行过程中的语句、表达式与值

^① 这里的含义是：由于所有语句被解析为可执行的语法树，因此最后一个被求值运算的结点的值就是整个语句的值。由于本书不把编译器 / 解释器原理作为重点，因此这里仅概要地提及这种效果的成因。

^② 然而遗憾的是，在这些问题上 ECMA 规范定义并不明确。尤其是在动态语言的实现及效果上，ECMA 几乎缺乏任何有约束性的描述。

👉 1.4.2.1 语法声明与语句含义不一致的问题

👉 1.4.2.7 构造绑定

第三种副作用是指语句导致的表达式运算，其典型负面效果便是经常被开发人员用来表演戏法的一种技巧：在“1.4.3 循环语句”小节中提到的在循环操作非循环变量的问题。导致这种问题的根本原因在于 JavaScript 中的一些表达式运算可以影响到值或对象自身，例如常见的用法：

```
var arr = [], x;
while ( x = arr.pop() ) {
    // ...
}
```

在这个语句中，“`x=arr.pop()`”是一个较常见的用法，但 `arr.pop()` 导致的出栈效果就是这个语句本身的副作用。事实上 `while` 语句只关心栈顶“是否是一个有效的元素”，而并不关心栈顶是否移除了一个元素。

另一个原因使这一局面迅速恶化：JavaScript 中可以广泛地使用一般表达式，且又允许在这其中使用表达式连续运算。举例来说，对于 IF 语句，可以写成如下这种复杂的形式：

```
if (a=1, b=2, c=3, d=4, x=a+b-c*d, --x) {
    // ...
}
```

我当然不知道为什么要写成这样，对于这个 IF 语句本身来说，只有“`--x`”这个运算的结果值有意义——注意，你还必须记得是“`x--`”还是“`--x`”，他们作为表达式的值含义是不同的。因此除了这最后一个表达式来说，所有其它表达式都成了该语句的副作用。

在 JavaScript 中，几乎所有语句（其语法成员中的判别式）都可以使用这种由连续表达式构成的、只有最后一个表达式有意义的语法，这成了灾难最为集中的地方。要知道，即使是 JavaScript 老手也难以想象下面这样的表达式在语句中会有怎样的效果：

```
// 语法问题①
delete x=y, y=z, z
```

① 尽管“`delete x, y, z`”这样的语法本身是可以被理解的，但在这里，整个表达式会导致错误。这需要从运算符优先级的角度上加以详细考量。由于 `delete` 的运算级高于“`=`”和“`,`”，因此该表达式并非用 `delete` 删除多个运算分量，也不是删除后三个连续运算的返回值 `z`，而是最先运算“`delete x=y`”，然后与后两个表达式构成连续运算，（语义上）应当返回 `z` 的值。但在不同的引擎中，对这一运算的理解又存在细节的差异。在 JScript 中这“`delete x=y`”被视为不合法的语法，而 SpiderMonkey JavaScript 中则是在运行期理解为“`delete x`”的结果值为等式左侧的运算元，且因该运算元不能被赋值而出现执行期错误。

```
// 将 z 传入 y, 以 z 值的字符串形式存取变量 x 的成员, 然后将 z 值减 1
x[y=z, z--] = 100
```

更何况在这种情况下，开发人员还要记住每一个语句中的语法成员关注的值的个数。例如对于下面的 `with` 语句^①：

```
// x,y 作为单值表达式运算, 但值被忽略, 变量 z 作为 with 语句操作的对象
with (x, y, z) {
    //...
}
```

不过有些看起来很象副作用的语法效果，却只是该语句的作用而非副作用。例如 `with` 语句的作用就是打开对象闭包，使后续语法的作用域发生变化，因此这一效果并不是该语句的副作用，而只能视为它的作用。与此类似的，`var`、`let` 等语句自身的效果——例如声明变量标识符也只是它们的作用而已。

1.3.[XXX]基本数据类型的结构化含义

复杂类型：数组、对象、字符串

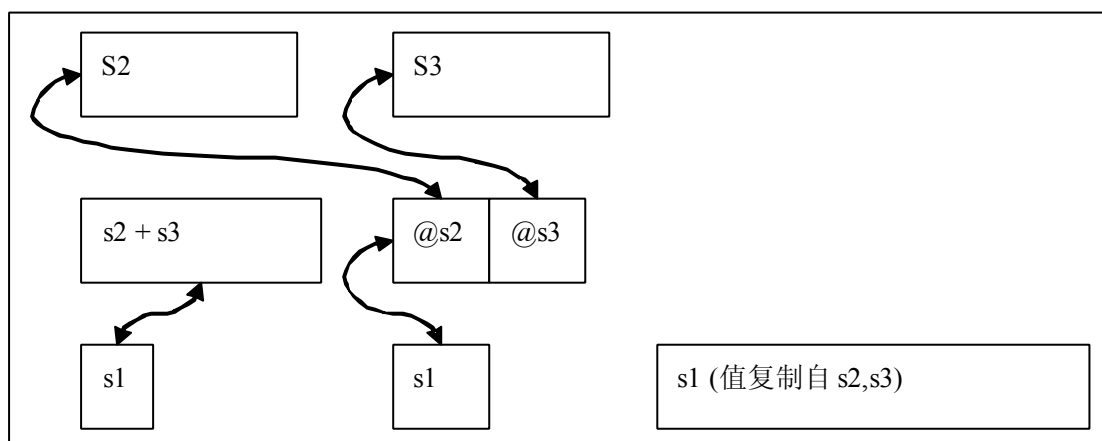
简单类型：`undefined`、数值和布尔值

其它：函数类型(不讲?)

对于数据结构的理解中，JavaScript 对对象、字符串和数组并没有“连续存储”的约定。在现实的实现中，字符串被处理成连续的字符集合。但是由于 JavaScript 禁止存储字符串中的单一元素（字符），因此“是否连续存储”对于该语言是没有意义的。我们可以假定如下代码：

```
s1 = s2 + s3;
```

在内存中的结构是“运算的引用”或者“顺序表的引用”，也或者是“连续的存储”：



对于此后的运算，都可以找到合理的处理方案。例如：

```
s4 = s1.substr(100);
```

^① SpiderMonkey JavaScript 1.7 的一些语法使这一问题加剧恶化。例如“`for (let [x,y]=[1,2], ...)`”这样的语法中，就需要关注不确定个数的值。

但是，“是否连续存储”的问题，在 JavaScript 中，是利用其“动态语言”的特性来消解的。这一点，我们放到第六章中去讲述。

1.4. JavaScript 中的原型继承

“面向对象”有三个基本特性，即封装、继承和多态。一般来说，三个特性都完全满足的话，我们称为“面向对象语言”，而称满足其中部分特性的语言为“基于对象语言”——这里使用了“基于对象”概念的异常多种解释中并不常用的一种，因为其它的解释会与后续的陈述混淆。

“对象系统”的继承特性，有三种实现方案，包括基于类（class-based）、基于原型（prototype-based）和基于元类（metaclass-based）。这三种对象模型各具特色，也各有应用。在这其中，JavaScript 没有采用我们常见的类继承体系，而是使用原型继承来实现对象系统。因此 JavaScript 没有“类（Class）”，而采用一种名为“构造器（Constructor）”的机制来实现类的某些功用^①。在本节中，为了叙述的方便，会用“对象（类）”来表明类的特性，而用“对象”（或“实例”、“对象实例”）来表明单一一个对象的特性。特别强调的是，在陈述“对象（类）”的特性时，相当于讲述由构造器或由构造机制带来的特性。

“原型继承（而非类继承）”是 JavaScript 最重要的语言特性之一。正是因此，才使得 JavaScript 拥有了丰富、多变且适用于动态语言的对象系统。本小节主要讨论这种继承体系，以及与此相关的构造、析构和属性（和方法）存取。

1.4.1. 空对象(null)与空的对象

很少有人会从这个话题开始讨论原型继承，但我必须先指出：在 JavaScript 中，“空的对象”是整个原型继承体系的根基。

在 JavaScript 中，null 是作为一个保留字存在的。null 不是“空的对象”，而是代表这样一个对象：

👉 属于对象类型；

👉 对象是空值的。

因为它是对象类型，所以你可以甚至可以去列举(for..in)它；又因为它是空值，所以没有任何方法和属性，因而列举不到内容。如下例所示：

^① 也因此 JavaScript 被称为“无类语言”。

```
// null 是一个属于对象类型的对象
alert( typeof null );

// null 可被列举属性
var num = 0;
for ( var propertyName in null ) {
    num++;
    alert(propertyName);
}
// 显示值 0
alert(num);
```

null 对象也可以参与运算，例如“+（加法和字符串连接）”或“-（减法）”运算。但因为它没有属性也没有方法，同时也没有原型，所以与这些相关的调用都会失败。由于它并不是自 `Object()` 构造器（或其子类）实例而来，因此 `instanceof` 运算会返回 `false`。

而所谓“空的对象”（也称为裸对象、空白对象等^①），就是一个标准的、通过 `Object()` 构造的对象实例。例如我们使用：

```
obj = new Object();
```

来得到的 `obj` 实例。此外，对象直接量也会隐式地调用 `Object()` 来构造实例，因此下面的代码也可以得到一个“空的对象”：

```
obj = { };
```

空的对象具有“对象”的一切特性。因此可以存取预定义属性和方法（`toString`、`valueOf` 等），而 `instanceof` 运算也会返回 `true`。

我有些时候会说空的对象是“干净的对象”。在缺省情况下，空的对象只有预定义属性和方法，而 `for .. in` 语句并不列举这些属性和方法。因此，所谓空的对象在 `for..in` 中并不产生任何效果。但是总有一些操作会使得空的对象在 `for..in` 中显得“并不那么干净”——会列举出一些属性名，而我们在下面将要讲到的原型操作就会有这种负面的作用^②。

^① 一些文档中也会称它为 `empty objects`。

^② 此外，在某些引擎（例如 SpiderMonkey JavaScript）中重写它的内部成员，也会导致这种负面的作用。

1.4.2. 原型继承的基本性质

在 JavaScript 的语言和对象系统的实现来讲，对象 (Object Instance) 并没有原型，而是构造器 (Constructor) 有原型。对象只有“构造自某个原型”的问题，并不存在“持有（或拥有）某个原型”的问题。

原型其实也是一个对象实例。原型的含义是指，如果构造器有一个原型对象 A，则由该构造器创建的实例 (Instance) 都必然复制自 A。

这里的“复制”就存在了多种可能性，由此引申出了动态绑定和静态绑定等等问题。但我们先不考虑“复制”如何被实现，而只需先认识到：由于实例复制自对象 A，所以实例必然继承了 A 的所有属性、方法和其它性质。

“原型也是对象实例”是一个最关键的性质，这是它与“类继承体系”在本质上不同。对于类继承来说，类不必是“对象”，因此类也不必具有对象的性质。举例来说，“类”可以是一个内存块，也可以是一段描述文本，而不必是一个有对象特性（例如可以调用方法或存取属性）的结构。

1.4.3. 空的对象是所有对象的基础

我们用下面的代码来考察一下最基本的 Object() 构造器：

```
// 取原型对象
obj = Object.prototype;

// 列举对象成员并计数
var num = 0;
for (var n in obj) {
    num++;
}

// 显示计数：0
alert(num);
```

可见，Object() 构造器的原型就是一个空的对象。

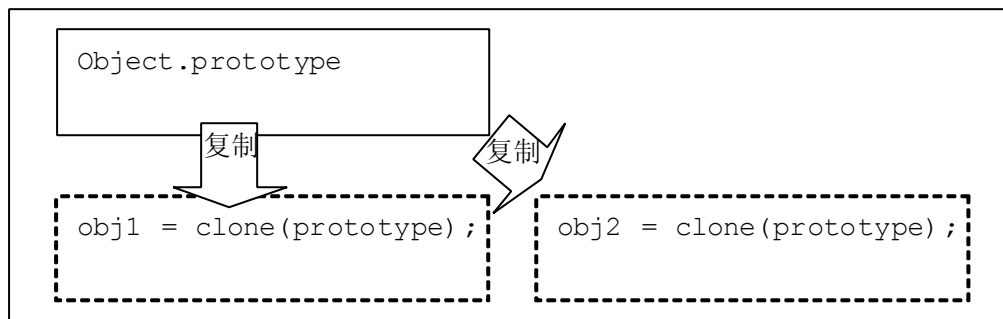
那么，这有什么意义呢？

这意味着下面的两行代码，都无非是自 Object.prototype 上复制出了一个“对象”的映像来——它们也是“空的对象”：

```
obj1 = new Object();
```

```
obj2 = { };
```

因此对象的“构建过程”可以被简单地映射为“复制”。如下图所示：

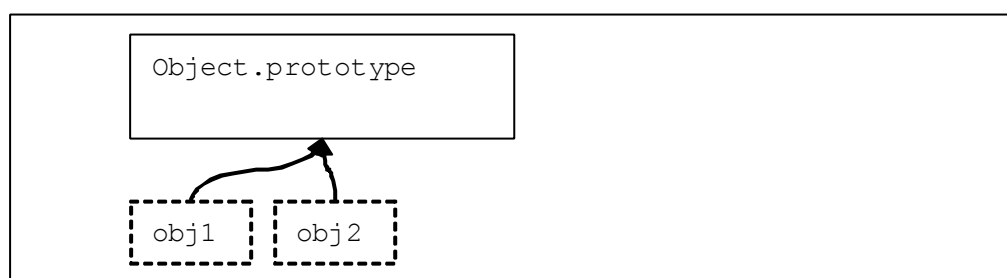


(图一)

1.4.4. 构造复制？写时复制？还是读遍历？

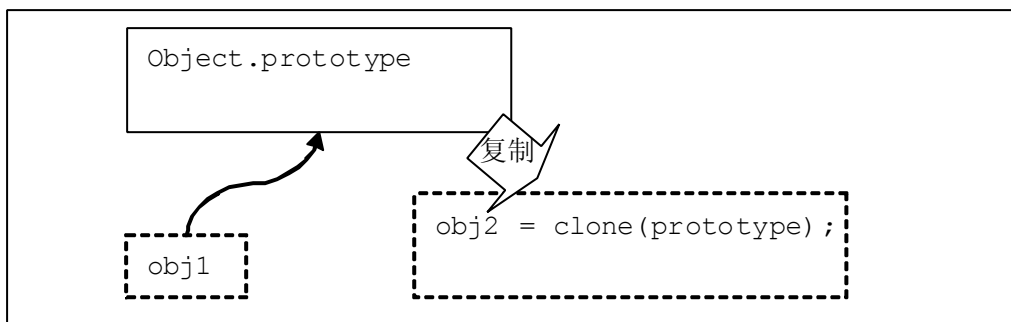
上面的这个图例假设每构造一个实例，都从原型中复制出一个实例来，新的实例与原型占用了相同的内存空间。这虽然使得 `obj1`、`obj2` 与它们的原型“完全一致”，但也非常地不经济——内存空间的消耗会急速增加。

另一个策略来自于一种欺骗系统的技术：写时复制。这种欺骗的典型示例就是操作系统中的动态链接库(DLL)，它的内存区总是写时复制的。这种机制的情况大致如下：



(图二)

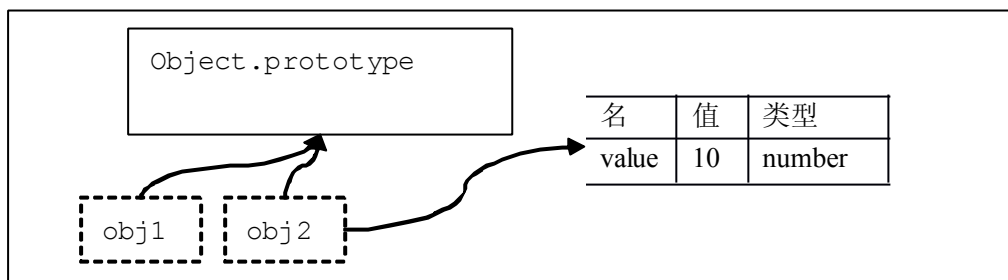
我们只要在系统中指明 `obj1` 和 `obj2` 等同于它们的原型，这样在读取的时候，只需要顺着指示去读原型即可。当需要写对象（例如 `obj2`）的属性时，我们就复制一个原型的映象出来，并使以后的操作指向该映象就行了。这种情况大致就变成了：



(图三)

这种方式的优点是我们在第一次写的时候会用一些代码来分配内存，并带来一些代码和内存上的开销。但此后这种开销就不再有了，因为访问映象与访问原型的效率是一致的。不过对于经常写操作的系统来说，这种法子并不比上一种法子经济。

JavaScript 采用了第三种法子：把写复制的粒度从原型变成了成员。这种方法的特点是：仅当写某个实例的成员时，将成员的信息复制到实例的映象中。这样一来，在初始构造该对象时的局面仍与“图二”一致。但当需要写对象属性（例如 `obj2.value=10`）时，会产生一个名为 `value` 的属性值，放在 `obj2` 对象的成员列表中：



(图四)

我们发现 `obj2` 并没有变化，它仍然是一个指向原型的引用，而且在操作过程中并没有一个与原型相同大小的对象实例创建出来。这样一来，写操作并不导致大量的内存分配，因此内存的使用上就显得经济了。

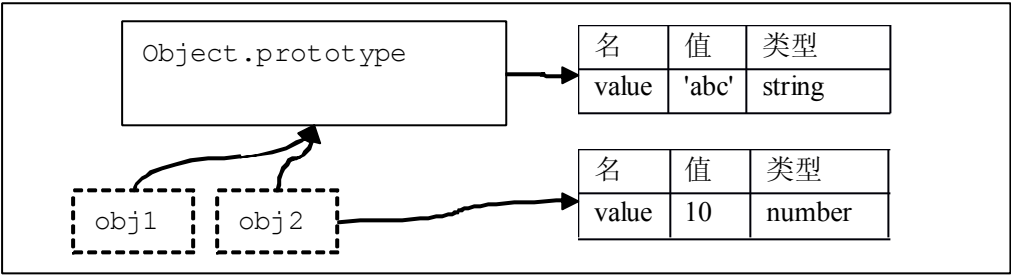
但是 `obj2`（以及所有的对象实例）需要维护一张成员列表。这个成员列表指向在 `obj2` 中发生了修改的成员名、值与类型。这张表是否与原型一致并不重要，只需要：

👉 规则 1：保证在读取时被首先访问到即可。

现在我们访问 `obj2.value` 时，就可以得到值 10 了。但是 `obj1.value` 呢？这时“读遍历”规则就起作用了：

👉 规则 2：如果在对象中没有指定属性，则尝试遍历对象的整个原型链，直到原型为空(`null`)或找到该属性。

因此访问 `obj2.value` 的结果，将取决于原型（以及整个原型链）的成员列表的情况了。如果图 4 中的原型也持有一张表，如下图：



（图五）

那么 `obj1.value` 自然就得到了值 'abc'。因此，我们也可以发现原型继承带来的另一个效果：`obj1.value` 与 `obj2.value` 类型并不相同。当然，如果原型没有上面这样的一张成员列表（或者说它的成员列表为空），那么 `obj1.value` 的值就将是 `undefined` 了。

注意上面的这个规则，其实是与“对象是什么”没有关系的。这个规则描述的是一种对象的成员的存取规则，以及存储这些成员时的数据结构约定。最后关于这个结构的唯一一点补充是：存取实例中的属性，比存取原型中的属性效率要高。很明显的例子是，在图五中存取 `obj2.value` 比 `obj1.value` 要少一个指针访问。

1.4.5. 构造过程：从函数到构造器

显然上面的规则只讲述了“对象形成的过程”，而并不讲述构造过程。也就是说，我们并没有解释“函数作为一个构造器，都做了些什么”。

其实函数首先只是函数，尽管它有一个“`prototype`”成员，但如果每声明一个函数都“先创建一个对象实例，并使 `prototype` 成员指向它”，那么也并不经济。所以，我们事实上可以认为 `prototype` 在函数初始时根本是无值的，实现上可能是如下的逻辑：

```

var __proto__ = null;

function get_prototype() {
    if (!_proto_) {
        __proto__ = new Object();
        __proto__.constructor = this;
    }
    return _proto_;
}

```

所以，函数只有在需要引用到原型时，才具有构造器的特性。而且函数的原型总是一个标准的、系统内置的 `Object()` 构造器的一个实例，不过该实例创建后 `constructor` 属性总先被赋值为当前函数。

这一点很好证明，因为 `delete` 运算符总是可以删去当前属性，而让成员存取到父类的属性值。所以：

```

function MyObject() {
}

// 1. 显示 true，表明原型的构造器总是指向函数自身
alert( MyObject.prototype.constructor == MyObject );

// 2. 删除该成员
delete MyObject.prototype.constructor;

//3. 删除操作将使该成员指向父代类原型中的值
// （显示值 true）
alert( MyObject.prototype.constructor == Object );
alert( MyObject.prototype.constructor == new Object().constructor );

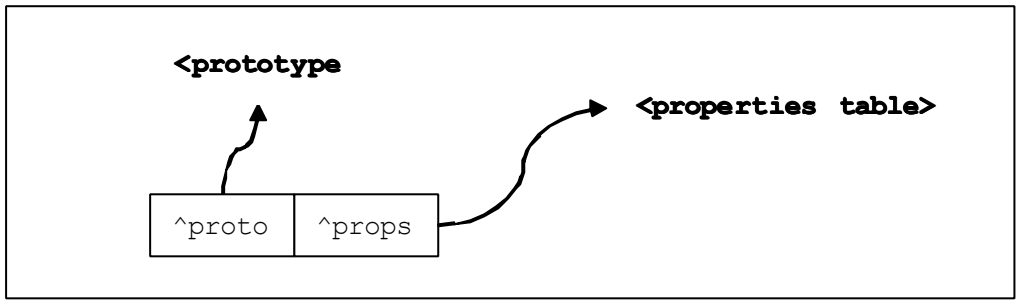
```

所以 `MyObject.prototype` 其实与一个普通对象——“`new Object()`”创建的实例并没有本质区别。然而当一个函数的 `prototype` 有意义之后，它就摇身一变成了一个“构造器”。这时，如果用户试图用 `new` 运算创建它的一个实例，那么引擎就再构造一个对象，并使该对象的原型链接向这个 `prototype` 属性就可以了。因此，函数与构造器并没有明显的界限，其中的唯一区别，只在于原型 `prototype` 是否是一个有意义的值。

当然，你也可以认为函数的 `prototype` 总是有意义的——只是本体以函数存在时它显得多余罢了。你怎样理解都没有关系，不过你得记住“构造器的 `prototype` 属性总是来自于 `new Object()` 产生的实例”这个基本假设，这在讲述“动态语言特性”时，将会成为一切推论的起点。

1.4.6. 预定义属性与方法

由构造过程我们了解到，JavaScript 中的对象实例本质上只是“一个指向其原型的，并持有一个成员列表的结构”。这个结构最简单的表示法如下：



可见，对象实例本身并没有什么特别的性质，甚至都不象一个“对象”（而象——或者说是——一个普通的结构体）。它所有的对象性质，来源于系统对原型的，以及对成员列表的理解。

而我们前面说过“空的对象是所有对象的基础”。也就是 `Object.prototype` 是所有对象的最顶层的原型。我们所谓的“空的对象”，以及“干净的对象”，只不过是满足以下条件的一个结构：

- 👉 `^proto` 指向 `Object.prototype`;
- 👉 `^props` 指向一个空表。

更进一步的推论是：我们所有的“实例”，之所以具有对象的某些性质，是因为它们的共同原型 `Object.prototype` 具有某些性质。

下面对这些对象实例的这些性质做一个分类。需要强调的是，某些性质并不是“原型继承”所必须的，而是 JavaScript 作为“动态语言”所必须的。这个分类是：

成员名	类型	分类
<code>toString</code>	<code>function</code>	动态语言
<code>toLocaleStr ing</code>	<code>function</code>	
<code>valueOf</code>	<code>function</code>	
<code>constructor</code>	<code>function</code>	对象系统：构造
<code>propertyIsEnumerable</code>	<code>function</code>	对象系统：属性
<code>hasOwnProperty</code>	<code>function</code>	
<code>isPrototypeOf</code>	<code>function</code>	对象系统：原型

对于一个具体的构造器来说，它除了具有上述普通对象的成员之外（因为

自身是一个对象)，还具有几个特别的、属于函数类型对象的成员。下面对构造器的特殊成员做一个分类：

成员名	类型	分类
call	function	函数式语言
apply	function	
caller	function	
arguments	object	动态语言
length	number	
prototype	object	对象系统：原型

1.4.7. 原型链的维护

1.4.7.1. 两个原型链

从上表来看，维护原型链似乎全是构造器的事情，因为只有构造器（函数对象）有一个名为 `prototype` 的成员。但事实上并非完全如此。从更前面的讨论来看，我们知道一个实例至少应该拥有指向原型的 `proto` 属性，这是 JavaScript 中的对象系统的基础。不过这个属性是不可见的，我们称之为“内部原型链”，以便和构造器的 `prototype` 所组成的“构造器原型链”（亦即是我们通常所说的“原型链”）区别开来。

下面列举一个示例，先构造了一个原型继承的对象系统：

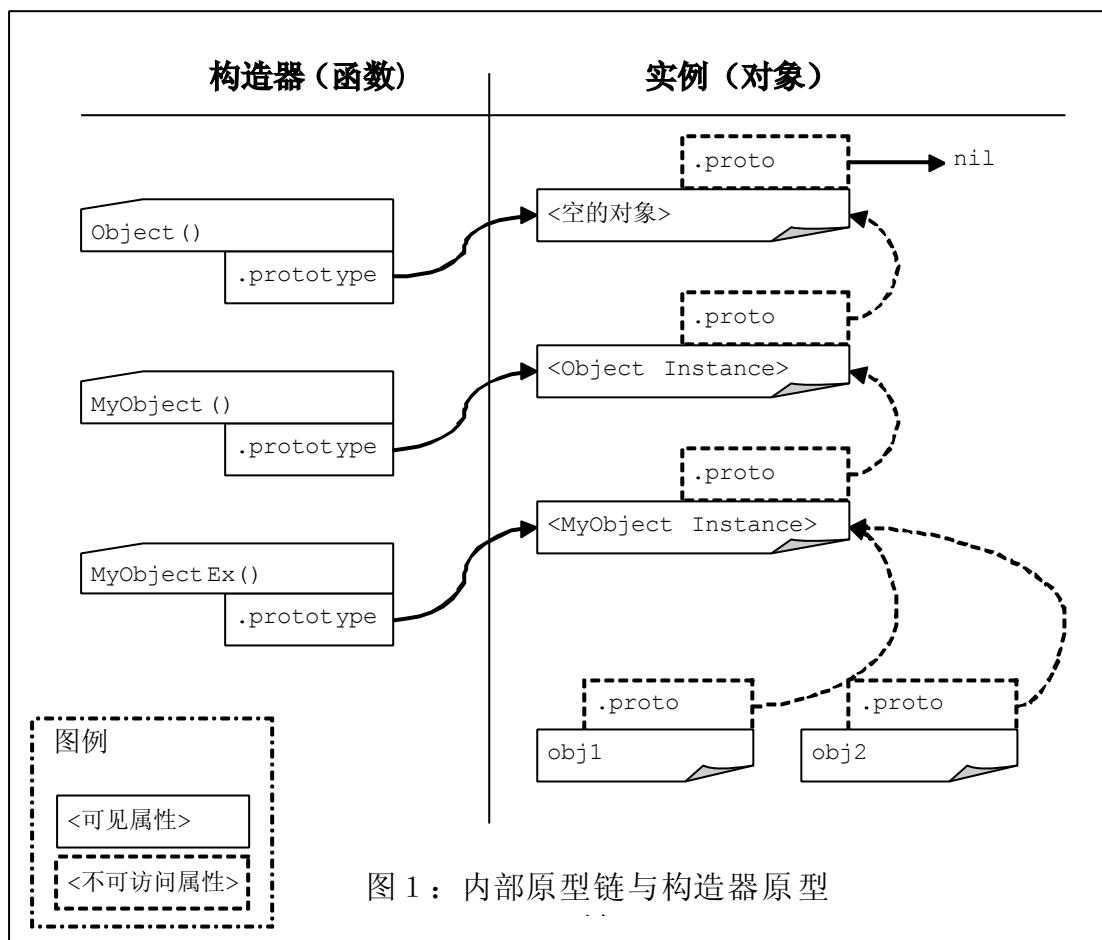
```
/**
 * 构造器声明
 */
function MyObject() {
}
function MyObjectEx() {
}

/**
 * 原型链
 */
MyObjectEx.prototype = new MyObject();

/**
 * 创建对象实例
 */
```

```
obj1 = new MyObjectEx();
obj2 = new MyObjectEx();
```

下图展示这段代码所构成的内部原型链与构造器原型链：



这个图说明构造器通过了显式的 `prototype` 属性构建了一个原型链，而对象实例也通过内部属性 `proto` 构建了一个原型链。从这个图上也可以发现，由于 `obj1.proto` 是一个不可访问的内部属性，所以没有办法从 `obj`（指代所有 `MyObjectEx` 的实例）开始访问整个原型链。

解决这个问题的法子是通过 `obj.constructor`。

1.4.7.2. constructor 属性的维护

一个构造器产生的实例，其 `constructor` 属性缺省总是指向该构造器。因此，下面的代码将显示为 `true`：

```
function MyObject() {
```

```
}  
  
var obj = new MyObject();  
alert(obj.constructor == MyObject);
```

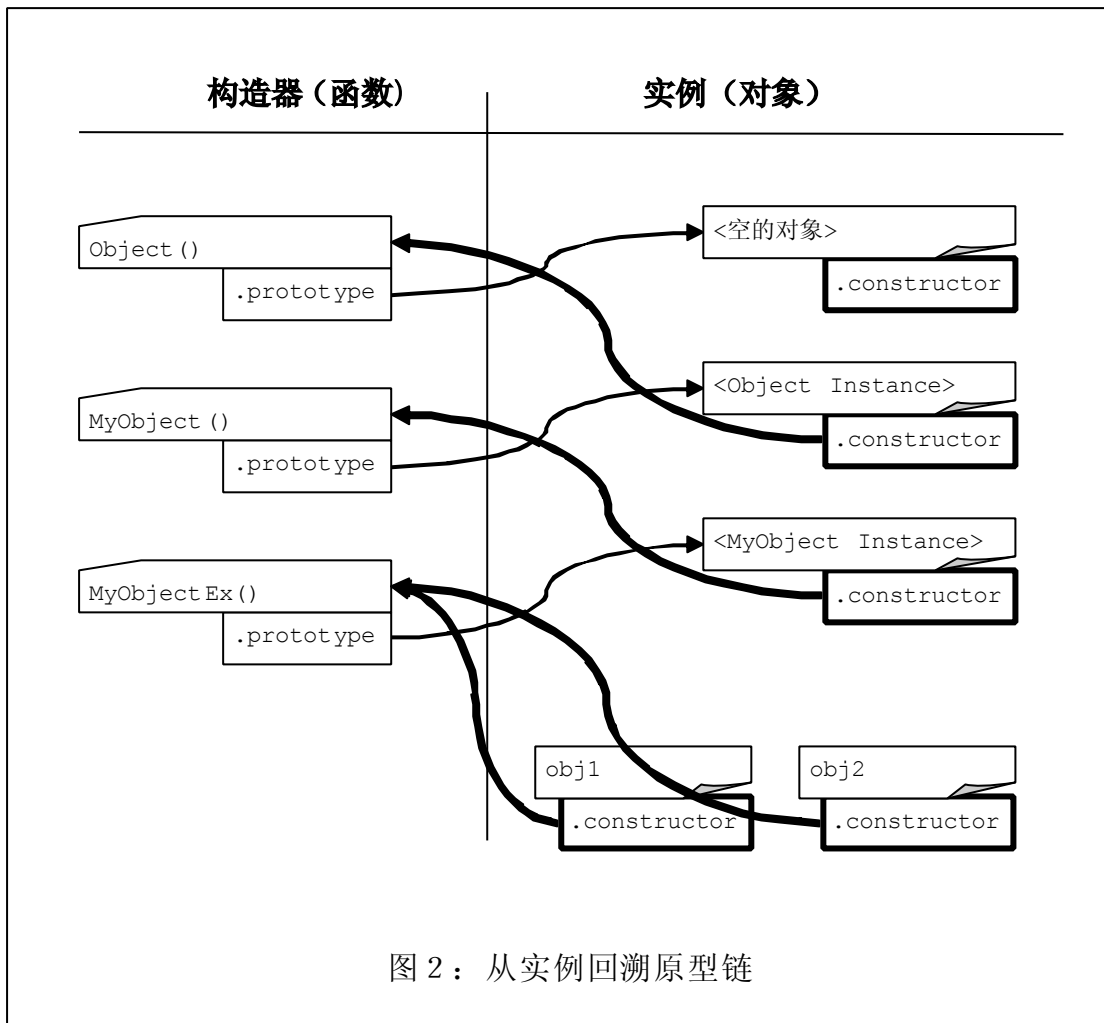
而究其根本源，则在于构造器(函数)的原型的 `constructor` 属性指向了构造器本身。所以下面的代码也显示为 `true`:

```
// (续上例...)  
alert(MyObject.prototype.constructor == MyObject);
```

所以，JavaScript 已经为构造器维护了原型属性。因此，一般情况下（强调一下，这里说的是“一般情况”），我们可以通过实例 `constructor` 属性来找到构造器，并进而找到它的原型。如下例：

```
// (续上例)  
alert(obj.constructor.prototype);
```

这样一来，我们在图 1 中，找到了一个连接点，使得我们在实例不能访问 `obj.proto` 的情况下，通过构造器来访问原型链。因此，我们“似乎”可以补完对图 1 中的构造器原型：



这看起来完美，但接下来我们来讲述 `constructor` 与原型继承的冲突。而这大概会让刚刚建立起来的、对“原型继承”的一点点好感备受挫折。

首先，下面的两个构造器的 `prototype.constructor` 也都毋庸置疑地、正确地指向了构造器自身：

```
function MyObject() {
}

function MyObjectEx() {
}
```

但下面的一行代码导致了问题：

```
/**
 * 等效于下面的代码：
 *   proto = new MyObject();
```



```
*   MyObjectEx.prototype = proto;
*/

// 构建原型链
MyObjectEx.prototype = new MyObject();
```

（为了叙述方便，我们使用上面的等效代码，）由于我们重置了 `MyObjectEx` 的原型属性，使之指向 `proto`。而 `proto` 又是 `MyObject` 的实例，那么显然：

```
proto.constructor = MyObject.prototype.constructor = MyObject;
```

因此在构造原型链之后，整个的 `MyObjectEx` 就出了问题：

```
MyObjectEx.prototype.constructor = proto.constructor = ... = MyObject;
```

那么请预测一下，根据我们前面讲过的“原型复制”的概念，下面的代码将显示什么呢？

```
var obj1 = new MyObject();
var obj2 = new MyObjectEx();
alert(obj1.constructor == obj2.constructor);
```

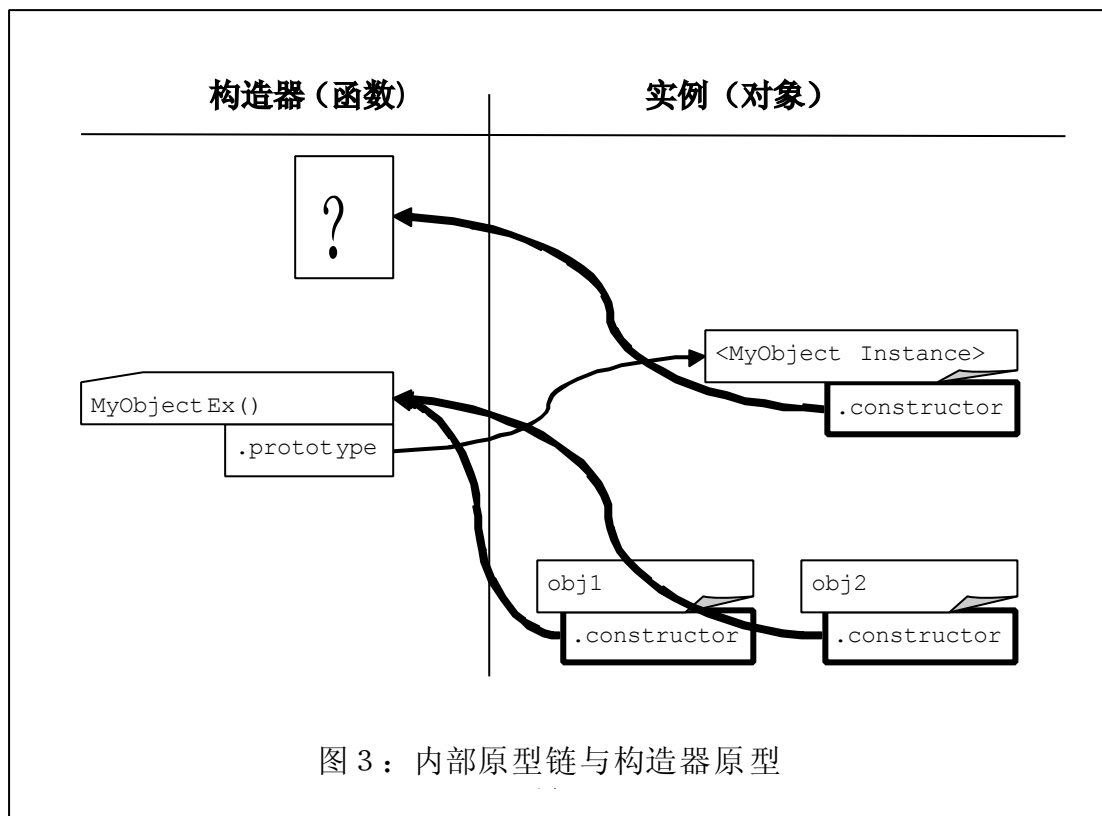
没错！将显示 `true`！结果是：`obj1` 与 `obj2` 是不同的构造器产生的实例，而它们的 `constructor` 属性却指向了相同的构造器！

到底是构造器出了问题呢？还是原型出了问题？

我们看到，原型继承要求的“复制行为”其实已经正确的发生了，问题是出在我们给 `MyObjectEx` 的原型时，应该“修正”该原型的构造器值。为此，一般的建议（例如《JavaScript 权威指南》）是这样处理：

```
// 方法一：直接构建原型链
MyObjectEx.prototype = new MyObject();
MyObjectEx.prototype.constructor = MyObjectEx;
```

也就是重置原型后，再修改原型的 `constructor` 属性，使之指向正确的构造器函数。但大家很快会看到一个问题：由于丢掉了原型的 `constructor` 属性，因此事实上我们也就切断了与原型父类的关系。仍以上图为例：



我们看到疑难。出于原型继承的目的，obj1、obj2 等实例需要属性 `MyObjectEx.prototype.constructor`

正确的指向 `MyObjectEx()` 构造器；而如果要正确的访问原型链，则 `<MyObject Instance>.constructor`

又应该指向 `MyObject()` 构造器，这与前一个需求又自相矛盾。

另一个方法是保持原型的构造器属性，而在子类构造器函数内初始化实例的构造器属性——这看起来有点令人发晕，但代码不过如下：

```
// (参见上例...)

/**
 * 方法二：正确维护 constructor，以便回溯原型链
 */
function MyObjectEx() {
    this.constructor = arguments.callee;
    // or, this.constructor = MyObjectEx;

    // ...
}
```

```
}

/**
 * 原型链
 */
MyObjectEx.prototype = new MyObject();
```

这样一来，`MyObjectEx()` 构造的实例的 `constructor` 属性都正确的指向 `MyObjectEx()`，而原型的 `constructor` 则指向 `MyObject()`。于是图 2 的回溯方法便可以成立了。虽然你会发现它的效率较低——因为每次构造实例时都要重写 `constructor` 属性——但是它是唯一有效的方法。

够啰嗦了吗？没办法，如果你不需要回溯原型链，那么你可以用第一种方法，否则你得按第二种方法来。

1.4.7.3. 内部原型链的作用

接下来会有人问：既然用户代码只需要正确维护 `constructor` 属性就可以回溯原型链，那么实例的内部属性 `proto` 有什么价值呢？换言之，内部原型链有什么价值呢？

这个问题与原型继承的实质有关，也与面向对象的实质有关。面向对象的继承性约定：子类与父类具有相似性。在原型继承中，这种相似性是在构造时决定的，也就是由 `new()` 运算内部的那个“复制”操作决定的。

如果我们改变了“继承”的定义，说：子类可以与父类不具备相似性。那么我们就违背了对象系统的基本特性。因此，你会发现子类实例有一个特性：不能用 `delete` 删除从父类继承来的成员。也就是说，子类必须具有父类的特性。即使你可以重写成员，改变它的实现，但在界面(`Interfaces`)上必然与父类一致。

为了达成这种一致性，且保证它不被修改。`JavaScript` 使对象实例在内部执有这样一个 `proto` 属性，并且不允许用户访问。这样，用户可以出于任何目的来修改 `constructor` 属性，而不用担心实例与父类的一致性。

简单地说，内部原型链是 `JavaScript` 的原型继承机制所需的。而通过 `constructor` 与 `prototype` 所维护的构造器原型链，则是用户代码需要回溯时才需要的。如果用户无需回溯，那么不维护这个“原型链”，也没有关系——例如上一小节中用于修正 `constructor` 属性的“方法一”。

1.4.8. 原型继承的实质

现在我们总算觉得原型继承有点价值了。

回顾一下，我们在此前已经（通过蹩脚的方式）解决了用户代码回溯原型链的问题，我们也讲述了原型继承模式下访问对象成员的实质。我们还讨论了原型继承中，“空的对象”是所有对象的基础。当然，这一论点的前提是：原型也是对象实例（所以“空的对象”可以被用作原型）。

下面我们详细讨论原型继承的应用与相关技术。

1.4.8.1. 原型修改

我们前面讲过“原型继承系统”的特性，包括：

- 👉 原型是一个对象；
- 👉 在属性访问时，如果子类对象没有，则访问其原型的成员列表。

根据这些特性（以及原型的定义）可以知道，我们如果修改一个构造器（或称为类）的原型，则所有由该类创建的实例都将受到影响——如果有其它子类继承自该类，则所有子类也将受到影响——因为在存取成员列表时必将回溯到该类。

修改原型是 JavaScript 中最常用的构建类系统的方法，它的好处在于可以在实例构造之后“动态地”影响到这些实例。也就是说，实例（对象）的特性不但可以在 `new` 运算中通过“构造”来产生，也可以在此后通过原型“修改”来产生。简单的示例说明如下：

```
// 构造器声明
function MyObject() {
}

// 构造完成后并没有成员 'name'
var obj = new MyObject();
alert('name' in obj);

// 通过原型得到了成员 'name'
MyObject.prototype.name = 'MyObject';
alert('name' in obj);
```

而这种修改原型的特性，是我们下面讨论原型继承的基础。

1.4.8.2. 原型继承

用来讲述原型继承的一个比较学术的示例，是做一个动物王国的模型。但是，无论有何等伟大的构想，我们也得从一只猫和一只狗开始：

```
/**
 * 示例 1
 */
// 1. 构造器
function Animal() {}; //动物
function Mammal() {}; //哺乳类
function Canine() {}; //犬科
function Dog() {}; // 狗
function Cat() {}; // 猫

// 2. 原型链表
Mammal.prototype = new Animal();
Canine.prototype = new Mammal();
Dog.prototype = new Canine();
Cat.prototype = new Mammal();

// 3. 示例函数
function isAnimal(obj) {
    return obj instanceof Animal;
}

// 4. 示例代码
var dog = new Dog();
var cat = new Cat();
document.writeln(isAnimal(dog));
```

结果输出 **true**——证明狗是一种动物，也证明我们的生物学学得不错。接下来我们要在这个系统上应用第一个规则：动物呼吸。我们该怎么做呢？

这就会用到我们上面讲述的“原型修改”的方法了：

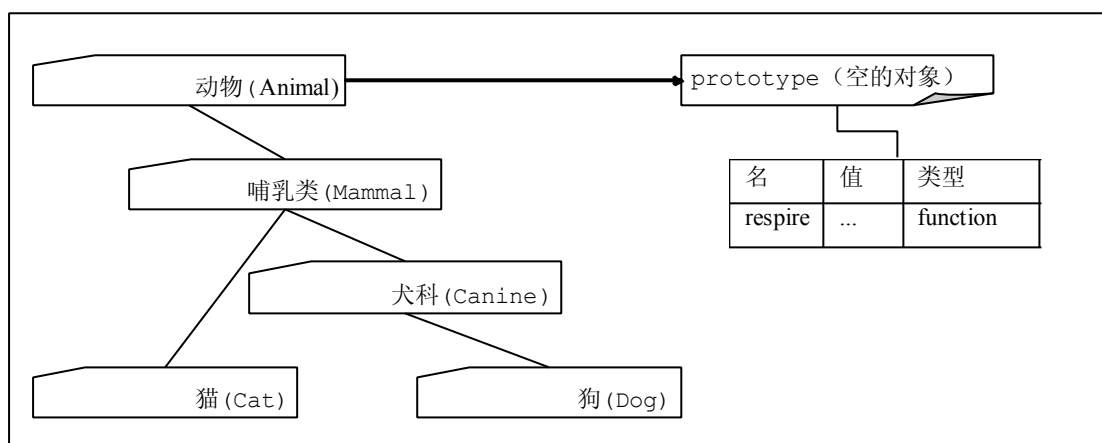
```
/**
 * 将下列代码插入到示例 1 的步骤 (1) 和 (2) 之间
 */
Animal.prototype.respire = function() {
    // 交换氧气与二氧化碳
}
```

现在我们就可以看见无论是 `dog`，还是 `cat`，都可以呼吸了：

```
// (续"示例 1")
```

```
alert( 'respire' in cat );  
alert( 'respire' in dog );
```

使用同样的方法，我们可以修改继承树上的任意分支（类），以使得它们的原型具有某些属性或方法。对于这个示例来说，该继承树形如：



注意我们构造这个继承树时，使 `Dog` 属于 `Canine` 这个分支，而 `Cat` 属于 `Mammal` 这个分支，这意味着 `Canine` 的原型修改不会影响到 `Cat`，而 `Mammal` 的原型修改会影响到 `Dog` 和 `Cat`。

我们也应该留意到，正是 JavaScript 为每个构造器初始化了一个“空的对象”，才使我们可以用前面的“原型修改”的方法重写构造器（类）的成员列表，而子类才可以继承这些属性——反之，如果原型是 `null`，或原型不是对象，那么这种子类继承父类的属性（以及重写父类的成员列表以影响子类）的特性就不可能成立。

1.4.8.3. 原型继承的实质：从无到有

如果 JavaScript 是一种“静态的语言”，那么通过上述过程（原型修改 + 原型继承）创建的所有实例将是一致的，而且对象继承树也会保持结构的稳定。由于它满足对象继承的全部特点，因此它已经是“面向对象的（静态）语言”了。

综合前面所述的内容，我们可以明确的说：原型继承与原型修改，前者关注于继承对象（在类属关系上）的层次，后者关注具体对象实例的行为特性。

在 JavaScript 中，原型的这两方面的特性是相互独立的，这也构成了“基于原型继承的对象系统”最独特的设计观念：将对象（类）的继承关系，与对象（类）的行为描述分离。

这与“基于类继承的对象系统”存在本质的不同。因为基于类继承设计时，我们必须预先考虑好某个类“是或者否”具有某种属性、方法与特质(Attribute)，如果某个类的成员设计得不正确，则它的子类、接口以及实例等等在使用中都将遇到问题。因而“重构”是必然、经常和更易出错的。

但在原型继承中，由于类继承结构与方法（等成员）的关系并不严格绑定，因此：

👉 “类属关系的继承性”总是一开始就能被设计正确的；

👉 成员的修改是正常的、标准的构造对象系统的方法。

但是，我们留意一下：“原型修改”似乎、好象、仿佛是一种动态语言特性——不是吗？的确，是这样的。这里正好就是动态语言与面向对象继承交汇的关键点。JavaScript 正是依赖动态语言的特性（可以动态地修改成员）而实现原型构建模式。这是一种所谓“从无到有（ex nihilo ("from scratch")）的模式。它首先为每一个构造器分配了一个原型，并通过修改原型构造整个对象树。接下来，如果你要访问一个实例的成员，那么可以在内部原型链中查找“成员列表”来实现。

在这里，所谓“从无到有”是指：在理论上我们可以先构建一个“没有任何成员”的类属关系的继承系统，然后通过“不断地修改原型”，从而获得一个完整的对象系统。尽管在实际应用时，我们不会绝对地将这两个过程分开，但“从无到有”的设计方法却是值得我们思考的。

1.4.8.4. 如何理解“继承来的成员”？

一个成员如果能被 for..in 语句列举出来，则它对外部系统是可见的^①，否则它不可见。一个不可见的成员仍然是可以存取的，对象成员可否存取的性质称为读写特性，某些成员不能写，称为只读的。我们本小节讨论的对象成员总是可读写的，但不一定是可见的。

子类从父类继承对象性质时，也会继承成员的可读写性与可见性。简单的

^① 这里我们使用了“可见性”这个名词，没有使用上一章说到的“可列举性”，是因为 propertyIsEnumerable() 方法在 ECMA 规范中存在设计错误，这使得“可列举性”这个词不能准确表达这里的含义。

说，父类的只读成员在子类中也“应当”是只读的^①；父类的不可见成员在子类中也“应当”是不可见的。在概念上，这是面向对象的继承性所要求的。然而源于实现方案的不同，JavaScript 的不同引擎对此的解释并不一致。

这至少存在两种解释，一种是继承成员名字，一种是继承成员性质。前一种方式将名字与性质直接关联起来，认为“该名字的成员是具有指定性质的”；而后一种方式则认为系统只是“为该名字的成员指定了特定的初始性质”。

这两种解释非常不同，这是导致 JScript(使用第一种解释)与 SpiderMonkey JavaScript(使用第二种解释)在“继承来的成员”方面存在巨大差异的根源。首先我们来看看在 JScript 中如何“继承成员的名字”：

```
// 例一：使用构造期重写
function MyObject() {
    this.constructor = null;
}
// 以下列举在 JScript 不会显示成员 constructor
var obj = new MyObject();
for (var n in obj) {
    alert(n);
}
```

在这个例子中，JScript 通过一张“需要隐藏的成员名字列表”来指写属性“constructor”是不可见的。因此即使使用 `this.constructor` 强制地重写它，也不会被列举出来。与此相似的，也可以直接声明一个直接量并指定该成员：

```
// 例二：使用直接量
obj = {
    constructor: null
}
```

同样该对象的 `constructor` 成员是不可见的。

如果在 SpiderMonkey JavaScript 运行，那么两个示例都将列举出 `constructor` 成员。这表明重写使得对象成员 `constructor` 的既有的性质（这里是指可见性、读写性等）被重置了。同时，在 `for..in` 列举时，成员的可见性不是依赖名字的，而是依赖这些可被重置的性质值。

下面，我们从深层的、对象系统实现的细节来看这个问题^②。JScript 没有

^① JScript 中不能设置对象成员的可读写性，因此本小节将跳过关于它的部分。需要补充的是，在 JScript 中的对象如果有读写性限制，那么该限制应该是在用 ActiveX 技术为 JScript 扩展类时，通过 ActiveX/COM 技术实现的，这不是 JScript 本身的特性。

^② 表面来看这是动态语言特性对对象系统的影响。ECMA 规范对动态语言产生的效果缺乏有价值的约定，的确是导致 JavaScript 引擎出现实现差异的原因之一。但我们这里讨论不同实现的具体细节，而非问责之。

在 JavaScript 引擎层面来实现对象成员的可见性，以及提供脚本代码对可见性的控制，这样的设计决定了它不必实现复杂的成员性质管理功能。在这些前提下，基于一张引擎内容保留的名字列表来控制可见性，是最小代价的实现方案。而 SpiderMonkey JavaScript 却不是这样，它引擎实现并面向使用者公开了成员性质的管理功能(面向高级语言的 SDK 中的 API)，例如你可以用这样的代码来修饰对象成员^①：

```
obj.__defineProperty__(<name>,<initValue>,<dontDelete>,<readOnly>,<dontEnum>);
```

这时，引擎的设计表明用户可以重新修改任意成员的性质。SpiderMonkey JavaScript 的策略是：当一个“继承来的成员”被重写时，该成员的性质信息的一个备份就被复制到了该对象的属性表中，并且性质将被初始化为一个普通成员的缺省值（上例性质中的 <dontDelete>,<readOnly>,<dontEnum> 分别为 false,false,false）。这样一来，我们就可以看到：一个在父类中不能被删除、列举且只读的成员被重写后，就变成了可删除、可列举和可写的。也惟其如此，该成员的重写效果才能被清除——在删除重写的成员后，将重新沿用父类的值与性质。

所以从实现上来说，成员的性质是基于名字理解的还是基于具体属性（在属性表中的）专属性质，决定了对象系统如何理解“继承来的成员”。当然，这也是引擎在设计上、应用上的一些具体需求带来的差异。

1.5. JavaScript 的对象系统

我们已经讨论了 JavaScript 的对象系统中的关键元素：原型继承，我们也提及到原型继承是一种“从无到有”的构建对象系统的方法。但是，我们还没有讲述构建对象系统的其它两个同样重要的元素：封装与多态。本小节将讨论这两个问题，此外也将会综述在 JavaScript 中构建对象系统的两种主要方式：类抄写和原型修改。

1.5.1. 封装

OOP 的封装，表达为 `private`、`protected` 等关键字限定的成员存取范围或作用域，以及对象不同继承层次上对成员的叠加。

在一般的（例如基于类继承的）对象系统中，封装特性是由语法解析来实

^① 在面向 NARCISSUS 的 SpiderMonkey JavaScript 编译版本中是可以在脚本中使用该方法的，否则用户只能通过高级语言调用引擎的 API 来调用一个类似的方法。

现的，也就是说依赖于“1.2 基本语法的结构化含义”讲述的“语法作用域”。而由于 JavaScript 是动态语言，因此它只能依赖“变量作用域”来实现封装特性。在前面，我们也讲到过 JavaScript 的变量作用域只有表达式、函数（局部）和全局三种。下表列举变量作用域对封装性的影响：

面向对象封装性	条件	语法作用域	JS 变量作用域
(partial)	声明在多个文件中可视	文件	
published	(语法效果同于 public)	(同 public)	(同 public)
public	访问不受限制，声明在类的外部可视	任意	全局
protected	该类及其派生类可视	类+子类	
internal	该程序内可视	项目	
protected internal	该程序内、该类及其派生类可视	项目+类+子类	
private	仅该类可视	类	局部

注* PartialType

该表表达的意思是：如果通过语法分析来实现这些封装性，则需要对类、子类、项目和文件等四种语法作用域进行分析。传统语言（及其编译器或语法解释器）显然都可以具备这种能力，因此实现这样的封装性并不困难。而在 JavaScript 中，基于“JS 变量作用域”来实现时，只能实现 public 和 private 这两种封装性。

读者可能看不到二者的必然性。因此我们先来看看下面的代码：

```
8  function MyObject() {
9      // 私有(private) 变量
10     var data = 100;
11
12     // 私有(private) 函数
13     function _run(v) {
14         alert(v);
15     }
16
17     // 公开(public) 属性
18     this.value = 'The data is: ';
19
20     // 公开(public) 方法
21     this.run = function() {
22         _run(this.value + data);
23     }
24 }
25
```

```
26 // 演示，最终将调用到_run() 函数
27 var obj = new MyObject();
28 obj.run();
```

我们发现，由于在 JavaScript 中“类”表现为“构造器”，而构造器本身就是一个函数。因此在（执行期的、变量的）作用域上，也就表现为函数的性质。因而这样的作用域，在事实上也就只有“函数内”与“函数外”的区别。

在 JavaScript 中，由于封装不是一种源代码级别的“说明（或声明）”，例如象 Java/Delphi/C++ 之类在源代码中“声明”某个属性是 `protected` 或 `private` 等等。因此它的封装性依赖于代码运行期间的“可见性”的效果。我们回顾前面所讲述过的内容，但基本上可以下一个定论：在 JavaScript 中不可能存在超过“变量作用域”级别的封装性。因此，我们可以排除掉上表中与“文件”、“项目”和“类+子类”等相关的封装特性。

于是，你可以看到结论：JavaScript 在封装性方面不可能做得更强了。

从目前已知的消息来看，JavaScript 2 计划放弃原型继承而采用类继承来实现对象系统。在 JavaScript 2 中，由于“类”、“构造器”与“函数”不严格地绑在一起，因此在引擎实现中，语法作用域的范围就比 JavaScript 1.x 更广，从而为实现更多的对象封装特性带来了空间——除此之外，JavaScript 2 还将提供命名空间等特性，这也增强了构建复杂的对象系统的能力。

当然，另一方面来说，JavaScript 2 也牺牲了一些 JavaScript 1.x 的简洁性。

1.5.2. 多态

多态性表达为两个方面：类型的模糊与类型的确认（或识别）。在一些高级语言中，他们分别被表达为“`as`”和“`is`”这两个关键词或运算。

JavaScript 是弱类型的，通过 `typeof` 运算考察变量时，它要么是对象 (object)，要么是非对象 (number, undefined, string 等)，绝不存在“像是某个对象或者某个类”这样的多态问题。反过来说，因为任何一个实例的类型都是 object，因此 JavaScript 这种语言本身就“类型模糊”的。

同样由于没有严格的类型检测，因此你可以对任何对象调用任何方法，而无需考虑它是否“被设计为”拥有该方法。对象的多态性被转换为运行期的动态特性——例如我们可以动态地添加对象的方法 / 成员，使它看起来象是某个

对象。下面的例子说明这种情况：

```
function Bird() {
    var bird = (arguments.length == 1 ? arguments[0] : this);

    bird.wing = 2;
    bird.tweet = function() { };
    bird.fly = function() {
        alert('I can fly.');
```

我们可以在这个例子中看到: `obj` 是不是 `Bird` 类型, 并不是“`fly`”的必要前提——`obj` 在创建时, 以及通过 `Bird(obj)` 运算转换后, 都是一个 `Object` 类型的对象, 但它是可以“飞(`fly`)”的。“能否飞行”只取决于它有没有 `fly` 方法, 而不取决于它是不是某种类型的对象。

下面我们进述类型识别的问题。

由于所有对象的 `typeof` 值都是“`object`”, 因此当在某些系统中要确知对象的具体类型时, 就需要使用 `instanceof` 运算来检测了。在“1.5.4 对象及其成员的检查”中我们已经讲过该运算, 它其实就等效于其它高级语言中的 `is` 运算。对于上例来说, 如果我们要使对象“能否飞行”取决于它是否是 `Bird` 构造器(等义于“类”)产生的实例, 那么应该使用类似下面的代码:

```
// (续上例)
// var obj = new Object();
// var bird = Bird(obj);

function doFly(bird) {
    if (bird instanceof Bird) {
        obj.fly();
    }
    else {
```

```

        throw new Error('对象不是 Bird 或其子类的实例. ');
    }
}

// 测试 2: Bird 的实例可以"fly", Object 的实例不能"fly".
doFly(new Bird()); // 能飞
doFly(bird); // 不能飞

```

然而多态性中的类型识别还不仅包括这些。另一个关键问题，是在类型继承中识别父类的同名方法。仍以上面的代码为例，但我们现在遇到的是一只鸵鸟：

```

function Ostrich() {
    this.fly = function() {
        alert('I can\'t fly. ');
    }
}

Ostrich.prototype = new Bird();
Ostrich.prototype.constructor = Ostrich;

// 演示：鸵鸟是鸟，但不能飞
var ostrich = new Ostrich();
doFly(ostrich);

```

但是对于 `doFly()` 这个函数来说，我们用以识别的表达式是：

```
(bird instanceof Bird)
```

这并没有错。因此这里也并不是 `instanceof` 运算的结果导致“飞”的失败，而是因为 `Ostrich` 类的 `fly` 方法覆盖了父类方法。我们没有理由说 `Bird` 的子类必须象 `Bird` 一样飞得很难看（例如 `Phoenix` 类一定是很艺术化地飞），而又“可能”必须具体飞的能力——因此子类必须依赖父类的某些能力来扩展新的方法。

但是，冲突出现了：我们在实现子类 `Ostrich` 时已经覆盖了 `fly` 方法。同样，我们实现 `Phoenix` 等等类时，也会覆盖这个方法。于是我们想要“依赖父类的某些方法”时，却发现“找不到这些方法”了。

对于这一点，`Delphi` 提供了 `inherited` 语句，并保留了关键字 `dynamic`（动态方法）和 `virtual`（虚方法）。但在 `JavaScript` 却无法实现相应的功能。简单地说，如果子类与父类存在同名方法，那么在对象理解“多态特性”时，是应该视为子类还是父类的方法呢？

好的，我们先给出答案。这个答案包括两个部分：

- 👉 JavaScript 中子类一定是覆盖父类的同名方法的。因此方法同名时，脚本总是直接调用子类方法；
- 👉 由于是（动态的）覆盖，所以 JavaScript 中子类方法总是无法调用父类的同名方法。

进一步的推论是：JavaScript 无法依赖父类的同名方法（或被子类覆盖的其它方法）。

继承父类的功能（而非仅是成员名称）是构建复杂的对象系统时所必须的特性。然而我们看到 JavaScript 自身并不具备这种能力，这与 JavaScript 初始设计时的主要应用环境有关：它是用于浏览器客户端快速开发的一种脚本语言，而不是用于构建大型系统的通用语言。

当 JavaScript 用作一种富浏览器客户端（RWC）中的开发语言，或者一种客户端应用开发的辅助语言（例如插件或外挂脚本）时，它在组织对象系统方面的缺点就凸显出来，成为亟待解决的问题。本书第二部分将在 Qomo 内核构建的过程中讨论相关的方法，并解决这些问题。

1.5.3. 事件

一些讨论对象系统的书籍会把 PME(Properties、Methods、Events)作为对象系统的完整的外在表现来讨论。与此类似的，接口系统现在也表达为 PME 这三个方面。

从结果上来说，这是不错的。但大多数高级语言自身其实并没有事件系统，这种情况下，事件不是一个对象系统实现中的必然需求，也是应用框架在使用该对象系统时实现的一个额外的机制。例如 Delphi 中的面向对象系统中，事件系统就（首先）是一个外来户，是 Windows 的消息系统的一个延伸。然后又因为这种延伸的需求，而使得 Delphi 中出现了 DispatchMessages()这样的方法来支持它^①。

同样的，JavaScript 中的“事件”也是外来户，JavaScript 引擎自身并没有事件系统^②。我们经常在浏览器开发中使用的 OnLoad、OnClick 这样的事件，

^① 在更早期的对象系统中，所谓 Messages 并不是指事件，而仅仅是指对象的方法。所以 Niklaus Wirth 才会说“（在对象系统中，）尽管过程现在称为‘方法’，调用一个过程现在表述为‘发送一条消息’”——这里的“消息”其实就是指方法调用。

^② 我们在讲述语法时提到过 JavaScript 中的“方法”其实是属性存取与函数调用的连续运算效果。既然方法也只是被“模拟”的，因此 Anders Hejlsberg 很形象的说 JavaScript 中的对象其实只是“属性包”。由此可见，

其实是由 DOM——一个由引擎宿主供应与维护的可编程对象模型来提供的。所以尽管大多数时候我们在用 JavaScript 来写事件的响应函数，或面向事件响应来架构系统，但事实上事件却并非 JavaScript 引擎的一个组成部分。

要用 JavaScript 来实现一个简单的事件框架是非常容易的，因为所谓“事件”的本质，仅仅是“在确定的时候发生的、可由用户代码响应的行为”而已。下面的示例说明这种结构：

```
function MyObject() {  
}  
MyObject.prototype.OnError = undefined;  
MyObject.prototype.doAction = function(str) {  
    try {  
        return eval(str);  
    }  
    catch(e) {  
        if (this.OnError) this.OnError(e);  
    }  
}  
  
// 1. 创建对象  
var obj = new MyObject();  
// 2. 添加事件处理句柄  
obj.OnError = function(e) {  
    //...  
}  
// 3. 调用方法，执行过程中可能触发 OnError 事件  
obj.doAction('aObj.tag = 100');
```

在这个示例中，如果用户代码不响应 OnError 事件，程序也可以正常运行；如果响应 OnError，也可以在处理句柄中使用“throw e”来重新触发异常。而这整个（有关于事件系统的）过程，对 JavaScript 引擎来说是透明的、未知的，引擎只是处理了 try..catch 语句，以及象 OnError、doAction 这些方法调用而已。

另外，在 JavaScript 中一些看起来象是事件的机制，其实并非是事件。例如 Array.sort()方法中需要回调的一个比例函数，就不能被理解为事件。还有在 SpiderMonkey JavaScript 中实现的 Object.watch()方法也需要一个观察对象属性变化的 handler，也不能理解为事件。尽管没有一种有说服力的方法来解释为什么它们不能被这样理解，但从 JavaScript 引擎层面扔掉有关“事件系统”的概念，确实有助于你理解引擎实现方面的更多细节。

PME 只是对象系统的外在表现形式，而非实现技术上的必须。

1.5.4. 类抄写？或原型继承？

到现在为止，我们已经讲过在 JavaScript 中构造对象系统的五种方法。包括：

```
// 方法一：在构造器中写 this 实例引用
// （参见：章节……）
function MyObject_1() {
    this.<propertyName1> = ...;
    this.<propertyName2> = ...;
}

// 方法二：在构造器中直接返回对象实例
// （参见：章节……）
function MyObject_2() {
    var data = this;

    return {
        // （使用直接量声明或 new() 构造一个实例）...
    }
}

// 方法三：修改原型
// （参见：章节……）
function MyObject_3() {
}
MyObject3.prototype.<propertyName> = ...;

// 方法四：重写原型
// （参见：章节……）
function MyObject_4() {
}
MyObject4.prototype = {
    // 使用直接量声明一个实例...
}

// 方法五：继承原型
// （参见：章节……）
function MyObject_5() {
}
// 使用 new() 构造一个实例
```



```
MyObject5.prototype = new Constructor_for_ParentClass();
```

这五种方法中，前两种方法根本没有利用 JavaScript 的“原型继承”特性，它们基本上是“类继承对象系统”的翻版。方法三和方法四虽然操作了原型，但利用的是“对象成员可以被修改”这种动态语言特性，因此，（对于类继承树的构建来讲，）事实上也没有“继承”特性。

从“修改对象成员”这个特性上来讲，方法五（继承原型）可以看作方法四的一种扩展方式。但也只有它才是 JavaScript 的原型继承性质的准确应用。这样看来，“构建原型继承关系”的唯一方法，是使用 `new` 语句来创建子类的原型。通过这种方法，继承原型会使得 JavaScript 维护对象系统的原型链，这也是唯一一种影响内部原型链的形式。

1.5.4.1. 类抄写

方法二与方法一的差异我们在“面向对象的语法概要”中已经讲述过。我们也提到过方法一是“类似于类继承系统”的，因此即使不用原型系统，我们也可以使用这种方法来构造复杂的对象系统。下面简单地举个例子：

```
/**
 * 公共函数：子类派生 extend()
 */
extend = function(subClass, baseClass) {
    // 暂存父类构造器
    subClass.baseConstructor = baseClass;
    subClass.base = {};

    // 复制父类特性(属性与方法)
    baseClass.call(subClass.base);
}

/**
 * 构建对象系统
 */
function Mouse() { /* 测试用 */ }

function Animal(name) {
    this.name = name;
    this.say = function(message) {
        alert(this.name + ": " + message);
    };
}
```

```

    }
    this.eat = function() {
        this.say("Yum!");
    }
}

function Cat() {
    Cat.baseConstructor.call(this, 'cat');

    this.eat = function(food) {
        if (food instanceof Mouse)
            Cat.base.eat.call(this);
        else
            this.say("Yuk! I only eat mice - not " + food.name);
    }
}
extend(Cat, Animal);

function Lion() {
    Lion.baseConstructor.call(this, 'lion');
}
extend(Lion, Cat);

/**
 * 测试
 */
var cat = new Cat();
var lion = new Lion();
var mouse = new Mouse();
var unknowObj = {};

cat.eat(mouse);          // Yum!
cat.eat(unknowObj);      // Yuk!
lion.eat(mouse);         // Yum!

```

在这个示例里，我们看到猫喜欢吃老鼠，而由于我不知道狮子对老鼠是否感兴趣，因此只能认为它也喜欢——我的生物学学得显然有些问题。

这个例子的独特之处在于：我们用 `extend()` 来维护了一个 `baseConstructor` 成员，这个成员总是指向父类的构造器。而子类实例的构造逻辑就变成了：先向父类传入 `this` 引用以抄写父类方法，再向子类传入 `this` 引用以抄写子类方法（如 `Animal.say` 等方法）；后者覆盖前者中的同名成员（如 `Cat.eat` 方法）。由

于整个构造过程，都是在不断地从“类构造器”向“this 引用”抄写成员，所以我们将这种方法称为“类抄写”。

在这样的构建过程中，不但能通过修改 this 引用来添加、改写子类成员（例如在 Cat() 中重写 eat 方法），也可以通过抄写具有所有父代类的成员。通过这样的方法，我们可以用“类抄写”的方法来构建复杂的对象系统。

类抄写有两个问题。第一个问题是以内存开销换取效率。

仍以在“1.4.8.2 原型继承”讲述的“动物王国”系统为例。如果我们是在“类继承系统中”去使“动物能呼吸”，则应修改 Animal() 类声明，以使得该类具有“呼吸”这样的行为。这样一来，根据类继承的定义，由于 Animal 类具有了“呼吸”行为，所以所有的子类也就具有了这种行为。

方法一的含义、效果都与此类同：我们先写 Animal() 构造器，然后要做的事是改写 Animal() 构造器——或者说类——以添加或抄写它的成员。我们来试试这个法子：

```
// 类继承的实例：修改示例 1 的构造器声明部分
function Animal() {
    this.respire = function() {
        // 交换氧气与二氧化碳
    }
}

// (下略...)
```

很明显的，这段代码必须写在构造器中——就象其它面向对象语言要求写在类声明中一样。这个方法好不好呢？并不怎么好，因为这意味着每次创建这个 Animal 的实例时，都需要给实例初始化一个名为 respire 的方法。

不幸的是，在这种实现方法之下，在构造的多个 Animal 实例之间，它们的 respire 方法并不是同一个函数——这意味着更多的内存开销。下面的代码证明这一点：

```
// 示例：重写构造器的方法会导致实例持有不同的方法 (或函数)
var obj1 = Animal();
var obj2 = Animal();

// 显示值：false
alert(obj1.respire == obj2.respire);
```

这个示例表明：由于 `obj1.respire` 与 `obj2.respire` 指向不同的引用（换言之，是代码体相同的两个不同函数），所以增加了内存开销。

但这种方法也有好处。由于这种方法是通过不断修改实例（`this`）的成员来“构造”对象，因此所有的属性都在实例（`this`）的属性表中。进一步的推论是：访问任何成员都不必回溯原型链，因而效率更高。

类抄写的第二个问题，是系统并不维护原型继承链。因此在类抄写构建的系统中，我们不能使用 `instanceof` 运算来检测继承关系。对于这个问题，上述的 `extent()` 给出了一半答案：维护一个父类的实例 `base`。除此之外，用类抄写构建的系统还应当自行维护一个类继承树，并用特定的方法或函数来检测实例与类（构造器）的关系。

1.5.4.2. 原型继承

那么原型继承又存在什么问题呢？

除了我们前面提到的：

- 👉 在维护构造器引用 (`constructor`) 和外部原型链之间无法平衡，和
- 👉 没有提供调用父类方法的机制

之外，原型继承很显然是一个典型的、以时间换空间的解决方案。由于在子类中读写一个成员，而又无法直接存取到该成员（的名字）时，将会回溯原型链以查找该名字的成员，因此直接的结果是：继承层次中邻近的成员访问更快，而试图访问一个不存在的成员时耗时最久。

但我们来想想现实的对象系统。我们其实最希望基类、父代类等实现尽可能多的功能，我们也希望通过较多的继承层次来使得类的粒度变小以便于控制。从这里来看，访问更多的层次和访问父代类的成员是复杂对象系统的基本特性。而且，我们总是希望在继承树的叶子结点上做尽可能少的工作的。如果不是这样，我们就没有必要构建对象系统了。

但是 JavaScript 的原型继承的特性，显然与这种现实需求冲突。根本的原因在于，JavaScript 原本就是为了一种轻量级的、嵌入式的、WEB 浏览器端的脚本语言而设计的，这种应用环境决定了它的空间占用是关键，而时间消耗则相对次要得多（早期的浏览器端并不承担较多的逻辑）。

1.5.4.3. 如何选择继承的方式

综合类抄写与原型继承二者的基本特点，我们应该注意他们正好是互补的两种方案：

- ☞ 类抄写时成员访问效率更高，但内存占用较大；而原型继承反之。
- ☞ 类抄写不依赖内部原型链来维护继承关系，因此也不能通过 `instanceof` 来做这种检测；原型继承却可以在 `new` 运算时维护这种继承关系，也可以检测之。

除些之外，原型继承时的“写复制”机制也决定了我们不能单纯地依赖原型继承。对“写复制”机制有较深了解的读者应该知道：写复制机制在“引用类型”与“值类型”数据中表现并不一致。具体来说，就是复制引用时，所有实例都将指向同一个引用——从语义上来讲也的确应当是这样。但我们也会有这样的需求：实例成员指向基于同一类型的不同实例的引用。例如一个存放“线程池对象”的容器中，每个线程池就需要一个独立维护的池，而不能直接使用父代类中的某个池的相同引用。而由此带来的问题实质上是较为严重的，因为这意味着我们必须给原型继承保留一个构造过程，在这个过程中来初始化一些引用类型的成员，使得它们能够指向不同的引用。这实质上又走回了老路：使用类抄写过程，来为每个实例摹写某些引用类型的成员。

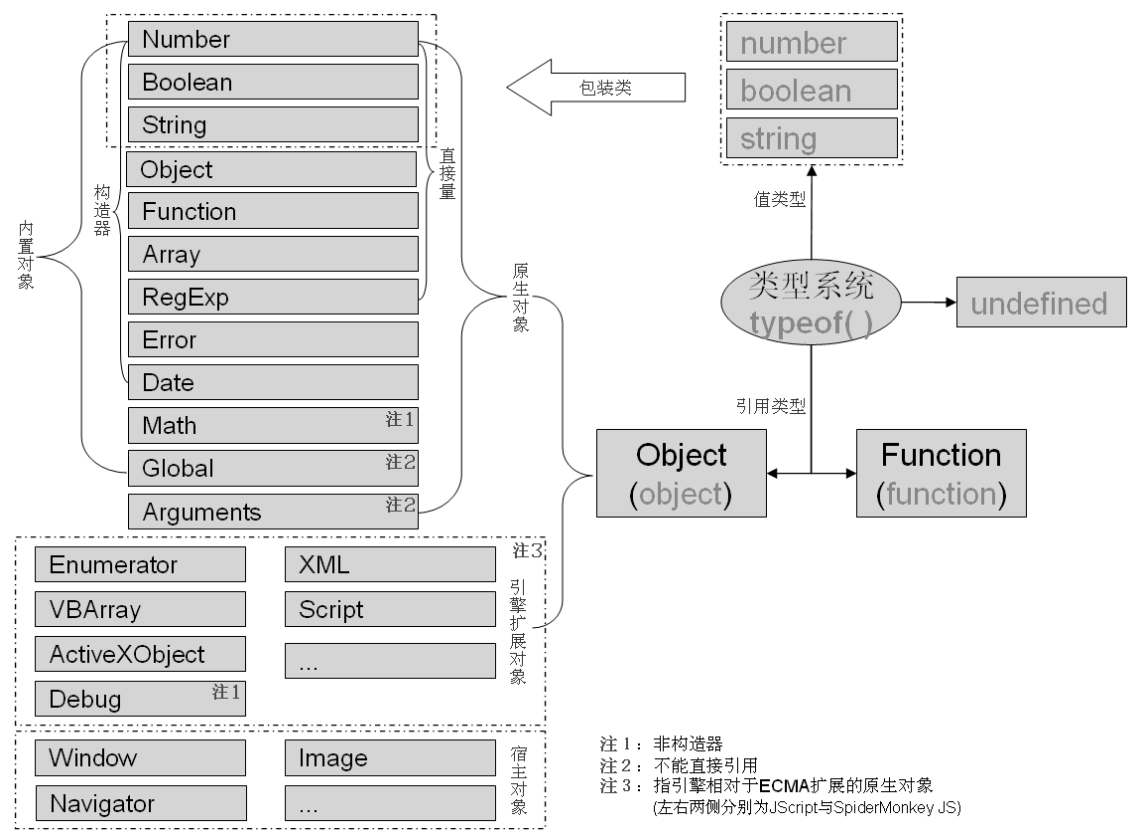
如今 JavaScript 应用的环境已经发生了非常多的改变，例如 Flash 中的 ActionScript、Windows 中的 WSH、Mozilla 中的 XUL / XBL，甚至在一些特殊的商用系统中我们也可以看到 JavaScript 来做控制语言（例如 Acrobat 和 Symantec 等公司的产品中对 JavaScript 的应用）。在这样的局面下，JavaScript 语言这种互补的特性产生了非凡的效用：一方面具有了构建大型对象系统的能力，另一方面也易于快速组织小功能构件（例如 Gadget）。

不过我们也应该注意到一个根本的问题：JavaScript 本身的优点也正是他的缺点。一方面，它能够组织大型对象系统，但又对大型对象系统中的封装和多态处理得不够，所以在大型应用（例如使用 ajax 技术的复杂的浏览器客户端）时常常缚手缚脚，心有余而力不足。另一方面，它能够组织小型的应用，但又因为“动态、函数式、原型继承”三方面的灵活性而带来了一种混杂的程序设计语言学知识体系，其结果是易学难精，而且是越深入底层则越容易感到混乱。

在继承方式的选择上，我认为仍然是应当择需而用：在大型系统上用类继承的思路，因而需要构建一种底层语言体系来扩展 JavaScript 的能力；在小型结构或者体系的局部使用原型继承的思路，因此应该更深入的学习 JavaScript 中不同语言的精髓。正是前者导致业界热推的 ajax 在底层都不可避免的有一些

对象系统的扩展机制（与 Qomo 项目所做的略同），而后者则正是我写这本书的基本动因。

1.5.5. JavaScript 中的对象（构造器）



ECMA 规范中说明，内置（Build-in）对象与原生（Native）对象的区别在于：前者总是在引擎初始化阶段就被创建好的对象，是后者的一个子集；而后者包括了一些在运行过程中动态创建的对象。按照该规范，JavaScript 的标准内置对象有 11 个。其中 7 个具备直接量声明的语法形式。Number、Boolean 与 String 的直接量形式声明将会定义值类型数据而非对象，值类型到对象类型可以通过包装类的形式来转换。这些相关的内容，在本书的以下章节中加以详述：

- 👉 1.2.1 变量的数据类型
- 👉 1.2.3 变量声明中的一般性问题
- 👉 1.5 包装类：面向对象的妥协

内置对象中，只以构造器形式存在，而没有直接量声明语法的只有 Error()

与 `Date()` 两个。本书中的下述章节对这两种内置对象有一些简单的介绍：

- 👉 1.4.5 流程控制：异常
- 👉 1.5.3 事件
- 👉 1.5.6 不能通过继承得到的效果
- 👉 1.7.3 出错处理
- 👉 1.7.6.3 其它类型的显式转换
- 👉 1.6 三天前是星期几？

`Math` 与 `Global` 对象存在一些特殊性。前者是一个对象而非构造器，因此不能被实例化；后者则在引擎初始时就被实例化一个单例，并作为全局可访问的对象使用，因此不能直接引用 `global/Global` 标识符。本书中没有地方讨论到 `Math` 对象，至于 `Global` 对象，也仅仅在“1.7.4.5 值类型数据的显式转换”章节中讨论到它的 `parseInt()` 和 `parseFloat()` 方法。需要补充说明的是，如果需要访问 `Global` 对象的方法，可以通过宿主对象——在浏览器环境中是 `window`——来实现，例如使用 `window.parseInt()` 来调用 `Global.parseInt()` 方法，二者的含义是一致的。

尽管除了 `Arguments` 之外，ECMA 规范中也还存在一些其它的原生对象（总的来说原生对象是包含内置对象的），但 `Arguments` 的确是这其中最为重要的一个。`Arguments` 对象是有关函数调用的一个对象，它总是由引擎在函数调用时动态创建并添加在函数闭包中。在：

- 👉 第四章 JavaScript 的函数式语言特性
- 👉 1.2 动态执行 (`eval`)
- 👉 1.3 动态方法调用 (`call` 与 `apply`)
- 👉 第七章 一般性的动态函数式语言技巧

中有大量的实例来讲述该对象的特点与应用。

引擎扩展对象是一个并不太大的集合，一般来说也比较确定，它们也属于引擎的原生对象（但不属于 ECMA 规范的原生对象）。在 JScript 中，`Enumerator`, `VBArray`, `ActiveXObject` 三个对象都与 COM/ActiveX 系统有关。其中，`VBArray` 与一种 COM 框架中的名为 `SafeArray` 的数组有关（VBScript 中的数组都是 `SafeArray`，所以这里的名字用成了 `VBArray`，看起来象是 JScript 与 VBScript 交换数据的一个中间对象）；`Debug` 对象仅只在 JScript 代码处理调试环境中有效，它不是构造器因此不能创建实例。在 SpiderMonkey JavaScript 中扩展了 `Script()` 和 `XML()` 两个对象，前者用于将一段脚本代码解析、编译成为一个全局

中的函数，后者仅实现于 SpiderMonkey JavaScript 1.6 以上版本，是基于一个名为 E4X(JavaScript for XML)的规范而实现的、能在 JavaScript 中直接支持 XML 数据的技术^①。除了 JScript 与 SpiderMonkey JavaScript 之外，其它引擎也有相对独立的对象扩展。这些不同引擎独自实现的扩展对象，一般不会被其它引擎接受，在移植中存在明显的兼容问题，因此建议慎重使用。

最后一类是宿主对象，宿主对象不是引擎的原生对象，而是由宿主框架通过某种机制注册到 JavaScript 引擎中的对象。这一类对象极为丰富，例如我们常见的 DOM 对象模型，以及浏览器框架（Window、Navigator 等）。宿主对象可能只是一个在宿主初始化引擎的过程中创建好的对象（例如浏览器中的 navigator），也可能是一个在宿主运行过程中才会创建出来的对象（例如 window、window.document 等）。宿主对象可能只是（动态创建 / 产生的）对象实例，也可能是宿主提供给用户代码使用的构造器函数（例如 Image()、XMLHttpRequest()等）。

有趣的是，一些宿主会把自己提供的对象 / 构造器也称为“原生对象”。例如 Internet Explorer 7 就把它提供的 XMLHttpRequest()称为原生的，与此相对的是在它的更早先版本中通过“new ActiveXObject('Microsoft.XMLHTTP')”这样的方法创建的对象。这种情况下，读者应注意到“宿主的原生对象”与“引擎的原生对象”之间的差异。

1.5.6. 不能通过继承得到的效果

一些 JavaScript 内置对象具有在对象系统的封装、多态与继承性之外的特殊效果，这些效果包括^②：

对象	特殊效果	备注
Number	包装类	值类型与它的包装类之间的关系是在引擎内部设定的，既不可能通过重写构造器（的标识符）来替代这种效果，也不能通过继承来使得这些包装类的子类继承这种效果。
Boolean		
String		
Array	自动维护的 length 属性	作为索引数组使用时，引擎为该对象隐式地维护 length 属性和（可能连续的）下标。Array 对象的某些方法作用在普通对象上时，也会

^① 与此类似的是 Microsoft 在 Internet Explorer 中实现的 XML DataLand——该技术用于在 HTML 页面中包含 XML 数据且支持 JavaScript 直接访问。但 MS XML DataLand 是在 HTML 层面实现的一种数据表示技术，而 E4X 是在 JavaScript 层面实现的一种可编程数据对象技术，二者存在本质上的差异。

^② 这里只讨论了 ECMA 标准中的原生对象，但同样的问题也会出现在引擎扩展的对象和宿主对象中。例如 JScript 中的 ActiveXObject() 具有从 COM 类库中创建对象的能力，而 DOM 对象模型中的 Image() 具有预装载图像的能力等等，这些特殊效果在继承过程中也会丢失。

		有维护 <code>length</code> 属性的效果。部分内容参见“1.6 关联数组：对象与数组的动态特性”。
Date	日期对象的相关运算	日期对象的方法(例如格式转换)仅能作用在由 <code>Date()</code> 构造器产生的对象实例上。
Function	可执行	
RegExp	可执行	(*)
Error	(无)	(**)
Object	(无)	

(*)仅在 SpiderMonkey JavaScript 中支持，参见“1.5.4.2 正则表达式特例”。

(**)也许你认为 `Error()` 对象具有“可被 `catch` 语法捕获”的特殊效果，但事实上不是。因为 `Error()` 对象没有任何特殊性，任何对象都可以由 `throw` 语句作为异常抛出，并被 `catch` 子句捕获。

表中列出的所谓“特殊效果”是不能通过继承得到的。举例来说，`Function` 函数的“可执行效果”是指类似如下的代码中：

```
func = new Function();
```

所得到的 `func` 对象“是一个可执行的函数”的效果。

接下来，按照对象系统的概念，我们有两种方法来扩展基于 `Function()` 的对象系统。其一是通过继承得到 `Function()` 的一个子类。例如：

```
function MyFunction() {  
}  
MyFunction.prototype = new Function();  
  
// 输出 true, 表明 myFunc 是一个函数对象  
myFunc = new MyFunction();  
alert( myFunc instanceof Function );
```

其二是直接指定构造器的原型，以使得不同的构造器使用相同的原型——本质上它们是同一个原型的不同引用，但由于没有使用 `new` 运算，因此不是存在子类和继承关系。例如：

```
function MyFunction() {  
}  
MyFunction.prototype = Function.prototype;  
...
```

上述的方法（也包括更多的方法^①）中，我们总能检测到 `myFunc` 是 `Function()` 的一个“子类的或使用相同原型构造的”实例，但是我们无论如何

^① 示例中使用了 `Function.prototype` 这个原型，事实上任何对象都可以用作原型，因此“`MyFunction.prototype = Function`”这种写法也是行得通的。

都不能使 `MyFunction` “得到”或“继承到”函数的可运行效果——如果继续考查上面这个例子，我们会发现得到的 `myFunc` 是不能执行的。下面的代码，就将触发一个由于 `myFunc` 对象“不是函数”而引出的异常：

```
// (续上例，) 尝试执行 myFunc，导致异常
myFunc();
```

所以这里说 `myFunc` 对象“不是函数”，即是指它的父类 `MyFunction()` 虽然从 `Function()` 中继承了“函数对象的所有属性”——这是完全符合对象继承的语义的，但不能继承它“可执行”的效果。

正因为内置对象 / 类的特殊效果不被对象系统继承，所以我们这里得到的 `myFunc` 只是一个“函数对象”，而不是一个“可执行的函数”。同理，表中列出的各种效果都不能通过继承或扩展对象系统来得到：一方面，这些效果被引擎绑定在特定的构造器上，而不是它们的原型上；另一方面，对象系统只负责维护内部原型链，以确保 `instanceof` 运算能正确检测这种关系，而不负责这些特定效果的实现与传递。

所以“派生引擎原生对象”的想法在 JavaScript 中变得不切实际。唯一可行的、安全的子类派生，是对那些没有任何特殊效果的对象做出的，例如 `Object()`、`Error()`，以及用户自己声明的构造器函数。

1.6.[XXX]综述

运算优先级

全局对象(全局闭包问题在函数式语言特性中讲)

代码的规范依赖于语法作用域（代码分块），而逻辑的灵活性则来源于为不同的语法作用域设计的“特殊的 GOTO”。这些“特殊的 GOTO”带来的不同层次的突破能力，构成了在不同代码分块中对“顺序执行”流程的纵向穿插。

在本章中我们严格地限制了对 JavaScript 结构化编程和面向对象编程方面的特性——我们仅仅讲了与这两种编程范型相关的那些特性。因此，从本章来看，JavaScript 一种对这两种范型都实现得不完整、不充分的语言，也不具有多少“有趣的”特性。综合这些与其它通用语言不同特性，就主要只有两点：

没有语句和代码块以上的语法单位，模块化的程序不高；

基本数据类型中除数值和布尔值外，其它的数据类型都没有存储含义；

类是采用原型继承方式来实现，而没有类继承结构

第四章 JavaScript 的函数式语言特性

函数式程序设计始于 Lisp。Lisp 的程序和数据都用表来表示，甚至这种语言的名字本身也就是“表处理系统 (List Processor)”的缩写形式。著名的函数式语言 Scheme 是 Lisp 的一种方言。

——《程序设计语言概念和结构》，Ravi Sethi。

1.1. 概述

在一些语言中，连续运算被认为是不良好的编程习惯。我们被要求运算出一个结果值，先放到中间变量中，然后拿中间变量继续参与运算。

这两个原因，一方面容易形成良好的代码风格。这个原因被阐释得非常多。例如我们被教育说，不应该下面这样写代码：

1.1.1. 从代码风格说起

在一些语言中，连续运算被认为是不良好的编程习惯。我们被要求运算出一个结果值，先放到中间变量中，然后拿中间变量继续参与运算。

这两个原因，一方面容易形成良好的代码风格。这个原因被阐释得非常多。例如我们被教育说，不应该下面这样写代码：

```
child = (!LC && !RC) ? 0 : (!LC ? RC : LC);
```

而应该把它写成下面这样：

```
if (LC == 0 && RC == 0)
    child = 0;
else if (LC == 0)
    child = RC;
else
    child = LC;
```

我承认我们应该写更良好风格的代码，我也曾经深受自己代码风格不良之苦并幡然醒悟。但是上面这个问题的本质，真的是“追求更漂亮的代码格式化风格(style)”吗？

例如我曾经有一个困扰，就是如何写 LISP/Scheme 的代码，“才会有更良

好的风格”？下面这段代码是一段 LISP 语言的示例：

```
; LISP Example function
(DEFUN equal_lists ( lis1 lis2)
  (COND
    ((ATOM lis1) (EQ lis1 lis2))
    ((ATOM lis2) NIL)
    ((equal_lists (CAR lis1) (CAR lis2))
     (equal_lists (CDR lis1) (CDR lis2)))
    (T NIL)
  )
)
```

答案是：没有比上面这个示例更良好的 LISP 语言风格了（当然，你愿意用四个空格替换两个空格，或者把括号写在一行的后面之类，是一种习惯而非“更良好风格”的必要前提）。

因此除了对风格的苛求之外，在另一个方面，“不支持连续运算”这种编程习惯（和代码风格）其实是为了更加符合冯·诺依曼的计算机体系的设计。看来，语言环境是风格限定与编程习惯形成的重要前提之一：对于一种语言来说，某种风格可能是非常漂亮的，但对于另一种来说，可能根本就无法实现这种风格。假如从过程式代码的风格来看上面这样的代码，那么除了还存有缩进之外，几乎毫无美观之处。

然而，事实上只有这种风格才能满足函数式语言的特性设定——因此问题的根源并不在于“代码是否更加漂亮”，而是 LISP——这种函数式语言——本身的某些特性需要这种“复杂的、非结构化的”代码风格。

那么这些影响到函数式语言的代码风格的特性是什么呢？

1.1.2. 为什么常见的语言不赞同连续求值

我们现在再来说连续运算求值。在一般的程序设计观念中，我们应该象下面这样写代码：

```
var desktop = new Desktop();
var chair1 = new Chair();
var chair2 = new Chair();
var me = new Man();

var myHome = new Home();
```

```
myHome.concat(desktop);
myHome.concat(chair1);
myHome.concat(chair2);
myHome.concat(me);
myHome.show(room);
```

看看，我们费尽心力，我们才创建了一个有桌子、椅子和人的房子，并进而有了个家，但这个家的简陋条件，实在是比监狱还差。然而我们已经付出了如此多的代码（还不包括那些类的声明与实现），因此我们如果要创建一个更加漂亮而有生气的家，上面这样的代码我们得写很多年。

为什么我们要这样写代码呢？因为我们从面向过程、面向对象一路走来，根本上就是在冯·诺依曼的体系上发展。在这个体系上，我们首先就被告知：运算数要先放到寄存器里，然后才能参与 CPU 运算。

于是我们得到了结论，汇编语言应该这样写：

```
MOV EAX, [0040123D]
MOV EDX, 123
CALL Ri_CreateProcesss
```

接下来，我们就看到过程式语言这样写：

```
var
    value_1 : integer;
    value_2 : integer;

begin
    value_1 := 100;
    value_2 := 1000;
    writeln(value_1 * value_2);
end.
```

然后，我们就看到了面向对象的语言应该这样写：

```
var
    value_1 : TIntegerClass;
    value_2 : TIntegerClass;

var
    calc : TCalculator;

begin
    calc := TCalculator.Create();
```

```
value_1 := TIntegerClass.Create(100);  
value_1 := TIntegerClass.Create(1000);  
  
calc.calc(value_1, value_2);  
calc.show();  
end.
```

在冯·依曼体系下，我们就是这样做事的。所以在《程序设计语言实践之路》这本书中，将面向对象与面向过程都归类为“命令式”语言，着实不妥。

1.1.3. 函数式语言的渊源

函数式语言却并非这样。函数式语言的特征之一就是连续运算，运算一个输入，产生一个输出，输出的结果即是下一个运算的输入。并不需要中间变量来“寄存”。从理论上来说：函数式语言不需要寄存器，或变量赋值。

然而为什么“连续求值”会成为函数式语言的基本特性呢？

要了解这一特性的实质，我们需要更远地回溯“函数式”语言的起源。我们得先回答这个问题：这种语言是如何产生的呢？

在 1930 年前后，在第一台电子计算机还没有问世之前，有四个著名的人物展开了对形式化的运算系统的研究。他们力图通过这种所谓的“形式系统”，来证明一个重要的命题：可以用简单的数学法则表达现实系统。这四个人分别是阿兰·图灵、约翰·冯·诺依曼、库尔特·哥德尔和阿隆左·丘奇。

图灵提出了现在称为“图灵机”的形式系统。后来人们认为，图灵机概念中最重要的是解决了 0、1 运算和 0、1 存储的问题。对于后者，即是说要求计算机具有存储系统(例如内存、硬盘等等)。约十年之后，冯·诺依曼成功地实现了这样的计算机系统 ENIAC，约定了计算机的五大部件：运算器 CA、逻辑控制器 CC、存储器 M、输入装置 I 和输出装置 O。其中，运算器基于的理论是 0、1 运算，而存储器 M 和输入输出装置 IO 则依赖于 0、1 存储。

电子计算机的历史存在很多的争议。这些争议包括：

这里要阐述的结论是：图灵机以及后来的冯·诺依曼体系的计算机系统都依赖于存储（这里指内存）进行运算。后来有人简单的归结这样的运算系统：通过修改内存来反映运算的结果。

然而，我们应用计算机的目的，是进行运算并产生结果。所以其实运算才是本质，而“修改内存”只不过是这种运算规则的“副作用”，或者说是“表现运算效果的一种手段”。

而阿隆左·丘奇提出的，就是一种更加接近于“运算才是本质”的理论，这是一种被称为 `lambda` 演算的形式系统。这个系统本质上是一种虚拟的机器的编程语言，他的基础是一些以函数为参数和返回值的函数。注意，我们在这里一定要强调“基础是一些‘以函数为参数和返回值’的函数”这一特性。

这种运算模式却一直没有被实现。大约到冯·诺依曼等人完成了 EDVAC 的十年之后，一位 MIT 的教授 John McCarthy（也是普林斯顿毕业生）对阿隆左·丘奇的工作产生了兴趣。在 1958 年，他公开了表处理语言 LISP。这个 LISP 语言就是对阿隆左·丘奇的 `lambda` 演算的实现。

但是，这时的 LISP 工作在冯·诺依曼计算机上！——很明显，这时只有这样的计算机系统——更加准确地说，LISP 系统当时是作为 IBM 704 机器上的一种解释器而出现的。

所以从函数式语言的鼻祖——LISP 开始，函数式语言就是运行在解释环境而非编译环境中的。而究其根源，还是在于冯·诺依曼体系的计算机系统是基于存储与指令系统的，而并不是基于运算的。

函数式语言强调运算过程，这也依赖于运行该系统的平台的运算特性。由于我们的确是将计算机设计成了冯·诺依曼的体系，所以在过去的很长的时间里，你看不到一个计算机(硬件)系统宣称在机器指令级别上支持了函数式语言。直到 1973 年，MIT 人工智能实验室的一组程序员开发了被称为“LISP 机器”的硬件。

阿隆左·丘奇的 `lambda` 演算终于得以硬件实现！

现在让我们回到最初的话题：为什么可以将语言分成命令式和运算式语言？是的，从语言学分类来说，这是两种不同类型的计算范型；从硬件系统来说，它们依赖于各自不同的计算机系统。

然而现在我们每个人手中的电脑毕竟都不是名为“LISP 机器”的硬件——支持大量“运算函数”的 RISC(复杂指令集)已经失败了，精简指令集带来了更少的指令和更确切的运算法则：放到寄存器里，然后再交由 CPU 运算。我们不能寄期望一种基于 A 范型实现的计算机系统同时支持 B 范型。换言之，不

能指望在 X86 指令集中出现适宜于 `lambda` 演算的指令、逻辑或者物理设计。

于是当前的现实变成了这样：我们(大多数人)都在使用基于冯·诺依曼体系的命令式语言，但为了获得特别的计算能力或者编程特性，这些语言也在逻辑层来虚拟一种适宜于函数式语言范型的环境。这一方面产生了类似于 JavaScript 这样的多范型语言，另一方面则产生了类似于 Python 的以函数式运算为主的虚拟机环境。

1.2. 函数式语言中的函数

并不是一个语言支持函数，这个语言就可以叫做“函数式语言”。函数式语言中的“函数 (function)”除了能被调用之外，还具有一些其它的性质。这包括：

- 👍 函数是运算元
- 👍 在函数内保存数据
- 👍 函数内的运算对函数外无副作用

我们下面分述函数在 JavaScript 中的这三种特性。

1.2.1. 函数是运算元

大多数语言都支持将函数作为运算元参与运算。不过由于对函数的理解不同，因此它们的运算效果也不一样。例如在 C、Pascal 这些命令式语言中，函数是一个指针，对函数指针的运算可以包括赋值、调用和地址运算。其中地址运算带来了“内存访问违例”的隐患。由于这种情况下函数被理解为指针，因此也可以作为函数参数进行传值（地址值），比较常见的情况是函数 A 的声明中，允许传出一个回调函数 B 的指针。下例是这样一个 Win32 API 的声明：

```
/**
 * Pascal 语言声明的 EnumWindows ()
 */
function EnumWindows(lpEnumFunc: EnumWindowsProc;
    lParam: LPARAM): BOOL; stdcall
/**
 * C 语言声明的 EnumWindows ()
 */
BOOL EnumWindows(WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

由于这里的 `lpEnumFunc` 将传入一个地址指针，这个地址显然可能来自另一个进程空间，或者当前进程无效的存储地址。因此这种函数调用过程中，以地址值为数据的参数传递，大大增加了系统的风险。

在 JavaScript 中，函数也是运算元，但它的运算只有调用。由于彻底地杜绝了地址运算，因此也就没了上述的隐患。更为重要的是，在 JavaScript 中函数只有运算元的含义而没有地址含义，因此函数入口参数表中的“函数参数”就与普通参数没有什么不同了。

1.2.2. 在函数内保存数据

函数式语言的函数可以保存内部数据的状态。在某些命令式语言中也有类似的性质，但与函数式语言存在根本不同。以（编译型、X86 平台上的）命令式语言来说，由于代码总是在代码段中执行，而代码段不可写，因此函数中的数据只能是静态数据。这种特性通常与编译器或者某些特定算法的专用数据绑定在一起（例如跳转表）。

除了这种情况之外，在命令式语言中，函数内部的私有变量（局部变量）是不能被保存的。从程序执行的方式来讲，局部变量在栈上分配，当函数执行结束后，所占用的栈被释放。因此函数内的数据不可能被保存。

在 JavaScript 的函数中，函数内的私有变量可以被修改，而且当再次“进入”到该函数内部时，这个被修改的状态仍将持续。下面的例子中，函数 `set_value()` 用于修改值，函数 `get_value` 用于获取值，我们将这两个函数置入到 `MyFunc()` 内部，用来考察该函数内部的数据状态：

```
var set, get;

function MyFunc() {
    // 初值
    var value = 100;

    // 内部的函数，用于该问 MyValue
    function set_value(v) {
        value = v;
    }
    function get_value() {
```

```

    return value;
}

// 将内部函数公布到全局
set = set_value;
get = get_value;
}

// 测试一
// 显示输出值: 100
MyFunc();
alert( get() );

// 测试二
// 显示输出值: 300
set(300);
alert( get() );

```

测试一表明：函数 `MyFunc()` 在执行结束后，内部数据值仍保持它的状态。而测试二则表明：`set_value()` 将影响到内部数据，这种影响后的状态也被保持。

在函数内保持数据的特性被称为“闭包(Closure)”，我们将在“.....”中更详细地讨论它。不过显而易见的好处是，如果一个数据能够在函数内持续保存，那么该函数（作为构造器时）产生的实例的多个方法就可以使用该数据进行运算；而在多个实例间，由于数据存在不同的闭包中，因此不会产生相互影响——以面向对象的术语来解释，就是说不同的实例拥有各自的私有数据（复制自某个公共的数据），多个实例之间不存在可共享的类成员。下例说明这个特性：

```

function MyObject() {
    var value = 100;
    this.setValue = function(v) {
        value = v;
    }
    this.showValue = function() {
        alert(value);
    }
}

var
    obj1 = new MyObject();
    obj2 = new MyObject();

// 测试: obj2 的置值不会影响到 obj1

```

```
// 显示结果值: 100;  
obj2.setValue(300);  
obj1.showValue();
```

1.2.3. 函数内的运算对函数外无副作用

这是函数式语言应当达到的一种特性。然而在 JavaScript 中这项特性只能通过开发人员的编程习惯来保证。

所谓对函数外无副作用，含义在于：

- 👉 函数使用入口参数进行运算，而不修改它（作为值参数而不是变量参数使用）；
- 👉 在运算过程中不会修改函数外部的其它数据的值（例如全部变量）；
- 👉 运算结束后通过函数返回向外部系统传值。

这样的函数在运算过程中对外部系统是无副作用的。然而我们注意到 JavaScript 允许在函数内部引用和修改全局变量，甚至可以声明全局变量。这一点其实是破坏它的函数式特性的。除此之外，JavaScript 也允许在函数内修改对象和数组的成员。这使得函数并不是仅仅通过它的返回值来影响系统，因此也不是正确的函数式特性。

所以在 JavaScript 中，只能通过开发人员的习惯来实现这一特性。这包括两点：不修改全局变量，以及不在函数内修改对象和数组成员。

当把“不在函数内修改对象成员”这个原则，与面向对象系统的另一个特性结合起来的时候，系统的稳定性就大大地增强了。这个特性就是通过接口（Interface）向暴露系统，以及通过读写器(get&setter)访问对象属性(attribute)。由于在这种对象系统中，对象向外部系统展现的都是接口方法（以及读写器方法），从而有效地避免了外部系统“直接修改对象成员”。

在这里补充面向对象系统的这一特性，是强调函数式中的“函数”所要求的“无副作用”这个特性，其实可以与面向对象系统很好地结合起来。二者并不矛盾，在编程习惯上也并非格格不入。

1.3. 从运算式语言到函数式语言

现在让我们回到最开始的话题：为什么“连续求值”会成为函数式语言的基本特性呢？这是因为函数式语言是基于对 lambda 演算的实现而产生的，其理论基础就是：

👉 （表达式）运算产生结果；

👉 结果（值）用于更进一步的运算。

至于从 LISP 开始引用的“函数”这个概念，其实在演算过程中只有“结果（值）”的价值：它是一组运算的封装，产生的效果是返回一个可供后续运算的值。

到了这里，我们应该认识到函数式语言中，“函数”并不是真正的精髓。真正的精髓在于“运算”，而函数只是封装“运算”的一种手段。

到了这里，我们如果“假设系统的结果只是一个值”，那么“我们必然可以通过一系列连续的运算来得到这个值”。不过，这句话要分成两半来看，我们下面先讲“连续运算”，然后再来讨论“结果只是一个值”的问题。

1.3.1. JavaScript 中的几种连续运算

1.3.1.1. 连续赋值

在 JavaScript 中，一种常见的情况就是连续赋值：

```
var a = b = c = d = 100;
```

我们把它写成下面这种格式，可能会让人更能理解(我并不是想说明这种语法风格更好。当然如果你想要为每个变量做注释，可能这是个不错的主意)：

```
1   var a =  
2       b =  
3       c =  
4       d = 100;
```

第四行的表达式“d=100”被首先运算。因为表达式有返回值，所以得到了运算结果“100”。接下来，该值参与下一个赋值表达式运算，变成了“c = 100”。如此类推，我们得到了连续赋值的效果。

所以，在别的某些语言中（例如 Pascal），连续赋值可能是一种“新奇的语法特性”，但在 JavaScript 语言中，它不过是一种连续表达式运算的效果。

1.3.1.2. 三元表达式的连用

我们前面提到过三元表达式（?:），这个表达式在 C 语言里是一种并不非

常推荐的语句。因此对于刚才那个例子：

```
child = (!LC && !RC) ? 0 : (!LC ? RC : LC);
```

总有人会在书籍中故意弱化或者丑化这个表达式的表现^①。其实开始的那段代码，一般人并不会那样写，即使要写成三元表达式，也应该如下：

```
child = (LC || RC) ? ( LC ? LC : RC ) : 0;
```

在函数式里，三元表达式其实不但应该使用，甚至被推荐连用。因为因为这样能够充分发挥连续运算的特性：

```
1  var objType = getFromInput();
2  var cls = ((objType == 'String') ? String :
3            (objType == 'Array') ? Array :
4            (objType == 'Number') ? Number :
5            (objType == 'Boolean') ? Boolean :
6            (objType == 'RegExp') ? RegExp :
7            Object
8  );
9  var obj = new cls();
```

在这个例子里，我们也可以看到一种良好的代码书写风格(至于第 8 行的括号是放在第 7 行的最末或新起一行，可以看成一种习惯)。但我们充分利用了表达式求值的特性。2~6 行的每个三元表达式的第三个运算元，其实都是下一行运算的返回结果。

显然，“运算产生值并参与运算”这一特性，使得上述的代码成为可能。否则，你可能需要写下面这样的代码（当然，你可能现在仍旧认为下面这样的代码更漂亮）：

```
var objType = getFromInput();

switch (objType) {
  case 'String': {
    obj = new String();
    break;
  }

  case 'Number': {
    // ...
  }
}
```

^① 例如某书某书。

```

    }

    // ...
    default: {
        obj = new Object();
    }
}

```

一部分理解了面向对象编程的“多态性”的开发人员可能会主张下面的代码：

```

var cls;
var objType = getFromInput();

switch (objType) {
    case 'String': {
        cls = String;
        break;
    }

    case 'Number': {
        // ...
    }

    // ...
    default: {
        cls = Object;
    }
}

var obj = new cls();

```

熟悉模式的开发人员则不慌不慢地提出他们的观点：

```

// ...
// (对于不同的语言，以上省略 10~50 行类工厂的实现代码)

var objType = getFromInput();
var fac = new Factory();
var cls = fac.getClass(objType);
var obj = new cls();

```

我们不必去评述这几种代码实现的优劣。就如同代码风格一样，在不同的体系之下，存在不同的评判标准。但现在的问题是：你在使用一门函数式语言，然而这些代码利用了函数式语言的特性了吗？

1.3.1.3. 一些运算连用

在前面这个三元表达式中的例子中，我们说明了行代码：

```
child = (!LC && !RC) ? 0 : (!LC ? RC : LC);
```

至少可以被改成如下的形式：

```
child = (LC || RC) ? ( LC ? LC : RC ) : 0;
```

然后我们又说，可以通过更好的代码格式化，来使得三元表达式连用的代码可读性得到提升。例如：

```
child = !( LC || RC ) ? 0
      : ( LC ? LC
        : RC );
```

但事实上在 JavaScript 中，一些语法约定基本不需要用户开发人员写上面这样的代码。正如这个例子，我们无非是想要得到 LC、RC 和 0 值之一。这可以借助连续的逻辑或 “||” 运算来得到。上面的代码等效于：

```
child = LC || RC || 0;
```

这行代码中，等号右边的表达式的意思是说：

- 👉 如果 LC 能被转换成逻辑 “true” (值为真)，则运算返回 LC 的实际值；否则，
- 👉 如果 RC 值为真，则返回 RC 的实际值；否则，
- 👉
- 👉 直到表达式结束，返回表达式最后一个运算元的实际值。

而这样的运算结果，正是我们需要得到的 child 值。

这段代码其实也正好说明了 JavaScript 中逻辑运算的实质——逻辑或运算并非为布尔值而专设。按照上面的规则，对布尔值进行 “或(||)” 运算的效果，只是一个特例而已。

1.3.1.4. 函数与方法的调用

我们上面举了一个不非常恰当的例子。这是因为孤立来看，我们要得到到一个对象的类类型，并不需要用那样复杂的三元表达式，**用类厂可能的确是不错的主意**。但我说那是一个孤立的问题。因为我们忽视了另外一项 JavaScript 特性：对象的构造、函数与方法的调用等等，本质上是都表达式运算，而非语句。

举例来说，我们可以用下面的代码完成对象的构造：

```
var obj = new ( (obj=='String') ? String : Object );
```

这行代码是用一个运算来作为 new 运算的入口参数——注意 new 不是语句的语法关键字，而是运算符。

所以在 JavaScript 中，我们事实上可以用下面的连续运算来完成上一小节的示例：

```
var obj = new (
  (objType == 'String') ? String :
  (objType == 'Array') ? Array :
  (objType == 'Number') ? Number :
  (objType == 'Boolean') ? Boolean :
  (objType == 'RegExp') ? RegExp :
  Object
);
```

连续的一组三元表达式的运算结果，成为了 new 运算符的输入，而 new 运算符后面的一对括号 “()”，在这里的起到的是强制运算符的作用。

更进一步的说，new 运算的结果(构造一个对象实例)，被作为赋值运算的运算元，然后赋给了变量 obj。

接下来的代码将会更加有趣。让我们对上面的代码做一小点点的修改：

```
1 alert(
2   (new (
3     (objType == 'String') ? String :
4     (objType == 'Array') ? Array :
5     (objType == 'Number') ? Number :
6     (objType == 'Boolean') ? Boolean :
```

```
7      (objType == 'RegExp') ? RegExp :
8      Object
9    )
10   ).toString()
11 )
```

是的。我故意将代码分隔成这个样子。这样可以使你更清楚地看到运算的层次。我们看到 `new` 运算在第 9 行得到了运算结果：一个对象实例。然后它被一对强制运算符给包括了起来(第 2~10 行)^①。强制运算的结果还是返回该实例，——我们在这里只是需要取得一个语法上清晰的效果——而该实例接下来就调用了一下方法 `toString()`。

如果方法调用不是表达式而是语句，那么上面这样的代码就不可能被写出来。所以，第 10 行代码的实质是，刚被创建的对象实例：

- 👉 先通过点运算符 “.” 进行了一次对象属性 `toString` 存取，
- 👉 然后通过运算符 “()” 进行了一次方法调用。

这两次运算的结果，返回了对象的 **序列化(流化、持久化)** 值。接下来这个值被送入 `alert()` 函数的入口(第 1~11 行代码)，最终显示输出。

1.3.2. 运算式语言

在本节中，我们将讨论一种新的编程范型：运算式语言。它满足说明式语言的两个特性：一是陈述运算，二是求值。

不同的运算式语言的编程能力是不同的，在本节中我们将例举两个这种类型的语言。需要注意的是，它们一开始时并不是以一个语言范型出现的——而更像是某个体系中的小功能而已。

1.3.2.1. 运算的实质，是值运算

将“值运算”换个说法，就是“求值”。如果说“运算的实质，就是求值”，那么大家会觉得顺理成章。但是，这里的“值”如果是指“值类型”的数据呢？

在这个设问中，我们其实已经向程序设计语言的本质走得更近了一步。为了说明这一点，我们先来考察一下 JavaScript 的各种运算的结果类型。下表对

^① 这里的强制运算是不必须的。因为 “.” 运算的优先级低于 “new” 运算，所以这里并不存在歧义。本例中使用了加强制运算，只是为了更好的表示运算次序。另外，事实上 “.” 具有最低的优先级次序。

此作出了完整的分析：

分类	名称	符号	说明	运算元	目标
计算运算	加法	+	将两个数相加	number	number
	减法	-	对两个表达式执行减法操作		
	乘法	*	将两个数相乘		
	除法	/	将两个数相除并返回一个数值结果		
	取余	%	将两个数相除，并返回余数		
	递增	++	给变量加一		
	递减法	--	将变量减一		
	一元正值	+			
	一元取反	-	表示一个数值表达式的相反数		
按位运算	按位与	&	对两个表达式执行按位与操作		
	按位左移	<<	将一个表达式的各位向左移		
	按位非	~	对一个表达式执行按位取非（求非）操作		
	按位或		对两个表达式指定按位或操作		
	按位右移	>>	将一个表达式的各位向右移，保持符号不变		
	按位异或	^	对两个表达式执行按位异或操作		
	无符号右移	>>>	在表达式中对各位进行无符号右移		
逻辑运算	逻辑与	&&	对两个表达式执行逻辑与操作	boolean	boolean
	逻辑非	!	对表达式执行逻辑非操作		
	逻辑或		对两个表达式执行逻辑或操作		
字符运算	连接	+	连接字符或字符串	string	string

分类	名称	符号	说明	运算元	目标
函数调用	函数调用	()	调用函数并返回结果值	function	*注 1
比较运算	比较	(比较运算)	返回比较结果	(任意)	boolean
赋值运算	赋值	=	将一个值赋给变量	(任意)	*注 1
	复合赋值	(赋值运算)	运算并将结果值赋给变量	(值类型)	(值类型)
对象	对象构造	new	创建一个新对象	function	object
	对象检查	instanceof	返回一个 Boolean 值, 表明某个对象是否为特定类的一个实例	object	boolean
	成员存取	[]或.	存取对象的成员	object (string)	*注 1
	成员删除	delete	删除对象的属性, 或删除数组中的一个元素	(标识符)	boolean
	成员检查	in	检查一个对象成员是否存在	object (string)	boolean
其它运算	typeof	typeof	返回运算元的数据类型的字符串	(任意)	string
表达式逻辑	三元条件 (*)	?:	根据条件执行两个表达式之一	boolean (表达式)	*注 2
	优先级	()	包含 JScript 运算符的执行优先级信息的列表	(表达式)	
	逗号	,	使两个表达式连续执行		
	void	void	避免一个表达式返回值		undefined

注 1: 取决于具体的值、变量或对象成员的数据类型。

注 2: 是表达式之间的运算关系, 结果只对表达式产生影响。

在上面这个表格中, 最令人惊讶的结论是: 所有的运算都产生“值类型”的结果值^①。那么, 这个结论到底有什么意义呢? 因为“运算都产生值类型的结果”, 且又因为“所有的逻辑语句结构都可以被消灭”, 所以结论是: “系统的结果必然是值, 并且可以通过一系列的运算来得到这一结果^②”。

^① “注 1”所标示的几处运算, 要么是使用引用类型来定位值, 要么就是在象赋值这样在本章中不讨论的运算。

^② 这是一项重要的结论。尽管在这里没有展开讲述, 但如果读者愿意了解一些计算系统基本模型方面的知识, 可以从该项结论为出发点, 了解一些关于函数式和数据流式语言的特性。例如 VAL 这种语言, 一方面它是典型的数据流式语言, 另一方面它也具有某些函数式特性。

为什么“结果必然是值”呢？因为我们的计算机其实只能表达值。对于任何复杂的现象（例如界面、动画或者模拟现实），其实在运算系统来看，只是某种输出设备对数值的理解而已，运算系统只需要得到这些数值结果，至于如何展示，则是另一个物理系统（或其它运算系统）来负责的事情。

所以运算的实质，其实是值的运算。至于象“指针”、“对象”这样抽象结构，在运算系统来看，其实只是定位到“值”以进行后续运算的工具而已。换言之，它们是不参与“求值”运算的。

1.3.2.2. 有趣的运算：在 Internet Explorer 和 J2EE 中

在 Internet Explorer 中，层叠样式表（CSS，Cascading Style Sheet）中会有一些运算过程，用于设定一些特殊的样式属性，例如颜色和 URL 地址：

```
<style>
/* 设置字体颜色 */
DIV {
  COLOR: rgb(127, 127, 0);
}

/* 设置背景图片的 url 地址 */
TABLE {
  BACKGROUND: url(http://127.0.0.1/bg.png);
}
</style>
```

这些过程是符合 CSS 规范的。但是 Internet Explorer5.0 及更高版本的浏览器对此做出了一些扩展，它使用一个名为 expression 的过程，来表明 CSS 属性需要通过一个计算过程来得到值。具体来说，如下例：

```
<style>
SPAN {
  BORDER: 1 solid red;
  POSITION: absolute;
}

#span_left {
  LEFT: 0px;
  WIDTH: 300px;
}

#span_right {
```

```
LEFT: 300px;  
WIDTH: expression(document.body.clientWidth - 300);  
}  
</style>
```

我们可以用下来的 HTML 代码来展示这个样式表的效果：

```
<body>  
<span id="span_left">300px</span>  
<span id="span_right" onresize="this.innerText = this.clientWidth + 'px'">  
</span>  
</body>
```

其效果如下图：



在上面的样式表中，我们用一个表达式运算来使 `span_right` 的宽度总是随当前网页的宽度而变化。我们的目的是要使 `left/right` 两个 `` 标签动态地填充网页上的左右两个部分。因此事实上我们也可以用如下方式写 `span_right` 的样式表：

```
/* 上例的一个更好的版本 */  
#span_right {  
  LEFT: expression(document.getElementById('span_left').currentStyle.width) ;  
  WIDTH: expression(document.body.clientWidth -  
    parseInt(this.currentStyle.left)) ;  
}
```

所以，这是 Internet Explorer 上的一个非常强大的功能。事实上，在 IE6.0 中，有一个名为 IE7 的开源项目就利用这种特性，将在 IE6.0 上的 CSS 样式表扩展出了与 IE7 等同的功能。

那么，Internet Explorer 中是如何扩展这样的一个功能的呢？

如果我们更加完整地考察这个 `expresstion()` 过程，就会发现它的一些独特之处，包括：

- 👉 它是 JavaScript 语法的脚本代码；
- 👉 它可以访问整个的文档对象模型（DOM，Document Object Model）；
- 👉 它只能是一个表达式，或一组用逗号分隔的表达式；
- 👉 它不能使用任何语句和语句分隔符（分号）；
- 👉 它可以声明并使用函数（函数中也可以出现语句），但函数只用于值运算；
- 👉 整个表达式的运算结果是值（用于赋给样式表属性）。

我们综合这些特性就可以发现，这个 `expresstion()` 中所包括的，其实是一种：

- 👉 消灭了语句的
- 👉 用表达式来运算求值的

JavaScript 语言的简化版本。

这是真正有趣的地方。事实上我们可以通过样式表中的 `expresstion()` 过程来完成所有的工作，而无需单独写其它的脚本代码。换言之，这里的 `expresstion()` 已经具备了整个 JavaScript 语言的编程能力。

不过在这里我们还要注意到 `expresstion()` 允许声明和使用函数这一特性。关于这一点，我们下一小节还会重新提及。

这一类的特性也出现在 J2EE 这种大型的语言系统中。在 JSTL 1.0 中，为了方便存取数据所自定义了这样一种语言（只能用在 JSTL 标签中），要求以 # 开始，将变量或表达式放在一对大括号之间，例如：

```
#{...}
```

由于它可以直接访问 `faces-config.xml` 中定义的名称、客户端 Request 中的参数，或者是 JavaBeans 中的成员等等数据，因此可以写出这样的代码来：

```
<f:view>
  名称: <h:outputText value="#{userBean.name}, #{param.name}"/>
</f:view>
```

这个语言名为 “JavaServer Faces(JSF) Expression Language(EL)”，JSEL 中还可以使用数值运算、逻辑运算、关系运算，以及数组和对象成员存取等表达

式。和刚才我们提到的 CSS 中的 `expresstion()` 过程一样，JSEL 提供的也是一个运算求值的结果；并且在表达式运算过程中，也不会出现语句这种语法元素。

这一类的语言，被称为表达式语言（Expression Language，EL）。我们前面论述过，我们可以消灭语言中的陈述运算逻辑的三种语句（顺序、分支和循环），并使代码具有完全等同的编程能力。所以我们也看到表达式语言具有完备的程序设计能力，是一种极端精华的编程范型。

为了将这个范型与直译的“表达式（Expression）”区分开来，在随后的文字中我们将称之为“运算式语言（范型）”——事实上也存在这样的翻译，但这种翻译一般强调的是名词性质的表达式。而我们在这里，将用这个名词来强调“通过运算求值来实现程序设计”的这样一个编程范型^①。

1.3.3. 如何消灭掉语句

读者应当注意到：由连续运算来组织代码，与用顺序语句来组织代码，是两种不同的风格。对于“运算式”这种新的语言风格，如果要想让它成为一种纯粹的、且完备的语言（范型），那么我们就需要让它通过“表达式运算”就完成全部的程序逻辑，这包括其它语言中的三种基本逻辑结构：顺序、分支与循环——是的，我们在讲述命令式语言时提到过这三种基本逻辑结构，它既用于组织代码，亦用于陈述逻辑。同样，运算式语言也需要这两种能力（如果我们不考虑代码写得多么凌乱、难懂的话，我们可以忽略前者）。

因此，为了让“（纯粹地）连续运算”能实现足够复杂的系统，我们要在消减掉“语句”这个语法元素的同时，通过表达式来陈述三种基本逻辑。我们将看到：在运算式语言中，语句是可以被消灭掉的。

1.3.3.1. 通过表达式消灭分支语句

单个分支的 IF 条件语句，可以被转换成布尔表达式。例如：

```
/**
```

```
* 示例 1：消灭条件分支语句 (无 else 分支)
```

^① expression language 通常被译作“表达式语言”，以这种方式称述对象时，主要说明它是一种叙述表达式规格、性质和功能语言，一般不作为程序设计语言，因此也不会指称某种编程范型，例如正则表达式(RegExp)是一种表达式语言，但并不是程序设计语言。在本书中，“运算式语言（expression language）”是确指一种程序设计语言范型，它通过处理表达式求值来完成整个程序设计过程。


```

*/
if (tag > 1) {
    alert('true');
}

// 转换成
(tag > 1) && alert('true');

```

IF 条件分支语句（一个或两个分支），总是可以被转换(三元)条件表达式。例如下面的代码：

```

/**
 * 示例 2：消灭条件分支语句
 */

// 1. 无 else 分支
if (tag > 1) {
    alert('true');
}

// 转换成
(tag > 1) ? alert('true') : null;

// 2. 有 else 分支
if (tag > 1) {
    alert('true');
}
else {
    alert('false');
}

// 转换成
(tag > 1) ? alert('true') : alert('false');

```

由于一个多重分支语句可以被转换成 IF 条件分支语句的连用，例如：

```

/**
 * 示例 3：多重分支语句与 IF 语句连用的等效性
 */
switch (value) {
    100:
    200: alert('value is 200 or 100'); break;
    300: alert('value is 300'); break;
    default: alert('I don\'t know.');
```

```

}

// 等效于
if (value == 100 || value == 200) {
    alert('value is 200 or 100');
}
else if (value == 300) {
    alert('value is 300');
}
else {
    alert('I don\'t know.')
}

```

所以 SWITCH 语句与 IF 语句连用等效。而 IF 语句连用则可以用(三元)条件表达式连续运算来替代：

```

/**
 * 示例 4：消灭 IF 语句连用
 * (参考示例 3)
 */

// ... (if 语句连用示例代码略)
// 转换为
(value == 100 || value == 200) ? alert('value is 200 or 100')
    : (value == 300) ? alert('value is 300')
    : alert('I don\'t know.');
```

1.3.3.2. 通过函数递归消灭循环语句

放开易用性不论，在常见的三种循环语句 while、do..while 和 for 是可以互换的，对这一点我想无需论述。因此，我们下面只以 do..while 语句为例，来讨论循环语句的问题。

循环语句可以通过函数递归来模拟，这一点其实也是经过证实的。下面简单地举一个例子：

```

/**
 * 示例 1：通过函数递归来模拟循环语句
 */
var loop = 100;
var i = loop;
do {
    // do something...
}

```

```

    i--;
}
while (i > 0);

// 用函数递归模拟上述循环语句
function foo(i) {
    // do something...
    if (--i > 0) foo(i);
}
foo(loop);

// 用函数递归模拟上述循环语句 (更能展现函数式语言特性的)
void function(i) {
    // do something...
    (--i > 0) && arguments.callee(i);
}(loop);

```

但是，如果用函数来模拟循环，那么必然存在一个问题，就是栈溢出。循环语句的一个良好特性就是开销很小，而在函数的递归调用过程中，由于需要为每次函数调用保留私有数据和上下文环境，因此将消耗大量的栈空间。

但是在递归中，也可以存在不占用栈的情况，这就是尾递归。简单的讲，尾递归是指在一个函数的执行序列的最后一个表达式中出现的递归调用。由于这个递归是最后一个表达式，那么当前函数不需要为下一次调用保持栈和运算的上下文环境。换言之，这种情况下，递归函数的多次调用中要么使用同一个栈（和上下文环境），要么根本就不用栈（和上下文环境）。

一个简单的现象方法就是：尾递归相当于在函数尾部发生的一个（无需返回的）跳转指令。由于这种特性，所以满足尾递归的函数，就可以在不消耗栈和上下文环境的情况下，用来替代循环语句。关于这种理论，在 SICP 中有过详细的解释，而在现实中，Scheme、Erlang 等语言都将尾递归作为一种重要的特性内置于编译器中。这些编译器内置尾递归（或强调必须使用严格的尾递归）的原因在于：在函数式等编程范型中，通过编译器的优化，可以无需“（循环）语句”这种编程元素来实现高性能的迭代运算。

然而不幸的是，目前已知的 JavaScript 的解释环境中并不支持这种特性。因此，我们在这里讨论函数式时，可以说“能够通过函数递归来消灭循环语句”，但在不支持尾递归（及其优化技术）的 JavaScript 中，这种实现将以大量栈和内存消耗为代价。

1.3.3.3. 其它可以被消灭的语句

由于可以不使用循环和 `switch`，所以标签语句也就没有存在的价值。与此相同的，流程控制中的一般子句（`break` 和 `continue`）也没有存在的价值。

结构化异常处理是结构化编程中的要素，而非函数式编程的要求，因此结构化异常处理所需的几个语句也没有存在价值。

前面说过函数式语言可以不使用寄存器，因此这种事实上只需要值声明，而不需要变量声明——值参与运算，变量其实是值的寄存。所以在函数式语言中，变量声明语句也是不需要的。

所以你会看到，在函数式语言中，除了值声明和函数中的返回子句之外，其它的语句都是可以被消灭的。但是，为什么这两种语句不能被消灭呢？

1.4. 函数：对运算式语言的补充和组织

我们现在面临着一个新的词汇“运算式语言”（EL，`expression language`）。这个需要强调这个词汇，是因为（目前）没有任何书籍将函数式语言的基础归结到这种运算范型中。

我们前面讲到，运算式语言的特点在于强调“求值运算”。我们也列举了 JavaScript 中的表达式运算符，我们看到这些运算符处理的运算元都是“值类型数据”。由此我们可以认为：JavaScript 语言特性中的某个子集，可以作为一个最小化的运算式语言来使用——事实上我们也看到在 Internet Explorer 中将 JavaScript 的一个子集作为 CSS 的表达式 (`expresstion`) 来使用。

我们也讨论到，如果要在运算式语言中完成全部的逻辑，那么它需要通过表达式或者运算来填补“被消灭掉的语句”，更确切地说，它必须有能力实现“顺序、分支和循环”三种逻辑结构。在 JavaScript 中，可以通过“连续运算符（逗号）”和“三元条件表达式”来替代顺序和分支。因此，最后的问题是：如何在表达式级别实现循环？

我们也在前面给出了答案：使用函数递归来消灭循环语句。在这一个小节里，我们将进一步讨论这个话题。

1.4.1. 函数是必要的补充

因为表达式运算是“求值运算”，所以有且仅有“当函数只存在值含义，且没有副作用”时，该函数才能作为表达式的运算元参与运算，并起到替代循环语句的作用。

显然，根据上面所述的“函数式语言中的函数”的特性，它确实可以充当这样的角色。因此，在一个纯粹的、完备的运算式语言中，函数是一个必要的补充。

但这只是理论上的理想化的假设：如果“函数”是满足函数式语言的三个特性的话。而正如我们在前面所讨论的那样，JavaScript 中的函数事实上仍然存在副作用。即使除去这个我们寄期望于“程序员的习惯”来解决的问题不谈，也还有一个无法回避的问题：JavaScript 中的函数并不支持尾递归。

不过这仍不会对 JavaScript 中的运算式语言特性构成太多的负面影响。因为 JavaScript 是一种多范型语言，所以事实上它可以在一个函数内部使用循环语句——当然，这会使它在表面上看起来并没有彻底地消除循环语句。

为什么说是“表面上看起来”呢？因为函数在这里只有值的含义，所以函数只有返回的值在影响系统的运算。由此的推论是：无论函数内部如何实现，事实上并不会影响到其外部的系统（也包括编程范型的纯洁性）。

当然，对于 JavaScript 来说，使用“包含循环语句的函数”来实现纯粹的运算式语言编程时，还需要抑制函数的副作用。关于这一点，我们已经在前面讲述过了。

当在 JavaScript 中需要一种纯粹的“运算式语言”时，函数是一个必要的补充。这首先体现在对循环逻辑的封装上。在“尾递归”与“利用多范型特性来包含循环语句”这两种方案上，JavaScript（非常偷懒地）选择了后者。但只要不产生副作用，我们仍然可以承认这是一种纯粹的运算式范型。

作为更加具体的实例，我们可以在 Internet Explorer 的样式表表达式中，使用这样的技术。在下在的示例中，我们需要通过循环查找顶层的元素(class=boxSpan)，来决定指定的元素(id=rightSpan)的大小，而这样复杂的逻辑，仍然可以通过在样式表中，使用如下的表达式来实现：

```
<style>
```

```

SPAN {
  BORDER: 1 solid red;
  POSITION: absolute;
}

.nb {
  BORDER-WIDTH: 0px;
}

.boxSpan {
  LEFT: 0px;
  WIDTH: 90%;
  HEIGHT: 600px;
}

#rightSpan {
  BORDER: 1 solid blue;
  WIDTH: 100px;
  LEFT: expression(
    (function(el, className) {
      while (el = el.parentElement &&
        el.className != className) { /* null loop */ };
      return (el ? el.clientWidth : 0);
    )
    (this, 'boxSpan') - parseInt(this.currentStyle.width)
);
}
</style>

```

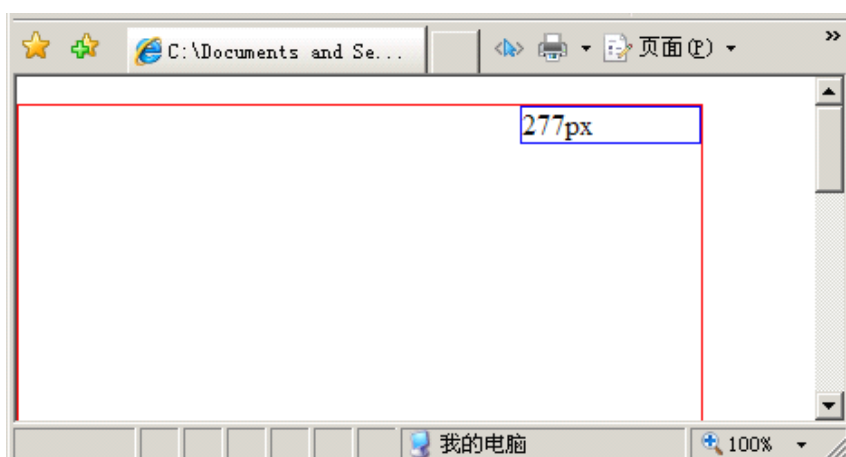
这个样式表的效果可以使用如下 HTML 代码来展示：

```

<body onload="document.getElementById('rightSpan').fireEvent('onmove')">
<span class="boxSpan" >
  <!-- 注：这里无论使用多少层次的 span，均不会影响到 rightSpan 的运算效果。 -->
  <span class="nb"><span class="nb"><span class="nb">
    <span id="rightSpan"
      onmove="this.innerText = this.currentStyle.left"></span>
    </span></span></span>
</span>
</body>

```

如下图所示：



我们看到，蓝色块(rightSpan)总是动态的浮在外框(boxSpan)的右上角。为了增加难度，我们在这里设定外框是使用了 `class=boxSpan`，因此不能用 `getElementById` 这种便捷的方法来找到该元素。在样式表的 `expression()` 中，我们声明了一个匿名函数并立即调用它，该匿名函数的返回值直接参与了后续的表达式运算。

正如我们所强调的那样，在匿名函数中我们使用了循环语句，但这并没有影响到外部表达式运算的纯洁性。因为对于这个表达式来说，该函数是如何实现的并没有关系，有且仅有它的值参与了运算的过程。

1.4.2. 函数是代码的组织形式

我们当然可以使用连续的表达式运算来完成足够复杂的系统，这一点在前面已经论证过了。但是如果我们要真的这样做，那么跟试图用一条无限长的穿孔纸带来完成复杂系统并没有区别——当代码（连续的表达式）达到某种长度之后，我们将难于阅读和调试，最终系统将因为复杂性（而不是可计算性）而崩溃。

在大型系统中，“良好的代码组织”也是降低复杂性的重要手段。对于运算式语言来说，实现良好的代码组织的有效途径之一，就是使用函数。如前所述，函数具有值特性、可运算、无副作用，因此在运算式范型中引入这样一个概念，并不会导致运算规则的任何变化。

所以，我们可以用函数来封装一组表达式，并更好的格式化它的代码风格。从语义上来讲，一个函数调用过程其实只相当于表达式运算中的一个求值：

```
// 表达式一  
a = v1 + v2 * v3 - v4
```

```
// 示例 1：表达式一等于于如下带匿名函数的表达式
a = v1 + (function() {
    return v2 * v3 - v4;
}) ()

// 示例 2：等于于如下表达式
a = v1 + (
    v2 * v3 - v4;
)

// 示例 3：使用非匿名函数的例子
function calc() {
    return v2 * v3 - v4;
}
a = v1 + calc()
```

对于运算式的语法解释过程来说，示例 1 与示例 2 之间的区别仅在于：一个是用匿名函数调用来求值，另一个是通过强制运算符“括号（）”来求值——当然，将前者改成非匿名的，在语义上也没有变化（如示例 3）。

所以在运算式语言中，函数不但是消减循环语句的一个必要补充，也是一种消减代码复杂性的组织形式。

1.4.3. 当运算符等义于某个函数

我们看到，为了实现足够复杂的系统，运算式语言需要“函数”来组织代码和消减循环语句。在前面的行文中，我们花了很长的篇幅，以命令式语言中的函数（**function**）的概念，来解释了运算式语言的这种需求。当然，这种函数除了“名字（**function**）”跟命令式语言中用得一样之外，也具有三种特别的“函数式”特性。

但是我们事实上从来没有解释过“函数式”是什么意思，我们只是反过来澄清过“并不是一个语言支持函数，这个语言就可以叫做函数式语言”。那么，如果要下个定义的话，那么我们是否能总结前文，说“函数式语言是一种用‘函数’来消减循环语句和组织代码的运算式语言”呢？

更深层的问题是：运算式是不是函数式的基础，而函数式又是不是运算式的某个分支呢？

这些问题的根源其实在于“函数式语言”中的这个“函数”，并不是我们在命令式语言中看到的例程（函数 `function` 和过程 `procedure`），也不是我们在 JavaScript 中看到的 `function` 关键字或 `Function` 类型。所以，仅凭“JavaScript 中函数是第一型的”就推论出“JavaScript 是函数式语言”，这种推论是不严谨的，或者说根本就是不正确的。

在认识“函数式语言”之前，必须明确它这个“函数”的含义。

1.4.3.1. “函数” == “lambda”

让我们先来看看“Vyacheslav Akhmechet^①”对 `lambda` 的一个解释：

“我在学习函数式编程的时候，很不喜欢术语 `lambda`，因为我没有真正理解它的意义。在这个环境里，`lambda` 是一个函数，那个希腊字母（ λ ）只是方便书写的数学记法。每当你听到 `lambda` 时，只要在脑中把它翻译成函数即可。”

简单地说，就是：函数 == `lambda`。所以更复杂的概念，例如“`lambda` 演算（`lambda calculus`）”其实就是一套用于研究函数定义、函数应用和递归的系统。

从数学上，已经论证过 `lambda` 运算是一个图灵等价的运算系统；从历史上，我们已经知道函数式语言就是基于 `lambda` 运算而产生的运算范型。所以，在本质上来讲，函数式语言中的函数这个概念，其实应该是“`lambda`（函数）”，而不是在现在的通用语言（我是指的象 C、Pascal 这样的命令式语言）中讲到的 `function`。

1.4.3.2. 当运算符等义于某个函数

我们来看一段普通的 C 代码（以下设 `bTrue` 为布尔值 `true`）：

```
// 示例 1：普通的 C 代码
if (bTrue) {
    v = 3 + 4;
}
else {
    v = 3 * 4;
}
```

^① 《函数式编程另类指南》的作者。

为了让代码简洁些，我们可以写成这样：

```
// 示例 2：使用函数的普通的 C 代码
```

```
function calc(b, x, y) {  
    if (b) {  
        return x + y;  
    }  
    else {  
        return x * y;  
    }  
}  
  
// 等效于示例 1 的运算  
v = calc(bTrue, 3, 4);
```

我们说上面这两种写法都是命令式语言的。下面我们将 JavaScript 作为“运算式语言”，用表达式来重写一下：

```
// 示例 3：使用表达式的 JavaScript 代码
```

```
v = (bTrue ? 3+4 : 3*4);
```

接下来我们提出一个问题，既然在这个表达式中，值 3 与值 4 是重复出现的，那么可不可以象示例 2 一样处理成参数呢？当然，也是可以的：

```
// 示例 4：使用函数来消减掉一次传参数
```

```
function f_add(x, y) {  
    return x + y;  
}  
  
function f_mul(x, y) {  
    return x * y;  
}  
  
// 与示例 3 等义的代码  
v = (bTrue ? f_add : f_mul)(3, 4);
```

我们注意示例 4 中一个问题：f_add()与 f_mul()其实本身并没有运算行为，而只是将“+”和“*”运算的结果直接返回。换言之，事实上这里的“+”与“*”运算符就分别等义于 f_add()与 f_mul()这两个函数。

所以对于上面代码，除开赋值运算符之外的“求值表达式”部分，我们改写成如下（当然，下面的代码并不能被正常执行，但形式上与示例 4 是一致的）：

```
// 示例 5
```

```
(bTrue ? "+" : "*")(3, 4);
```

最后，我们改变一下代码书写习惯（改变书写代码的习惯其实对很多开发人员来说甚为艰难，但我们这里只是尝试一下而已）。新的代码风格是这样约定的：

- 👉 表达式由运算符和运算元构成，用括号包含起来；
- 👉 运算元之间的分隔符使用空格；
- 👉 对于任何运算符来说，运算必须写在前面，然后再写运算元；

注意我们这里没有改变任何逻辑，而只是换用了新的书写方法和顺序。那么新的代码应该写成这样：

```
((?: bTrue "+" "*") 3 4)
```

我们再接着约定：

- 👉 对于三元表达式(?:)来说，?:号改用 if 来标识（至于三个运算元，按前面的规则，跟在运算符后面并用空格分隔即可）；
- 👉 运算符可以用作运算元（这意味着"+"和"*"中的字符串引号可去掉）
- 👉 对于布尔值 true 来说，使用#f 标识。

这里我们只是用了新的符号标识。新的代码应该写成这样：

```
// 示例 6：新的代码风格  
(if #f + *) 3 4)
```

这里，我最终给出答案：示例 6 其实是一行 scheme 语言代码。而 scheme 是 LISP 语言的一个变种，是一种完全的、纯粹的“函数式语言”。

从一个 JavaScript 表达式，到一行 scheme 代码的过程中，我们只做出了一个假设：如果运算符等义于某个函数。如果我们再把这个过程逆转回来，那么同样也可以说：在 JavaScript 中可以用函数来替换运算符。这个例子可以写成这样：

```
// 示例 7：JavaScript 中用函数替代运算符  
var f_mul = function(x, y) {  
    return x * y;  
}  
var f_add = function(x, y) {  
    return x + y;  
}  
function f_if(b, v1, v2) {  
    return b ? v1 : v2;  
}  
  
//与示例 4 完全等义的代码  
f_if(bTrue, f_mul, f_add)(3, 4);
```

我们再一次回顾前面说到的“函数式语言中的函数”的三个特性：

- 👉 函数是运算元
- 👉 在函数内保存数据
- 👉 函数内的运算对函数外无副作用

我们看到，示例 7 中的 `f_mul()`、`f_add()` 和 `f_if()` 完全满足了这三个特性。最后，我们得到结论：“当运算符等义于某个(`lambda`)函数”时，我们前面所讲述的运算式语言其实就是一种“非常纯粹的”函数式语言了。

1.4.3.3. JavaScript 语言中的函数式编程

在上面提到的这行 `scheme` 代码中：

```
(if #f + *) 3 4)
```

“`if`”、“`+`”和“`*`”都是函数，而“`#f`”、“`3`”和“`4`”都是运算数（或者说值）。所以整个的 `scheme` 语言的编程模式，就变得非常简单：

```
(function [arguments])
```

也就是说，整个编程的模式被简化了函数(`function`)与其参数(`arguments`)的运算，而在这个模式上的连续运算就构成了系统——整个系统不再需要第二种编程范型或冗余规则（例如赋值等）。

在我们上面的例子中，我们将 `JavaScript` 中的某些特性作为新范型——一种（基于表达式的）“运算式语言”来讨论，并证明它具有计算系统的完整的逻辑与运算能力。然后我们引入了“函数式语言中的函数”，说明“如果运算符等义于某个(`lambda`)函数”，那么所谓的运算式语言也就成了“函数式语言”了。然而我们也知道，我们不能通过重新定义 `JavaScript` 规范来使得它表现出这种“函数式语言的纯粹性”。因此，我们需要提炼并阐述一个 `JavaScript` 中的最小特性集，使它满足“函数式语言”的特性，以便在后文更好的讨论这些特性。

这样的—个特性集是：

- 👉 只使用表达式和函数，通过连续求值来组织代码(在函数外消除语句)；
- 👉 在值概念上，函数可作为运算元参与表达式运算；
- 👉 在逻辑概念上，函数等义与表达式的运算符（参数是其运算元，返回值是运算结果）；
- 👉 函数严格强调无副作用。

这样的语言特性集的要点在于：关注运算，以及运算之间的关系。使用者必须认识到：连续运算的结果就是我们想要的系统目标。因而我们无可避免地要去面对一种“连续运算”的代码风格，我们的选择仅在于：把这种风格写得漂亮点，或放弃说“函数式语言不是我想要的”。

——当然，很明显我在这里写这本书并不是想要达到后一个目标。

1.5. JavaScript 中的函数

我们在前面已经约定，在 JavaScript 中使用函数式风格编程，应首先使用“运算式语言”的风格，用表达式连续运算来组织代码，并且注意如下概念：在运算元中，除了 JavaScript 中默认的数据类型，重点强调函数可以作为运算元参与运算；在运算符中，强调函数可以等义于一个运算符。

我们已经在“1.9 实现二叉树

JavaScript 的语法：表达式运算”中全面地讨论过表达式运算。在本小节中，我们将详细讨论 JavaScript 中的函数的种种特性。这些特性是 JavaScript 的函数能够成为“函数语言中的(lambda)函数”的根源——或表现为某些不足。当然，反过来说，正是为了让 JavaScript 成为一种函数式语言，设计者才为 function 这种类型添加了这些语言特性。

1.5.1. 可变参数数，与值参数传递

在一些语言中，函数入口参数有多种调用约定，例如值参数、变量参数、传出参数等等。常见的有：

调用约定	传参顺序	清除参数 责任	寄存器 传参	实现目的	其 它
pascal	由左至右	例程自身	否	与旧有过程兼容	较少使用。
cdecl	由右至左	调用者	否	与c/c++模块交互	一些语言缺省使用该约定。
stdcall	由右至左	例程自身	否	Windows API	Windows API 通常使用该约定。

在 JavaScript 中，函数参数值只支持一种调用约定。它的特点表现为：

- 从左至右向函数传入参数；
- 只接受值参数（const 传值约定）；
- 传入参数个数（相对于函数声明时的形式参数）是可变的。

传入可变个数的参数的最简单（却并不常用）的技巧，可能是用它来替代函数内的局部变量声明。例如：

```
function Array.prototype.clear(len) {  
    len = this.length;  
    if (len > 0) {  
        this.splice(0, len);  
    }  
}
```

对于这个例子中的 `clear()` 方法，很显然 `len` 是完全不需要的一个形式参数。既然它只用于在函数内暂存一下 “`this.length`” 的值，那么显然可以声明为一个局部变量。因此在这里，它被声明成函数形式参数 `len` 的唯一可能的理由只是：省掉 `var` 这个关键字^①。

包括 JavaScript 自身的许多内部函数或方法，也都会经常用到“传入可变个数的参数”这种技巧。例如：

```
// 字符串 split 方法，separator 参数缺省值为','。当 limit 缺省时返回整个数组。  
stringObj.split([separator[, limit]])  
  
// 数组的 concat 方法，item1..itemN 参数数是可变的  
arrayObj.concat([item1[, item2[, ... [, itemN]]]])
```

在具体到函数的代码实现时，对于 `split()` 这种形式的函数声明，通常我们会将形式参数都声明出来。例如（`str` 参数是为了避免声明成 `string` 的对象方法）：

```
function s_split(str, separator, limit) {  
    // ...  
}
```

接下来，`s_split()` 函数按如下规则来理解 `separator`：

- 👉 如果值未传入，或为 `undefined`，则返回包含单个 `str` 元素的数组；
- 👉 如果值为空字符串，则将字符串分解成单个字符的字符串数组；
- 👉 如果值为其它值，则转换为字符串，并使用该字符串分解 `str`。

对于规则三，举一个实例来说，如果传入 `null`，`null` 转换成字符串将是 `'null'`，因此对于如下调用来说，将返回数组 `['_', '_']`：

```
s_split('_null_', null);
```

当实际函数调用时，如果指定参数缺省（没有传入实际值），则它的值为 `undefined`。因此识别的代码如下：

^① 我并不赞同这种改变函数形式参数语义的“奇怪用法”，但这的确在某些代码包中出现，并且也是经常被置疑的用法。在这里讲到它，更多的是释疑，而非认同。

```
// 示例一：对于可变参数的识别方法 (1)
function s_split(str, separator, limit) {
    if (separator === undefined) {
        myArray = [str];
    }
    else {
        // ...
        // split to myArray
    }

    // 注：当 limit 值大于结果数组长度时，将返回数组本身
    if (limit !== undefined &&
        limit < myArray.length) {
        myArray.length = limit;
    }

    return myArray;
}
```

现在出现了一个问题：在使用缺省参数（不传入实际值）与使用 `undefined` 作为参数时，该参数的值都是 `undefined`！那么我们如何在代码中区分下面这两种情况呢？

```
// 参数传入 1
s_split(str, undefined);
// 参数传入 2
s_split(str);
```

理论上说，对于 `split()` 这个函数的设计，两种调用方法应该返回完全相同的结果。但是我们的问题只是（在某些可能的情况下）检测二者的不同，因而需要一个这样的方法。JavaScript 对此给出的方法就是：使用 `arguments.length` 来检测实际传入参数的个数。例如：

```
// 示例二：对于可变参数的识别方法 (2)
function s_split(str, separator, limit) {
    // ...
    alert(arguments.length);
}

// 输出：2
s_split(str, undefined);
// 输出：1
s_split(str);
```

`arguments` 也用于在缺少形式参数的情况下进行处理，例如前面提到的 `arrObj.concat()` 方法。形式上我们可以如下这样声明一个类似的函数：

```
function a_concat(arr) { // item1, item2 , ... [, itemN]
    // ...
}
```

我们没有办法对 `item1...itemN` 写出形式参数来，因此只能略去不写。这在具体的代码实现时，应该用如下的方法处理：

```
// 示例三：对缺少形式参数的识别方法
function a_concat(arr) {
    for (var i=1; i<arguments.length; i++) {
        if (arguments[i] instanceof Array) {
            //...
        }
        else {
            arr[arr.length] = arguments[i];
        }
    }
}
```

即使是在有形式参数，也仍然可以使用 `arguments` 来访问。例如上例中，第 0 个形式参数为 `arr`，但也可以使用 `arguments[0]` 来访问它，而且二者完全等效——这种“完全等效”还包括对 `arguments` 的修改。例如：

```
// 示例四：访问 arguments 数组与形式参数完全等效
function myFunc(value) {
    arguments[0] = 100;
    alert(value);
}

// 输出：100;
myFunc('abc');
```

在上面这个例子中，我们看到：不论是通过 `arguments` 还是形式参数，都可以修改入口参数的值。对于其它的一些语言来说，这个修改能否影响到原来的值，取决于参数声明的形式：

```
// pascal 风格 1: 可以在函数内修改参数
function Func_1(var str: String): string;
// pascal 风格 2: 不能在函数内修改参数
```



```
function Func_2(const str: String): string;
```

JavaScript 不提供上述第一种风格的参数声明方式，以及其它类似于 in、out 等约束的参数声明方式。JavaScript 的函数参数声明，等效于上述的第二种风格。也就是说，你永远不可能在 JavaScript 中通过修改参数来影响到函数外部的变量——不过这里必须强调的是，你可以通过成员赋值或方法调用，来影响对象的成员。例如：

```
obj = {  
  value: 100  
}  
function myCalc(obj) {  
  //...  
  obj.value = 2 * obj.value;  
}  
  
// 一般的代码风格  
myCalc(obj);  
alert(obj.value);
```

在这种情况下，“函数没有对外的副作用”只能是程序员的编程风格上的约束。因此如果你试图让代码“更加函数式”，那么更加合理的做法是：用方法来封装成员存取，并在某个函数之外去调用方法。因而对于上面的例子，相对良好的函数式风格应是如下：

```
obj = {  
  value: 100,  
  setValue: function(value) {  
    return (this.value = value);  
  },  
  getValue: function() {  
    return this.value;  
  }  
}  
  
// 函数式风格的代码  
function myCalc(v) {  
  // ...  
  return v*2;  
}  
  
// 输出: 200  
alert(  

```

```
obj.setValue(myCalc(obj.getValue()))
);
```

当然，如果确定 `myCalc()` 是 `obj` 对象的特定逻辑，那么上面的代码也可以是：

```
obj = {
  value: 100,
  calc: function() {
    return this.value * 2;
  }
}
// 输出：200
alert(obj.calc());
```

1.5.2. 非惰性求值

在下面这个例子中，代码的两个输出值将是什么呢？

```
/**
 * 示例 1：在参数中使用表达式时的求值规则
 */
var i = 100;
// 输出 A
alert( i+=20, i*=2, 'value: '+i );
// 输出 B
alert( i );
```

在这个例子中，输出 A 将显示数值 “120”，输出 B 则将显示数值 “240”。对于第一个输出，我们大概可以理解：因为 `alert()` 只接受一个参数，这个参数的值为 “`i+=20`” 的运算结果，因此是 120。

`alert()` 并没有接受第二、三个参数，但用于传入第二个参数值的表达式 “`i*=2`” 却完成了运算，并实际地向 `alert()` 传入了该值——只不过 `alert()` 没有使用它而已。由于该值已经完成了运算，所以在这行代码结束后，变量 `i` 已经被赋值为 240 了。于是，输出 B 将显示数值 240。

这里就体现了 JavaScript 的“非惰性求值”的特性——对于函数来说，如果一个参数是需要用到时，才会完成求值（或取值），那么他就是“惰性求值”的，反之则是“非惰性求值”。而 JavaScript 使用“非惰性求值”的很大一部分原因，在于它的表达式还支持赋值，这也就意味着表达式会产生对系统的副作用。类似于这行代码：

```
alert( i+=20, i*=2, 'value: '+i );
```

在语义上就是应该对系统产生副作用的。但如果参数是“惰性求值”的，那么只有当 `alert()`——或者别的什么函数——使用到第二个参数时，表达式才会完成运算，也才会产生副作用。然而，应该注意到的这种（被假设的）情况是的实质是：表达式在语义上“将”产生副作用，而副作用是否发生却只能取决于参数“是否被使用”。

我们总不可能让静态的语义理解，与动态的执行效果发生这种冲突：程序员写出代码却不知道它会何时产生副作用。因此，在“允许赋值表达式存在”的这种情况下，更加合理的做法的确是像 JavaScript 那样采用“非惰性求值”。这样，函数被调用时的形式就决定了它将产生哪些副作用，以及将传入哪些值到函数用去参与运算——尽管这些值可能根本就不会被用到。

显然，你应该也已经看到了问题：由于值可能根本不被函数内部用到，因此它的运算也可能是完全无意义的：既不产生副作用，也不被使用。例如上面例子中的参数三，它完成了：

- 👉 将数值 `i` 转换成字符串；并，
- 👉 与字符串 `'value: '` 合并成新字符串

这样两个运算，然而运算结果却根本不被 `alert()` 这个函数使用——这显然是一种 CPU 资源的浪费。

反过来讨论一下“惰性求值”。如果在上述的表达式中不存在副作用，例如我们采用如下的参数传入：

```
/**
 * 示例 2: 在参数的表达式中消除副作用
 */
var i = 100;
alert( i+20, i*2, 'value: '+i );
alert( i );
```

那么对于第一个 `alert()` 调用来说，第二、三个参数“是否运算后再传入函数”就并不重要，即使它们完全不运算（你可以想像成没有这两个参数），也不会对第二个 `alert()` 调用造成任何影响。

这意味着在“惰性求值”的语言中，如果参数不被用到，那么它无论是直接量还是某个表达式，都不会占用 CPU 资源，而语言的引擎也可以将它直接优化掉。作为一个更加明显的例子，下面的代码在“惰性求值”的语言中将什么也不会发生，而由于 JavaScript 是“非惰性求值”的语言，因此会进入死循

环：

```
function lock() {
    while (true);
}

function NullFunction() {
}

alert( NullFunction(
    /* arguments..., */
    lock()
));
```

如果上面的代码中还传入更多的参数，那么这所有的参数都将被运算——无论它们有多大的运算量，并最后进入那个 `lock()` 循环。然而，我们知道 `NullFunction()` 其实什么也不做、什么行为也没有，我们调用了一个什么行为也没有的函数，系统却死锁了。

这是 JavaScript 为它支持多语言范型，尤其是支持一种表达式与函数都“可能”对外产生副作用的语言范型（例如命令式语言）而付出的代价。尽管一般开发人员不会写出上例这样的“用来死锁”的代码，但是 JavaScript 也因此也丢掉了函数式语言的一种很好的特性。

1.5.3. 函数是第一型

在中文中解释“第一型(first-class data types)”会是一件比较艰难的事，因为这是个难于解释的词汇。它更完整的说法是“第一类数据类型”，其中的“第一类(first-class)”是有确定含义的修饰词。采用相同构词法的概念还有“第一类值(first-class values)”、“第一类函数/实体(first-class functions/entity)”、“第一类表示类型(first-class representation types)”、“第一类自然数据类型(first-class natural data types)”等等。

在上述所有概念中，所谓“第一类(first-class)”所表示的意思，就是表明指称的目标是“不可分解的、最高级别的、不被重述的”等。在一些解释中，直接套用社会学中的“一等公民(first-class citizens)”来解说 first-class，虽然同样未能说清楚，但起码让人有了直观的概念^①。

^① 事实上，“first-class data types”最早确实引申自“一等公民(first-class citizens)”这个概念。它出自英国科学家 Christopher Strachey 在 1960 年的一篇论文《functions as first-class citizens》。Christopher 是指称语义的奠基者，分时系统(Compactable Time-Sharing System, CTSS)概念的创立者，据说他的名字与 C 语言中的“C”有

与一般程序设计语言中的数据类型概念比较，“基础类型（或基元类型）”和“**first-class data types**”的概念是近同的。所谓基础类型，是指在语言中用来组织、声明其它复合类型的基本元素，它在语言 / 语法解释器级别存在，无需用户代码重述。换言之，“第一型(**first-class data types**)”通常是指基础类型。更加直观地说，他表现为如下特性^①：

- 👍 能够表达为匿名的直接量（立即值）；
- 👍 能被变量存储；
- 👍 能被其它数据结构存储；
- 👍 有独立而确定的名称（如语法关键字）；
- 👍 可（与其它数据实体）比较的；
- 👍 可作为例程参数传递；
- 👍 可作为函数结果值返回；
- 👍 在运行期可创建；
- 👍 能够以序列化的形式表达；
- 👍 可（以自然语言的形式）读的；
- 👍 能在分布的或运行中的进程中传递与存储；
- 👍

很显然 JavaScript 能够满足上述全部特性。但大多数宿主并非独立进程或分布式系统，因此最后一条特性无法体现。不过，因为 JavaScript 函数能够序列化字符串并跨进程与主机传递，因此它天生具备这种能力。

要让 JavaScript 中的“函数(function)”要能够替代运算符，并起到“**scheme 函数(scheme 函数式语言中的函数)**”的作用，其最重要的一条前提就是“让函数可以作为运算元”。也就是说，（如前面列举的特性，）既可以作为数据值存储与向函数传入传出，又可以作为函数来执行调用。这其实意味着它必然是“第一型的”。而“函数即可以是运算符，也可以是运算元（被运算的数据）”——亦即是函数可以作为函数的参数（运算符可以作为运算元）这一特性，在函数式语言中有一个专门的名词，叫“高阶函数”。

所以事实上“高阶函数”与“函数是第一型”两个特性是相依存的。JavaScript 中的第一型就是指六种基本类型：**undefined**、**string**、**boolean**、**number**、**object**

着一些关系。

^① Wiki: http://en.wikipedia.org/wiki/First-class_object

和 `function`。

1.5.4. 函数是一个值

我们经常听到有人说“JavaScript 中所有的东西都是对象”。其实，从函数式语言的角度来看却正好与此相反，变成了“所有的东西都是值”。函数是第一型——可以作为值来使用、传递等，这也充分体现了上述这个观点。

因为函数是值，所以函数可以被作为值来存储到变量，也可声明它的直接量。例如：

```
// 将一个函数直接量赋值给变量 func 存储
var func = function() {
    //...
}

// 声明一个(命名的)函数变量
function myFunc() {
    // ...
}
```

因为函数是值，所以函数直接参与表达式运算。例如：

```
// 直接参与布尔运算
if (!myFunc) {
    // ...
}

// 与字符串等其它类型数据混合运算
value = 'the function context: ' + myFunc;
```

因为函数是值，所以它也可以作为其它函数的参数传入，或作为结果值传出——上一小节说这是“高阶函数”的特性，放在本小节中，作为“值的特性”来解释，也是一样的。例如：

```
// 函数作为参数传入
function test(foo) {
    // 函数参与表达式运算，以及作为函数返回值传出
    return (foo !== myFunc ? foo : function() {
        // ...
    });
}

// 将 test() 函数调用的返回值作为新的函数调用
test(myFunc)();
```

但是，JavaScript 的函数的对象特性有一个非常特殊的地方：通过 `new` 方式构造的函数对象，其 `typeof` 值仍然为 `'function'`。

1.5.5. 可遍历的调用栈

调用栈是 JavaScript 的一个非常有用的特性，很多高级的 JavaScript 功能都需要使用它来扩展。这些特殊的技巧将在本书更靠后的章节中去讲述，在这一节里，主要讲述这个特性本身。

一个静态的、未调用的函数只是一个值。一旦它(例如函数 F)调用时，系统就将当前正在运行的函数(例如函数 A)入栈，并保留函数 A 的执行指针。当函数 F 执行完之后，函数 A 出栈，并继续执行指针后的代码。例如这样的代码：

```
function F() {  
    // ...  
}  
function A() {  
    F(x, y, z);  
}
```

下面的形式代码说明上述代码在执行过程中栈的构造，以及函数调用的初始化过程：

```
function BaseArray() {  
    // 基本数组的构造器，拥有 length 属性，支持下标存取  
}  
function _stack_push(cp) {  
    // 指定的代码指针(cp, Code Point)入栈  
}  
function _stack_pop() {  
    // 指定的代码指针出栈，返回一个 cp  
}  
function _set_cp(cp) {  
    // 让引擎从指定的 cp 位置开始执行，引擎会分析 cp 位置所在的  
    // 闭包(closure)并装载该闭包环境  
}  
function _cp_function(foo) {  
    // 将指定函数 foo 装入引擎，并获取代码指针  
}  
  
/**
```

```

* 以下为形式代码: F(x, y, z);
*/
function _A() {
    _stack_push($$CP);

    F.arguments = new BaseArray(x, y, z);
    F.arguments.callee = F;
    F.caller = _A;
    _set_cp(_cp_function(F));

    _set_cp(_stack_pop().next());
}

```

这个形式代码中最重要的是 callee 与 caller 的赋值，这就是 JavaScript 为用户可访问的调用栈准备的全部。

1.5.5.1. callee: 我是谁

callee 在 JScript 5.5、JavaScript1.2 以下版本中是没有实现的，即使是现在它也用得不多。但由于匿名函数和函数可被重写的特性的存在，使得 callee 变得不可或缺。下面两例说明在没有 callee 成员时的问题：

```

12 // 示例 1: 匿名函数调用自身
13 void function() {
14     // 在这里如何调用函数自身?
15 }();
16
17 // 示例 2: 函数如果在递归过程中被重写
18 var loop = 0;
19 function myFunc() {
20     otherFunc();
21     alert(loop++);
22     myFunc();
23 }
24
25 function otherFunc() {
26     myFunc = null;
27 }
28
29 myFunc();

```

在示例 2 中，另一种常见的（类似）情况可能是 myFunc 是一个时钟触发

过程^①。虽然 `otherFunc()` 的代码写得有点不可理喻，但它表明的意思可能只是：“`myFunc`”这个全局变量名被第三方代码重写。很显然 `loop` 值只能被正常显示一次，当第一次迭代结束后，`myFunc()` 值被重写了，于是当代码执行到第 11 行时，便出错了。看起来，解决的法子是不要使用全局变量名，但如果我们不给 `myFunc()` 一个全局变量名，那么问题就回到了示例 1 —— 我们如何调用匿名函数自身呢？

所以从 JavaScript 1.2 开始使 `arguments` 对象拥有一个成员：`callee`，该成员总是指向该参数对象 (`Arguments`) 的创建者函数。由于 `arguments` 总是在函数内部可以直接访问，因此也就总是可以在函数内部识别“我是谁”。所以，如果不考虑函数名重写或匿名的问题，下面的等式是成立的：

```
function myFunc() {  
    // 下列等式成立  
    arguments.callee === myFunc.arguments.callee;  
    arguments.callee === myFunc;  
}
```

这样一来，无论是否匿名，函数的递归就总是可以写成：

```
void function() {  
    // ... (略)  
    arguments.callee();  
}();
```

1.5.5.2. caller: 谁呼(叫)我

在函数内部识别自身（我是谁），只是解决了匿名递归的问题，并不是我们说到了“遍历调用栈”的问题。如果我们要遍历函数 A 调用函数 B（以及调用函数 A 自身）的所形的栈，那么我们就需要用到 `caller` 这个成员。从前面的形式代码中我们可以看到，这个成员是 `Function` 的一个属性。

所以我们先讲述 `callee` 属性的原因仅在于：如果我们要遍历栈，则必须先找到指定的（包括匿名函数在内的）函数，然后才能使用 `caller` 来访问它^②。下面的代码演示如何遍历调栈：

```
var  
    _r_function = /^function\b *([$\w\u00FF-\uFFFF] +) *\n/m;  
  
function showIt(func) {  
    .....  
}
```

^① 这种问题在时钟触发的代码中重现时还会牵涉到函数闭包及其效果，因此我们会放在下一章中讲。

^② JavaScript 中，也可以直接用 `arguments.caller` 来访问。但这是非标准的用法。

```

var name = (!(func instanceof Function) ? 'unknown type' :
  _r_function.test(func.toString()) ? RegExp.$1 :
  'anonymous');
alert('-> ' + name);
}

function enumStack(callback) {
  var f = arguments.callee;
  while (f.caller) {
    callback( f = f.caller );
  }
}

function level_n() {
  enumStack(showIt);
}

function level_2() {
  // ...
  level_n();
}

function test() {
  level_2();
}

test();

```

我们看到如下的显示：

```

-> level_n
-> level_2
-> test

```

亦即是 `enumStack()` 函数被调用时的栈信息——排除 `enumStack()` 函数自身。

接下来我们提一个问题：既然 `caller` 是 `Function` 的一个成员。那么在栈上如果出现两次、多次同一个函数，该函数的 `caller` 又如何处理呢？——更加严重的是，如果这个函数是来自几次不同的调用，又怎么办呢？

答案是：`JavaScript` 中并不能识别这种情况，所以导致遍历出错。下面的示例说明这种情况：

```

// ... (续上例)

```

```

var i = 0;
var i_max = 1; // <-- 当这里置值大于1时，将导致 enumStack() 进入死循环
function f1() {
    f2();
}
function f2() {
    if (++i < i_max) {
        f1();
    }
    enumStack(showIt);
}

f1();

```

我们这个例子构造得稍稍复杂了一点，其逻辑基本上就是：f1()调用 f2()，然后 f2()再调用 f1()，使用变量 i 来决定调用的次数，因此这个循环调用也是可以中断的。当我们设置 i_max 为 1 时，则在 f2()中不会重新调用 f1()。因此它输出的栈是：

```

-> f2
-> f1

```

当我们设置 i_max 大于 1（例如置值 2）时，则正确的栈应该是：

```

-> f2 // caller 指向 f1
-> f1 // caller 指向 f2
-> f2 // caller 指向 f1
-> f1 // caller 指向 host

```

但我们发现，当列举到第二遍时，f1.caller、f2.caller 就开始相互指向对方，从而导致死循环：

```

-> f2 // caller 指向 f1
-> f1 // caller 指向 f2
-> f2 // caller 指向 f1 // <-- 由于指向的 f1 仍是上一次迭代中的函数实例，导致下述结果
-> f1 // caller 指向 f2 // <---这里的 host，在第二行的位置被覆盖了
-> (死循环)...

```

因此，看起来非常漂亮的列举栈的功能，在这里的例子中失效了。当然，对于 JavaScript 自身来讲，这并不是问题。因为在引擎内部会有一个真实的、不依赖函数引用或函数名的调用栈。这正是因为引擎内部和外部的脚本环境并不一致，所以我们可以调试器中看到上例的正常调用栈，而在脚本代码中，

却没有任何办法来列举它^①。

也许是因为这样的原因，尽管在 Mozilla 的 JavaScript(基于 SpiderMonkey) 中实现了该方法，但在同出一源的 Safari for Mac 的 WebKit 引擎中，就没有在 Function 对象中实现 caller 属性。然而从 WebKit 项目组得到的反馈却是：ECMA Script 标准反对使用——很遗憾，这也是事实。

1.6. 闭包

让 JavaScript 的“纯函数式风格代码”看起来好看一些的基本诀窍在于：用表达式运算来实现局部逻辑，用函数声明来组织整体的代码；用表达式来完成最终的运算与输出。一个简单、直观的代码块示例如下：

```
function f1(){
    // expressions
}

function f2() {
    // expressions
}

// main()
// 这里的 f1, f2 用作入口参数，是强调函数无副作用的特性
alert(function(x, y) {
    return x() + y();
}(f1, f2));
```

这当然意味着“函数”将会大量出现。而我们需要保证“函数式风格”的纯粹性，以免回归到命令式语言的老路上去，就需要强调“在内部保存数据和对外无副作用”这两大特性。而这两个特性，在 JavaScript 中都是通过“函数闭包（function closure）”来实现的。

除了函数闭包之外，在 JavaScript 中还存在一种特殊的“对象闭包（object closure）”。它是与 with 语句实现直接相关的一种闭包。因为这种特殊性，我们将在“1.6.2.2 对象闭包带来的可见性效果”小节单独地讨论它。而在其它章节中的所谓“闭包”，都是特指函数闭包。

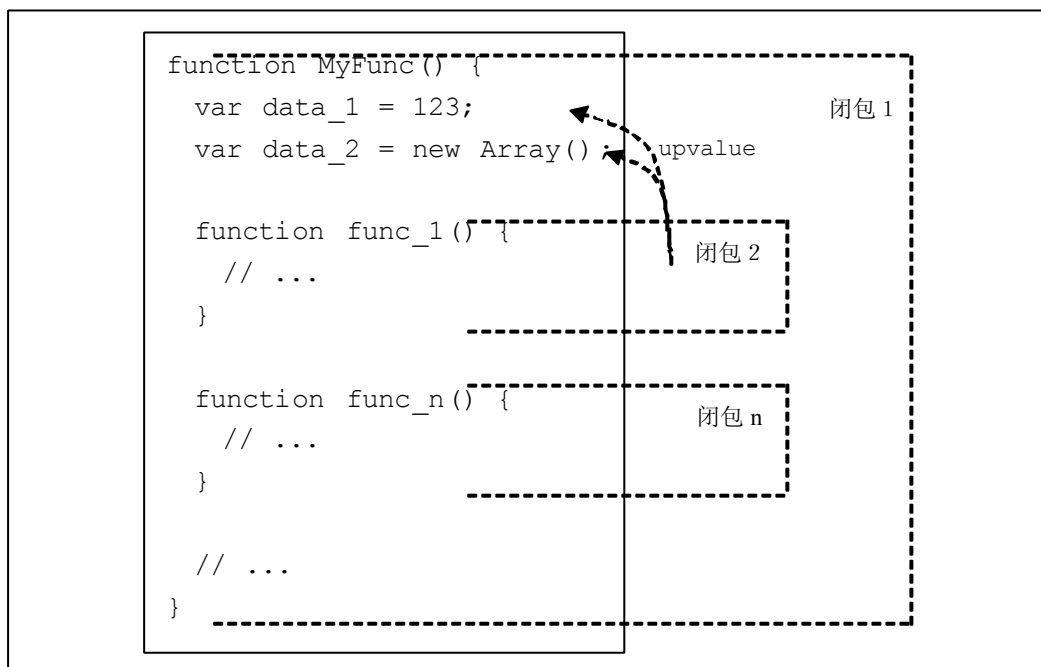
^① 关于这个问题，我们在后面关于 JScript 5 的兼容实现中，讨论到 call/apply 的实现及其限制时，会再次提到这个问题。我们的处理方法，是在 _safe_caller() 函数中，加入列举层数的限制。当然，这仍不能避免栈上出现同一函数时，_safe_caller() 失效的问题。

1.6.1. 什么是闭包

闭包(Closure)与函数有着紧密的关系，以至于许多人将函数与闭包等同起来讨论，但结果却总是讨论不清楚。事实上在 JavaScript 中，一个函数只是一段静态的代码、脚本文本，因此它是一个代码书写时，以及编译期的、静态的概念；而闭包则是函数的代码在运行过程中的一个动态环境，是一个运行期的、动态的概念。由于引擎对每个函数建立其独立的上下文环境，因此当函数被再次执行或进入函数体内的代码时，就将会得到闭包内的全部信息。

闭包具有两个特点：第一是闭包作为与函数成对的数据，在函数执行过程中处于激活（即可访问）状态；第二是闭包在函数运行结束后，保持运行过程的最终数据状态。因此函数的闭包总的来说决定了两件事：闭包所对应的函数代码如何访问数据，以及闭包内的数据何时销毁。对于前者来说，涉及作用域（可见性）的问题；对于后者来说，涉及数据引用的识别。

闭包包括的是函数运行实例的引用、环境（environment，用来查找全局变量的表）、以及一个由所有 upvalue 引用组成的数组，每个闭包可以保有自己的 upvalue 值。下图说明这种结构关系：



上图并不能很好地传达出“闭包是运行期概念”这样的信息，它只是从静态的、视觉效果上说明闭包、子函数闭包、upvalue 之间的关系（闭包 n 与闭包 2 有相同的 upvalue，但图中未能标示出）。需要记住的是：

- 👉 在运行过程中，子函数闭包（闭包 2~n）可以访问 `upvalue`；
- 👉 同一个函数中的所有子函数（闭包 2~n），访问一份相同值的 `upvalue`。

下例简单地说明第二条特性：

```
function MyFunc() {  
    var data = 100;  
  
    function func_1() {  
        data = data * 5;  
    }  
  
    function func_n() {  
        alert(data);  
    }  
  
    func_1();  
    func_n();  
}  
  
// 由于 func_n 与 func_1 使用相同的 upvalue 变量 data，因此在 func_n() 中可以  
// 显示 func_1() 对该值的修改。返回结果值：500  
MyFunc();
```

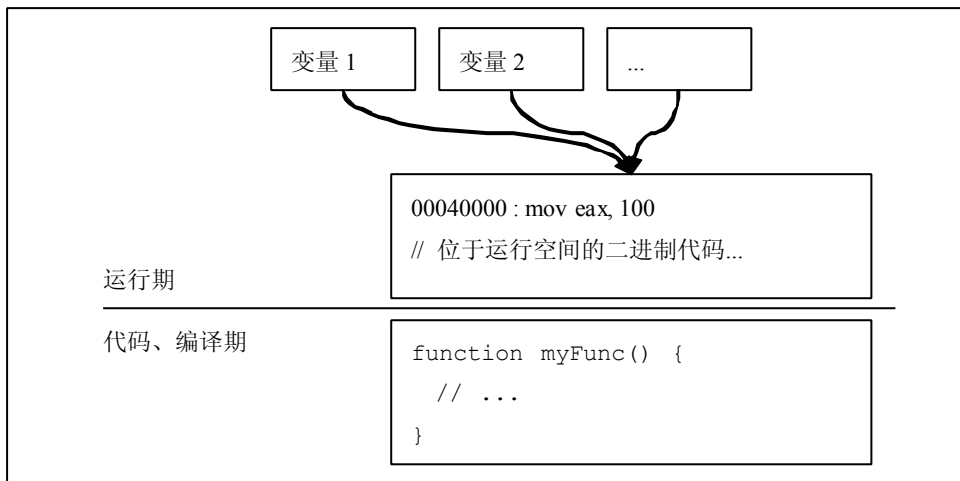
(from: <http://homepage.yesky.com/96/2612596.shtml>)

<http://www.360doc.com/showWeb/0/0/119543.aspx>

(refernce: http://blog.csdn.net/ruby_cn/archive/2004/11/23/192588.aspx)

1.6.2. 什么是函数实例与函数引用

在书写代码的过程中，函数只是一段代码文本。对于编译语言来说，这段文本总被编译成确定的代码，并放在确定的位置执行。因此在编译语言里，一段代码文本，与运行期的代码实例，实际上是等同的、一对一的概念。又由于函数可以被多个不同的变量引用，因此一个函数的代码块在运行期可以对应于多个变量（函数入口地址指针）。如下图所示：



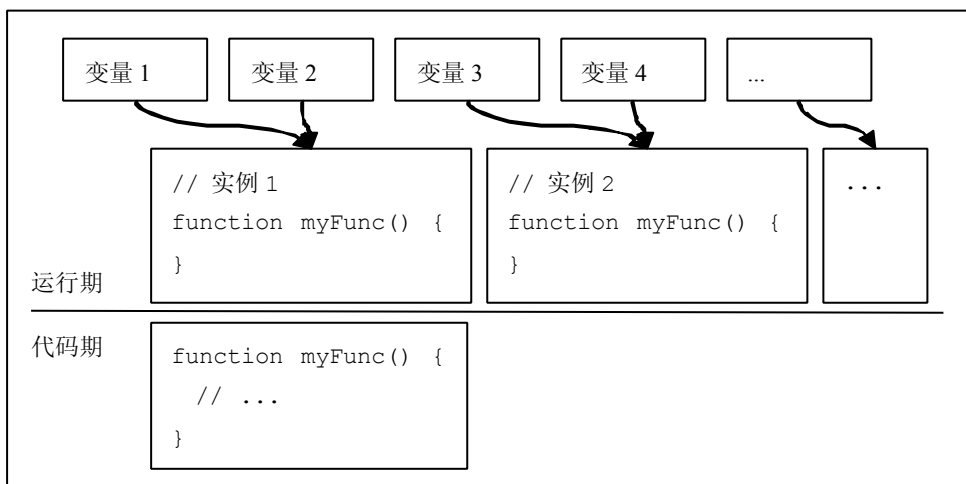
在 JavaScript 中也可以具有这种关系。下面的例子说明这种情况（一个函数有多个引用）：

```
function myFunc() {
    // ...
}

var f1 = myFunc;
var f2 = myFunc;

// 返回值 true，表明变量 f1 与 f2 指向同一个函数实例。反过来说，也就是 f1 与 f2 是
// 该函数实例的多个引用。
alert( f1===f2 );
```

但在 JavaScript 中，却并不是单单只有这种关系。更复杂的情况是：一个函数代码可以有多份函数实例，一个函数实例可以有多个函数引用。因此变成了下图的样子：



下面这个例子，说明在 JavaScript 中同一个函数代码块可以有多个函数实例：

```
function MyObject() {  
    function func() {  
        // ...  
    }  
  
    this.doFunc = func;  
}  
  
var obj1 = new MyObject();  
var obj2 = new MyObject();  
  
// 显示 false, 表明这是两个不同的函数实例  
alert( obj1.doFunc === obj2.doFunc )  
// 显示 true, 表明两个函数实例的代码块完成相同  
alert( obj1.doFunc.toString() == obj2.doFunc.toString() )
```

也许有读者认为上例是 new 运算的效果，但事实上这与 new 运算无关，也与 JavaScript 的对象系统无关。下例说明这一点：

```
function myFunc() {  
    function func() {  
        // ...  
    }  
  
    this.doFunc = func;  
}  
  
var obj = new Object();  
  
// 1. 进入 myFunc, 取 func() 的一个实例  
myFunc.call(obj);  
// 2. 套取函数实例的一个引用, 暂存  
var func = obj.doFunc;  
// 3. 再次进入 myFunc, 取 func() 的一个实例  
myFunc.call(obj);  
// 4. 比较两次取得的函数实例, 结果显示 false, 表明是不同的实例  
alert( func === obj.doFunc );  
// 5. 显示 true, 表明两个函数实例的代码块完成相同  
alert( func.toString() == obj.doFunc.toString() )
```


1.6.3. （在被调用时，）每个函数实例至少拥有一个闭包

许多书籍总是不能把函数与闭包讲清楚，其中的原因之一，就是不能将函数、函数实例与函数引用的分开来。关键在于：闭包是对应于运行期的函数实例的，而不是对应函数（代码块）的。由于闭包对应于函数实例，那么我们只需要分析哪些情况下产生实例，就可以清楚地知道运行的闭包环境。

前面已经例举过，下面的情况只产生引用，不产生实例：

```
// 例一：没有函数实例产生
function myFunc() {
}
var f1 = myFunc;
var f2 = myFunc;

// 显示 true
alert( f1 === f2 );
```

对象的实例只持有原型中的方法的一个引用，因此也不产生（方法）函数的实例：

```
// 例二：没有函数实例产生
function MyObject() {
}
MyObject.prototype.method = function() { /* ... */ };
var obj1 = new MyObject();
var obj2 = new MyObject();

// 显示 true
alert( obj1.method === obj2.method );
```

下面的例子也是常用的构造对象的方法，但它会产生多个函数实例（与例 2 的原型构造方法正好不同）：

```
// 例 3：有函数实例产生
function MyObject() {
    this.method = function() { /* ... */ };
}

var obj1 = new MyObject();
var obj2 = new MyObject();

// 显示 false;
alert( obj1.method === obj2.method );
```

可见，常见的两种构造对象的方法，产生对象实例的效果并不一样（我们这里强调的是方法的闭包性质）。下面的示例，就交叉利用了例 2 与例 3 两种不同特性，来实现构造器及其原型，从而在 JavaScript 中构造出复杂的对象：

```
// 构造器函数
function MyObject() {
    var instance_data = 100;

    this.getInstanceData = function() {
        return instance_data;
    }

    this.setInstanceData = function(v) {
        instance_data = v;
    }
}

// 使用一个匿名函数去修改构造器的原型 MyObject.prototype，以
// 访问该匿名函数中的 upvalue.
void function() {
    var class_data = 5;

    this.getClassData = function() {
        return class_data;
    }

    this.setClassData = function(v) {
        class_data = v;
    }
}.call(MyObject.prototype);

// 创建对象
var obj1 = new MyObject();
var obj2 = new MyObject();

// 输出 100
// 表明 obj2 与 obj1 的 getInstanceData 是不同函数实例，因此在访问不同闭包的 upvalue
obj1.setInstanceData(10);
alert(obj2.getInstanceData());

// 输出 20
// 表明 obj 与 obj1 的 getClassData 是同一个函数实例，因此在访问相同的 upvalue
obj1.setClassData(20);
```

```
alert(obj2.getClassData());
```

除了构造对象实例的情况，我们也常常在函数中将内部函数作为返回值。下面的例子，随函数的执行次数产生多个 **MyFunc()** 函数实例：

```
// 例 4: 有函数实例产生
function aFunc() {
    function MyFunc() {
        // ...
    }
    return MyFunc;
}
var f1 = aFunc();
var f2 = aFunc();

// 显示 false
alert( f1 === f2 )
```

下面两个例子与上例中的 **aFunc()** 一样，都将在多次调用中，产生多个函数实例：

```
// 例 5: 有函数实例产生
function aFunc_1() {
    var MyFunc = function () {
        // ...
    }
    return MyFunc;
}

// 例 6: 有函数实例产生
function aFunc_2() {
    return function() {
        // ...
    };
}
```

我们也有机会返回同一个的函数实例，只是比较麻烦一点：

```
// 例 7: 有函数实例产生
var aFunc_3 = function () {
    var MyFunc = function () {
        // ...
    }
}
```

```
// 返回一个函数到 aFunc_3
return function() {
    return MyFunc;
}
}()

// 多次调用 aFunc_3() 将得到 MyFunc 的同一个实例
var f3 = aFunc_3();
var f4 = aFunc_3();

// 显示 true
alert( f3 === f4 )
```

在这个例子中，我们实际上在 `aFunc_3` 内部创建了一个匿名函数，这个匿名函数（即使是多个实例）访问 `upvalue` 时，将会得到相同的数据。因此多次调用 `aFunc_3` 时，就会得到同一个 `MyFunc()` 实例——匿名函数的 `upvalue` 中的那个实例。

上面的各种形式在各个 JavaScript 的代码包中都经常出现。但本质上就是“产生”与“不产生”函数实例两种形式，进而形成两种闭包、两种保护数据和提供运行上下文环境的形式。

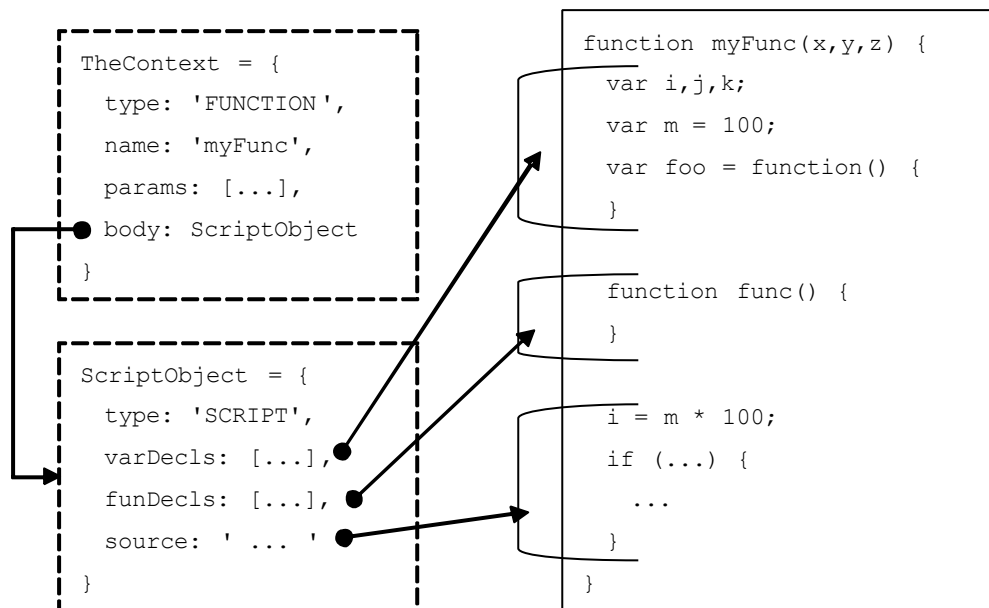
1.6.4. 函数闭包与调用对象

《JavaScript 权威指南》对“调用对象”作出了三点说明：

- 👉 4.6 对象属性与变量没有本质差异
- 👉 4.6.1 全局变量其实是“全局对象(global object)”的属性
- 👉 4.6.2 局部变量其实是“调用对象(call object)”的属性

为了解释上述话题，《JavaScript 权威指南》引入了“JavaScript 执行环境（执行上下文，`execution context`）”的概念^①。这其中所说到的“全局对象(global object)”与“调用对象(call object)”其实是下图中的 `ScriptObject` 结构：

^① 在我早期的一些公开文档中，我会把执行环境与对象上下文（`Object Context`）混同起来，根本的原因便在于我把“全局对象 / 局部对象（`global object / call object`）”与 JavaScript 中的 `Object()` 对象实例混作一谈。事实上，在 JavaScript 环境中，对象实例是没有、也并不需要“上下文环境”的。因为对象是存储系统，不是执行系统，其主要运算是成员存取操作“`.`”和“`[]`”——我们甚至讨论过“在 JavaScript 中并没有真正的方法调用运算”（参见 1.5 面向对象编程的语法概要）。这种对象实质上只是一个“名字—值”对照表（其自身也可作为引用与值参与运算）。但这些都与“函数”完全无关，与运行期的“上下文环境”也完全无关。



该图中，TheContext 结构只描述函数作为对象时的外在表现，例如名称是 myFunc，参数有哪几个等等。它的 body 指向一个 ScriptObject，这个 ScriptObject 就包含了函数代码体中全部的语法分析结构，包括：内部变量表 (varDecls) 和内嵌函数表 (funDecls)，以及除此之外的全部代码 (source)。

下面讲述有关于该结构的基本规则。

1.6.4.1. “调用对象”的局部变量维护规则

规则一：在函数开始执行时，varDecls 中所有值将被置为 undefined。因此我们无论如何访问函数，变量初始值总是 undefined。例如：

```
// 变量初始化
function myFunc() {
  alert(i);
  var i = 100;
}

// 输出值总是 undefined
myFunc();
myFunc();
```

由于 varDecls 总是在语法分析阶段就已经创建好了，因此在 myFunc() 内部，即使是在 “var i” 这个声明之前访问该变量，也不会有语法错误，而是访

问到位于 `ScriptObject` 结构中的 `varDecls` 中该变量的初值：`undefined`。由于 `varDecls` 总是在执行前被初始化，因此第二次调用 `myFunc()` 时，值仍是 `undefined`。

规则二：函数执行结束并退出时，`varDecls` 不被重置。正因为 `varDecls` 不被重置，所以 JavaScript 中的函数能够提供“在函数内保存数据”这种函数式语言特性。需要说明的是，该规则与规则一并不冲突。上例中第二次调用 `myFunc()` 时，没有显示“在函数内所保存的数据（该例中是数值 100）”的原因是：第二次执行函数时，在进入函数前 `varDecls` 被再次初始化了。

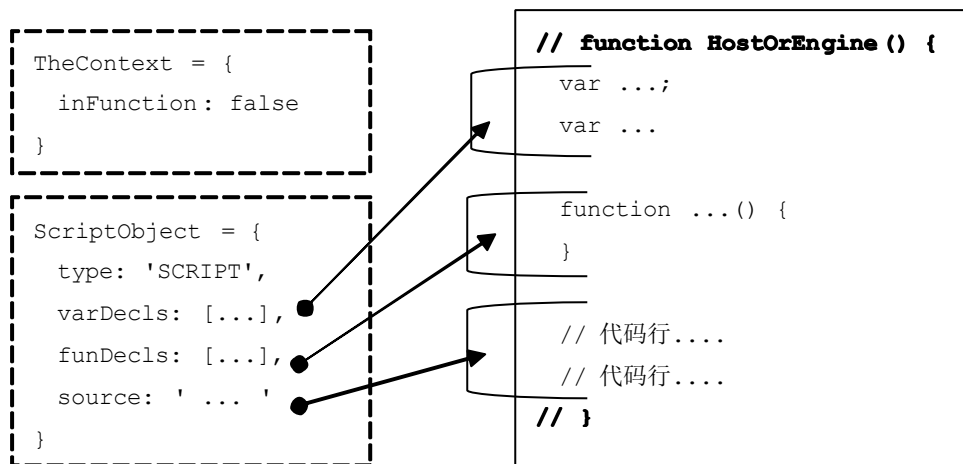
如果我们不再次执行该函数，而通过一些方法来考察该函数的内部，就可以看到该数据被保存了。关于这个示例，请参见“ ”。

规则三：函数内数据持续（即“在函数内保存数据”）的生存周期，取决于该函数实例是否存在活动的引用，当没有活动引用时，由内存回收机制销毁函数和“调用对象(`ScriptObject`)”。

1.6.4.2. “全局对象”的变量维护规则

我们上面例举的是一个函数内部的局部变量、嵌套函数和代码。`ScriptObject` 结构对于“全局对象”来说，仍然是完全适用的。唯一不同的是，从代码的语义分析上，我们找不到一个全局的“`TheContext`”结构。

不过在 JavaScript 引擎内部，这个全局的 `TheContext` 结构仍然是存在的。这种情况下，我们可以把全局代码看作是某个函数中的“`SCRIPT`”代码块。基本结构如下：



全局的变量维护规则与关于“调用对象”的变量维护规则并不冲突。但由于（虚构的）HostOrEngine()存在着特殊性，因此：

- 👉 由于该函数从来不被再次进入，因此不会被重新初始化；
- 👉 由于该函数有且仅有一个实例，且总是被系统持有引用，因此它自身和内部数据总是不被销毁。

需要强调的是，由于这些特殊性，下面关于生存周期的讨论中将不包括“全局对象”。

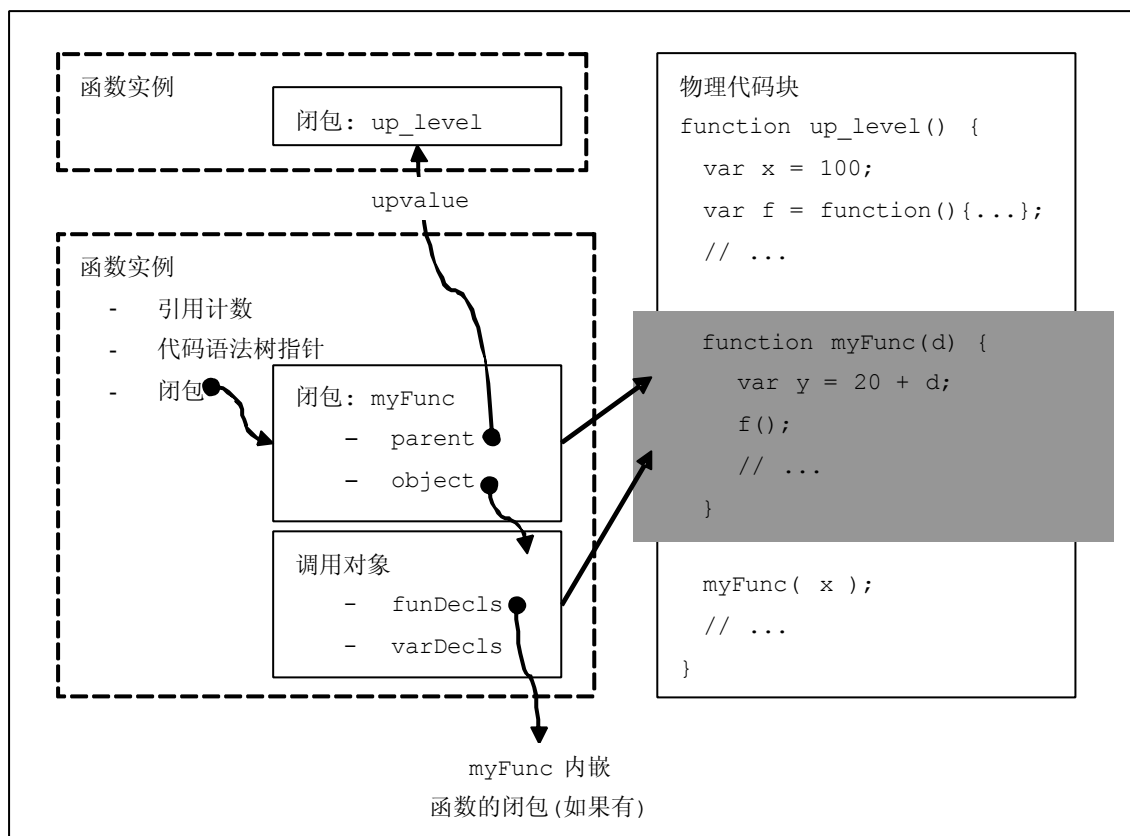
1.6.4.3. 函数闭包与“调用对象”的生存周期

读者应该注意到，上述的“调用对象”其实在语法分析期就可以得到。有了这个在语法分析期得到的 ScriptObject 作为原型，接下来事情就好办了。因为在运行期该函数实例有一个函数闭包，所以在执行它时，引擎将会：

- 👉 创建一个函数实例；然后，
- 👉 为该函数实例创建一个闭包；然后，
- 👉 为该函数实例（及其闭包）的运行环境从 ScriptObject 复制一个“调用对象”。

因此这三个运行期的结构都是一体出现的，一般情况下看起来就象处于同一个生存周期之中。虽然在后续的章节中你会看到其它特殊的情况，但在这里，读者先理解为同一生存周期亦无可——我们先讨论这种闭包、“调用对象”的生存周期，依赖于具体函数实例被引用、释放引用和销毁的周期的情况。

下面结合示例代码做进一步解释：



在这个例子中，`myFunc()`访问了两个存在依赖的变量。其一是参数 `d`，通过函数入口传入。这种情况下，它可能在函数内被持有，或仅作求值运算。但在这个例子中，语句：

```
var y = 20 + d;
```

仅将参数 `d` 用作表达式求值运算，因此在 `myFunc()`闭包退出后，`d` 的引用状态被复位。也就是说，在语句：

```
myFunc( x );
```

中传入的变量 `x`，在 `myFunc()`中并没有被持有。

但第二个变量 `f`，就与此不同。变量 `f` 是在闭包中通过 `upvalue` 访问到的 `up_level()` 中的变量。接下来的问题是：由于 `f` 是闭包外的引用，因此当函数 `myFunc()` 还存活时，函数 `up_level()` 也必须存活。而且由于 JavaScript 是动态语言，因此 `up_level()` 的代码体中随时可能会修改变量 `f`；又由于 `myFunc()` 可能被外部其它变量引用，因此在 `up_level()` 执行结束后，变量 `f` 也不应该被清除（这正是函数式语言中函数闭包的特性）。

因此当在一个闭包使用了 `upvalue`，那么 `upvalue` 所在的闭包，就为该函数

所依赖。

除了在闭包内通过标识符显式地引用 `upvalue`，从而导致闭包与闭包间的生存周期关联之外，还有一种导致关联的情况。这正好也与前面提到的一个例子有关系。我们在讨论参数 `d` 时，说到过“（参数）可能在函数内被持有，或仅作求值运算”。但是我们上面的例子只讲述了“作求值运算”而不导致引用。而另一种，则会因为在函数内被持有，而导致闭包间产生关联关系。例如：

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

    function myFunc(foo) {
        var _foo = foo;
        // ...
        function aFunc() {
            _foo();
        }
    }

    myFunc(f);
}
```

在这个例子中，函数 `f` 被 `myFunc()` 中的变量 `_foo` 持有了一个引用。尽管“对函数外无副作用”，以及函数 `f` 是通过入口参数传递的，但仍然不可避免地进行了引用计数。在这里也应该注意到一个细节，前例中使用的代码是：

```
var x = 100;
// ...
function myFunc(d) {
    var y = 20 + d;    // d是入口参数，传入值类型变量 x
    // ...
}
```

而在本例中使用的代码是：

```
var f = function() {...};
// ...
function myFunc(foo) {
    var _foo = foo;    // foo是入口参数，传入引用类型变量 f
    // ...
}
```

我们在两个例子中，传入的数据类型和在函数内使用的方法都不一致。真正导致闭包持有外部变量的原因是：当 `foo` 是一个引用，且被闭包内的某个变量赋

值、传递或收集（例如数组的 `push()` 操作）。

由于是动态语言的缘故，一个变量 / 入口参数是否是引用类型只能在运行期动态地获得，而无法通过代码的上下文语义来分析。所以我们不能有更好的、差异更明显的例子来对比上述两种情况。而读者应知道，这个“引用与释放引用”的运算是运行期的、不可预知的行为。这便足够了。

1.6.4.4. 引用与泄漏

在前面，我们讨论了闭包间基本的引用规则；也讨论到闭包与调用对象的生存周期，基本上也可以归结为“函数实例”的生存周期。

在编译语言，以及以编译成二进制代码为主的静态语言中，函数体总是在文件的代码段中，并在运行期被装入标志为“可执行”的内存区。因此，事实上我们不会认为代码、函数等等这些会有“生存周期”。我们在大多数时候会认为“引用类型的数据结构”——例如指针、对象等——具有引用、生存周期和泄漏的问题。同样，在 **JavaScript** 中我们也说过，函数也是一个引用类型的“数据 / 值”——尽管我们没有从这个角度上去论证它必然存在引用、生存周期与泄漏的问题，但事实上的确可以这样做。

除了这个原因之外，事实上我们也应该注意到，由于全局函数事实上也可以视为宿主对象的属性（例如浏览器中的 `window` 对象），因此很多情况下，**JavaScript** 中的函数都具有“作为方法时依赖于对象生存周期”和“作为闭包时依赖函数实例生存周期”这样双重的复杂性。

当然，也有独立于对象系统而存在的“纯粹的函数”，例如匿名函数、内嵌函数等。很多时候它们只用于运算，而并不赋值到某个对象成员（从而变成对象方法）。但如果一个函数不被显式 / 隐式地赋值，那么它的函数实例在使用之后就必然被废弃——因为没有被引用。换言之，引用正是“赋值”这样的运算带来的。

也许有人会质疑：函数返回值是否有引用效果呢？例如：

```
function myFunc() {  
  function foo() {  
    //...  
  }  
}
```

```
    return foo;
}
```

这个例子其实可以用于证明函数返回（`return` 子句）并不导致引用。因为如果我们这样来使用 `myFunc()` 函数：

```
myFunc()();
```

那么函数 `foo()` 在返回后立即被执行了，然后就会释放。因此 `return` 子句并不是导致引用的必然条件。相反，如果我们下面这样使用：

```
func = myFunc();    // <-- 在这里产生一个引用
func();
```

那么将产生引用，而后 `myFunc()` 与 `foo()` 的生存周期就将依赖于变量 `func` 的生存周期了。

除了上述这种被变量引用的情况，在 JavaScript 中最常见的其实是对象属性的引用。这主要指如下四种情况下：

- 👉 对象在构造时，使用 `this` 引用进入构造器函数；或
- 👉 对象在调用方法时，使用 `this` 引用进入函数；或
- 👉 某个函数使用 `call/apply` 方法调用，并传入某个对象作为 `this` 引用；或
- 👉 调用一个函数时，对象从参数入口传入，

在下面的代码中，我们以第四种情况为例来讲述一下函数实例被创建和引用的过程（注意下例中的匿名函数共产生了三个函数实例，但表现出来的生命周期并不一致）：

```
function MyObject(obj) {
    var foo = function() {
        // ...
    }

    if (!obj) return;
    obj.method = foo;
}

// 示例 1
MyObject();

// 示例 2
MyObject(new Object());
```

```
// 示例 3
var obj = new Object();
MyObject(obj);
```

在示例 1 中，`MyObject()`被调用，在函数内部有一个匿名函数的实例被创建，并被赋值给 `foo` 变量，但因为参数 `obj` 为 `undefined`，所以该函数实例没有被返回到 `MyObject()`函数外。因此 `MyObject()`执行结束后，闭包内的数据未被外部引用，因此闭包销毁，`foo` 引用指向的匿名函数也被销毁。

在示例 2 中，传入参数 `obj` 是一个有效的对象，于是匿名函数被赋值给 `obj.method`，因此建立了一个引用。在 `MyObject()`执行结束的时候，该匿名函数与 `MyObject()`都不能被销毁。但随后，由于传入的对象未被任何变量引用，因此立即销毁，`obj.method` 的引用得以释放。这时 `foo` 指向的匿名函数没有任何引用、`MyObject()`内部也没有其它数据被引用，因此开始销毁过程。

在示例 3 中开始的过程与示例 2 一致，但由于 `obj` 是一个在 `MyObject()`之外具有独立生存周期的外部变量，JavaScript 引擎必须对这种持有 `MyObject()`闭包中的 `foo` 变量（所指向的匿名函数实例）的关联关系加以持续地维护，直到该变量被销毁，或它的指定方法(`obj.method`)被重置、删除^①时，它对 `foo` 的引用才会得以释放。例如：

```
// 1. 重新置值时，关联关系被清除
obj.method = new Function();

// 2. 删除成员时，关联关系也被清除
delete obj.method;

// 3. 变量销毁(或重新置值)导致的关联关系清除
obj = null;
```

对于上述的第三种情况，在对象销毁时，该对象所持有的所有函数的闭包将失去对该对象的引用。而当一个函数实例的所有引用者都被销毁时，函数实例（及其闭包、调用对象）被销毁。

这看起来很完美：在纯粹的函数式语言中，在一个闭包中构造的对象只能被自己持有，或者通过函数返回来被其它闭包引用——这显然是能被引擎感知的——所以函数与闭包总能在确知的情况下被销毁。而在 JavaScript 中，由于函数内无法做到无副作用，因此它必须为每一个在函数内创建的对象创建一个

^① 该项不适用于使用 `var` 声明的变量，因为 `var` 声明的变量不能被 `delete` 操作删除。

引用列表，只有当所有函数内的对象（不仅仅是内嵌函数）都不再被引用时，该函数才处于自由（free）的状态。

然而由于有些对象总是不能被销毁（例如在 DOM 中存在的泄漏），或在销毁时不能通知到 JavaScript 引擎，因此也就有些 JavaScript 函数总是不能被销毁。这是在某些具体的宿主环境中，常常因为宿主的使用方法导致 JavaScript 存在泄漏的根源。一个通常的例子是：

```
// code from codeproject.com, By volkan.ozcelik.
<html>
<head>
<script type="text/javascript">
function LeakMemory() {
    var parentDiv = document.createElement("<div onclick='foo()'>");
    parentDiv.bigString = new Array(1000).join(
        new Array(2000).join("XXXXX"));
}
</script>
</head>
<body>
<input type="button" value="Memory Leaking Insert"
    onclick="LeakMemory()" />
</body>
</html>
```

这个例子中，LeakMemory()中创建了一个 DOM 对象 parentDiv，然后该对象又持有了在 LeakMemoery()中创建的一个大数组。当 LeakMemory()函数退出时，因为 DOM 并不释放临时的 parentDiv 对象，所以 JavaScript 也不能释放 LeakMemory 闭包和那个创建的大数组，于是产生了内存泄漏^①。

1.6.5. 函数实例拥有多个闭包的情况

一般情况下，一个函数实例只有一个闭包，当闭包中的数据（闭包上下文）没有被引用的情况下，该函数实例与闭包就被同时回收了——我们在此前也主要讲述这种情况。

但也存在函数实例有多个闭包的情况——这非常罕见。下面特别构造了这样一个例子来说明这种情况：

^① 这个泄漏在 IE7 中已经被修复。但如果你将 IE6/5 的 JScript 用在 IE7 中，也一样会出问题；反过来将 IE7 中装载 IE5/6 的 JScript 引擎，问题也不再存在。因此这个泄漏事实上是由宿主环境而非脚本引擎导致的。

```

1  var checker;
2
3  function myFunc() {
4      if (checker) {
5          checker();
6      }
7
8      alert('do myFunc: ' + str);
9      var str = 'test.';
10
11     if (!checker) {
12         checker = function() {
13             alert('do Check:' + str);
14         }
15     }
16
17     return arguments.callee;
18 }
19
20 // 连续执行两次 myFunc()
21 myFunc()();

```

在这个例子中，**myFunc** 函数将执行两次。在第 21 行的第一次执行结束时，在第 17 行代码处将“函数实例自身的一个引用”（**callee**）作为结果值返回；接下来，在 21 行处会遇到第二个函数调用运算符，于是 **myFunc** 函数就被第二次调用了。由于第二次执行的只是一个引用，因此这个示例中 **myFunc** 只创建过一个函数实例。

运行这个例子将输出三个信息：

```

do myFunc: undefined
do Check: test.
do myFunc: undefined

```

其中第一、三个信息都由第 8 行代码输出。输出 **undefined** 的原因在于：

- 👉 函数被调用时，函数内的局部变量被声明并被初始化为 **undefined**；
- 👉 局部变量表被保存在该函数闭包中的 **varDecls** 域中。

第 8 行代码用于检测该值——并且我们强调它的值的确是 **undefined**。但是我们也注意到第二个输出信息是 **"test."**，根据代码流程，这个值是第二次调用 **myFunc** 时输出的——因为在第一次调用 **myFunc** 时，**checker** 还被赋过值呢。

而 `checker` 是在第一次调用中被赋的值，它是一个局部函数引用；并且在它的闭包中，还引用了 `upvalue` 变量 `str`。由于这个变量是第一次调用的 `myFunc` 函数的闭包中的，因此在第一次 `myFunc` 调用结束后，闭包并没有被销毁——闭包中存在被其它对象引用的变量 / 数据。

所以，`myFunc` 第二次调用、并以函数形式调用全局变量 `checker` 时，`checker` 实际上是输出了“第一次 `myFunc` 调用过程中形成的闭包”中的 `str`——显然，这个值是 `"test."`。这就是第二行输出信息的由来。

这个例子传达出的信息是：

- 👉 JavaScript 中函数实例可能拥有多个闭包；
- 👉 JavaScript 中函数实例与闭包的生存周期是分别管理的；
- 👉 JavaScript 中函数被调用时总是初始化一个闭包；而上次调用中的闭包是否销毁，取决于该闭包中是否有被（其它闭包）引用的变量 / 数据。

我们注意到这里提及“函数实例与闭包的生存周期是分别管理的”。因此一个函数实例（以及其可能的多个引用）的生存周期，与闭包是没有直接关系的。换言之，会存在函数变量没有持有闭包的情况。这是因为：

- 👉 闭包创建自函数执行开始之时；接下来，
- 👉 在执行中闭包没有被其它对象引用；接下来，
- 👉 在函数执行结束时闭包被销毁了。

而这时函数实例及其引用（例如变量、对象成员、数组元素等）都还存在，只是没有与之对应的闭包了。所以第 1.6.3 小节的标题是“（在被调用时，）每个函数实例至少拥有一个闭包”，以强调“在调用过程中”这样的事实。

本小节则另外强调，函数实例在没有被调用时可能有对应的（一个或多个）闭包，也可能没有闭包——当然，还可能闭包失效了，但未被引擎的内存管理器回收，而这就不在本书要讨论的范围中了。

1.6.6. 语句或语句块中的闭包问题

一般情况下，当一个函数实例被创建时，它唯一对应的一个闭包也就被创建。在下面的代码中，由于外部的构造器函数被执行两次，因此内部的 `foo` 函数也被创建了两个函数实例（以及闭包）并赋值给 `this` 对象的成员：

```
function MyObject() {  
  function foo() {  
  }  
}
```

```

        this.method = foo;
    }

    obj1 = new MyObject();
    obj2 = new MyObject();

    // 显示 false, 表明产生了两个函数实例
    alert( obj1.method === obj2.method );

```

这在函数之外（语句级别）也具有完全相同的表现。在下面的例子中，多个匿名函数的实例被赋给了 `obj` 的成员：

```

var obj = new Object();
for (var i=0; i<5; i++) {
    obj['method' + i] = function() {
        //...
    };
}

// 显示 false, 表明产生了多个函数实例
alert( obj.method2 === obj.method3 );

```

尽管是多个实例，但它们仍然是共享同一个外层函数闭包中的 `upvalue` 值——在上例中，外层函数闭包指的是全局闭包。因此下面的例子所表现出来，仍然只是闭包中对 `upvalue` 值访问的规则，而并非闭包或函数实例的创建规则“出现了某种特例”^①：

```

var obj = new Object();
var events = {m1: "clicked", m2: "changed"};

for (e in events) {
    obj[e] = function(){
        alert(events[e]);
    };
}

// 显示 false, 表明是不同的函数实例
alert( obj.m1 === obj.m2 );

// 方法 m1() 与 m2() 都输出相同值
// 其原因，在于两个方法都访问全局闭包中的同一个 upvalue 值 e
obj.m1();

```

^① 这个例子引自一份存有误解的网络文档(<http://blog.csdn.net/g9yuayon/archive/2007/04/18/1568980.aspx>)。


```
obj.m2();
```

在这个例子中，方法 `m1` 与 `m2` 究竟输出何值，取决于前面的 `for..in` 语句在最后一次迭代中对变量 `e` 的置值。某些引擎中总是保证 `for..in` 的顺序与 `events` 中声明时的属性顺序一致（例如 `SpiderMonkey`），但也有一些引擎并没有这项约定。因此上例在不同的引擎中表现的结果未必一致，但 `m1()` 与 `m2()` 输出值总是相同的。

按照这段代码的本意，应该是每个函数实例输出不同值。对这个问题的处理方法之一，是再添加一个外层函数，利用“在函数内保存数据”的特性来为内部函数保存一个可访问的 `upvalue`：

```
var obj = new Object();
var events = {m1: "clicked", m2: "changed"};

for (e in events) {
  obj[e] = function(aValue) { // 闭包 lv1
    return function() { // 闭包 lv2
      alert(events[aValue]);
    }
  }(e);
}

/* 或用如下代码，在闭包内通过局部变量保存数据
for (e in events) {
  function() { // 闭包 lv1
    var aValue = e;
    obj[e] = function() { // 闭包 lv2
      alert(events[aValue]);
    }
  }();
}
*/

// 下面将输出不同的值
obj.m1();
obj.m2();
```

由于闭包 `lv2` 引用了闭包 `lv1` 中的入口参数，因此两个闭包存在了关联关系。在 `obj` 的方法未被清除之前，两个闭包都不会被销毁，但 `lv1` 为 `lv2` 保存了一个可供访问的 `upvalue`——这除了私有变量、函数之外，也包括它的传入参数。

很明显，上例的问题在于多了一层闭包，因此增加了系统消耗。不过这并非不可避免。我们要看清楚这个问题，其本质是要一个地方来保存 `for..in` 列举中的每一个值 `e`，而并非是“需要一个闭包来添加一个层”。那么，既然列举过程中产生了不同的函数实例，自然也可以将值 `e` 交给这些函数实例自己去保存：

```
var obj = new Object();
var events = {m1: "clicked", m2: "changed"};

for (e in events) {
  (obj[e] = function() {
    // arguments.callee 指向匿名函数自身
    alert(events[arguments.callee.aValue]);
  })
  .aValue = e;
}

// 下面将输出不同的值
obj.m1();
obj.m2();
```

1.6.7. 闭包中的标识符(变量)特例

在下面两个小节中：

👉 1.2.4.5 变量作用域中的次序问题

👉 1.2.4.6 变量作用域与变量的生存周期

在对函数内声明的变量进行阐述的时候，说标识符(变量)在定义后值为 `undefined`，并可以在函数内访问。例如：

```
var name = 'test';
function myFunc() {
  // 输出 1
  alert(name);

  // 输出 2
  var name = 100;
  alert(name);
}
```

在这个例子中，输出 1 不会访问到全局变量 `name`，是因为 `name` 已经在函数闭包内被声明（但未被赋值）过；输出 2 则输出该局部变量 `name` 赋值后的值。

但是完整地说来，函数闭包内的标识符系统应该包括：

- 👉 `this`
- 👉 局部变量 (`varDecls`)
- 👉 函数形式参数名 (`argsName`)
- 👉 `arguments`
- 👉 函数名 (`funcName`)

其中 `arguments` 是语言内定的标识符，无需声明即可使用的。

上面提供的标识符系统是有优先顺序的。这种优先顺序（及其隐含规则）甚至超出了我们前面讲到过的变量作用域规则、闭包规则。首先我们来看看下面这个例子：

```
// 示例 1: 输出值 'hi', 而非函数 foo.toString()
function foo(foo) {
    alert(foo);
}
foo('hi');
```

```
// 示例 2: 输出数值 100 的类型 'number', 而非参数对象 arguments 的类型 'object'
function foo2(arguments) {
    alert(typeof arguments);
}
foo2(100);
```

```
// 示例 3: 输出参数对象 arguments 的类型 'object'
// (注: 在 JScript 中, arguments 作为函数名可以被声明, 但调用该函数导致脚本引擎崩溃)
function arguments() {
    alert(typeof arguments);
}
arguments();
```

这几个示例表明：

- 👉 形式参数名优先于内置对象 `arguments`
- 👉 内置对象 `arguments` 优先于函数名
- 👉 综合上两项（以及参考示例 1 的效果），形式参数名优先于函数名

所以，从优先级上来看是 “`argsName > arguments > funcName`”。但是如果与局部变量 `varDecls` 比较来看，又会是如何呢？下例说明这种情况：

```
// 示例 4: 输出值 'test'
```

```
function foo(str) {  
    var str;  
    alert(str);  
}  
foo('test');
```

这个例子中，变量 `str` 显然没有作为局部变量(`varDecls`)处理，而是函数入口处的形式参数，所以才输出值 `'test'`。但接下来这个例子却又有点不同：

```
// 示例 5: 输出值'member'  
function foo(str) {  
    var str = 'member';  
    alert(str);  
}  
foo('test');
```

这个例子表明：局部变量(`varDecls`)优先级高于函数形式参数。这又与上一个例子是矛盾的。比较来看，我们可以直观地认为：

- 👉 当形式参数名与未赋值的局部变量名重复时，取形式参数值；
- 👉 当形式参数与有值的局部变量名重复时，取局部变量值。

我并不确知这样实现的根源，但是这显然带来了一种语义上的合理性：

```
// 示例 6: 输出值'test'  
function foo(str) {  
    var str = str;  
    alert(str);  
}  
foo('test');
```

如果仅认为“局部变量(`varDecls`)优先级高于函数形式参数”，那么该代码“`var str=str`”赋值运算的左右两侧都应该是局部变量 `str`；而局部变量声明后初值为 `undefined`，所以该行代码的结果“（看起来）应该”是 `undefined`。

然而这可能与我们的目的不一致^①，我们的目的，可能是将入口参数 `str` 赋值局部变量 `str`。那么，就只能是上述两条假设都同时能立，才能使接下来这行“`alert(str)`”既访问局部变量，又能输出有效值。这一语义上的合理性是：赋值运算符的左侧是局部变量——这是由 `var` 关键字决定的，而右侧是形式参数——这是因为此时局部变量 `str` “尚未赋值”。

这样一来，赋值语句就成立了，接下来 `str` 变成有值的局部变量，而 `alert()` 也就会访问它了。不过，这里的“有或没有赋值”是指显式的赋值操作。仅以

^① 当然，开发人员预期目的未必如此，但语言设计者总得为这样的代码而设定一个“可被理解的语义”。

变量声明来讲，它是会有一个 `undefined` 初值的，而“是否是 `undefined`”并不是上述判断的条件。

1.6.8. 函数对象的闭包及其效果

这里说的“函数对象”，是特指使用 `Function()` 构造器创建的函数。它与函数直接量声明、匿名函数不同，它在任意位置创建的实例，都处于全局闭包中。亦即是说，`Function()` 的实例的 `upvalue` 总是指向全局闭包。下例说明这一点：

```
var value = 'this is global.';

function myFunc() {
    var value = 'this is local.';
    var foo = new Function('\
        alert(value);\
    ');

    foo();
}

// 显示'this is global.', 表明 foo 访问到全局闭包中的 value 变量
myFunc();
```

这其中的原因，在于 `Function()` 构造器传入的参数全部都是字符串，因此不必要与函数局部变量建立引用。由此带来的一个便利是：在任意多层的函数内部，都可以通过 `Function()` 创建函数而不至于导致闭包不能释放。因此，以上一小节的示例而言，更加妥当的写法应该是：

```
var obj = new Object();
var events = {m1: "clicked", m2: "changed"};

for (e in events) {
    obj[e] = new Function('alert(events["' + e + '"])');
}

// 下面将输出不同的值
obj.m1();
obj.m2();
```

这样，系统就不会维护多余的闭包，也不会在函数实例上添加多余的成员了。不过你也应该注意到，这里用于构造函数的字符串

```
'alert(events["' + e + '"])'
```

中，变量 `e` 是一个字符串。这的确是一个限制：在 `Function()` 构造器中，在通过这个字符串参数来保证没有对变量的引用时，变量是被转换为字符串值来使用的——因此，如果该变量转换成字符串时会失去原意（例如 `Object/ActiveXObject` 等），那么这个方法显然不可行。

除开上述的限制，你还是可以拿这个特性来做很多事。例如下面的函数用来为你的函数定制一个特别的 `toString()` 效果：

```
function myFunction(name) {
    var context = [
        "return 'function ", name, "()\n",
        "{\n",
        "  [custom function]",
        "\n}"
    ];
    return new Function(context.join(''));
}

function aFunc() {
    //...
    //...
    //...
}

// 显示默认的 toString 效果
alert(aFunc);

// 显示一个定制的 toString 效果
aFunc.toString = myFunction('aFunc');
alert(aFunc);
```

因为使用了 `new Function()`，所以在 `myFunction()` 中产生的函数实例不会引用它的入口参数 `name`。这样一来，无论多少次调用 `myFunction()`，都不会建立更多的闭包——如果写成下面的代码，尽管结果一致，但系统需要维护的 `myFunction()` 实例 / 闭包数却会大量增加：

```
function myFunction(name) {
    return function() {
        return 'function ' + name +
            '()\n{\n  [custom function]\n}';
    }
}
```

1.6.9. 闭包与可见性

作为动态语言实现上的一种结果，语法作用域、变量作用域、变量生存周期等等在 JavaScript 中变成了不同的概念。本书分别在下面的章节中讲述过它们：

- 👉 语法作用域：语法形式上的结构化效果，讲述于“1.2.2 模块化的层次：语法作用域”；
- 👉 变量作用域：亦即变量的可见性^①，讲述于“1.2.4 模块化的效果：变量作用域”；
- 👉 变量生存周期：变量分配存储到销毁的过程，讲述于“1.6.4 函数闭包与调用对象”。

除了变量作用域（及其可见性）之外，在 JavaScript 中还存在对象成员的可见性问题——这是与 `with` 语句相关的语言特性。本章节将引入“对象闭包”的概念来解释这一特性。

1.6.9.1. 函数闭包带来的可见性效果

在表现上来看，JavaScript 对可见性的规定是：

- 👉 变量在全局声明(用或不用 `var`)，则在全局可见；
- 👉 变量在代码任何位置隐式声明(不用 `var`)，都在全局可见。
- 👉 变量在函数内显式声明(用 `var`)，则在函数内可见；

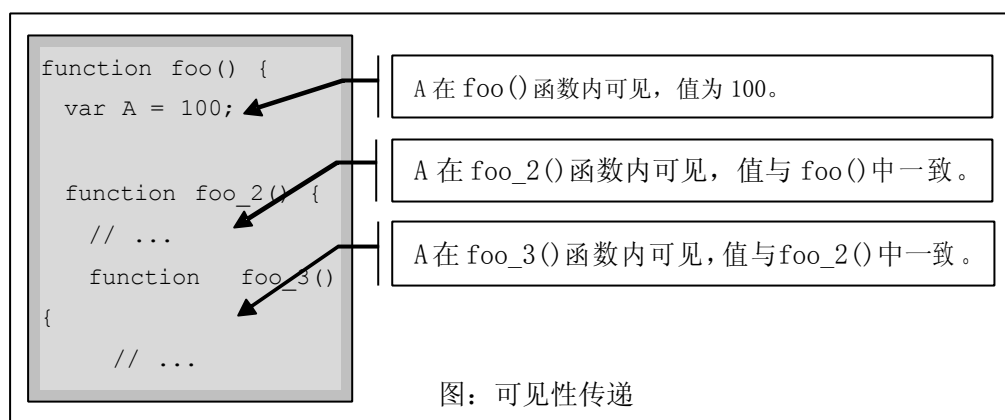
如下图所示：

^① 标识符可见性包括对象成员标识符、标签标识符、变量标识符等的可见性，因此它与变量作用域并不严格等义，而我们这里只强制变量的可见性。

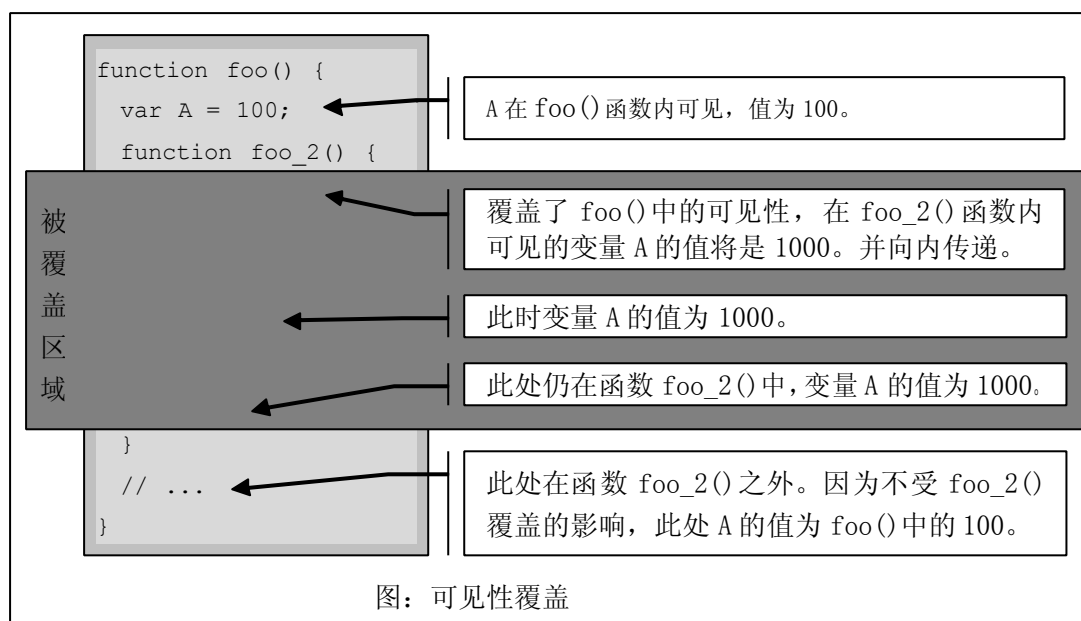
示例一	示例二	示例三
<pre>var v1 = 100; v2 = 1000; function foo() { // ... function foo_2() { // ... } }</pre>	<pre>function foo() { v1 = 100; function foo_2() { v2 = 1000; } }</pre>	<pre>function foo() { var v1 = 100; function foo_2() { var v2 = 1000; } }</pre>
可见性: 全局可见 原因: v1, v2 全局声明	可见性: 全局可见 原因: v1, v2 隐式声明	可见性: 函数内可见 原因: 函数内显式声明
图: 变量声明与可见性		

以下两条是对上述规则的补充:

- 👉 可见性传递: 变量在 A 函数内可见, 则在所有 A 的内层嵌套函数中可见。如下图所示:



- 👉 可见性覆盖: 在函数内显式声明变量 A, 将覆盖当前可见的同名函数。如下图所示:



但在具体的实现上，JavaScript 只要求在语法分析期：

- 👉 将一个函数代码体中所有用 `var` 关键字声明的变量记入它自己的 `ScriptObject` 的 `varDecls` 域，然后
- 👉 设定访问规则“如果当前函数的 `ScriptObject.varDecls` 域中不存在该变量声明，则通过‘闭包 `.parent`’来取得上一层函数的 `ScriptObject.varDecls` 作为 `upvalue`”

那么必然会得到 JavaScript 中那些“看起来非常复杂”语言特性。以上图为例：

- 👉 可见性传递的实质是 `foo_2()` 与 `foo_3()` 都访问到来自于 `foo()` 的同一份 `ScriptObject.varDecls`；
- 👉 可见性覆盖的实质是 `foo_3()` 在 `foo_2()` 的 `ScriptObject.varDecls` 中找到了名为 `A` 的变量，因而不必再向上层回溯；
- 👉 “变量在代码任何位置隐式声明(不用 `var`)，都在全局可见”的实质，是该变量标识符不在函数所有的 `parent` 的 `ScriptObject.varDecls` 中存在时，必然回溯至顶层的全局函数闭包（通常实现为宿主对象）的中的 `varDecls`，并在该位置“隐式地”声明了一个变量。

综上所述，我们在上一章讲述“JavaScript 的非函数式语言特性”时，讨论到的 JavaScript 中的“变量作用域”几个问题，包括（参见 1.2.4 模块化的效果：变量作用域）：

- 👉 只有表达式、局部和全部三种作用域；
- 👉 变量生存周期不依赖于语法的声明顺序；
- 👉 生存周期只有两个：函数内的局部执行期间，和函数外引擎的全局执行期间。

其中除了“表达式作用域”依赖于匿名函数的特性之外，其它的所有现象，

其实都是“函数闭包”这种特性带来的效果。更加确切地说，它们仅仅依赖于在“1.6.4 函数闭包与调用对象”中讲述的调用对象（ScriptObject）中的 varDecls 域的访问规则。

最后补充一点，正是由于每一个函数实例都有一份 ScriptObject 的副本，因此“不同的闭包访问到的私有变量不一致”。

1.6.9.2. 对象闭包带来的可见性效果

所谓“对象闭包”，是指使用 with 语句时与 with() 所指示对象相关的闭包，该闭包被动态创建并添加到执行环境当前的闭包链的顶端^①。如果我们将闭包的可见性理解为“闭包的 upvalue 系统”与“闭包内的标识符系统”，对象闭包与函数闭包在前者上是一致的，但对后者的处理并不相同。下表说明这种差异：

标识符系统	函数闭包	对象闭包	说明
this	有	没有	(*)
局部变量 (varDecls)	有		(**)
函数形式参数名 (argsName)	有	没有	仅与函数声明相关
arguments	有	没有	仅与函数调用过程相关
函数名/对象名 (funcName/objName)	有	没有	对象直接量不能具名
对象成员名	没有	有	(***)

(*) 在对象闭包中使用 this 引用，将通过 upvalue 访问到它上一层的函数中的 this 引用。

(**) 对象闭包中的 var 声明效果存在引擎差异。

(***) 在函数闭包中，arguments 也是函数的一个属性。但函数内能直接访问 arguments，是因为函数闭包将一个引用了该属性的变量添加到闭包标识符系统中，与（函数作为对象时的）“对象闭包”并没有什么关系。

在对象闭包中用 var 声明变量的效果，与具体的引擎实现有密切关系。在目前的考察中，JScript、SpiderMonkey JavaScript、Adobe ActionScript 等都将 with 语句中的 var 关键字理解为所在函数闭包中的变量声明。如下例：

```
// 示例 1
function foo() {
  with (this) {
    var value;
  }
  alert(value);
}
```

^① 在 SpiderMonkey 中，with 语句不是打开对象闭包的唯一方式。首先，用 Object.eval() 可以让一段代码执行在对象闭包中，例如 “obj.eval('value = 100')”；将在 obj 的对象闭包中执行代码。此外，Script() 对象也可以强制在一个对象的闭包中执行代码，例如 “new Script('value = 100').exec(obj)” 就与上面的作用是一致的。

```
}  
// 显示'undefined'  
foo();
```

由于“`var value`”被理解为 `foo()` 函数中的一个变量声明，且该变量初值为 `undefined`，因此该示例显示 `'undefined'`。

当该关键字 `var` 被理解为 `foo()` 中的变量声明时，就存在了一个问题：如果 `with` 所指示的对象闭包中存在同名的属性，那么 `value` 应当操作哪个标识符？例如：

```
// 示例 2  
var aObj = { value: 'hello' };  
function foo() {  
  with (aObj) {  
    var value = 1000;  
    alert(aObj.value); // 显示值: 1000  
  }  
  alert(value);  
}  
// 显示'undefined'  
foo();
```

在这个例子中，上述脚本引擎采用了一致的策略：由于对象 `aObj` 存在 `value` 属性，因此“`value = 1000`”将向 `aObj.value` 置值。也就是说，由于当前最顶层的闭包是 `aObj` 的对象闭包，且该闭包存在“`value`”这个标识符（或属性名），因此应当向它置值。

表面看起来这很合理，但它与上一项处理策略是冲突的。因为：

- 👉 “`var value`”向 `foo()` 函数的闭包声明了一个局部变量；而
- 👉 “`value = 1000`”操作的是 `aObj` 对象闭包中的一个属性

这使得在不同的阶段中，同一行语句存在面向两种不同目标的语义与执行效果。这显然是不合理的。

为了避免上述的困惑，一些引擎选用了别的方案。例如用 Apple Safari 中的 `JavaScriptCore`（基于 `KJS` 引擎）来测试示例 2 时，第二个输出就会显示值“1000”而不是“`undefined`”。这是因为 `KJS` 引擎系列将“`var value = 1000`”作为一个语义来理解，所以即使 `aObj.value` 属性存在，也将操作 `foo()` 函数闭包中的 `value`。

无论如何，上述这些引擎都认为在对象闭包中声明的标识符总是位于上层

的函数闭包中——换句话说，因为只有函数闭包具有 `varDecls`，所以也只有它才能接受标识符声明^①。但除此之外，也有一些脚本引擎在对象闭包中解释 `var` 的语义——这也意味着该引擎会在执行过程中使用 `var` 动态声明变量，例如 `DMonkey`。下面的代码说明在 `DMonkey` 中，`with` 语句中的 `var` 在对象闭包中声明了一个 `'value'` 变量标识符，且该标识符不是“对象的一个成员”：

```
// 示例 3: 测试 DMonkey 中的效果
var value = 10;
var aObj = { value: 'hello' };
function foo() {
  with (aObj) {
    var value = 1000;
    alert(aObj.value); // 输出字符串: 'hello'
    alert(value); // 输出值: 1000
  }
  // 上面的 var 只作用于对象闭包内部，所以这里的 value 指向了函数外的全局变量，输出值: 10
  alert(value);
}
foo();
```

`DMonkey` 与 `KJS` 相同之处在于“`var value = 1000`”语句都不会影响到 `aObj.value` 的值；不同的是，`KJS` 认为该行语句的效果作用于函数闭包 `foo()`，而 `DMonkey` 认为效果作用于 `aObj` 的对象闭包中。

对象闭包存在一个最明显的易用性问题，就是对“对象直接量”闭包中如何访问该对象自身。这个问题在匿名函数闭包中同样存在，但是我们总可以通过 `arguments.callee` 来访问到该匿名函数。但是对象闭包中没有 `arguments`，也没有（对该对象有效的）`this` 引用，那么下面的代码有什么办法去访问到 `with()` 操作的对象呢：

```
with ( {} ) { // <-- 这里声明了一个匿名的对象直接量
  // <--这里如何访问对象直接量自身呢?
}
```

在 `with` 语句中访问对象自身至少有三个目的：其一是为该对象添加成员，其二是访问下标（对于数组）或无法用标识符存取的成员，其三是使用类似于 `for..in` 这种需要操作对象的语句。这些问题可以通过将对象直接量赋给一个标识符的方法来解决，例如：

```
var x;
```

^① `VJS` 与 `SpiderMonkey JavaScript` 引擎共同存在的这种前提，使它们在处理 `with` 语句所打开的对象闭包中存在“条件化函数声明（conditional function declaration）”时达成了一致。关于条件化函数声明，请参见“1.4.2.1 语法声明与语句含义不一致的问题”。

```
with ( x={} ) { // <-- 这里声明了一个匿名的对象直接量
  // 通过变量 x 访问对象引用
}
```

也可以通过下面的 `self()` 函数来解决：

```
function self(x) {
  return x.self = x;
}

// 将所声明的对象直接量用作 self 的参数
with ( self({}) ) {
  // 通过 self 成员访问对象引用
  self.value = 100;
  for (var i in self) {
    // ...
  }
}
```

函数闭包与对象闭包既有相关性，也有各自的独立性。对象闭包总是动态地添加在闭包链的顶端，而函数闭包则依赖于函数声明时的、静态的语法作用域限制。由于这样的缘故，二者可能出现不完全一致的情况，这很容易让人产生（至少在表面上的）困惑。例如：

```
1  /**
2   * 两种闭包的交叉作用
3   */
4  var obj = { value: 200 };
5  var value = 1000;
6  with (obj) { // <-- 对象闭包
7    function foo() { // <-- 具名函数 foo() 的闭包
8      value *= 2;    // <-- 依赖于函数静态位置所决定的闭包链
9    }
10   foo();
11 }
12
13 // 显示 200
14 alert(obj.value);
15 // 显示 2000
16 alert(value);
```

在第 6 行，我们可能预期是要限定 6 至 11 行中所有对 `value` 存取都指向 `obj.value`。但由于 7 至 9 行在一个函数（的闭包）作用域里，因此 `with` 的限定事实上只对行 10 有效（尽管行 27 并不直接存取 `obj.value`）。于是，我们看到

第 8 行通过变量作用域的影响而访问到了全局变量 `value`：使它的值乘 2，变成了 2000。

这段代码中同时出现了两种闭包。在形式上来看，`foo()` 的函数闭包位于 `obj` 的对象闭包中，因此 `foo()` 中访问 `value` 时，应该通过闭包链来存取到 `obj.value`。但事实上，`foo()` 函数的闭包链是在语法分析期就决定了的：它被添加在全局闭包之后；而另一方面，`with()` 在运行期打开了 `obj` 的对象闭包，由于 `with` 语句也处在全局闭包中，因此 `obj` 的对象闭包就被“静态地”添加到了全局闭包之后。结果是，`obj` 对象闭包与 `foo()` 函数现在处于并列位置。所以在第 8 行就无法访问到 `obj.value` 了——`foo()` 函数的闭包链上找不到 `obj` 的对象闭包。

另一种存在两种闭包的情况是：在函数中打开函数对象自身的闭包。例如：

```
function foo() {  
  with (arguments.callee) {  
    // 这里即处理 foo() 函数的函数闭包中，又处于它作为对象时的对象闭包中  
  }  
}  
foo();
```

我们在“1.12 使用对象闭包来重置重写”的最后部分，在讲述到有关 `Function.prototype.bind()` 的技术时，其实就用到了这种两种闭包同时存在的效果，尽管它没有带来特别的“好处”，但的确让我们看到了两种闭包不同的性质。

1.6.9.3. 匿名函数的闭包与可见性效果

有一种做法可以避免上面提到的“`foo()` 函数中不能访问 `obj.value`”的问题。这种做法是：将函数 `foo()` 添加为对象 `obj` 的方法。例如：

```
1  (部分代码参见上例)  
2  with (obj) { // <-- 对象闭包  
3    obj.foo = function() { // <-- 匿名函数的闭包  
4      value *= 2; // <-- 依赖于函数闭包所在的当前闭包链  
5    }  
6    obj.foo();  
7  }
```

这样一来，`obj.value` 值将变成 400，而全局的 `value` 值不变。这是因为匿名函数的闭包也是如同对象闭包一样，动态地添加到当前闭包链的顶端——而代码执行到第 3 行时，“当前闭包”正好是 `with` 所打开的 `obj` 对象闭包。

由于这里的“添加到闭包链顶端”的行为只是（引擎针对于）匿名函数自身的行为，所以它与上述代码的赋值操作无关。也就是说，即使该匿名函数没有添加为对象 `obj` 的方法——而仅是即用即声明，那么它所操作的仍然是对象闭包中的 `value` 值。例如：

```
1    (部分代码参见上例)
2    with (obj) { // <-- 对象闭包
3        void function() { // <-- 函数闭包
4            value *= 2;
5        }();
6    }
```

更细致地追究这个问题，其实匿名函数“添加到闭包链顶端”也与“执行”无关，而是在它作为一个直接量被“创建”时，由引擎动态添加在闭包链上的。也就是说，匿名函数直接量的创建位置决定了它所在闭包链的位置。例如：

```
function foo() {
    function foo2() {
        // foo2() 函数内的局部变量声明
        // ...

        // 使外部变量 m 引用到该匿名函数
        m = function(varName) {
            return eval(varName);
        }
    }
    foo2();

    // aVarName 是 foo2() 中的任意局部变量名
    var m;
    var aFormatStr = 'the value is: $(aVarName)';
    var rx = /\$\{(.*)\}/g;
    alert(aFormatStr.replace(rx, function($0, varName) {
        return m(varName);
    }));
}
foo();
```

这个例子中，可以通过 `aFormatStr` 来取得任意 `foo2()` 函数中的局部变量的值。而前提是：让外部代码持有 `foo2()` 的一个匿名函数的引用。通过这种方法，`foo2()` 函数无论在任何位置——包括 `foo()` 的闭包内，或者任何 `foo()` 无法访问到

的位置，函数 `foo()` 都可以访问到它的内部成员^①。

1.7. 综述

特性一：语句存在返回值；

特性二：任何值(包括无值与函数)都参与运算；

特性三：闭包

特性四：语句的本质：在于描述表达式求值的逻辑，或者辅助表达式求值

特性五：当 JavaScript 作为一种表达式语言使用

特性六：`return` 是唯一明确定义返回值的语句，也是唯一能使“语句（仅指 `return` 子句）的返回值”直接参与到运算的语句。除此之外，所有的语句如果要参与运算，必须通过 `eval()` 函数——`eval()` 是 JavaScript 的动态语言特性。

<http://www.i170.com/user/yqj2065/Article-33958>

^① 这一技术被内置于 Qomo V2 的内核，并由此带来了一种类似于 PHP 中的允许 `echo()` 输出带有变量名的字符串、并动态将这些变量转换为值的技术。

第五章 JavaScript 的动态语言特性

动态和静态结合的主流是融合各个领域的特点。经典的、面向对象的、函数型的、动态的，我们从所有这些吸收可取之处，比起以前，生硬地嵌入（另一种语言的东西）将越来越不可取了。

——《微软架构师谈编程语言发展》，Anders Hejlsberg

1.1. 概述

显然，只需要在这个“绑定”的概念上加上限定词“代码执行期”，就可以得到动态和静态绑定的概念了。

JavaScript 中主要的动态语言特性包括：

- 👉 重写
- 👉 对象与数组的动态特性
- 👉 动态执行系统
- 👉 动态类型系统

1.1.1. 动态数据类型的起源

最早期的动态语言，据知是 1960 年由 Kenneth E.Iverson 在 IBM 设计的 APL，与同时期在贝尔实验室的 D.J.Farber、R.E.Griswold 和 F.P.Polensky 三人设计的 SNOBOL。这两种语言的共同特性表现为：动态类型声明和动态空间分配。

所谓动态类型声明，是指语言的变量是无类型的，只有当它们被赋值后才会具有某种类型；所谓动态空间分配，是指变量在赋值时才会为其分配空间。把这两种特性换成现在通常的概念，就是：变量可以理解为一个无类型指针，只有在指针被分配一个确定的内存空间时，才可以获知该指针指向内存区的大小，以及可能的数据类型(*)。

尽管《程序设计语言概念(Concepts of Programming Language)》中认为 APL 与 SNOBOL 对后期的语言并没有产生什么影响(***)，但除开直接确指的某种语言而言，“动态类型系统”思想的提出，对后来的编程系统确实具有不容小

视的影响。

在 COM 体系中的 variants(***)

(*)注：《程序设计语言概念 6th》中，这被称为“显式堆动态变量”，而 JavaScript 中的动态类型系统被称为“隐式堆动态变量”(p151)。不过所谓显式与隐式，只是在词法分析上是否具有明显的类型识别过程，并不强调是否采用相同的“动态”实现机制。

(**)注：SNOBOL4 是已知最早支持模式匹配的语言（COPL p179）；APL 则是至今所设计的最强大的数组处理语言（COPL p188）。

(***)注：《程序设计语言概念 6th》p147 中指出“为变量提供动态类型绑定的语言必须使用纯解释器实现”，而事实上 COM 设计理念中打破了这种规则，由于对类型的高度抽象与统一（我是指 IDL 对类型系统的规定），因此 COM 被设计为一个二进制规范，你显然可以用任何编译语言来提供 COM 组件以及使用其中的动态类型系统。

1.1.2. 动态执行系统的起源

1.1.1.1. 编译系统、解释系统与编码

自从第一份能够被有意义地书写于其它介质（我的意思是泛指计算机存储系统之外）的代码出现以来，一个重要的问题就被提了出来：要让计算机理解这份代码，就需要一个翻译系统。

翻译系统有编译器与解释器两类。一般情况下，编译器将代码翻译成计算机可以理解的、二进制的代码格式，并置入存储系统（例如存为二进制可执行文件）；解释器则用一个执行环境来读入代码，然后执行这份代码——这里主要是指单纯解释执行的语言系统。

对于解释执行的系统来说，显然我们不必要总是逐字符读入并解释、执行。由于一份代码如果被写定，那么执行时通常不需要改变，因此我们可以先将解释过程做一次，由源代码转换为中间代码(*)，然后执行系统只需要处理中间代码即可。这样的好处是，执行系统可以变成虚拟执行环境，在不同的平台上用各自的虚拟执行环境来处理相同的中间代码，即可实现跨平台应用——这也是 Java 和 .NET 的基本实现思路。

中间代码是较新的概念，它特别容易与纯编译器时代的操作码（Operation Code，OpCode）混淆。对于纯编译器来说，OpCode 所指的已经是机器码了。但中间语言也有它自己的 OpCode。例如 .NET 框架中的中间语言 (MSIL，Microsoft Intermediate Language)，就有其对应的 MSIL OpCode。一些并不使用中间语言机制的，也将虚拟执行环境中可运行的中间代码称为（某种专有的）OpCode，例如 PHP 的 Zend 编译器，就有一种 Zend OpCode。

语言系统、指令系统与操作码是三个不同但相关联的概念，例如 .NET 架构中的 MSIL，MSIL Instruction 和 MSIL OpCode。

但是直接执行中间语言仍然是效率极低的（尽管比执行源代码要高），因此出现了即时编译器（JIT，Just In Time），即时编译由于只处理中间语言而不需要做复杂的语法解释和错误处理，因此编译过程的实时性效好；而编译结果是本机的机器码，因此执行效率也很高。

动态执行系统一般依赖于解释和即时编译系统——不过目前的实现中，JavaScript 没有即时编译系统（例如 DMonkey 的所谓编译，只是保存代码的语法解释树）。

1.1.2.2. 动态执行

我们可以设计一种动态类型的语言，并让它被静态编译而不能被“动态执行”。尽管在早期，通常以“动态类型、动态绑定”作为对动态语言特性的基本约定，但在本书中将“动态执行”也作为这种语言的基本特性之一。

所谓的“动态执行”是指可以随时载入一段源代码文本并执行它。因此一种有“动态执行”能力的动态语言，需要解释系统的支持。

在计算机语言的早期历史中，人们就习惯于使用“动态执行”的方法来操作计算机系统了。

1.1.3. 脚本系统的起源

脚本系统最早不是作为“程序设计语言”的面貌出现的。早期的 Shell、批处理或者某些文字处理规则语言，都满足脚本系统的两个条件：

- 👉 脚本描述规则(不一定是语法)
- 👉 脚本解释和执行环境

但准确地来说，“脚本”与“脚本语言”并不是一回事。在实际使用中，某些录制的宏（例如录制键盘和鼠标操作），也是一种用于回放的“脚本”，但它们并不是“脚本语言”。

批处理是一种脚本语言，例如 DOS 批处理。这些批处理也具有语言的某些要素：关键字、逻辑语句或语法、声明和处理过程（函数或命令）。不过，DOS 批处理也具有更加专业的称谓：DOS Shell。批处理与 Shell 脚本没有明显的界限，一般只是称功能较弱或没有复杂逻辑能力的为批处理，更强的则称为 Shell 脚本——例如某些 UNIX Shell 比 DOS Shell 要强大得多。

再往前溯源，可以在 Unix 操作系统的历史中找到脚本系统的起源。在还没有出现 Unix 的时代，在 1965-68 年，AT&T(美国电话及电报公司)、G.E.（通用电器公司）和 MIT(麻省理工学院)推动了 Multics (MULTiplexed Information and Computing Service) 计划。在六十年代末，Bell Labs（贝尔实验室）也正式参加该项目，但又很快退出了。虽然后来这个计划以失败而告终，但正是 Bell Labs 的参与，使得 Ken Thompson 与为 Multics 研究小组的一员。接下来以 Thompson 为主要推动力，（至少）产生了两项巨大的影响：

- 👉 在操作系统史上，Thompson 为了让他在 Multics 计划中开发的一个名为“太空旅行(Space Travel)”的游戏程序能够在一台废弃的 PDP-7 机器上运行起来，着手写了一套操作系统(*)，这套操作系统名为 Unics (UNiplexed Information and Computing System)，取意于“un-MULTiplexed”。后来，在 1971 年间更名为 unix，成为现在众所周知的操作系统。
- 👉 在程序设计语言史上，Multics 基于当时电脑的主要操作方式“批处理(Batch Processing)”的一次处理多条指令的思想，开发了一个“Multics Command Language”(**)。后来 Thompson 在 PDP-7 上实现 Unics 时，引用这一构思，实现了第一个 unix shell (command interpreter)，诞生于 1971 年(***)。也是脚本语言类(shell)的最初起源(****)。

(*)可见偏执也是一种生产力。

(**)Multics Command Language 由 Peter Deutsch、Calvin Mooers、Christopher Strachey 等实现于 1967 年，也包括 E. L. Glaser, R. M. Graham, J. H. Saltzer 等的一些设计思想与实现。Multics Command Language 的更早的影响来自于 BESYS 和 CTSS 上的命令语言(Command language)，以及 TRAC T64 上的宏语言(Macro language)。

(***) 通常称为 Thompson shell，1971 年至 1975 年随 Unix 第一版至第六版发布。而我们常说的 sh，则是指 Stephen Bourne 在 1977 年在 Version 7 Unix 中针对大学与学院发布的 Bourne Shell。它用于替代 Thompson shell，不过它们的可执行程序的名字却是一样的。Bourne 或许更习惯于用"Shellish"来称之为

“外壳”，而更官方的释义，则称 sh 是一种 “Command shell interpreter and script language for Unix”。

(****)以 shell 作为脚本语言的起源，可以参考《程序设计语言——实践之路》P793 对 Perl 语言的起源解释。

晚至 1978 年，Bill Joy 在加州大学伯克利分校时编写了 C Shell，1979 年随 BSD 首次发布。同时期，在 unix 系统上还出现了一个名为 awk 的宏与文本处理语言（Macro and Text-processing language），也被普遍认为是一种脚本语言(*)。awk 主要用于处理文本，即是我们现在所谓正则表达式(RegExp, Regular Expression)的前身。而 awk 的设计思想就受到我们前面讲的动态数据类型语言 SNOBOL 的影响。

(*)它的创建者后来将它正式命名为 “样式扫描和处理语言”。

正是因为 awk 与 shell 这两种早期的脚本语言系统，使得许多在介绍 “脚本语言” 的文章总是解释 “系统管理员们是最早利用脚本语言的强大功能的人”，以及 “处理基于文本的记录是脚本语言最早的用处之一”。

不过如果从功用的角度上来讲，那么 shell、以及脚本语言最早受到的影响应该来自于 1960 年的 IBM360 系统中(*)，该系统中提供了一个任务控制语言（JCL, Job Control Language），其基本思想是 “用于控制其它程序（used of control other programmes）”。

(*)《CSCI: 4500/6500 Programming Languages - Scripting Languages Chapter 13》PDF

1.1.4. 脚本只是一种表面的表现形式

“JavaScript 是一种脚本语言” 这样的定义肯定是不不会错的。但是这样的定义并不确指它有什么特别的语言特性。因为 “脚本” 只是一种表现形式或者记述语法的形式，而并不用于限定特性。

简单的说，你可以将 PASCAL、C、PROLOG 这些语言等等全部实现成 “脚本语言”，但这种举措并没有对这些语言的实质有任何特别的改变。事实上，这些语言的确都有相应的脚本语言系统的实现。

以 unix 上的 sh 为代表的脚本语言，大约比 APL 和 SNOBOL 提出的 “动态类型系统” 晚出现约十年，因此我们不能将 “脚本语言” 与 “动态语言” 混

为一谈。本书在这一章中主要讨论“动态语言特性”，因此强调脚本只是一种表现形式——不过在大多数情况下它更适用于实现动态语言，并强调“脚本化”也非 JavaScript 这种语言（以及动态语言类型）的本质特征。同样，一些与脚本化相关的特性，也疏离了语言本质：

- 👉 JavaScript 是嵌入式的语言：JavaScript 的早期实现，以及现在主要的应用都是在嵌入在浏览器中，以浏览器为宿主的。但这并不代表 JavaScript 必须是一个嵌入式引擎。在一些解决方案中(*)，JavaScript 也可以作为通用语言来实现系统。事实上，JavaScript 引擎和语言本身，并不依赖“嵌入”的某些特性。
- 👉 JavaScript 是用作页面包含语言（HTML Embedded、ServerPage）：JavaScript 的主要实现的确如此。例如在 HTML 中使用<SCRIPT>标签来装载脚本代码，以及在 ASP 中使用 JScript 语言。但是，这种特性是应用的依赖，而非语言的依赖。大多数 JavaScript 引擎都提供一种 Shell 环境，可以直接从命令行或系统中装载脚本并执行(*)，而无需依赖宿主页面。

(*)例如 jslib。

(*)例如 WSH 中的 WScript.exe，以及用 Java 实现的 Riho 引擎。

除开这些表面的现象，我们将下面的一些特性归入动态语言的范畴，并在后面加以详述：

- 👉 解释而非编译：JavaScript 是解释执行的，它并不能编译成二进制文件。的确存在一些 JavaScript 的编码系统（encode），但并没有它的编译器。
- 👉 可以重写标识符：可以随时重写（除关键字以外的）标识符，以重新定义系统特性。这种特性也被称为“动态绑定”，但 JavaScript 重写的性质，比动态绑定所能做的更多。

其它的一些来自于动态语言系统自身的定义的特性，包括：

- 👉 动态类型系统：JavaScript 在运算过程中会根据运算符的需求或者系统默认的规则转换运算元的数据类型。此外，变量在声明时是无类型的，直到它被赋予某个有类型含义的值。所以 JavaScript 既是弱类型，也是动态类型的。
- 👉 动态执行：JavaScript 提供 eval()函数，用于动态解释一段文本，并在当前上下文环境中执行。
- 👉 丰富的数据外部表示：通常情况下你总是可以将一个变量序列化成字符串，即使是一些扩展的（非内置的）类型的数据，也可以通过定制它的外部表示方法。而反过来，你也总是可以通过直接量的方式来声明或创建一个数据。

这些也将在随后的章节中予以详述。

1.2. 动态执行(eval)

如前所说，动态执行起源很早。但基本可以分为动态装载与动态执行两个

阶段。例如 dos 批处理中，可以使用 `call` 命令来执行另一个批处理（并传入参数）：

```
// a.bat
echo now execute a.bat
call b.bat

// b.bat
echo now exec b.bat
dir *.exe
```

但是批处理只能按文件装入代码，而不能处理文件中的代码内容，例如并没有将 `b.bat` 中的 `"*.exe"` 替换成 `"*.txt"` 的指令 / 命令。

而在 JavaScript 中，动态执行的对象是“代码文本”，它将装载与执行分成两个阶段。对于后者来说，执行的只是一个字符串文本，至于该字符串文本是来自 internet，还是本地文件，并不是动态执行系统所密切关注的。

因此接下来主要只讲述动态执行^①，这主要是由 `eval()` 方法带来的效果。ECMA Script v3 规范对重写 `eval()` 方法作出了限制，因此你不应当重写该方法或通过它的引用来调用它^②。此外，`eval()` 的参数只接受（唯一一个）字符串值，如果参数是其它类型的数据——也括字符串对象实例，那么 `eval()` 只是原封不动地返回该值，而不会有其它任何效果。

1.2.1. 动态执行与闭包

JavaScript 的代码总是要运行在一个闭包环境中，这样它才会有一个 `ScriptObject` 用以访问当前闭包中的变量表与内嵌函数表（`varDecls` 和 `funDecls`），并且能够通过闭包的 `parent` 属性访问到外层闭包。这是我们上一章所讲述的主要内容之一。

由于代码总要运行在一个闭包中，因此 `eval()` 也需要一个闭包环境。然而这成了一个问题：系统有全局闭包、当前函数闭包两种。那么究竟应当为 `eval()` 的代码准备哪个闭包呢？

^① 动态装载在 JavaScript 中是由宿主提供的能力。例如在 WSH 中提供了 `FileSystem` 对象来装载本地文件，而在浏览器环境中提供了 `XMLHttpRequest`，或 `Microsoft.XMLHTTP` 这个 ActiveX 对象来装载文件——这正是 Ajax 在浏览器环境中得以实现的基础。

^② 不过在 JScript 与 SpiderMonkey JavaScript 中都忽略了这项规范，因此你可以重写它或将它赋值其它对象的成员、变量，并通过这些成员或变量来调用该函数。但是，重写后的新函数无法维护 `eval()` 方法对“当前函数闭包”的理解，因此新函数总是将代码执行在某一个确定的闭包环境中，这使得 `eval()` 基本没有重写的价值。

如果换成更加确切的问题，那么请思考下面的代码

```
// 示例 1
var i = 100;
function myFunc(ctx) {
    var i = 'test';
    eval('i = "hello."');
}
myFunc();

// 输出值 100
alert(i);
```

中，`eval()`的代码块，究竟是修改了值为 100 的全局变量 `i`，还是修改了值为 'test' 的局部变量 `i`？

根据经验，一般的开发人员都可以给出答案：它应该修改局部变量 `i`，使它从值 'test' 变成值 'hello.'。这的确没错，但我们想想，`eval()` 是 JavaScript 的全局对象 `Global` 提供的方法，而如果要访问 `Global` 对象的方法，可以通过宿主对象——在浏览器环境中是 `window`——来实现。也就是说，理论上讲，下面的代码与上面的示例应该是一致的：

```
// 示例 2
var i = 100;
function myFunc(ctx) {
    var i = 'test';
    window.eval('i = "hello."');
}
myFunc();

// (在 Mozilla 引擎中，)输出值 'hello.'
alert(i);
```

然而非常之不幸，示例 2 的代码在 IE 中效果同于示例 1，而在 Mozilla 中将会修改全局变量 `i`。我们知道在两种浏览器中，前者使用的是 JScript 引擎，而后者使用的是 SpiderMonkey JavaScript 引擎。而这个问题的根源，正在于不同的 JavaScript 引擎对 `eval()` 所使用的闭包环境的理解并不相同^①。

^① 在 SpiderMonkey 中，`eval()` 不仅是一个宿主对象 (`window` 或其它) 的方法，而且也是所有对象的一个方法。它可以如同 `with` 语句一样用于打开该对象的闭包。因此 `window.eval()` 的实际效果就是在 `window` 对象的闭包中执行代码。关于这一点，请参阅“1.6.2.2 对象闭包带来的可见性效果”。

1.2.1.1. eval 代码专享闭包

一些引擎在实现 eval 时，采用一种简省的方法来为 eval 的代码块创建闭包。这种简省的方法如下面的伪代码所示：

```
function _parseContext(ctx) {  
    // 对代码上下文分析，并返回语法树  
    return <语法树>  
}  
  
function _evaluate(scope, tree) {  
    // 按语法执行代码  
}  
  
function _eval(ctx) {  
    var scope = _current_scope; // 取 eval 所在环境中的当前闭包  
    var tree = _parseContext('function() {return (' + ctx + ')}');  
    return _evaluate(scope, tree);  
}
```

也就是说，它通过将代码包裹成匿名函数的形式，来使得代码可以在当前函数内部执行，并且可以访问当前函数闭包内的变量。如果将这种 eval() 展开，则会是如下的情况：

```
// 展开前  
function myFunc() {  
    var i = 100;  
    eval( 'i = "hello.'" );  
}  
  
// 展开后  
function myFunc() {  
    var i = 100;  
    (function() { // <-- 匿名函数  
        return (i = "hello.");  
    })();  
}
```

很显然，在这种方法中，该匿名函数可以正常地访问 myFunc() 的闭包，但是在该匿名函数中声明变量，对 myFunc 函数却不会有影响。例如：

```
// 展开前  
function myFunc() {  
    var i = 100;
```

```
eval( 'var i = "hello."' );
}
```

这个例子中的代码起码会出现两个问题：

- 👉 `eval()`的代码块中, `var` 声明将导致内部的匿名函数中出现一个变量 `i`。但开发人员的本意, 可能只是重新声明 `myFunc()`中的变量 `i`。因此这会有歧义的代码。
- 👉 由于 `return (var i = "hello.")`是一句语法错误的代码, 因此一般使用这种方法实现 `eval` 的引擎将会出现异常。

这种引擎的特点, 是不能在 `eval()`的代码块中, 在最外层的位置使用 `var` 声明变量。一方面这会导致语义问题, 另一方面, 在上面你已经看到了: 这种处理方法会导致代码不能被执行。

很遗憾的是, Mozilla Firefox 的早期版本, 与 Safari 使用的 WebKit (直至 v2.0) 都是使用的这种处理方法。

1.2.1.2. `eval` 可以使用全局闭包

允许 `eval()`使用全局闭包并不是一个什么好主意——因为从闭包的概念来说就不应当让 `eval()`对函数外产生副作用。但事实上连 Mozilla 的 JavaScript 引擎都具有这样的实现, 这表现为一种 `eval()`的使用技巧, 如下例:

```
// (本例建议在 mozilla firefox 环境中测试)
var i = 100;
function myFunc() {
    window.eval('i = "test"');
}
myFunc();

// 输出值'test', 表明调用 myFunc() 时修改了全局闭包中的变量 i
alert(i);
```

应该留意到这里的 `eval` 是使用的 `window.eval` 方法而非系统的 `eval` 函数。但是事实上差异并非由 `window` 对象 (或其方法) 导致, 换做下面的代码效果也是一样的:

```
eval.call(window, 'i = "test"');
```

由于后面这种情况只是指明调用 `eval()`中传入的 `this` 对象为宿主对象。因此事实上这是 Mozilla 中的 JavaScript 引擎针对宿主的一种特殊实现, 这种实现也的确是旨在给 `eval()`提供一种访问全局闭包的能力。

相对照的，在 Internet Explorer 中的 JScript 的 `eval()` 就没有这种能力。无论是使用 `window.eval` 调用，还是使用 `window` 作为传入的 `this` 实例，都不可能让 `eval()` 得到访问全局闭包的能力。不过在 JScript 中可以使用另一种方法来得到完全相同的效果，即在 `window.execScript()` 方法中执行的代码“总是”在全局闭包中执行。例如：

```
// (本例建议在 Internet Explorer 环境中测试)
var i = 100;
function myFunc() {
    window.execScript('i = "test"');
}
myFunc();

// 输出值'test'，表明调用 myFunc() 时修改了全局闭包中的变量 i
alert(i);
```

由于 `window` 是缺省对象，因此下面的代码是等效的：

```
execScript('i = "test"');
```

而且，JScript 运行在名为 **ActiveScript** 的脚本环境中，该 `execScript()` 方法也提供跨脚本语言的代码能力。因此也可以在第二个参数中指定语言的种类^①。例如：

```
execScript('i = "test"', 'JScript');
或
execScript('dim i = "test"', 'VBScript');
```

JScript 引擎使用 `execScript()` 来将 `eval` 在全局闭包与函数闭包中的不同表现隔离开来，而 Mozilla 的 JavaScript 引擎则使用 `eval()` 函数的不同调用形式来区分它们。二种实现方法确有不同，但对此具有相同理解的是：在全局闭包与函数闭包中动态执行代码时，应该具有不同的表现和效果。

就目前所知，包括 KJS 等引擎在内，除了上述的差异之外，在使用 `eval()` 动态执行时，它们都采用第三种方案“在 `eval` 中使用当前函数的闭包”。

1.2.1.3. `eval` 使用当前函数的闭包

一般情况下，`eval()` 总是使用当前函数的闭包。基本上来说，这是最理想的情况，这种设计的实现效果，就如同本小节的第一个示例所展示的：

^① 我们通常见到的 Internet Explorer 中的脚本语言是 JScript 和 VBScript。但事实上可以通过 WSH 加载更多的脚本系统，因此第二参数可以变成“LUA”、“Python”或者其它。这取决于你的（或浏览器用户的）系统中是否安装有该种脚本引擎。关于这些可用引擎的列表可以参看：<http://www.mvps.org/scripting/languages/>

```
// 示例 1
var i = 100;
function myFunc(ctx) {
    var i = 'test';
    eval('i = "hello."');
}
myFunc();

// 输出值 100
alert(i);
```

由于 `eval()` 使用当前函数闭包，所以上例中影响到了局部变量 `i`，而全局变量并不发生变化。当然下面的推论也就是正确的了：在 `eval()` 代码中使用 `var` 声明的变量是局部变量。例如：

```
// 示例 2
var i = 100;
function myFunc(ctx) {
    var i = 'test';
    eval('var test = "hello."');
}
myFunc();

// 输出值 100
alert(i);
```

然而接下来的问题就复杂了。既然 `eval()` 代码中允许声明局部变量，那么就必须保证代码的语法树能在 `eval()` 过程中动态维护——这可能已经超出了一些语言系统在语法与语义上的理解能力。

为了详细解释这种动态执行能力的复杂性，我们再复述一下 JavaScript 代码解释执行的过程。

- 👉 在 JavaScript 中，代码文本是先被解释为语法树，然后按照语法树来执行的；
- 👉 在每次执行语法树中的一个函数（的实例）时，会将语法树中与该函数相关的形式参数、函数局部变量、`upvalue` 以及子函数等信息复制到一个结构中，该结构称为 `ScriptObject`（调用对象）；
- 👉 `ScriptObject` 动态关联到一个闭包，闭包与 `ScriptObject` 具有不同的生存周期；
- 👉 按照语法树来执行函数体中的代码，需要访问变量时，先考察 `ScriptObject` 中的局部变量等，最后考察 `upvalue`。

在这个过程中，闭包是用来创建并存储值的一套系统；`ScriptObject` 是用来考察标识符和脚本内容的一套系统。而我们在上面这个例子中，由于 `eval()`

试图在当前闭包环境中新声明一个变量，因此该变量名需要被添加到 `ScriptObject` 中，而它的一个值则需要被存放在当前闭包。

这显然意味着 `ScriptObject` 与闭包都是需要动态维护的。然而，不同的脚本引擎对此的理解也并不一致。例如在 KJS(safari)中，被 `eval()` 执行的代码块中就不能在最外层用 `var` 声明一个变量，也不能声明一个命名的局部函数。这其中的原因，就在于它不能动态地维护 `ScriptObject` 中的这个标识符系统。

1.2.2. 动态执行过程中的语句、表达式与值

`eval()`总是将被执行的代码文本视为一个“代码块 (block)”，代码块中包含的是语句、复合语句或语句组。我们在前面讨论语法时讲述过：在“语句”这种语法结构中，表达式可以解释为表达式语句。因此，代码块事实上就变成了“由单个或多个语句组成”。

我们知道，在 JavaScript 中语句是存在返回值的，该值由执行中的最后一个子句/表达式的值决定（除空语句和语句中的控制子句之外）。由此得到的推论是：`eval()`总是试图返回所执行的代码块的值——由代码块的最后一个语句决定。下面是第一种的情况（没有返回值）：

```
// 例 1. 返回值为 undefined
alert(eval('for (var i=0; i<10; i++);'));
```

这个例子中，循环的代码体（空语句），以及循环的控制子句(三个表达式)都不会有返回值，所以返回 `undefined`。也许部分读者会认为是空语句在返回 `undefined`。但事实上并不是这样——空语句不影响返回值^①，如下例：

```
// 例 2. 返回值为 6
alert(eval('i=6; for (var i=0; i<10; i++);;;'));
```

返回值为赋值表达式语句“`i=6;`”的结果值，也由此可见，循环体中的、以及其后的空语句都没有影响到该返回值。

在以前的章节中，我们说到直接量也可以被理解为单值表达式和单值表达式语句。因此下面两行示例在语义上其实完全一致：

```
// 例 3. 返回值为 6
alert(eval('6;'));
alert(eval('6'));
```

也许读者会认为第二行是在返回数字值“6”，因此第二行代码只是表达式或值

^① 这可能是一项隐式的规则，也可能只是在编译期就经过优化而扔掉了空语句。

而非语句，并由此推论出它与第一行代码存在语义上的不一致。其实不然，事实上第二行'6'之后还应该有一个虚拟的代码文本结束符（用在语法分析器中，表示文本结束），而这个文本结束符其实与语句分隔符";"等义，用来表明语句结束。因此，对于解释器来说，上面两行代码的分析结果都会得到“单值表达式语句”，从而存在语句返回值。

所以，这里要强调的是：`eval()`表示的是一个（动态的）语句执行系统，而非一个动态的“取值 / 赋值”系统。“返回值”只是 JavaScript 中语句的副作用导致的效果，而并不是为了“值 / 对象的序列化”而专门设计的。

由此带来的问题却相当令人烦恼。例如，尽管我们可以用下面的代码来得到字符串、数值或布尔值：

```
// 例 4. 用 eval() 来获取值的一般方法
alert(eval('true'));
alert(eval('"this is a string."'));
alert(eval('3'));
```

等等，但是我们不能用同样的代码来得到一个直接量的对象：

```
// 例 5. 用 eval() 来获取“对象直接量”的错误方法
alert(eval('{ name: "MyName", value: 1 }'));
```

要知道，同样的代码如果不是放在 `eval()` 中，就会是正常的：

```
obj = { name: "MyName", value: 1 };
```

这其中的原因，就在于 `eval()` 其实是将下面代码：

```
{ name: "MyName", value: 1 }
```

中的一对大括号视为一个复合语句的标识，因此接下来：

- 👉 第一个冒号成了“标签声明”标识符；
- 👉 （“标签声明”的左操作数）`name` 成了标签；
- 👉 `"MyName"` 成了字符串直接量；
- 👉 `value` 成了一个变量标识符；
- 👉 对第二个冒号不能合理地作语法分析，出现语法分析期异常；
- 👉

同样的原理（与语法分析流程），下面的代码不会异常，但返回的却是数值"1"，而不是一个对象：

```
// 例 6. 用 eval() 来获取“对象直接量”的错误方法，返回数值 1
alert(eval('{ value: 1 }'));
```

解决这个问题的方法，是将这里的直接量声明（的单值表达式），变成一般表达式语句。明确地指出“由大括号引导”的一段代码是“值”的方法，是使用强制表达式运算符“()”。例如：

```
// 例 7. 用 eval() 来获取“对象直接量”的正确方法，返回对象
alert(eval('{ value: 1 }'));
```

这样一来，由于表达式运算符“()”的操作数必须是表达式或值，因此相当于强制声明了“由大括号引导”的一段代码表示对象声明。而这里的表达式运算符，则表明它自己是一个一般表达式。再综合我们前面所讲述过的，`eval()`就将这段代码视作了由单个“一般表达式语句”组成的“语句块”。

不过在 JScript 中存在一个例外：函数直接量（这里指匿名函数）不能通过这种方式来获得。例如下面的代码：

```
var func = eval('(function() { })');
// 输出'undefined'
alert(typeof func);
```

这种情况下，可以具名函数来得到它^①。例如：

```
var func;
eval('function func() { }');
// 输出'function'
alert(typeof func);
```

但是你可能遇到必须使用匿名函数的情况（不打算使用上例那样确定的函数名），这时就需要稍稍复杂一点的代码（使用表达式运算）。例如：

```
// var func = eval('(function() { }).prototype.constructor');
// var func = eval('({$:function() { }}).$.');
// 或
var func = eval('【function() { }】[0]');

// 输出'function'
alert(typeof func);
```

1.2.3. 奇特的、甚至是负面的影响

1.2.3.1. 变量作用域可能变化

```
var i = 100;
```

^① SpiderMonkey JavaScript 正好相反：可以用 `eval()` 返回一个匿名函数，而对具名函数却只返回 `undefined`。回顾前面所讲的内容，这仍然是由两个引擎对“函数声明的语句含义”的理解不同所导致的。

```
function myFunc(ctx) {  
    alert('value is: ' + i);  
    eval(ctx);  
    alert('value is: ' + i);  
}
```

在这段代码中，并不能保证两次 `alert()` 的显示值一致。因为 `eval()` 使用了 `myFunc()` 的闭包和调用对象，因此也有机会来修改它。例如：

```
myFunc('var i = 10;');
```

代码的执行效果将是显示：

```
value is: 100  
value is: 10
```

动态作用域规则在早期解释性语言中被采用，后来则被大多数语言放弃。不过现在仍能在一些 `shell` 类的脚本系统中找到它应用（例如 `unix shell`、`bash`）

1.2.3.2. 代码的不可编译性

支持动态执行的代码是不能真实编译的——这里的真实编译，是指编译成为二进制的机器代码。这是因为我们动态执行的代码是面向一个标识符系统的，例如下面这行代码：

```
eval('myName = "...";');
```

这行代码中包括：

- 👉 标识符 `myName`;
- 👉 运算符 `=`
- 👉 字符串直接量 `"..."`

如果是针对机器代码的执行系统，那么运算符应该对应于某个函数，而标识符与直接量都应该对应于某个存储地址——也就是回归到“计算系统 = PDIO”中的 PD（Processing, Data）问题中。

然而，标识符系统与存储系统在编译过程中来说，是一张命名与地址对照表。接下来的问题就是：如果编译代码持有对照表，则该代码是可逆的；如果编译代码不持有对照表，则执行系统无法在运行期查找标识符——因此上面的动态执行将无法实现。

与此相关的技术和解决方案包括：

- 👉 像 .NET 或 JAVA 一样提供中间代码和虚拟机，中间代码中包括标识符，但逆

向工程后代码（例如在.NET 中的中间汇编语言）的可读性会降低；

- 👉 像早期解释性语言的伪编译系统一样，编译过程只用于形成语法树，标识符被包含在该伪编译代码中，这种情况下的代码是完全可逆的。

在现在的 JavaScript 领域中，JScript 提供一个名为 JSEncode 的工具程序，但它连编译器都不算（只是一种编码系统，是完全可解码的）；在 DMonkey 等一些 JavaScript 引擎中，提供编译目标为“语法树+标识符表”系统的伪编译方案；而在 JScript.NET 和 DLR 中，则提供基于.NET 的、编译为中间语言的解决方案。所有的这些系统，都未能（也不可能）在“支持动态执行代码”的前提下实现真实编译。

1.3. 动态方法调用(call 与 apply)

JavaScript 中有三种执行体^①。一种是 `eval()` 函数入口参数中指定的字符串，该字符串总是被作为当前函数上下文中的语句来执行（在某些引擎中允许作为全局代码中的语句执行）；第二种是 `new Function()` 中传入的字符串，该字符串总是被作为一个全局的、匿名函数闭包中的语句行被执行；第三种情况，执行体就是一个函数，可以通过函数调用运算符“`()`”来执行。

由于第二种与第三种都是面向某个函数的，因此他们存在一个完全相同的动态执行机制：除了使用函数调用运算符来执行之外，也可以使用 `apply()` 和 `call()` 方法作为动态方法来执行。

下例说明这种执行的一般性用法：

```
function foo(name) {  
    alert('hi, ' + name);  
}  
  
// 示例：call 与 apply 的一般性使用  
foo.call(null, 'Guest');  
foo.apply(null, ['Guest']);
```

在这个例子中，函数 `foo()` 使用 `call()` 与 `apply()` 方法来进行动态调用时的效果并没有什么不同。其实这两个方法的差异也仅仅在于调用时的参数不同——其中，`apply()` 的第二个参数传入一个数组或 `arguments` 值，因此具有相当的灵活性，而且一般来说效率也较 `call()` 方法稍高。

^① 在“包装类”的章节中你将看到在 JavaScript 中的第四种，此外在“类型转换”的章节中，你会看到由宿主对象系统带来的第五种执行体。

1.3.1. 动态方法调用中指定 **this** 对象

我们曾经说过 JavaScript 中并没有严格的“方法”。所谓的“对象方法”，只是用成员存取运算符——句点(.)和中括号([])——找到对象成员，然后将该成员作为函数执行而已。调用对象方法，与作为普通函数调用该成员并没有本质的不同，惟一的差异仅在于二者传入的 **this** 引用。

所以，如果我们要将一个普通函数作为一个对象的方法调用，或者将 A 对象的方法作为 B 对象的方法调用，惟一要做的，也仅是改变一下 **this** 引用。在 JavaScript 中，`apply()`和 `call()`都提供这样的能力——它们的第一个参数用于指定 **this** 对象。在前面的例子中，我们指定为 `null` 值，这表明执行中传入缺省的宿主对象。

下例说明指定 **this** 对象来进行动态方法调用的一般方法：

```
function foo() {  
    alert(this.name);  
}  
  
var obj1 = { name: 'obj1' }  
var obj2 = new Object();  
obj2.name = 'obj2';  
  
foo.call(obj1);  
foo.apply(obj2);
```

这个例子中，`obj1` 与 `obj2` 的效果并没有什么不同——同时你也应该注意到我们省略了 `call()`、`apply()`第二个以后的参数，这相当于在调用 `foo()`时不传入任何的参数。

由于我们在方法调用中能查询 **this** 引用以得到当前的实例。因此我们也能用下面的代码来传送 **this** 引用：

```
function foo() {  
    alert(this.name);  
}  
  
function MyObject() {  
    this.name = 'MyObject';  
}  
MyObject.prototype.doAction = function() {  
    foo.call(this);  
}
```

```

}

// 测试
var obj3 = new Object();
obj3.doAction();

```

在 `doAction()` 中，我们将调用这个方法时传入的 `this` 值传递给了 `foo()` 函数——在最后的测试中，这个 `this` 就是 `obj3`（不过由于初始化的问题，它的 `name` 是 `'MyObject'`）。

用同样的方法，我们也可以方便的传递参数。例如：

```

function calc_area(w,h) {
    alert( w * h );
}

function Area() {
    this.name = 'MyObject';
}
Area.prototype.doCalc = function(v1, v2) {
    calc_area.call(this, v1, v2);
}

// 调用 calc_area() 来运算面积
var area = new Area();
area.doCalc(10, 20);

```

这里我们使用了 `call()` 方法，因此需要声明 `v1,v2` 两个形式参数。这对于书写更通用的方法（或更基础的类）来说要能有点多余。那么我们也可以用下面的方法实现 `doCalc()`：

```

Area.prototype.doCalc = function() {
    calc_area.apply(this, arguments);
}

```

正是由于 `apply()` 方法的存在，我们也可以构造一个数组作为参数传入。一个相对复杂的例子是这样：

```

Area.prototype.doCalc = function(v1) {
    var slice = Array.prototype.slice;
    calc_area.apply(this, [v1*2].concat(slice.call(arguments, 1)));
}

```

在这个例子中，`slice` 指向 `Array` 原型中的一个函数（方法与普通函数是一致的），然后我们调用 `slice()`，取出 `arguments` 中第二个及其后的元素，并返回到一个

数组中。最后我们将这个数组合并到一个数组直接量（只有一个值：v1 值乘 2）的尾部。这些操作（的最后的 `concat()` 运算）返回的，正是一个数组，因此就可以用在 `calc_area.apply()` 的调用中了。

不过，如果仅是为了“将 v1 的值乘以 2”这样的目的，我们其实并不需要如此大费周章。我们应该注意到的是，这里 `apply()` 与 `call()` 约定的数组参数或 `arguments`，其实是一个对象，而非一个值。那么，我们是不是可以直接修改对象的成员（数组 / `arguments` 中的元素）呢？答案是：可以。如下例：

```
Area.prototype.doCalc = function(v1) {  
    v1 *= 2;  
    calc_area.apply(this, arguments);  
}
```

JavaScript 中，形式参数直接引用自 `arguments`。因此修改形式参数（例如这里的 v1）是会直接影响到 `arguments` 的。因此下面的代码与之完全相同：

```
Area.prototype.doCalc = function() { // <-- 注意这里没有声明形式参数  
    arguments[0] *= 2;  
    calc_area.apply(this, arguments);  
}
```

1.1.1. 丢失的 this 引用

在早期我曾以为宿主环境会限制一些变量引用自其它某些特殊对象的方法。例如在 Firefox 中执行下面的代码：

```
// 示例 1  
print = document.writeln;  
print('this is a test.');
```

会导致异常，而在 IE 中则没有问题。但后来我发现这并不是宿主环境或脚本引擎的限制，而仅仅是因为宿主对某些对象方法的实现存在不同。上面代码异常的原因，仅是因为 Firefox 实现 `writeln()` 这个方法时，在方法内需要使用 `this` 引用^①——IE 的 `writeln()` 则不需要，因此表现正常。

同样的问题也会出现在我们自己编写的代码中。很多时候，方法声明是放在构造器声明之外的，这种方法更容易被“意外地”直接调用。例如：

```
// 示例 2  
function calc() {  
    alert(this.getValue() * 2);  
}
```

^① Firefox 宿主以及 SpiderMonkey JavaScript 引擎对 `this` 的使用有很多奇特之处，例如上一节我们讲述过的 `eval()` 对 `this` 对象的识别问题。

```

}

function MyObject() {
    var value = 100;
    this.getValue = function() {
        return value;
    }
    // 方法引用自外部的一个函数
    this.calc = calc;
}

// 作为方法调用是正常的
var obj = new MyObject();
obj.calc();
// calc 直接调用时出错，与示例 1 在 Firefox 中异常是同一问题
calc();

```

所以上一小节的技巧也可以用于解决这些问题。对于示例 1 来说，我们可以为 Firefox 实现一个与 IE 兼容的 `writeln()` 方法：

```

// 示例 3：在 Firefox 中的 document.writeln() 的兼容性修补
document.writeln = function(f) {
    return function(){
        f.apply(document, arguments);
    }
}(document.writeln);

print = document.writeln;
print('this is a test.');
```

1.3.2. 栈的可见与修改

上面的例子带来了一种风险：既然 `arguments` 是一个对象，那么我们在被调用函数中修改其元素，是不是会影响到调用者中的参数值呢？

答案是否定的。下面的例子说明这一点：

```

function func_1(v1) {
    v1 = 100;
}

function func_2(name) {
    func_1.apply(this, arguments);
}
```

```

    alert(name);
}

// 显示传入参数未被修改，值仍为'MyName'
func_2('MyName');

```

尽管看起来 `func_1` 与 `func_2` 中使用的 `arguments` 是同一个，但事实上在 `func_1.apply()` 调用时，`arguments` 被做了一次复制：值数据被复制，引用数据被创建引用。因此，`func_1` 与 `func_2` 中的 `arguments` 看起来是相同的，其实却是被隔离的两套数据。

但这种隔离的效果非常微弱。因为在上例中，对于 `calc_area()` 这个函数，它在被调用时，它的栈上就放着 `doCalc()` 这个函数。我们其实是可以访问到 `doCalc()` 的，例如我们把上面的 `func_1()` 改成下面这样：

```

function func_1() {
    alert( arguments.callee.caller === func_2 );
}

```

那么当执行 `func_2()` 时，`func_1()` 就会显示值 `true`。所以外层的函数对于内部调用的函数来说，是可见的。

因此，风险仍然存在：尽管 `arguments` 在 `apply()` 与 `call()` 时是通过复制加以隔离的，但是调用栈对于被调用函数来说仍然可见，被调用函数仍然可以访问栈上的 `arguments`。如下例：

```

function func_3() {
    arguments.callee.caller.arguments[0] = 100;
}

function func_4(name) {
    func_3();
    alert(name);
}

// 显示传入参数的值被修改为数值 100
func_4('MyName');

```

在 `func_3()` 中，我们通过 `callee` 与 `caller` 访问到栈上的函数的参数，我们修改了 `func_4()` 中的形式参数 `name`。然而，这个行为对于 `func_4()` 来说却并不知晓。因此这是极端危险的。

在实现时，类 `Arguments` 与 `Array` 通常是共享一个父类的——这是因为它

们都有一个需要自维护的 `length` 属性。因此我们事实上也可以将 `Array` 原型中的某些方法 `apply()` 在 `Arguments` 的实例上。这也是上一小节中

```
slice.call(arguments, 1)
```

这行代码能被成功调用的根本原因。然而这也进一步加大了调用栈上的风险：我们不但能修改 `arguments` 中某些参数的值，也可修改 `arguments` 传入值的个数。如下例：

```
function func_5() {
    Array.prototype.push.call(arguments.callee.caller.arguments, 100);
}

function func_6(name) {
    func_5();
    alert(arguments.length);
}

// 显示传入的参数数已经变成 2
func_6('MyName');
```

类似的 `Array` 原型方法还可以包括 `push`、`pop`、`shift`、`unshift`、`splice`、`sort` 等，当然也可以包括基于这些方法的一些更复杂的实现。请留意这里还包括 `sort()` 函数，所以事实上这种方法还可以改变 `arguments` 传入值的顺序：

```
function func_7() {
    Array.prototype.sort.call(arguments.callee.caller.arguments);
}

function func_8(v1, v2, v3) {
    func_7();
    alert([v1, v2, v3]);
}

// 显示结果为 1,3,5，参数顺序发生了变化
func_8(1, 5, 3);
```

1.3.3. 兼容性：低版本中的 `call()` 与 `apply()`

`call()` 与 `apply()` 究竟是运行在当前函数上下文中，亦或者是全局域中呢？这个问题关系到 `call()` 与 `apply()` 可访问的函数闭包。答案是：`call()` 与 `apply()` 总是保证函数运行在“函数自己所在的闭包环境”中。

更加简单的说法是：函数总是运行在自己所在的闭包环境中，这与它是否是通过 `call()` 或 `apply()` 调用无关。这与上一章所述的内容是一致的：函数的闭包，是由函数实例和函数在代码文本中的物理位置所决定的。

然而这带来了一些对低版本进行兼容性改造的问题。举例来说，在 JScript 5.0 中，函数是没有 `call()` 与 `apply()` 方法的，因此我们一般需要通过下面的方法来实现兼容：

```
1 // code from http://hexmen.com/blog/, by Ash Searle
2
3 Function.prototype.apply = function(thisArg, argArray) {
4     // ...
5     thisArg = (thisArg == null) ? window : Object(thisArg);
6     thisArg.__applyTemp__ = this;
7
8     // youngpup's hack
9     var parameters = [], length = (argArray || '').length >>> 0;
10    for (var i = 0; i < length; i++) {
11        parameters[i] = 'argArray[' + i + ']';
12    }
13
14    var functionCall = 'thisArg.__applyTemp__(' + parameters + ')';
15    try {
16        return eval(functionCall)
17    } finally {
18        try { delete thisArg.__applyTemp__ } catch (e) { /* ignore */ }
19    }
20 }
21
22 Function.prototype.call = function(thisArg) {
23     return this.apply(thisArg,
24         Array.prototype.slice.apply(arguments, [1]));
25 }
```

这个兼容性实现中，行 6 用于将当前函数（`this`）变成 `thisArg` 所指对象的一个 `__applyTemp__` 方法，而第 18 行用于在退出 `apply()` 之前删除该方法；第 11 行则使用一个局部变量 `parameters[]` 来暂存 `argArray` 中传入的各个参数，第 14 行则用于将 `parameters[]` 拼接成一行代码文本，使之类似于如下代码：

```
thisArg.__applyTemp__(argArray[0], argArray[1], ...);
```

最后，在第 16 行时使用 `eval()` 来执行上面这行代码。由于 `eval()` 执行时使用的是当前函数的闭包（参见章节“1.2.1 动态执行与闭包”），因此可以访问到

thisArg、argArray 等标识符。

然而这样的兼容性 apply()实现忽略了一个问题：不同引擎的 eval()中，对被执行代码中的函数调用栈的理解，是不相同的。确切地说，因为并没有约定 eval()中的函数调用对栈的理解，因此各种引擎之间无法达成一致。

下面的代码用于显示被 eval()执行的函数的 arguments 与 caller 引用：

```
function foo() {  
    alert(arguments.callee.caller);  
}  
  
function myFunc() {  
    eval('foo()');  
}  
  
myFunc();
```

这段代码在 JScript 中将显示 null，而在 SpiderMonkey JavaScript 中将显示 myFunc()。也就是说：

- 👉 JScript 中，eval()中的代码运行在 myFunc()的闭包中，但没有继承它的调用栈；
- 👉 SpiderMonkey JavaScript 中，eval()既保证代码运行在 myFunc()闭包中，也保证继承 myFunc()的调用栈。

由此带来的问题是：在 JScript 中如果用前面的方案来实现 apply()与 call()，那么被调用的函数 __applyTemp__()将无法访问调用栈。而你应该知道，真正的 apply()却没有这个限制。例如：

```
function foo() {  
    alert(arguments.callee.caller);  
}  
  
function myFunc() {  
    foo.apply(this);  
}  
  
myFunc();
```

这在 JScript 与 SpiderMonkey JavaScript 中都将显示 myFunc()。

于是我们不得不用另外一种方法，来使得 JScript 低版本中的 apply()兼容性实现，具有能访问调用栈的能力——这就是使用函数调用。下面的例子省略了上面关于拼接代码文本的部分，重在强调对调用栈的构造：

```

1  Function.prototype.apply = function(thisArg, argArray) {
2      // 有关于该代码文本和__applyTemp__方法的代码，请参见上例
3      var functionCall = 'return thisArg.__applyTemp__(argArray[0], ...)';
4
5      var __local_function = eval('(function() {' + functionCall + '})');
6      return __local_function();
7  }

```

这个 `apply()` 的实现中，`__local_function()` 是 `apply()` 闭包中的一个局部变量(嵌套的、匿名的函数)，因此可以访问到 `apply()` 闭包中的各个标识符；又由于 `__local_function()` 是被作为函数调用的（第 6 行），因此它可以继承 `apply()` 的调用栈。

在这样的实现方案中，剩下的最后一个问题是：在目标函数（例如前面例子中的 `foo()` 函数）中，访问 `caller` 将总是得到 `apply()` 函数——而事实上它应该得到更靠栈底的那个函数（例如前面例子中的 `myFunc()` 函数）。

这个问题只能通过另一种方案来解决，以实现更安全的 `caller` 访问。在这种访问中，必须跳过 `apply()` 与 `call()` 函数。例如我们可以用下面的代码：

```

__safe_caller(foo, 2);
// 或
__safe_caller(arguments.callee, 1);

```

来替换下面这种常见的用法：

```

foo.caller.caller
// 或
arguments.callee.caller

```

这里的 `__safe_caller()` 的实现代码如下：

```

function __safe_caller(foo, lv) {
    while (foo && lv>0) {
        while ((foo = foo.caller) &&
            ((foo == foo.apply) ||
             (foo == foo.call) ||
             (foo.caller == foo.apply))) { /* null loop */ };
        lv--;
    }
    return foo;
}

```

1.4. 重写

JavaScript 的重写是一个代码执行期的行为，在代码的语法分析期，引擎既不对重写行为进行任何的预期，也不对其进行限制。因此，重写可能发生的问题之一，就是在运行中会发现冲突，或因为错误的、意外的重写而导致不可预料的代码逻辑错误。

针对原型和构造器的重写，会影响到重写前创建的实例的一些重要特性——例如继承性的识别。因此这种重写通常被要求在引擎最先装入的代码中进行。令人遗憾的是，开发人员通常无法保证这一点。所以在多个 JavaScript 扩展框架复合使用的情况下，经常会因此而出现不可控制的局面。

在不同的 JavaScript 语言环境中，存在的重写限制是不相同的。但是基本上来说，在引擎级别上的重写限制主要还是指保留字与运算符（这当然可以理解，起码我们所知道的许多语言都有这种限制）。但是在引擎之外，我们却不得不面临更多的限制，本小节对这些引擎之外的重写限制也会略有提及。

重写是 JavaScript 中各种语言特性相互作用的粘合剂。例如对象系统就依赖原型重写与原型修改来构造大型的继承系统，而对象成员的 `delete` 操作也是重写行为中的一种。但本小节不讨论它们。有关它们的叙述，请参见：

- 👉 1.5.2 对象成员列举、存取和删除
- 👉 1.4.8 原型继承的实质
- 👉 1.5 JavaScript 的对象系统

1.4.1. 原型重写

原型继承的一些问题是难于规避的，例如原型重写。正因为原型是可以重写的，所以事实上你可以用同一个构造器构造出两个完全不同的实例来。下例说明这种情况：

```
1  function MyObject() {  
2  }  
3  var obj1 = new MyObject();  
4  
5  MyObject.prototype.type = 'MyObject';  
6  MyObject.prototype.value = 'test';  
7  var obj2 = new MyObject();  
8
```

```

9   MyObject.prototype = {
10     constructor: MyObject,    //<-- 重写原型时应维护该属性
11     type: 'Bird',
12     fly: function() { /* ... */ }
13   }
14   var obj3 = new MyObject();
15
16   // 显示对象的属性
17   alert(obj1.type);
18   alert(obj2.type);
19   alert(obj3.type);

```

在这个例子中，1~7 行代码创建了 obj1 与 obj2 两个对象，由于行 5~6 只进行了“原型修改”，因此根据小节“1.4.8 原型继承的实质”中讲述的原理，我们知道 obj1 与 obj2 是“类同的”两个实例。第 9~13 行则将原型重写成为一个新的对象。不幸的是，这种重写事实上破坏了原型继承链，其直接的结果就是：obj3 与 obj1、obj2 完全不同——这种不同甚至直接扩展到继承关系的识别上。例如：

```

20   // (续上例)
21   // 显示 false
22   alert(obj1 instanceof MyObject);
23   alert(obj2 instanceof MyObject);
24   // 显示 true
25   alert(obj3 instanceof MyObject);

```

前两行代码显示 false 会让人疑惑，因为从上下文来看，obj1 与 obj2 的确是 MyObject() 构造器的实例。

如果我们去追究这个问题更深层面的原因，那么我们或许愿意去考察一下对象实例的 constructor 属性。然而这将更趋复杂：

```

26   // (续上例)
27   // 显示 false
28   alert(obj1 instanceof obj1.constructor);
29   // 显示 true
30   alert(obj1.constructor === MyObject);

```

第 28 行显示 false，潜在的含义是说“该对象不是由它的构造器构造器”；而第 30 显示 true，则只是重复了第 22~23 行的结论：因为它们的构造器仍然指向 MyObject()。

第 28 行代码表现出来的问题是：我们无法保证在 JavaScript 中，对象与其

构造器“必然”存在某种相似性。然而你应该知道，这已经违背了“面向对象系统”的基本原则。

在 JavaScript 中，一个构造器的原型是可以被重写的，重写意味着“此前的一个原型被废弃”。该构造器的实例中：

- 👉 旧的实例使用这个被废弃的原型，并受该原型的影响；
- 👉 新创建的实例则使用重写后的原型，受新原型的影响。

由于重写的存在，使同一个构造器可能有多套原型系统，这虽然给系统带来了非常大的复杂性，但在本质上并没有违反原型继承的任何规则。

1.4.2. 构造器重写

上一小节讨论重写构造器原型，是一种隐式地导致原型继承关系丢失的情况。事实上，显式的构造器修改也会导致继承关系丢失。不过相对而言，这比上一种情况要“合理”一些。如下例：

```
1 // 示例 1: 重写 - 执行期重写声明过的标识符
2 function MyObject() {
3 }
4 var obj1 = new MyObject();
5
6 MyObject = function() {
7 }
8 var obj2 = new MyObject();
9
10 // 显示 true
11 alert(obj2 instanceof MyObject);
12 // 显示 false
13 alert(obj1 instanceof MyObject);
14 // 显示 true
15 alert(obj1 instanceof obj1.constructor);
```

这个例子中，由于在第 6 行重写了 `MyObject()`，因此在进行 `instanceof` 检查时，（在第三行中）使用先前的 `MyObject()` 创建的 `obj1` 不能通过检查。这个问题可以归结为标识符被重写，固而第 13 行中的 `MyObject` 不是 `obj1` 创建时的 `MyObject`，基本上是一个合情合理的解释。而另一方面，第 15 行显示 `true` 值，也表明这种重写并不影响该实例的继承关系。

所以第 13 行表现出来的“（显式的）继承关系丢失”可以说是一种代码逻辑

辑上的假象。只是由于开发人员可能并不清楚何时发生了重写——例如被一个切面系统（Aspect，参考“1.6AOP”中有关的概念），在代码不能感知的情况下重写了构造器——因而可能会导致后续代码的一些意外。

还有一种构造器（函数）的重写会更早地发生，就是具名函数的重复声明。上例的一个修改版本如下：

```
1 // 示例 2: 语法分析期的覆盖
2 function MyObject() {
3 }
4 var obj1 = new MyObject();
5
6 function MyObject() {
7 }
8 var obj2 = new MyObject();
9
10 // 显示 true
11 alert(obj2 instanceof MyObject);
12 // 显示 true
13 alert(obj1 instanceof MyObject);
14 // 显示 true
15 alert(obj1 instanceof obj1.constructor);
```

从代码的物理结构来看 obj1 与 obj2 是两个不同的 MyObject()构造器创建的对象实例。然而在逻辑上来讲，这两个函数声明在语法分析期就出现了标识符覆盖，也就是后面的声明覆盖了前面的。因此在执行期就只有后一个声明是真正的 MyObject()了，因此 obj1 与 obj2 其实都是（物理顺序中的）第二个 MyObject()构造器的实例。

然而一点细微的差异就会改变上述的事实：

```
1 // 示例 3: 重写 - 执行期重写变量
2 MyObject = function() {
3 }
4 var obj1 = new MyObject();
5
6 // (以下同示例 1)
```

这个例子与示例 1 的效果是一致的，代码上的不同仅在第 2~3 行变成了赋值语句。因此 MyObject 是赋值的变量，而不是语法分析期的标识符。比较它与例 2，则是将“语法声明覆盖”变成了“变量重写”，而这两者的效果是炯然不同的。

示例 2 除了会带来视觉效果上的误解之外，并没有什么副作用——不过你在调试系统时也许会为之深受困扰。示例 1 与示例 3 是同质的，因此导致的问题与现象也是一致的。

1.1.1.1. 语法声明与语句含义不一致的问题

示例 2 展示的“语法声明阶段重写构造器”还存在一个问题，就是语法声明与语句含义不一致的问题。所谓“语法声明”与“语句含义”非常难于陈述清楚，请先看下面的代码：

```
// 示例 4：通过赋值发生的重写
// 重写
(Object = function()) {
  }).prototype.value = 100;

// 显示值 100
var obj = new Object();
alert(obj.value);
```

这段代码比较容易理解，因为重写是一行赋值语句，那么它的返回值也就是重写后的构造器函数，上面的代码为该函数的 `prototype` 属性声明了一个 `value` 属性，置初值 100。所以该构造器的实例 `obj` 的 `value` 属性值就是 100。

然而我们换用一下重写的方法，变成“语法声明阶段重写”呢？如下：

```
// 示例 5：语法声明阶段重写
// 重写
(function Object()) {
  }).prototype.value = 100;

// 显示值 undefined
var obj = new Object();
alert(obj.value);
```

两段代码的结构完全一致，重写效果也一致（都发生在 `obj` 实例构造之前），但为什么新的例子中 `obj` 的 `value` 值就是 `undefined` 了呢？——准确的说，为什么它没有 `value` 成员？

这里的问题的根源，在于函数直接量的语句含义与我们预期的并不一致。对于一个函数直接量来说：

```
function Object() {
  // ...
```

```
}
```

它表明在语法分析期声明一个名为“Object”的标识符，并使该标识符指向其内部的代码上下文（的语法分析树）。这些行为在语法分析期就已经完成了——这也就是重写的过程。然而直接量还有一层语句含义：一个直接表达式。由于它也是直接量表达式，所以当我们强制它为表达式运算

```
(function Object(){  
  // ...  
}) // <-- 这一对括号是强制表达式运算符
```

时，它就有了一个值含义，因此也就可以访问它的 `prototype` 成员：

```
(/* 函数直接量 */) .prototype
```

但是问题在于这个时候——在执行期的、将直接量作为表达式的时候——它没有了语法期命名的含义，所以事实上：

```
(function Object(){  
  // ...  
})
```

完全等效于一个匿名函数：

```
(function (){ // <-- 在作为语句的表达式执行时，没有“声明/重写标识符”的副作用  
  // ...  
})
```

所以代码：

```
(function Object(){  
  // ...  
}) .prototype.value = 100;
```

事实上是在“执行期”修改一个匿名函数的原型(`prototype` 属性)。而该匿名函数在该行代码之后没有任何的引用，也就被立即释放了，未对系统有任何的影响。

下面的用于检测语法声明与它的语句含义是否相同的引用：

```
// 示例 6：检测表达式的语法声明效果  
var MyObject2 = (function MyObject(){  
  // ...  
});  
  
// 显示 false，表明不是同一个引用  
alert( MyObject2 === MyObject );
```

而下面的代码则显示 `MyObject2` 的代码体，注意它与 `MyObject` 是不一样的：


```
// (续上例)
alert( MyObject2 );
```

代码执行后的显示结果为（这个结果非常奇怪）：

```
(function MyObject(){
    // ...
})
```

同样的问题也出现在下面这样的代码中：

```
var obj = {};
obj.foo = function foo() {
    // 显示 true, 代码内容相同
    alert(foo.toString() === arguments.callee.toString());
    // 显示 false, foo() 函数不是当前的调用者
    alert(foo === arguments.callee);
}
obj.foo();
```

obj.foo 方法与函数 foo 不是同一个引用。尽管他们的 toString()值相同（而上例是 toString()值不同），但这个问题与上例是一致的。

然而，在 SpiderMonkey JavaScript 中，上面的代码会有其它表现。其关键原因在于：SpiderMonkey JavaScript 不承认在表达式中声明的标识符^①。也就是说，下面的语句不会有声明标识符的效果，也就不会有 Object()构造器的重写：

```
// 示例 7: (在 SpiderMonkey JavaScript 中)不能导致 Object() 重写
myFunc = (function Object() {
    // ...
})();
```

即使去掉上面的表达式运算符“()”，Object()构造器也不会被重写。因为“=”号表明赋值运算，所以右边也会被强制理解为表达式：

```
// 示例 8: (在 SpiderMonkey JavaScript 中)不能导致 Object() 重写
myFunc = function Object() {
    // ...
};
```

正因为 SpiderMonkey JavaScript 不承认在表达式中声明的标识符，所以要在语法声明阶段重写一个构造器，绝对不能将这个具名函数声明“嵌入到”任何表达式中。因此在 SpiderMonkey JavaScript 中测试时，本小节前面的示例 6

^① 在 SpiderMonkey JavaScript 中被称为“函数表达式（function expression）”。

会显示错误信息“`MyObject` 未定义”，而示例 5 则根本不会有 `Object()`构造器重写的效果。

需要强调的是，虽然 SpiderMonkey JavaScript 不承认表达式中声明的标识符，但如果表达式被作为“表达式语句”来理解时，它仍然是具有标识符含义的。这表现为另外一种特性，该特性在 SpiderMonkey JavaScript 中被称为“条件化函数声明（conditional function declaration）”，一方面它不具有直接函数声明（function declaration）在语法解释周期声明标识符的特性，另一方面有具有在执行期隐式地声明标识符的特性，如下例^①：

```
// foo(); // this throws an error
if (true)
  function foo() { alert('hi.') }
foo();
```

在 SpiderMonkey JavaScript 中，被注释的第一行 `foo()`调用会产生异常^②，因为第三行的 `foo()`函数是有条件的声明的。当条件为 `true` 时，函数被成功声明，因此最末一行是可以成功执行的；当条件为 `false` 时，最末一行也会抛出异常。

但是从本质上来说，这只是因为 SpiderMonkey JavaScript 支持在运行期对语句中声明的函数作解析。而 `if` 语句后的这个函数声明被理为“直接量表达式语句”，作为语句的语法元素时，SpiderMonkey JavaScript 就能按照函数声明的语法效果了进行理解了。同样的，即使是在 `if` 语句之后，如果将它作为表达式中的运算元，也是不能达到声明函数的效果的。例如：

```
if (true)
  ( function foo() { alert('hi.') } ); // <--作为连续表达式运算的运算元
foo(); // this throws an error
```

1.4.2.2. 对象检测的麻烦

在 JavaScript 中，我们可以有两种方法来检测一个变量是否是对象：

```
var v1 = new String();
var v2 = 'string';

// 方法 1: 显示 true
alert( v1 instanceof Object );

// 方法 2: 显示 'object'
alert( typeof v1 );
```

^① 《Core JavaScript 1.5 Reference》

^② JScrip 则在语法分析期解析函数声明的标识符，因此这里不会出错。

```
// 对于字符串直接量，以下检测表明它不是一个'对象' (String 构造器的实例)
// 显示 false
alert( v2 instanceof Object );
// 显示'string'
alert( typeof v2 );
```

然而，如果我们重写了 `Object()`，那么方法 1 的检测将不能通过。也就是说，使用 `new` 运算与构造器（除 `Object` 自身）来构造的对象将存在误判。例如：

```
function Object() {
}

var v1 = new String();
var v2 = new RegExp();
var v3 = new Array();
// ...

// 显示 false
alert( v1 instanceof Object );
alert( v2 instanceof Object );
alert( v3 instanceof Object );
```

在大型系统中，这种困扰是令人不快的。因为我们的确有可能通过构造器来检测变量并做出响应——这不但包括系统内置的构造器，也包括用户声明的构造器函数。一个实际的例子是这样：

```
function checkInstance(obj) {
    if (obj instanceof Array)
        //...
    else if (obj instanceof String)
        // ...
    else if (obj instanceof Object)
        // ...
    else
        // ...
}
```

如果使用“检测 `constructor` 属性”这样的技术，代码也可能是这样：

```
function checkInstance_2(obj) {
    switch (obj.constructor) {
        Array : ...
        String : ...
```

```
Object : ...  
default : ...  
}  
}
```

凡上述这种使用构造器（函数）的标识符来直接引用它的，都可能存在问题。解决的方法，是保留一个局部的引用以避免其它的代码重写这些构造器：

```
checkInstance_2 = function() {  
    var _Array = Array;  
    var _String = String;  
    var _Object = Object;  
  
    // 返回一个匿名函数  
    return function(obj) {  
        switch (obj.constructor) {  
            _Array : ...  
            _String : ...  
            _Object : ...  
            default : ...  
        }  
    }  
}  
}(); // <-- 不要忘记这个函数调用运算符
```

这样一来，我们总是可以保证 `checkInstance_2()` 在系统中具有它的一致性：无论在它初始化之前 `Array` 等构造器是否重写，在它初始化之后总是以当时的构造器引用来检测 `obj.constructor`。

然而，最致命的问题则出现在其它两个方面：

1. 重写构造器之后，值类型的包装类使用哪个构造器？
2. 重写构造器之后，直接量声明的是哪个构造器的实例？

这些问题仍然将在后文有关包装类的章节中详述。

1.4.2.3. 构造器的原型(`prototype` 属性)不受重写影响

更为致命的是：`JavaScript` 规范并没有对“重写内置构造器^①”的效果作出任何解释。例如，如果我们重写了 `Object()` 这个构造器，那么系统还能构造出

^① `Object`、`Array` 之类并不是保留字，因此被重写是正常的。不过总会带来一些困扰。

对象来吗^①？

当然是能的。而且如果重写了 `Object()`——以及其它的内置构造器，那么当使用“`new Constructor()`”这样的语法来构造对象系统时，仍然是正常的。但是，既然我们重写了 `Object()`，那么 `prototype` 又从哪里来呢？

答案是：构造器的原型创建自系统内部的对象系统，而不是可被外部覆盖的标识符 `Object`，因此原型总能被创建。下例说明这一点：

```
// 1. 备份一个系统内部的 Object()
var PureObject = Object;

// 2. 重写
Object = function() {
}

// 3. 声明构造器
function MyObject() {
}

// 4. 删除 constructor 成员，以便访问到父类的同一属性
delete MyObject.prototype.constructor;

// 5. 显示 true，表明构造器的原型(对象)创建自 PureObject;
alert( MyObject.prototype.constructor == PureObject );
```

所以构造器——我们这里指任意的构造器函数——的原型属性，并不受 `Object()` 类的重写的影响，它总是创建自一个系统引擎中的对象构造器。

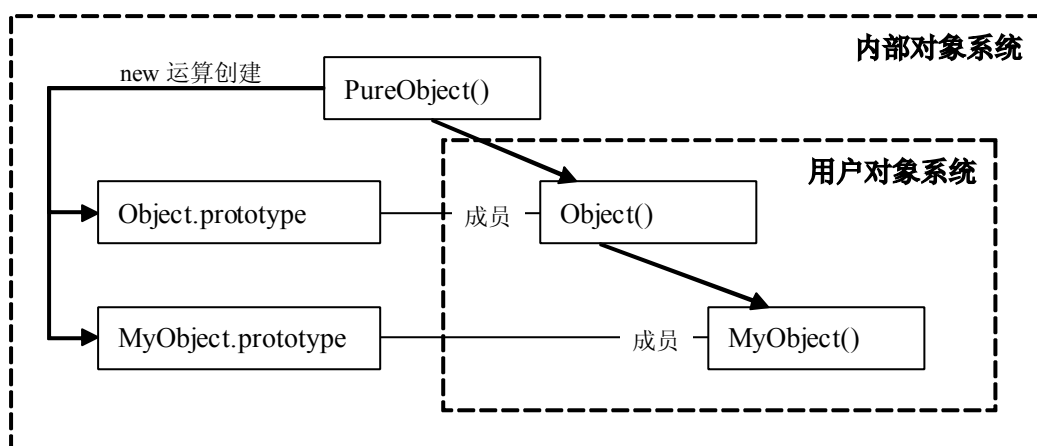
1.4.2.4. “内部对象系统” 不受影响

在上面的例子中，我们备份了一个引擎中原始的、真实的包装类的引用。如果我们修改这个备份（示例中的 `PureObject`）的 `prototype` 属性，会有什么样的效果呢？

对于 `PureObject` 来说，一方面，从引擎的角度上来说，它是系统所有对象的根原型；另一方面，标识型 `Object` 指向的构造器已经被重写了，这意味着用户希望基于一个新的祖先类来构建对象系统。

所以问题也具有两面性。为了陈述这两个方面，我们把以 `PureObject` 为祖先类的称为“内部对象系统”，而以重写后的 `Object` 为祖先类的称为“用户对对象系统”。那么其基本关系如下：

^① 要知道，在“1.4.5 构造过程：从函数到构造器”章节中，我们说过“构造器的 `prototype` 属性总是（缺省地）来自于 `new Object()` 产生的实例”——这里说“缺省地”是指如果你没有重写这个 `prototype` 属性的话。



上图表明当用户试图通过重写 `Object()` 来创建一个“用户对象系统”的时候，他事实上会在这个系统与“内部对象系统”之间遇到一道栅栏。这道栅栏会保证内部对象系统不受任何影响：

- 👉 确保任何构造器的原型总是来自于该 `PureObject` 的一个实例；
- 👉 `PureObject` 的原型修改与重写可以影响全部对象系统。

因此如果修改 `PureObject.prototype` 的成员，那么这个修改行为将传递到整个对象系统；如果重写 `PureObject.prototype`，那么重写之后新构建的对象都会受到影响（参考“1.4.1 原型重写”小节）。

1.4.2.5. 让用户对象系统影响内部对象系统

上面的对象结构中，用户对象系统被拦在栅栏之外，无法影响内部对象系统。然而这道栅栏非常脆弱，甚至象窗户纸一样一捅就破。

有一种非常简单的方法，可以让用户对象系统对内部对象系统产生影响。这种方法是“让用户对象系统持有 `PureObject.prototype`”，例如：

```
// 1. 备份一个系统内部的 Object()
var PureObject = Object;

// 2. 重写
Object = function() {
  // ...
}

// 3. 持有内部对象系统的原型
Object.prototype = PureObject.prototype;
```

现在，我们对 `Object` 原型的任何修改都将影响到系统全局——包括我们前面说到的“不受影响”的构造器原型。例如：

```
// (续上)

// 4. 修改原型
Object.prototype.hello = function() { alert('hi!') };

// 5. 新的构造器(直接量或赋值语句皆可)
function MyObject() {
}

// 6. 显示'hi!', 表明 MyObject() 构造器也受到了新的“用户对象系统”的影响
var obj = new MyObject();
obj.hello();
```

如果你认为步骤 1~3 过于繁琐，而且还会多出一个名为 `PureObject` 的变量，或者你怀疑在你重写之前已经有第三方代码已经重写过，又或者你可能使用函数直接量的方式声明了 `Object()`，而这导致在所有有效的代码（例如备份那个 `PureObject` 引用）之前 `Object()` 就已经被语言分析器重写了。那么，你可以用下面的代码简单地做同样的事：

```
// 从对象直接量的构造器中得到系统原始的 Object.prototype
function Object() {
    // ...
}
Object.prototype = {}.constructor.prototype;
```

1.4.2.6. 构造器重写对直接量声明的影响

直接量声明总是会调用“内部对象系统”来构造对象。因此，尽管除了 `undefined` 之外的所有直接量，都有对应的构造器，但是构造器与直接量却是两套系统——只是，在初始时二者是关联的。

当构造器被重写之后，直接量只与重写前的构造器相关，与重写后的就没什么关系了。因此重写不会对直接量声明构成什么影响，例如：

```
// 1. 取一个系统缺省的字符串直接量
var str1 = 'abc';

// 2. 重写 String() 构造器
String = function() {
}
String.prototype.name = 'myString';
```

```
// 3. 取重写后的字符串直接量
var str2 = '123';

// 4. 如果 name 成员有值，则证明重写会影响到直接量
alert( str1.name );
alert( str2.name );
```

在 JScript 中，显示两个 undefined 值。并且，所有七种直接量声明，都满足该项测试。

1.4.2.7. 构造绑定

在 SpiderMonkey JavaScript 中，上一小节的例子的效果会有些不同。对于三种值类型（string、boolean、number）直接量，函数直接量(function)，以及正则表达式对象(RegExp)直接量来说，上例的效果与结论都是一致的。但是，对于 Object 与 Array 来说却并非如此。下面的示例是上一示例 for Object()的版本：

```
// 示例 1
var obj1 = {};
Object = function() {
}
Object.prototype.name = 'myObject';
var obj2 = {};

// 如果 name 成员有值，则证明重写会影响到直接量
alert( obj1.name );
alert( obj2.name );
```

在 firefox 中测试，则显示：

```
undefined // <-- 已构造的 obj1 没有受到影响
myObject  // <-- 新构造的 obj2 使用了重写后的构造器原型
```

这表明当重写构造器之后，直接量声明将使用“新构造器的原型”来构造对象，但不会影响到旧的、已构造的对象实例。

那么，在 SpiderMonkey JavaScript 中重写 Object 后，直接量声明究竟只是使用原型，还是使用构造器来构造呢？在 firefox 中运行下面的测试代码：

```
// 示例 2
Object = function() {
    this.value = 'myValue';
}
```



```
var obj3 = {};  
alert(obj3.value);
```

结果将显示 “myValue”，这表明：在 SpiderMonkey JavaScript 中，

```
obj = new Object();  
// 与  
obj = {};
```

实际上是完全等效的。

这种特性被称为构造绑定。从本质上来说，这是因为在 SpiderMonkey JavaScript 中，Object()与 Array()的直接量声明形式总是通过“创建标识符所对应的构造器的实例”来生成对象实例的^①：

👉 （在 SpiderMonkey JavaScript 中，）对象类型的直接量声明，**与它的构造类直接绑定**。直接量声明过程本质上就是（对应的）构造类创建实例的过程。

这种构造绑定特性之所以重要，是因为它可以用原型继承中的构造器理论，完整地解释我们在 Array 与 Object 重写，并使用直接量声明时所遇到的任何问题。下面以最复杂的 Object 重写来加以叙述：

```
// 示例 3  
function Object() {  
    this.value = 100;  
}  
Object.prototype.name = 'MyObject';  
  
var obj = {};  
// 显示'MyObject'与 100  
alert( obj.name );  
alert( obj.value );
```

我们在这个例子中重写了 Object()，根据

👉 1.4.5 构造过程：从函数到构造器

👉 1.4.2.3 构造器的原型(prototype 属性)不受重写影响

所述，新的构造器函数具有如下特性（伪代码描述）：

```
// 1. 先从引擎内核取出原始的 Object 构造器  
PrueObject = Object;  
// 2. 重写构造器  
Object = function() { ... }  
// 3. 新函数作为构造器时，必须为它的 prototype 属性创建一个对象，这需要使用原始的 Object
```

^① 再次强调，在 JScript 中没有实现构造绑定，因此 Object 与 Array（以及其它）的直接量声明并不导致“重写的构造器创建新实例”的过程。

```
Object.prototype = new PrueObject();  
// 4. 该原型的 constructor 属性应指向函数自身  
Object.prototype.constructor = Object;
```

由于对象直接量被绑定在它的构造器函数（的标识符）上，因此 “obj = {}” 这行代码，就等同于：

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

因此，上例可以被展开成下面的代码：

```
// 展开示例 3 的代码  
PrueObject = Object;  
Object = function() {  
    this.value = 100;  
}  
Object.prototype = new PrueObject();  
Object.prototype.constructor = Object;  
Object.prototype.name = 'MyObject';  
  
var obj = new Object(); // <-- 这里与直接量声明完全等义  
// 显示 'MyObject' 与 100  
alert( obj.name );  
alert( obj.value );
```

这时我们再考察这个对象 obj 的属性，包括构造器、原型、继承性等都可以被合理地解释了。

我们最后再强调一下构造绑定的实质是创建直接量的过程“与其构造器的标识符有关”。这也表现在函数闭包内部的标识符系统上。举例来说，下例中 arr_1 与 arr_2 所使用的 Array() 构造器就不是同一个：

```
void function() {  
    function Array() {  
        alert('rewrited...');  
    }  
    // 使用内嵌的、重写的构造器 Array(), 所以将显示 'rewrited...'  
    var arr_1 = [];  
}();  
// 使用全局的、未被重写的构造器  
var arr_2 = [];
```

1.4.2.8. 内置构造器重写的概述

我们可以用下面的代码备份一个构造器（例如 `MyObject`）的引用，然后重写它：

```
// 方法 1
void function() {
    // 备份，在匿名函数中声明变量以避免污染全局
    var _MyObject = MyObject;
    // 重写
    MyObject = function() {
        //...
    };
}();
```

下面的代码更为精简：

```
// 方法 2
MyObject = function(_MyObject) {
    return function() {
        //...
    };
}(MyObject);
```

在重写后，如果用户需要使用相同的原型来创建对象，那么还可以用下面这样的方法：

```
// 方法 1 与原型
void function() {
    var _MyObject = MyObject;
    (MyObject = function() {
        //...
    }).prototype = _MyObject.prototype;
}();

// 方法 2 与原型
MyObject = function(_MyObject) {
    function NewMyObject() {
        //...
    }
    NewMyObject.prototype = _MyObject.prototype;
    return NewMyObject;
}(MyObject);
```

对于用户声明的构造器来说，这种方法是完全有效的。然而这些方法对于引擎内置的构造器来说，就可能有问题，因为某些构造器会有一些系统特性——例如前面讲到的构造绑定。准确地说，在 JavaScript 中，除了 `Error` 与 `Object` 可以正常地（和使用原型）继承子类之外，其它所有的内置构造器都无法使用上面的代码来重写构造器或直接引用原型。它们重写的方案，可以参考下面的示例：

```
// 方法 3
Array = function(_Array) {
  return function() {
    // ... 应当处理的入口参数
    return new _Array( ... );
  }
}(Array);
```

大多数情况下，我完全不建议重写内置构造器，除非你完全知道重写的后果，以及框架在底层上确需这样来实现。

1.4.3. 对象成员的重写

严格来说，原型重写是对象成员重写的一种特例。在不考虑宿主的情况下，JavaScript 对象中的几乎所有成员都可以被重写。与重写 `prototype` 属性一样，一些成员的重写是有特殊意义的，例如成员 `toString` 与 `valueOf` 的重写（参见“1.7 类型转换”）。而另一部分则没有什么特别的含义，例如重写成员 `call` 与 `apply`^①。

某些成员重写会带来标识符名称引用的问题，例如在函数内部重写 `Function.arguments`（参见“1.6.7 闭包中的标识符(变量)特例”）。此外，重写也会导致对象成员产的“可见性”这种性质的丢失。不过对于可见性的问题，不同的引擎处理也不完全一致，有关问题可以参见“1.4.8.4 如何理解“继承来的成员”？”。

1.4.3.1. 成员重写的检测与删除

我们总是可以检测一个成员是否被重写。这并不难做到，JavaScript 给对象基类添加了一个内置的 `hasOwnProperty` 方法。该方法不检测一个实例的父代

^① 与此类似的，重写 `RegExp` 的 `exec()` 方法也没有什么特殊性——即不会影响 SpiderMonkey JavaScript 中的 `rx()` 语法，也不会影响其它方法调用。

类，因此如果 “obj.hasOwnProperty('aName')” 返回 true，就表明 aName 是在该对象 obj 中被重写的——当然如果对象 obj 是一个构造器的原型，那么也就可以检测成员是否从原型链上继承而来，或是在当前原型中重写的。

```
function MyObject(tag) {
    if (tag) this.value = 1000; // 重写
}
MyObject.prototype.value = 100;

var obj1 = new MyObject(false);
var obj2 = new MyObject(true);

// 显示 false, true. 表明 obj1.value 没有重写，而 obj2.value 被重写过。
alert(obj1.hasOwnProperty('value'));
alert(obj2.hasOwnProperty('value'));

// 重写 obj1.value，并检测。显示 true
obj1.value = 'rewritten';
alert(obj1.hasOwnProperty('value'));
```

但是继承可以是多层的，所以在 JavaScript 中出现了“属性是由哪个原型来实现的（而又有哪些原型只是重写了它）”的问题。因此，我们可能面临的问题是：需要不断地访问父代类及其原型，来检测成员重写的情况。

接下来就回到了以前讨论过的一个话题：JavaScript 需要用户代码来维护外部的原型链（参见 1.4.7 原型链的维护）。所以，根本上来说，有效检测一个成员是否重写的关键，在于用户是否维护了一个有效的原型链。

当然，如果用户并不检测一个对象的原型是否（相对于其父代类）发生了重写，那么这种回溯是不必须的。然而，一旦一个属性是在对象的原型中添加的，那么你就不能直接从对象中删除它，而只能从原型（以及其父代类的原型）中删除它——不过这并不安全，因为它会影响到该类创建的其它实例。例如：

```
// 在原型中声明属性
function MyObject() {
    // ...
}
MyObject.prototype.name = 'MyObject';

// 创建实例
var obj1 = new MyObject();
var obj2 = new MyObject();
```

```
// 下面的代码并不会使 obj1.name 被删除掉
delete obj1.name;
alert(obj1.name);

// 下面的代码可以删除 obj1.name. 但由于是对原型进行操作, 所以也会使 obj2.name 被删除
delete obj1.constructor.prototype.name
alert(obj1.name);
alert(obj2.name);
```

基于上述的分析, 在一个正确维护原型链的对象系统中 (参见“1.4.7.2 constructor 属性的维护”), 下面的 `deleteProperty()` 函数可以检测到重写指定属性名(prop)的原型并删除:

```
// 从对象 (以及其原型中) 删除属性
function deleteProperty(obj, prop) {
    if (prop in obj) {
        do {
            if (obj.hasOwnProperty(prop)) break;
        }
        while (obj.constructor && (obj = obj.constructor.prototype));
    }
    delete obj[prop];
}

// 在原型中声明属性
function MyObject() {
    // ...
}
MyObject.prototype.value = 100;

// 一个正确维护原型链的派生类
function MyObjectEx() {
    this.constructor = arguments.callee;
}
MyObjectEx.prototype = new MyObject();

// 创建实例
var obj1 = new MyObjectEx();
var obj2 = new MyObjectEx();

// 删除成员 'value'
// (注意将导致所有子类对象的该成员被删除)
deleteProperty(obj1, 'value');
```

```
alert(obj1.value);
alert(obj2.value);
```

但是由于原型链上可能有多次重写，所以即使使用该函数，也不能保证在一次就能删除掉该对象的指定成员。因此某些情况下会需要一个加强版本：

```
function deleteProperty(obj, prop, forced) {
    do {
        if (!(prop in obj)) break;
        do {
            if (obj.hasOwnProperty(prop)) break;
        }
        while (obj.constructor && (obj = obj.constructor.prototype));
        delete obj[prop];
    }
    while (forced);
}

// 正确维护的原型链
function AObject() {}
function MyObject() {
    this.constructor = arguments.callee;
}
function MyObjectEx() {
    this.constructor = arguments.callee;
}
MyObject.prototype = new AObject();
MyObjectEx.prototype = new MyObject();

// 多次重写的过程
AObject.prototype.value = 100;
MyObject.prototype.value = 1000;

// (示例略)
...
```

1.4.4. 宿主对重写的限制

多数不能重写的情况，是出自宿主环境的限制。例如 Internet Explorer 通过 ActiveX 控件来扩展的对象，很多成员就是不能被重写的——这取决于该对象的接口约定——最常见到的例子，是 XMLHttpRequest 对象的属性值是只读

的。当然，也有一些不是那么麻烦就可以验证的例子，例如在 Internet Explorer 中试试下面的代码（包括 HTML）：

```
<input type="file" id="fileUri">
<script>
var el = document.getElementById('fileUri');

// 在 HTML 中，file 类型的 input 的 value 是不可写的
el.value = "set a path ...";
</script>
```

这种重写限制是否导致异常，也取决于宿主环境。同样是上面这个例子，在 Internet Explorer 中这个重写只是被忽略，而在 firefox 里，则导致一个异常。不过这只是一个特例，对于其它的一些重写限制，反过来也是适用的。

1.4.4.1. 重写与引用

一般来说，如果我们重写一个变量，那么效果只是该变量被修改，而该变量的引用是不会受到影响的——如果反过来修改引用，也是成立的。例如：

```
// 示例 1：重写变量，其它变量对它的引用不受影响
var v1 = { name: 'MyName' };
var v2 = v1;

// 1. 重写 v1
v1 = 100;

// 2. 显示 'MyName'，表明 v2 仍然是被引用的对象，但与 (当前的) v1 已经不同了
alert( v2.name );
```

重写不会改变引用者，是“引用”的基本特性，也是一个好的特性。这一点在 JScript 中得以很好地保障——尽管这也同样会给你带来麻烦。例如：

```
// 示例 2. 重写对象成员，被引用的原始变量不受影响

// 1. 显示 true，表明二者是同一个引用
alert( window.setTimeout === setTimeout );


// 重写 setTimeout
window.setTimeout = function() { };

// 2. 显示 false
// 这里由于 window.setTimeout() 未被重写，因此与 (当前的) setTimeout 已经不同了
alert( window.setTimeout === setTimeout );
```


不过看起来另有一些引擎则认为系统有必要维护 `setTimeout` 与 `window.setTimeout` 之间的关系。例如 SpiderMonkey JavaScript，当重写 `window.setTimeout` 时，`setTimeout` 也是同时被重写的。因此，上面的例 2 中的第二个输出就会是 `true`。

我们需要进一步认识这两种现象之间的差异。首先，我们通常所谓的“全局”在 JavaScript 中受到两方面的影响，一是宿主对象（以下用 `window` 对象代指），二是全局对象。这种影响可以模拟化的描述如下：

```
with (host_object) { // host object, window etc.
  with (global_object) { // engine global object
    // <-- 用户代码未使用 var 而导致的隐式变量声明在这里①
    // 所有用户代码都被在此后装入并执行
  }
}
```



在这里维护引用关系

在 JScript 中，一个隐式声明的位置在 `window` 对象之外。而按照约定，使用 `'varName in window'` 检测应当返回真值。于是，JScript 采用了一种策略来维护这种关系：变量们于 `global_object` 的对象闭包内，同时通过一个引用（或内部变量）作为 `window` 对象的成员。

JScript 采用这种（必须维护 `window` 与变量名之间的关系的）机制的原因，是因为 `window` 对象对引擎来说其实是一个外部对象。JScript 脚本引擎来自一个名为 `ActiveScripting` 的系统，而 `window` 来自浏览器的 `DOM` 系统。其中的区别之一在于：JScript 引擎中的对象成员都可以删除，例如 JScript 的全局对象的成员其实是可以删除的：

```
// 删除 global object 的成员
delete parseInt;
delete eval;
...
```

而 `window` 对象的成员（包括动态添加的成员）不可以删除。例如下面的代码导致异常：

```
// 删除 window 对象的成员导致异常(这由不同的宿主对象自行决定)
delete window.self;
delete alert;
...
```

① 隐式声明的变量被动态地添加到闭包（这里指调用对象 `ScriptObject` 块）的 `varDelcls` 中，这些动态添加的内容是可以被 `delete` 运算清除的。而使用 `var` 显式声明的变量在语法解释期就被添加到 `varDelcls`（的前端），这个结构在运行前即被创建好，因此不是动态的，也不能用 `delete` 动态删除。

所以现在你应该可以了解：为什么使用“`window.XXX`”方式添加的成员“可以用 `for..in` 列举，但不能删除”，而隐式变量声明的全局变量“不能用 `for..in` 列，但能删除”，其根本原因在于它们位于不同的对象闭包域中。

而为了使“`in`”运算符在这些不同的域中表现一致^①，JScript 强加了一层外部关系。然而在 `delete` 删除过程中，JScript 未能清除上述关系。所以在 JScript 中，在“`delete varName`”操作之后，一方面即可以检测到“`varName in window`”仍为真值，另一方面又存取异常（因为它实际上已经被删除了）。

在 SpiderMonkey JavaScript 中却没有这种关系。SpiderMonkey JavaScript 的 `window` 对象是一种引擎级别的原生对象^①，而且引擎支持对象属性的可见性声明（以及其它的性质声明）。所以它的全局环境是这样构造的：

```
with (host_object) { // host object, window etc.
  // <-- 用户代码未使用 var 而导致的隐式变量声明，作为 host_object 成员添加在这里
  with (global_object) { // engine global object
    // 所有用户代码都被在此后装入并执行
  }
}
```

所以现在这些隐式变量其实都是宿主对象的成员。而要达到它不被 `for..in` 列举的目标，SpiderMonkey JavaScript 只需要设置该成员的可见性为 `false` 即可，并不需要维护它与 `global_object` 的关系。也就是说，表面看来是 SpiderMonkey JavaScript 维护了全局变量 `setTimeout` 与 `window.setTimeout` 属性的关系，实质上不过是 `window.setTimeout` 成员的某种性质声明、以及闭包访问的效果。

1.4.4.2. 宿主环境的限制

在宿主环境中，重写行为不单单受到引用的限制，也会受到被重写成员的一些性质的限制。常见的性质是“只读”，这种情况下，该成员是不能写的。除此之外，重写也可能触发某种特别的行为，从而导致重写失败或远离开发人员的预期。例如在浏览器环境中的 `location` 对象：

```
// 显示'object'，表明 location 是一个对象
alert( typeof location );

// 显示 true，表明该对象可以作为字符串使用，这时值等于 toString() 的值
alert( location == location.toString() );

// 显示 true，表明其值也等于 location.href 的值
```

^① 这并不是说 `window` 对象是内置于 SpiderMonkey JavaScript 的。而是因为该引擎采用 C 语言实现，所谓“对象 / 类”只是注册到引擎内的一个普通结构(struct)，即不是复杂的 COM，也不是其它语言(例如 Java)的内部对象。因此任何置入到 SpiderMonkey JavaScript 的外部对象，与其原生对象的差异都非常微小。

```
alert( location == location.href );
```

那么我们重写这个 `location` 对象，其效果又是如何呢？这取决于不同宿主环境的约定。在 WEB 浏览器宿主中并，重写 `location` 意味着网页切换到新的网址。也就是说“`location = XXX`”事实上相当于如下代码^①：

```
// 1. 内部的_location.href 置值器声明
_location = {
  set href: function(src) {
    // 打开以 src 指定的新网页
  }
}
// 2. window.location 成员的读写器声明
window = {
  get location = function() {
    return _location;
  },
  set location = funcction(src) {
    _location.href = src;
  }
}

// 3. 如下代码将最终触发_location.href 置值，并导致新网页打开
window.location = 'XXX';
```

所以对于宿主对象 `window` 来说，一个成员在重写时的效果，是由该宿主对象自己来决定的。无论它是通过 COM/ActiveX 技术，还是通过成员的属性读写器来实现，其最终结果都是一样的——成员重写可能带来一个复杂的、未确知的、可被定制的行为。

现在，我们说到过重写宿主对成员的两种可能效果，一种涉及是否维护引用，一种涉及是否触发行为。但这两种效果事实上都是有可能规避的，不过这需要利用 JavaScript 在语法声明阶段的重写。例如在 JScript 中：

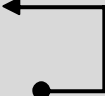
```
// 示例：通过具名函数在语法分析阶段中重写 setTimeout
function setTimeout() {
}
// 显示 true
alert( window.setTimeout === setTimeout );
```

^① 这里的代码使用的是 SpiderMonkey JavaScript 的扩展语法，并不能直接执行的。这些代码,主要用来说明在 SpiderMonkey JavaScript 中向引擎注册宿主对象 `windows`，以及说明它的各个成员的读写器的可能方法。在 JScript 中，这一过程是由 ActiveX Control 来实现的，在 COM/ActiveX 技术框架下，对象的每个成员都可以有独立的读写器函数。这些在概念上与 SpiderMonkey JavaScript 是一致的，只是实现不同。

```
// 示例: 在 var 声明在语法分析阶段重写 location
var location = 'replaced'
// 显示'replaced'
alert( window.location );
```

对此，我们回顾前面所述的如下结构：

```
with (host_object) { // host object, window etc.
  with (global_object) { // engine global object
    // <-- 用户代码未使用 var 而导致的隐式变量声明在这里①
    // 所有用户代码都被在此后装入并执行
  }
}
```



在这里维护引用关系

在 JScript 中全局对象与宿主对象的上述关系是在引擎开始执行第一行用户代码之前就完成了的——例如 `host_object` 从 `global_object` 的变量表中抄写 `setTimeout/location` 等变量并作为 `host_object` 的成员。因此当执行上面示例中的“`location = 'replaced'`”时，`window.location` 已经被（在语法期）重写过，失去了“触发行为”的效果，从而变成了合法和有效的。

然而在 SpiderMonkey JavaScript 中，结构却是下面这样：

```
with (host_object) { // host object, window etc.
  // <-- 用户代码未使用 var 而导致的隐式变量声明，作为 host_object 成员添加在这里
  with (global_object) { // engine global object
    // 所有用户代码都被在此后装入并执行
  }
}
```

由于语法分析期这些“用户代码中的全局变量名”就落到了 `host_object` 的域中，并与 `host_object` 中既有的成员名重名而导致了重写，但这种重写又因为 `host_object` 对这些成员设置了特别的存取性质（例如只读或读写器函数），从而导致引擎在装载这些脚本时就发生了存取违例或调用写函数出错。例如在 Mozilla Firefox 中装载如下代码，就导致一个名为“`NS_ERROR_DOM_SECURITY_ERR`”的浏览器异常（而非 SpiderMonkey JavaScript 脚本引擎错误）：

```
<script>
function location() {
}
</script>
```

① 隐式声明的变量被动态地添加到闭包（这里指调用对象 `ScriptObject` 块）的 `varDelcs` 中，这些动态添加的内容是可以被 `delete` 运算清除的。而使用 `var` 显式声明的变量在语法解释期就被添加到 `varDelcs`（的前端），这个结构在运行前即被创建好，因此不是动态的，也不能用 `delete` 动态删除。

1.4.5. 引擎对重写的限制

JavaScript 语言是不支持对关键字与运算符的重写的。但除了语言本身的这种约定，还有什么是不可重写的吗？是的，还有。例如你不能重写直接量：

```
// 下面的代码导致异常
null = null;
1 = 2;
```

相信读者能很容易地理解“为什么直接量不能被重写”——这是一个与存储相关的问题，更确切的说：常量不可写。然而下面的代码还是让人困惑（尽管它没有什么意义）：

```
// 下面的代码并不会导致异常
(1).toString = null;
'abc'.toString = null;
```

不过这已经涉及到包装类的问题了，我们将会放在更靠后的章节中去讲述。下面我们讲述一些常见的、源于引擎实现的重写限制。

1.4.5.1. this 的重写

“this”引用不能被重写，是最常见的引擎对重写的限制。例如：

```
function MyObject()
{
    this = null;
}

// 以下代码将产生异常
this = null;
new MyObject();
```

但这个问题仍有些细节值得探讨。在 JScript 中，this 是作为关键字处理的（尽管它不在引擎的保留字列表中），因此类似于“window.this”或“aObj.this”这样的代码在语法分析期就会导致异常，是一个静态的语法限制。然而，在 SpiderMonkey JavaScript 中，this 是作为一个不可写的对象引用来处理的，所以仅在执行期才会导致异常。尽管上例的重写会导致异常，但下面的代码却会有出人意料的事情发生：

```
// （下面代码仅在 SpiderMonkey JavaScript 中测试）
window.this = 'hello';
// 显示 window 对象
alert(this);
```

```
// 显示 window.this 属性值 'hello'
alert(this.this);
```

但是这并不意味着用户代码可以改变对象闭包中的 `this` 引用^①，例如下面的代码中并不能通过 `this` 引用访问到 `aObj`（这里 JScript 可以用 `'this'` 字符串形式声明该直接量的成员，以避免引擎的语法检测）：

```
var aObj = { value: 100 };
var value = 1000;
// 显示值: 1000
with ( {this: aObj} ) {
    alert( this.value ); // <-- 这里的 this 仍然指向宿主对象
}
```

这是因为在 `with()` 语句块中所能访问到的 `this` 引用取决于引擎规则。同样的原因，你也不能在匿名函数外包一层对象闭包（即使它具有一个名为 `this` 的成员），试图改变“以普通函数(而非对象方法)执行时，`this` 引用指向宿主对象”这条规则。因为在任意时候直接使用 `this` 这个变量，都是由引擎来决定当前的引用对象的，而与闭包链上有没有 `'this'` 这个标识符无关。

1.4.5.2. 语句语法中的重写

大多数语句本身不会导致重写^②。尽管在其语法元素中使用赋值表达式可以带来重写效果，但这是赋值表达式导致的，而非语句本身导致的重写行为。

一些语句本身有“暂存对象引用”的行为，最明显的有如 `with` 语句，就是用于操作一个对象闭包的。而在“暂存对象引用”的影响下，在语句中重写对象的行为将变得很特殊。例如：

```
var obj = {
    name: 'myName',
    value: 'hi'
}
// 1. 暂存: for 语句暂存了 obj 对象的一个引用
for (var i in obj) {
    // 2. 重写: 该重写不会影响到 for 语句暂存的 obj 对象引用，以及它对该对象的列举效果
    obj = {};
}
```

除了 `with` 语句以及 `for`、`for..in`、`while`、`do..while` 等循环语句之外，`switch`

^① 这也是在章节“1.6.2.2 对象闭包带来的可见性效果”中我们使用 `self` 属性名，而不是 `this` 作为属性名的原因。

^② 反之，例如下一小节将讲到的 `catch()` 子句自身就具有重写行为。

语句也会暂存对象引用。但特殊的是，`switch` 语句中只有一次成功检测 `case` 分支的机会，因此下面这个示例构造得令人困惑：

```
1  var obj = obj1 = {}
2  var obj2 = {};
3  switch (obj) {
4      case obj = obj2: alert('obj2'); break;
5      case obj1: alert('obj1'); break;  //<--跳转到该分支
6  }
7  // 显示 true
8  alert(obj === obj2);
```

这个示例主要检测 `obj` 引用是 `obj1` 还是 `obj2`。从初始状态到第 3 行，`obj` 引用都指向 `obj1`，而第 3 行是一个 `switch()` 语句，因此它缓存了一个 `obj` 的引用。第 4 行代码完成了一次重写，使它指向了 `obj2`。如果这次重写对 `switch` 的缓存是有效的，那么第 4 行的 `case` 分支就会检测成功，因为分支检测的正是 `obj2`。然而我们看到结果仍将显示 'obj1'。所以第 4 行的重写并没有影响到缓存。

如果从如此细致的粒度来考查语句，那么 `if` 语句也有类似的性质。例如：

```
9  // (续上例)
10 if (obj == (obj = obj1)) {
11     alert('obj1');
12 }
```

在第 10 行中，表达式 “`obj = obj1`” 将有两个语句效果：

- 👉 将 `obj` 重写为 `obj1` 的引用
- 👉 返回 `obj1` 引用

但在外部的 `if()` 语句中，由于 `if` 是缓存了 `obj` 引用，因此它仍是重写前的值“`obj2` 的引用”。所以上面的第 11 行代码不会执行到。

语句“暂存对象引用”的效果只针对引用本身，而不包括它的成员。因此在语句块中修改成员的话，就因引擎的不同而有不同的表现了。例如，如果在 `for..in` 列举过程中 `delete` 一个还未被列举的属性（并且成功），ECMA 规范约定它将不能再被列出。但并非所有的引擎都会这样做，例如 `SpiderMonkey`、`JScript` 等都是按照该约定实现的，但 `Adobe ActionScript` 则仍将列出全部成员，而在退出 `for..in` 循环之后这些成员又将不复存在（例如不再出现在下一次列举中）。这也表明引擎对语句“所持有（暂存）的对象”的理解并不完全相同。

1.4.5.3. 结构化异常处理中的重写

在结构化异常处理的 `catch (exception) { ... }` 块中，我们可以在 `exception` 位置声明任意一个变量名。由于这个变量名是被动态地声明的，因此也可能重写当前闭包中的、或闭包链上的标识符。ECMA 规范对此约定：该变量将仅在 `catch` 块中的 `catchStatements` 部分使用，并在退出该块时从闭包中移除。但不同的引擎的实现仍有差异，例如 JScript 并没有遵守这项约定，因此在退出 `catchStatements` 语句后，该变量标识仍将影响到其后的 `finally` 或其它语句：

```
var ex = 100;

try {
    alert(ex); // 显示 100
    throw new Error('ERROR! ');
}
catch (ex) {
    alert(ex); // 显示捕获到一个 Error() 对象实例
}
// 此后的输出中，JScript 仍显示 Error() 对象实例，而据 ECMA 规范则应当显示值 100
finally {
    alert(ex);
}
alert(ex);
```

JavaScript 实现 `finally{...}` 块的“总是被执行”的语法效果的方法，是在 `try` 块（因为任意原因）退出之前“挂起”`try/except` 块中的代码行，然后转入 `finally` 块执行代码。然而如果我们使用 `return` 子句退出，并试图返回内部的话，就存在了一个疑难：在挂起 `return` 子句的过程中，`finally` 块是否能重写 `return` 返回的值呢？对于下面这个例子：

```
function foo(x) {
    try {
        return x;
    }
    finally {
        x = x*2;
    }
}
// 显示值 100
alert( foo(100) );
```


答案是“否”——上例返回的并不是 `x*2` 的值。如果用户试图使 `finally{...}` 中的修改有效，那么应该在 `finally` 块中使用 `return` 子句，例如：

```
...  
finally {  
    x = x*2;  
    return x;  
}
```

但是这个问题还有更加复杂的细节：在上面这个例子中，我们使用的是一个值类型数据（数值 100）。这种情况下，从执行过程来说，`finally` 语句“挂起”的将会是“`return x`”这个语句，而变量 `x` 是一个值类型数据，因此在“挂起”时就已经被取值了，于是我们在 `finally{...}` 中的代码对变量 `x` 的修改不会影响到它（这个已经被取走的值）。然而，如果我们在这里使用对象并修改它的成员，那么由于“挂起”时取走的只是引用，所以 `finally{...}` 中的代码仍然会造成影响^①。例如：

```
function foo(x) {  
    try {  
        return x;  
    }  
    finally {  
        x.push(100);  
    }  
}  
// 显示返回数组字符串形式 '1,2,3,100'  
alert( foo([1,2,3]) );
```

1.5. 包装类：面向对象的妥协

JavaScript 中存在两套类型系统，其一是元类型系统（`meta types`），是由 `typeof` 运算来检测的，按照约定该类型系统只包括六种类型（`undefined`、`number`、`boolean`、`string`、`function` 和 `object`）；其二是对象类型系统（`object types`），对象类型系统是元类型的 `object` 中的一个分支。

接下来出了一个问题：按照 JavaScript 的语言设计的本意，认为“（除了 `undefined` 之外）所有的一切都是‘对象’”。如果是这样，那么 `number` 元类型与 `Number` 对象类型，以及其它元类型与相应的对象类型是如何被统一呢？

为了实现“一切都是对象”的目标，JavaScript 在类型系统上做出了一些

^① 重写变量 `x`，与重写变量 `x` 的成员之间存在区别。所以如果重写变量 `x` 引用本身，也是不会有影响的。

妥协，其结果是：可以将元类型数据通过“包装类”变成对象类型数据来处理①。这些元类型数据与包装类的关系如下表：

元类型	直接量	包装类	说明
undefined	undefined	无	不需要包装类
boolean	true, false	Boolean	
number	数值	Number	
string	'...', "..."	String	
function	function() { ... }	无 (*)	参见 1.5.4
object	{ ... }		

(*)function 也是对象，因而它与 object 类型一样，不需要通过包装类来转换。关于该问题的详细细节，请参见章节“1.5.4 其它直接量与相应的构造器”。

这样一来，从理论上说，对象类型系统中的每一个实例，以及元类型数据通过包装类转换而来的数据，都将是“对象”。为了区分包装前、后的数据，我们在本节中所述的“对象”仅指这种对象，它具有如下特性：

- 👉 typeof(obj) 值为 'object' 或 'function'
- 👉 obj instanceof Object 值为 true

除此之外的数据，我们称之为元类型数据，或元数据。

在开始本小节的讨论之前，我们先强调一下，在元数据经过“包装类”包装后得到的对象，与原来的元数据并不再是同一数据，只是二者的值相等而已。

1.5.1. 显式包装元数据

JavaScript 支持一种特殊语法的显式包装。这种语法是将类构造器当成普通函数使用，该函数能将参数值进行包装，并以该类构造器的一个实例传出。这种语法看起来类似于一些通用语言中的类型强制转换：

```
var instance_a = Constructor(value_a);
```

但从语言的实现来说这与类型强制转换完全不同：强制转换是在同一数据（相同内存地址的不同引用）上进行的，而这种语法则产生一个新的数据。

这里的 Constructor 首先包括以下三种值类型的包装类：

- 👉 Number()
- 👉 Boolean()

① 从另一个方面来讲，JavaScript 为了使得部分数据得到更加高效的处理，因此保留了独立于对象类型系统的元类型系统，从而使 JavaScript 没有成为一种象 Java 一样“纯粹的”面向对象语言。

String()

下例说明如何使用这种显式包装：

```
var
    v1 = 100;
    v2 = 'hello, world!';
    v3 = true;

// 使用构造器的显式包装
alert( Number(v1) );
alert( String(v2) );
alert( Boolean(v3) );
```

此外，这种技术也被应用在 JavaScript 内建的 `Object()` 类上。但实现的语义并不相同——并不是将一个“Object 值转换为 Object 对象”，因为 `Object` 不是元类型中的值（值类型数据）。当在 `Object()` 类上使用这种技术时，`Object` 传入参数仍然只能是 `boolean`、`number`、`string` 三种值类型数据之一，它表明“将这些值类型数据通过显式包装转换为等同的对象数据”。因此下列代码的效果其实与上例是相同的：

```
// (续上例)

// 使用 Object() 的显式包装
alert( Object(v1) );
alert( Object(v2) );
alert( Object(v3) );
```

在上面的例子中，显式包装的结果是产生了一个与包装前同值的对象——`alert()` 的输出证明了这一点。而这个同值的对象也具有一些自身的、对象系统的特性。例如：

```
// (续上例)

var obj1 = Object(v1);
var obj2 = Number(v1);

// 返回结果值 true 和 'object'
// (obj1 与 obj2 在下面测试中的结果是相同的)
alert( obj1 instanceof Number );
alert( typeof obj1 );
```

对于这些检测的细节，我们将在下一小节中予以详述。

1.5.2. 隐式包装的过程与检测方法

对于元数据来说，如果它是用作普通求值运算或赋值运算，那么它是以“非对象”的形式存在的。例如下面这行代码：

```
'hello, ' + 'world!'
```

在这行代码中，“+”运算符两侧的数据都是以“非对象”的形式在参与运算，也就是直接做值运算。当元数据作对象系统的相关运算（参见“1.5 面向对象编程的语法概要”中的说明）时，下面的运算是不会有包装行为的，

// 示例 1：分析对象运算过程中，运算是否产生包装行为

```
var data = 100;

// 1. instanceof 运算不对原数据进行“包装”
data instanceof Number

// 2. 如下导致异常，因为不能对元数据做 in 运算
'constructor' in data
```

而在做下面的运算时，它就需要通过包装先将元数据变成对象：

```
// (续上例)

// 3. 成员存取运算时，“包装”行为发生在存取行为过程中
data.constructor
data['constructor']

// 4. 所谓方法调用，其实是成员存储后的函数调用运算，因此“包装”行为发生的时期同上
data.toString()

// 5. 做 delete 运算时，“包装”行为仍然发生在成员存取时
delete data.toString
```

综合上述 3 ~ 5 的例子，我们可以知道，所谓元数据到对象的“隐式包装”，其实总是发生在成员存取运算符中^①。

那么我们如何检测被包装后的这个对象呢？我们知道，对象方法调用时，会传入一个 **this** 引用，而这个 **this** 引用必然是一个“真实的对象”。因此如果是对元数据做方法调用运算，那么就可以检测到这个“被包装后的对象”。如

^① 除此之外，专用于对象的 **for..in** 语句和 **with** 语句也会导致“包装”过程，基本上，读者可以将此理解为这两个语句的副作用。

下例：

// 示例 2：通过方法调用来获取被包装后的对象

```
var data = 100;
Object.prototype.getSelf = function() {
    return this;
}

var _this = data.getSelf(); //<-- 包装行为发生这个存取运算过程中
// 显示类型为 object
alert( typeof _this );
// 显示值为 100
alert( _this );
```

用同样的方法，我们可以获得元数据、直接量与包装类的全部性质。下面是这种检测的完整实现代码：

// 示例 3：检测元数据、直接量与包装类的性质

```
Object.prototype.getSelf = function(){
    return this;
}

Object.prototype.getClass = function(){
    return(this.constructor);
}

Object.prototype.getTypeof = function(){
    return(typeof this)
}

Object.prototype.getPrototypeOf = function(){
    return(this instanceof this.getClass());
}

var data = 100;
Object.prototype.getSelf = function() {
    return this;
}

// 样本：七种可声明的直接量
var samples = [
    '', // 字符串
    100, // 数值
```

```

true, // 布尔值
function() {}, // 函数
{}, // 对象
[], // 数组
/./ // 正则
];

// 检测
var v1, v2, attr, cls;
for (var i=0; i<samples.length; i++) {
    v1 = samples[i];
    v2 = v1.getSelf();
    cls = v1.getClass();

    attr = [
        typeof v1,
        v1.getTypeof(),
        v1 instanceof cls,
        v1.getInstanceof()
    ];

    alert(attr);
}

```

该项检测的结果如下表所示（部分数据，其它数据参见“1.5.4 其它直接量与相应的构造器”）：

（表：直接量、包装类、构造器的相关特性一节 1）

元类型	直接量(v1)	包装类(v2)	特性			
			typeof <value>		<value> instanceof <class>	
			v1	v2	v1	v2
undefined	undefined		'undefined'	(*)		
string	", ""	String	'string'	'object'	false	true
number	数值	Number	'number'			true
boolean	true, false	Boolean	'boolean'			true

(*)undefined 没有包装类，因此不能检测它的构造器产生的实例(v2)，或进行 instanceof 运算。

对 string、number 与 boolean 的 v1 值检测的结果表明：

👉 进行 typeof 检测，都不是 'object'，这表明“元数据类型”不是对象；

👉 进行 `instanceof` 检测，值都是 `false`，表明它们都不是通过对象系统（构造器）创建的。

这两项特征真实地反映了“元数据类型”或“值类型”（相对于引用类型）数据的特性。

1.5.3. 包装值类型数据的必要性与问题

包装类是 JavaScript 用来应对“可以调用值类型数据的方法，使它看起来象是对象调用”的处理技术。这与后来在 .NET 中出来的“装箱(boxing)”是一样的^①，只是 JavaScript 将这种技术称为“包装(warping)”而已。

下例中：

```
Number.prototype.showDataType = function() {  
    alert('value:'+ this + ', type:' + (typeof this));  
}  
  
var n1 = 100;  
alert(typeof n1);  
n1.showDataType();
```

在函数外部调用 `typeof` 查看 `n1` 的类型会是 `number`，而当调用 `n1` 的 `showDataType()`方法时，`n1` 的类型却变成了 `object` 类型。我们前面讲述过，当进行

```
n1.XXXX  
// 或  
n1['XXXX']
```

这样的对象成员存取操作时，JavaScript 用“包装类(Number)”为 `n1` 临时创建了一个对象。因此上面调用 `n1.showDataType()`这个方法的过程，看起来就好像：

```
// 与 n1.showDataType() 等效的代码  
new Number(n1).showDataType();
```

与这行代码相同的是：当 `showDataType()`调用结束后(准确的说是在该对象的生存周期结束时)，临时创建的包装对象也将被清理掉。

显然变量 `n1` 并不等同于包装后的“`new Number(n1)`”，因此“值类型数据的方法调用”其实是被临时地隔离在另外的一个对象中完成。而同样的原因，

^① JavaScript 没有类似于“拆箱(unboxing)”的过程，因为它的函数形式参数不支持值的传出。

无论我们如何修改这个新对象的成员，这种修改也会不影响到原来的值：

```
// 声明值类型数据，并修改它的成员
var str = 'abc';
str.toString = function() { // <-- 这里的重写是无意义的
    return '';
}

// 显示'abc'，表明对 str 包装后对象的修改，不会传递到原变量
alert(str);
alert(str.toString());
```

这个例子是可以预期的。然而有些函数对入口参数值也会做类似的处理，其效果就会变成不可预期，如下例(参见 1.2.1.1 值类型与引用类型)：

```
var str = 'abcde';

function newToString() {
    return 'hello, world!';
}

function func(val) {
    // 在这里，如果 val 是对象则修改它的成员，否则修改值类型包装的结果对象(并随后废弃该对象)
    val.toString = newToString;
}

// 试图显示'hello, world'，但实际仍然显示'abcde'
func(str);
alert(str);
```

1.5.4. 其它直接量与相应的构造器

在 JavaScript 中，三种元类型都有其相应的包装类与直接量声明。除了 undefined 不参与讨论之外，元类型中剩下还有 object 与 function。在这两种类型中，object 元类型中又有 Object、Array 与 RegExp 三种对象存在直接量声明语法。这几种类型与其相应的构造器——注意这里没有称为包装类——之间的关系如下表所示：

（表：直接量、包装类、构造器的相关特性一节 2）

元类型	直接量(v1)	构造器(v2)	特性	
			typeof <value>	<value> instanceof <class>

			v1	v2	v1	v2
function	function(){ .. }	Function	'function'	'function'	true	true
object	{ ... }	Object	'object'	'object'	true	true
	[...]	Array				
	/.../...	RegExp	(*)		true	true

(*)RegExp 在 JScript 与 SpiderMonkey JavaScript 中实现并不一致：对于前者，这里的返回值（如你所愿地）将是 'object'；对于后者，检测 v1 与 v2 的 typeof 值将（令人不可思议地）返回 'function'！

在 JavaScript 中，这些引用类型的直接量总是由它对应的构造器产生的一个对象实例。在声明直接量时，相对应的构造器会被调用并创建一个对象实例。但在没有实现“构造绑定”机制的引擎中，这种关系会受到构造器重写的影响。

1.5.4.1. 函数特例

上表中的 function 很明显是一个特例。但正是这个特例，表现出函数两个方面的重要特性：

- ☞ typeof 对函数直接量(v1)与包装后的对象(v2)检测都是返回 'function'，这表现了 JavaScript 作为函数式语言的重要特性：函数是第一类的；
- ☞ 对 v1 与 v2 作 instanceof 检测，它们总是 Function 的一个实例，这表现了 JavaScript 作为面向对象语言的重要特性：函数是对象。

1.5.4.2. 正则表达式特例

如果说函数的特例是可以理解的，那么在 SpiderMonkey JavaScript 中，正则表达式的特例可以说是对 JavaScript 类型系统的颠覆。因为在 SpiderMonkey JavaScript 中，无论是用直接量还是构造器创建的正则表达式对象，都具有一种奇怪的特性：typeof 值是 function。

将正则作为 function 处理的原因之一，是因为在某些较早的版本中允许一种特殊的语法：

```
var rx = /. /g;
// 显示 a
alert( rx('abcd') );
```

这种语法相当于如下的效果：

```
rx.exec('abcd') ;
```

也就是说，是 `exec()` 方法的一种简便的调用形式。非常明显的，如果 `rx(...)` 这样的语法成立，则这对括号必须被理解成“函数调用运算符”，而引擎就必须因此将正则表达式的类型置为 `'function'`——否则脚本执行系统（或称引擎中的模块）中就会因为“对一个非函数的对象做函数调用运算”而导致异常。

正是为了逃避脚本执行系统对正则表达式对象的检测，SpiderMonkey JavaScript 为正则表达式伪造了 `'function'` 这个类型信息。伪造的过程则非常简单：在 `typeof` 运算中，当发现运算是 `RegExp()` 的一个实例时，返回类型信息 `'function'`。因此这个伪造的结果只是^①：

- 👉 欺骗 `typeof`，返回 `'function'`
- 👉 欺骗脚本执行系统，使 `rx(...)` 语法成立并被能执行
- 👉 执行时调用 `rx.exec`

而并没有改变正则表达式其它的任何性质。例如，我们检测正则对象实例的构造器属性，或做 `instanceof` 检测时都将按标准的 (ECMA 标准或以 JScript 为参考的) 正则表达式处理：

```
var rx = /. /g;
// 显示 true, true, true
alert( rx.constructor === RegExp );
alert( rx instanceof RegExp );
alert( rx instanceof Object );
```

1.6. 关联数组：对象与数组的动态特性

索引数组与关联数组是从数组的下标使用方式上来区分的一种方法^②。所谓索引数组，是指以序数类型作为下标变量，按序存取元素的数组；所谓关联数组，则是指以非序数类型来作为下标变量来存取的数组。

在 JavaScript 中，（使用索引存取的）数组的下标必须是值类型——如果试图将一个引用类型（`object/function`）或 `undefined` 类型值用作函数下标，则它们将被转换为字符串值来使用。

而 JavaScript 中的值类型数据包括 `number`、`boolean` 与 `string`，这其中只有 `boolean` 是序数的，其它两种则是非序数的（`number` 在 JavaScript 中实现为浮

^① 几乎没有什么书籍提及到这个奇怪的特性及其成因。我为此专门阅读了 SpiderMonkey JavaScript 的源代码，其中的注释表明，这是为兼容旧版而保留的一项特性。因此我完全不建议读者使用该特性，但是在写有关 SpiderMonkey JavaScript 的兼容代码时，必须将这项特性考虑进去，以避免在使用 `typeof` 时出现误判。

^② 另一种常见的区分方法是动态数组与静态数据，是按照数组对存储的使用方式来区分的。从这个角度上来说，JavaScript 的数组是一种动态数组，这也是适合于实现关联数组的一种存储策略。

点数)。因此 JavaScript 必然以关联数组为基础，来实现“(使用索引存取的)数组”这种对象类型。

1.6.1. 关联数组是对象系统的基础

使用关联数组作为实现方案的另一个原因——也许是更加本质的原因——则是在 JavaScript 中，关联数组其实是实现对象系统的基础。也就是说，早在有 Array 类型之前，系统已经为 Object 类型实现好了关联数组，而 Array 只是这种特性的一种应用罢了。

关联数组下标是非序数的，所以它看起来更象是一张“表”，这大概是它在 C++中被称为 map，或在 python 中被称为字典的原因了。JavaScript 的对象的“表特性”非常明显：可以使用“[]”来作为成员存取符，而且成员可以是任意的字符串——而无论该字符串是否可以作为标识符。

更进一步确指地说，JavaScript 中对象（的原型）所持有的属性表^①，就是一个关联数组的内存表达。因而：

👉 所谓属性存取，其实就是查表；

继续从这样的实现细节来考察 JavaScript 的对象，那么：

👉 所谓对象实例，其实就是一个可以动态添加元素的关联数组；

👉 所谓原型继承，其实就是在继承链的关联数组中查找元素。

——由此看来，JavaScript 的内部实现并不怎么神秘。

1.6.2. 用关联数组实现的索引数组

JavaScript 中的数组 (Array 对象类型)，首先是一个对象。因此它的属性存取也是查表，例如：

```
aArray["length"]
```

便是进入了一个关联数组存取的过程。同样，我们去看它作为索引数组时候的特性：

```
aArray[0]
```

其实也与前面完全一样、毫无二致。如果你非要说有什么不同，那大概是将字符串 "length" 换成了数值 0。当然，事实上你也可以使用字符串 0：

^① 参见：XXXXXXXXXX

```
aArray["0"]
```

这与上面的存取效果没有任何的不同。

所以 JavaScript 中的索引数组，只是用数字的形式（内部仍然是字符串的形式）来存取的一个关联数组。由于在这一特性上，它与普通对象没有任何的区别，因此你完全可以用 "in" 运算，或 for ... in 语句来考察它的成员——我这里指的是数组下标 / 元素。例如：

```
var aArray = ['a', 'b', 'c', 'd'];

// 显示 true
alert('1' in aArray);
// 列举元素 0~3
for (i in aArray) {
    alert( i + '=> ' + aArray[i] );
}
```

一般来说，我们对索引数组中的元素进行考察时，应使用递增 / 减序的 for 语句。例如通常实现在数组中查找对象的 indexOf 方法：

```
Array.prototype.indexOf = function(obj) {
    for (var i=0, imax=this.length; i<imax; i++) {
        if (this[i] === obj) return i;
    }
    return -1;
}

var aArray = ['a', 'b', 'c', 'd'];
alert( aArray.indexOf('b') );
```

如同前面所讲述的，由于 JavaScript 中的索引数组并不是真正“连续分配”的有序个元素，因此这样的索引方法并不会真正地带来什么效率——也许这在其它高级语言中是更有效的，亦或有特殊的优先方案。而且当数组变得更加无序、自由存取时，这种列举的效率可能会更差。例如我们可能声明一个数组为一百万个（或更多^①）元素大小，但事实上却没有一个元素被存入数组——JavaScript 的数组在存储特性上是动态数组，是即需即分配的。因此对这种数组做上述列举，会产生大量的虚耗：

```
var aArray = new Array(100000);
for (var i=0, imax=aArray.length; i<imax; i++) {
```

^① JavaScript 的数组的元素数的理论上限是一个无符号 32 位整数大小（大约 40 亿）。

```
// 列举 aArray.length 次  
}
```

然而我们应该记得这个数组本质上仍然是一个关联数组，因此它并非必须使用这个增 / 减序列举的方法。所以你也可以使用一下 `for..in` 来解决问题：

```
var aArray = new Array(100000);  
for (var i in aArray) {  
    // 列举次数 = 真实存在的元素数 + 少数可被列举的重写或修改后的原型成员  
}
```

但在使用这种方法的时候，我们需要注意的是，所列举的元素将不再是某种特定的序列（例如上例中的 `0..100000-1`）。关联数组是无序的，因此我们不能保证 `for..in` 是准确地按照索引数组那样的增 / 减序列来列举。

在 `Array` 对象类型的实现上，JavaScript 主要是保障了 `length` 属性和数组元素维护。基于数组这种数据结构本身的特性，JavaScript 在以下情况中隐式地维护 `length` 属性：

- 👉 使用 `push`、`pop`、`shift`、`unshift` 方法在数组首尾增删元素时；
- 👉 使用 `splice` 方法在数组中删除元素时；

又因为 JavaScript 的数组是动态数组，所以具有如下的特性：

- 👉 给大于或等于 `length` 值的指定下标赋值时，会隐式地重设 `length` 属性值；
- 👉 可以显式地重写 `length` 属性来调整数组大小。

然而，JavaScript 数组的“基于关联数组实现”的这一事实上，带来了更多的疑问。其中最关键的一项是：如果数组是不连续的，那么在变更数组内部成员（及其顺序时），是否要为“索引数组”维护某种连续性？

这取决于“索引数组”的效果是一种真实存取值，还是一种运算规则的假象——我想这样的陈述过于坚涩，而我的意思不过是“JavaScript 的索引数组的连续性，只是一种表现效果”。

在一般语言中，对于一个索引数组来说，如果某个下标没有存放值，那么它的取值应当是空值（`null`、`nil` 或空串等）。在 JavaScript 中，这个效果被解释为“应当是 `undefined` 值”——这样的设定与一般语言是没有什么差异的。但是一般语言为了实现这个效果，得将该数组长度所示的内存区间填以 0 值，因此这是一种实现效果。而 JavaScript 并不为该特性做出任何专门的实现，按照对象系统所约定的“访问不存在的对象成员的值为 `undefined`”，所以数组自然

便有了上面的效果，因而我们说这只是一种表现效果。

正因为这是一种表现效果，所以用 `for` 语句增 / 减序列举时，它就“表现为”索引数组；用 `for..in` 列举时，它就“是”关联数组。而更进一步的推论就是，根本不必为数组“维护某种连续性”——那只是不同语句的“表现效果”。

然而有趣的地方就在这里。我们设定下面这样一个数组：

```
var aArray = new Array(1000*10000); // 1千万个元素
aArray[1] = 3;
aArray[3] = 1;
aArray[5] = 5;
aArray[9999] = 99999;
```

接下来我们问：如果用内部方法 `sort()` 排序该数组，那么会有多少个元素参与运算呢？答案是：仅有上面的四个元素参与运算。下面的测试代码说明这一点：

```
// (续上例)
function func(v1,v2) {
    alert(v1,',',v2,'<br>');
    return (v1>v2 ? 1 : v1==v2 ? 0 : -1)
}
aArray.sort(func);
```

结果显示为^①：

```
9,1
9,5
9,3
1,3
3,5
5,1
3,1
```

但到底是 `undefined` 值不参与运算呢，还是在数组内部是列举有效成员而不是使用 `length` 遍历呢？这样的检测很方便，只需要在 `aArray` 中填入一个 `undefined` 值：

```
aArray[100] = undefined;
```

然后再用上面的例子测试。其结果是：在添加了元素 `aArray[100]` 之后，输出结果没有变化，因此 `undefined` 值是不参与 `sort()` 运算的。

^① JScript 与 SpiderMonkey JavaScript 在显示结果上不一样，看起来后者的排序算法效率更高。

尽管 JScript 与 SpiderMonkey JavaScript 都不处理 `undefined`——因此上例在两个引擎中都不输出 `undefined` 值——但是在列举方法上，二者的选择却不相同：SpiderMonkey JavaScript 将根据 `length` 值列举全部元素，而 JScript 则充分利用关联数组的特性，只列举有效的元素。因此这个例子在实际运行时仍是 JScript 效率更高。

更进一步的测试表明^①：

- 👉 方法 `concat`、`slice`、`splice` 表现为上述相同的效果^②；
- 👉 方法 `pop`、`push` 不受这个问题的影响；
- 👉 JScript 在 `shift`、`unshift`、`reverse` 方法上采用了与 SpiderMonkey JavaScript 一样的方案。

最后，请不要尝试用极庞大（例如 1 亿或更多元素）的数组去测试 `join`、`toString`、`toLocaleString` 与 `valueOf` 这些方法，那会是一场灾难——这些方法很明显地会考察每一个存在或不存在的数组元素。

1.6.3. 干净的对象

这里的“干净的对象”指的是没有声明任何成员的对象直接量，也就是通过“`{ }`”来得到一个没有任何“可被列举”的成员的对象的对象——也可以是通过“`new Object()`”创建的对象。例如：

```
// 示例 1：“干净的对象”没有任何可列举的成员
var obj = {};
for (n in obj) {
    alert(n);
}
```

如果我们试图对象来构造一个用来列举的表，并使用“`in`”运算快速查找该表的话，那么我们会迫切地希望用来构造这个表的对象是“干净的”。如下例：

```
// 示例 2：困扰'in'运算的污染问题
var obj = {
    vl: 1,
```

^① 这是一个与引擎及其版本相关的测试。JavaScript 不会在数组操作中“意外地（相对于 `pop`、`push` 等方法）”增加或减少有效元素的个数。此外，可以使用给指定索引赋值、或用 `delete` 删除指定索引来增删有效元素，而这是数组作为对象时所表现出来的“关联数组”特性。

^② 这几个方法（以及大多数数组的方法）在不同引擎中表现的效果不同。在 SpiderMonkey JavaScript 中 `concat()` 方法会处理值为 `undefined` 的有效元素，而 JScript 则会忽略；在 Opera JavaScript 中，`slice()` 只返回一个包含最大有效元素的数组，但在 JScript 与 SpiderMonkey JavaScript 中，则会尽可能修正返回数的 `length` 属性。

```

v2: 1,
v3: 1
// ...
}

// 导致污染的修改原型操作
Object.prototype.vv = 0;

// 检测对象 obj 是否有某个成员。
// 显示值 true。但如果我们希望 obj 是“干净的”，那么它就不应当有 'vv' 成员。
alert( 'vv' in obj )

```

正是因为我们难以避免一个对象被污染，因此在一些解决方案中才会提供一个名为“**safed in operator**”的函数，以使得用户代码可以有效地检测对象成员。下例说明这种方案：

```

// 示例 3: safed in operator
// (使用上例中的 obj 对象)
$in = function(p, o) {
    var $o = {};
    return function(p, o) {
        return !(p in $o) && (p in o);
    }
}();

// 检测对象 obj，显示值 false
alert( $in('vv', obj) );

```

然而我们仍然不能确保 obj 在声明时就一定“不包括 'vv' 成员”。所以上面这种方案仍然是不够安全的。

更加安全的方法，是保护 **Object()** 构造器，让后续开发人员不能重写系统的 **Object()** 的原型成员。这样一来，在“用户对象系统”与“内部对象系统”之间的那道栅栏，在一定程度上就可以保障“干净的对象”这种有趣的特性——如果没有人刻意利用我们前面讲述的那些技巧来越过这道栅栏的话。下例说明这种方法（在 **SpiderMonkey JavaScript** 与 **JScript** 中是不同的）：

```

// 示例 4: 用重写 Object() 来隔离污染
// (本例中 obj1 与 obj3 是空的对象，obj2 与 obj4 是带初始化成员的对象)

// 1. 重写
function Object() {

```



```

}

// 2. 在 JScript 中的方法
var obj1 = { };
var obj2 = {
    v1: 1,
    v2: 2,
    v3: 3
}

// 3. 在 SpiderMonkey JavaScript 中的方法
function AObject() {
    for (var i=0, args=arguments, argn=args.length; i<argn; i++) {
        this[args[i]] = args[++i];
    }
}
var obj3 = new AObject();
var obj4 = new AObject(
    'v1', 1,
    'v2', 2,
    'v3', 3
);

// 4. 污染的修改原型操作
Object.prototype.vv = 'hi...';

// 5. 在 IE 中显示 false
alert( 'vv' in obj2 );
// 6. 在 IE 与 FF 中都显示 false
alert( 'vv' in obj4 );

```

然而最后你会发现，再干净的对象也终究是对象。因此总有 `toString` 之类的缺省方法与属性。如果需要在查找的目标中包括这些名称，仍然会导致出错。仍以上例为基础，如果我们要在 `obj1~obj4` 中添加一个名为 `'vv'` 的成员，又该如何检测它呢？解决这个问题的方法之一，是放弃使用 `in` 运算，而使用 `hasOwnProperty()`：

```

// (续上例)

// 7. 当没有重写特殊名称时，返回为 false
alert( obj1.hasOwnProperty('toString') );

```

```
// 8. 重写, 使 obj1 (包括 obj1~4) 成员列表中具有特殊的名称
obj1.toString = null;

// 9. 检测, 返回值 true
alert( obj1.hasOwnProperty('toString') );

// 10. 检测 hasOwnProperty 是否被重写的方法
obj1.hasOwnProperty = null;
alert( {}.hasOwnProperty.call(obj1, 'hasOwnProperty') );
```

显然, 如果用户需要一个通用的, 能够正确检测的字典 (意思是指字典中存在 `name/value` 的值对, 并可以检索 `name` 是否存在, 以及通过 `name` 获取到 `value`), 那么可以利用“干净的对象”的思想来实现这个方案:

```
// 示例 5: 实现字典类的解决方案

// 重写
function Object() {
}

// 字典对象 (参见示例 4)
function Dict() { // AObject()
    for (var i=0, args=arguments, argn=args.length; i<argn; i++) {
        this[args[i]] = args[++i];
    }
}

// 工具函数
function inDict(dict, name) {
    return {}.hasOwnProperty.call(dict, name);
}

function getItem(dict, name) {
    return inDict(dict, name) ? dict[name] : undefined;
}

// 使用示例
var dict1 = new Dict();
var dict2 = new Dict(
    'name1', 1,
    'hasOwnProperty', 2
);
// 分别显示 false, undefined, true, 2
```

```
alert( inDict(dict1, 'hasOwnProperty') );
alert( getItem(dict1, 'hasOwnProperty') );
alert( inDict(dict2, 'hasOwnProperty') );
alert( getItem(dict2, 'hasOwnProperty') );
```

1.7. 类型转换

JavaScript 中不可能通过声明来辨别一个变量的数据类型。例如对于下面这行代码：

```
var value;
```

我们显然不能确切地说 `value` 是什么类型——但这个说法稍稍有点问题，因为我们知道 JavaScript 中 `undefined` 也是一种类型，所以说一个已声明但无值的变量，其实是 `undefined` 类型。

但这样一来，我们其实会得出另一个推论：JavaScript 中数据的类型并不是由变量声明来决定的，而是由它包含什么值来决定的。这看起来很拗口。但下面的例子会出进一步的解释：

```
/**
 * 示例：JavaScript 中通过变量存放的值来说明变量的数据类型
 */
var
    value_javascript = 2;
alert( typeof value_javascript ); // 显示 number;

value_javascript = 'abcd';
alert( typeof value_javascript ); // 显示 string;
```

由于“变量能存放任何的值”，所以 JavaScript 并不需要其它强类型语言中的类型转换——JavaScript 没有专门的类型转换运算符。但是“变量声明时没有类型含义”也带来了一个问题：既然解释器并不知道源代码中的变量是什么类型，那么也就无法检错。例如下面这行代码：

```
value_3 = value_1 + value_2;
```

在对源代码做语法解释期间，并不能确知 `value_1` 与 `value_2` 到底是什么数据类型，因此也无法说明二者是否能进行“+”运算——下面还会说到，这时也无法确定“+”所表达的是算术的“求和”还是字符串的“连接”。

由于变量的值的类型只能在代码执行中过程才能获知，因此 JavaScript 也就只能采用“运算过程中执行某种类型转换规则”来解决不同类型间的运算问

题。例如一个字符串与一个数字的“+”运算，事实上就是将数字转换为字符串，然后进行字符串连接运算。所以才会有下面的代码运行结果：

```
var value_1 = '123';
var value_2 = 456;

value_3 = value_1 + value_2;
alert(value_3); // 显示字符串: '123456'
```

——而接下来的问题，就是同样是“+”号运算，为什么是数字转换成字符串，而不是反过来让字符串转换成数字（这样，上面的代码结果值就应该是数值 579 了）呢？这也是“某种类型转换规则”所包含的内容。

在本章节的内容中，我们类型之间相互转换的规则。但在这些主要内容开始之前，我们会对宿主环境下的某些特殊情况做一个小的讨论，以免这些问题使得后续讨论变成混乱。

1.7.1. 宿主环境下的特殊类型系统

我们先来看一个例子。也许读者认为可以通过下面的方法来实现一个能在 IE 和 FireFox 中通用的 `print()` 函数：

```
// 示例 1: 更通用的 print()
print = function() {
    document.writeln.apply(document, arguments);
}
myPrintf('this is a test.');
```

这样的方法用在别处的确是不错的方案，而在 IE 中、针对于 `document.writeln` 这个例子则不行。但这并不是脚本引擎的问题，而是 IE 在实现 DOM 时，将 `document.writeln` 实现成了一个对象（object），而不是一个函数（function）。正因为 `writeln()` 并非一个在引擎级能识别的函数，因此也不可能拥有 `apply`、`call` 等方法，这便是示例 1 在 IE 中将会导致错误的根源了。

宿主环境允许将对象作为函数来执行，这是令人诧异的事。但这终归还是在同一个类型体系之中，也就是说 `typeof` 值仍然是六种 JavaScript 基本类型之一，更复杂的情况是：JScript 允许宿主扩展这个基本类型系统。

下面这个例子用到了 IE 的 xml 扩展，在 xml 扩展中又存在一套类型系统。它尽管与 JScript 的类型系统大致相同，但还是有些差异：

```
var xml = document.createElement("xml");
var rs = xml.recordset;
```

```

rs.Fields.Append("date", 7, 1);
rs.Fields.Append("bin", 205, 1);
rs.Open();
rs.AddNew();
rs.Fields.Item("date").Value = 0;
rs.Fields.Item("bin").Value = 12345;
rs.Update();
var date = rs.Fields.Item("date").Value;
var bin = rs.Fields.Item("bin").Value;
rs.Close();

// 显示两个特殊的类型值'date','unknow'
alert( typeof date )
alert( typeof bin );

```

我们注意到这里出现了'date'和'unknow'这两个特殊类型。而且接下来我们还会发现，这些扩展不但改变了类型系统，而且还破坏了对对象的引用规则：

```

// (续上例)
var date2 = date;
// 以下两项输出都将是 false
alert(date2 == date);
alert(date == date);

```

还记得在 JavaScript 中的一个“有且只有 NaN 是自身不等值”的规则吗？——而这里你看到了变量 `date` 自身是不等值的，而且它显然不是 NaN。同样，你看到 `date2` 应该是 `date` 的一个引用，但他们仍然不等。是的，这样的类型系统与引用规则已经破坏了我们所知的、最核心的那些 JavaScript 语言标准。

下面我们简要说明一下这种问题的成因，它其实是由 JScript 的基础引擎——ActiveScript 的实现方案带来的一些问题。更具体的说，是 Windows 的 COM 系统带来的问题。在这个系统中，所有的数据类型有一个基础定义：变体（Variants）。而 Variants 不但包含 JavaScript 自身识别的六种元类型，也包括其它十余种公共数据类型——这些与类型相关基本规则其实也是后来实现 .NET Framework 中的 CTS（Common Types System）的基础。此外，更加复杂的是：COM 系统允许变体的复合类型，例如一个数据既可以是整型，也可以同时是字符型。

接下来，ActiveScript 允许通过脚本（这里既包括 JScript，也包括 VBScript 等等）来访问 COM 组件。换言之，JScript 必然会面临一种包含了自身不可识别的类型信息的外部组件。在这种情况下，脚本引擎就会为 `typeof` 返回六种

元类型之外的类型值——例如上面的 `date` 与 `unknown`。

出于同样的原因，`typeof` 也可能返回一个复合的变体类型。尽管大多数人未能留意，但这种情况实在并不少见。如果我们直接使用 `Microsoft.XMLHTTP` 组件（在 IE 浏览器中实现 Ajax 的基本技术），那么就会发现这个 `XmlHttp` 从服务器端返回的数据也是一种特殊的类型。下例借用 `VBScript` 的部分代码来检测这种情况：

```
<!-- 以下代码运行在 IE 浏览器环境中 -->
<script>
// 使用 VBScript 代码来实现 getTypeValue () 与 getTypeName () 两个函数
void function() {
    window.execScript (''+
        'Function getTypeValue(v) \n'+
        '  getTypeValue = VarType(v) \n'+
        'End Function\n' +

        'Function getTypeName(t) \n'+
        '  getTypeName = TypeName(t) \n'+
        'End Function', 'VBScript'
    );
}();

// 使用 ajax 技术，读取一个 GB2312 内码的网页
var xmlHttp = new ActiveXObject('Microsoft.XMLHTTP');
var srcUrl = 'http://www.sina.com.cn/';
xmlHttp.open("GET", srcUrl, false);
xmlHttp.send(null)

// 检测返回码
if (xmlHttp.status == 200) {
    // 显示 responseBody 的类型值与名称，分别是 8209 与 'Byte()'
    alert( getTypeValue(xmlHttp.responseBody) );
    alert( getTypeName(xmlHttp.responseBody) );

    // 显示 JScript 所识别的类型：unknown
    alert( typeof(xmlHttp.responseBody) );
}
</script>
```

输出的类型值（8209）等于如下两个常量的和：

```
// 变体类型值，引用自 MSDN
```

```
// Variable type is unsigned char.  
VT_UI1 = 17; // 亦即时字节类型 (vbByte)  
  
// Variable type is a safe array.  
VT_ARRAY = 0x2000; // 十进制值 8192
```

所以它的类型名为 'Byte()', 是以 VBScript 形式说明的 “字节数组”。这个类型在 JScript 系统看来, 则理解为 'unknown', 所以是不可以直接操作的数据类型。

本小节通过对 JScript 类型系统的实现进行分析, 意在指明 JavaScript 的类型系统在不同的宿主环境下可能存在特殊性——因此本章节所述的内容并不能直接应用于不同的宿主环境。为了尽可能避免这些特殊性的影响, 本章接下来的内容只讲述 ECMA Script 标准中所述元类型系统与对象类型之间的类型转换规则。需要强调的是, 这些转换规则是 JavaScript 类型系统设计的基础, 但并不限制或强制具体引擎、宿主的实现与扩展。

1.7.2. 值运算：类型转换的基础

现在先让我们回到元类型系统上来。我们仍然清晰地记得, 元类型其实只有两类: 值类型和引用类型。在这个类型系统中, 只有函数与对象是引用类型的, JavaScript 的对象系统衍生自元类型 object, 函数式语言特性则基于元类型 function。

回顾 “1.3.2.1 运算的实质, 是值运算”, 我们注意到引用类型自身其实并不参与值运算。对于运算系统来说, 引用类型的价值是:

- 👉 标识一组值数据;
- 👉 提供一组存取值数组的规则;
- 👉 在函数中传递与存储引用 (标识)。

所以我们必然面临的问题是: 所谓的类型转换, 其实是指

- 👉 值类型之间的转换^①;
- 👉 将引用类型转换为值类型 (以参与运算)

这样的两种情况。而这, 就是 JavaScript 中类型转换的全部了。

^① undefined 类型是一个特例: 不必讨论某种类型到 undefined 类型的转换。原因很简单, 任何 “存在的值”, 不可能变成 “不存在的(undefined)”。所以这种想法在逻辑上就不通。

1.7.3. 隐式转换

我们把通过函数或方法调用，明确地将类型转为另一类型的称为显式转换。相对应的，不通过函数或方法调用的情况，就称为隐式转换。后者主要是指在用户代码在不明确指示数据类型的情况下，JavaScript 引擎根据代码的上下文推断所需要的数据类型，并完成操作数转换和值运算的过程。这一过程通常发生在表达式运算和某些语句（语义）对数据的强制理解上。

下面分别讨论这两种情况。

1.7.3.1. 运算导致的类型转换

在语法分析期，JavaScript 引擎首先解析到运算符，并将运算符操作的运算元（标识符或直接量值）置入语法树——但在这一过程中并不对运算元的类型做任何的推定。在运行期，引擎执行语法树上（由运算符指定）的运算时，将首先根据运算符所支持的“运算元的类型”进行一次操作数的隐式转换。

这里所说的“运算元的类型”是由运算符决定的，亦即是元类型系统中的值和引用类型。在这里进行的一次语法推断是：是检查引用还是求值运算？在“1.3.2.1 运算的实质，是值运算”中已经分析过这一过程的简洁性。当一个运算符是针对确定数据类型的值运算时，引擎将立即进行一个（对原变量 / 值无副作用的）隐式转换。例如一元运算符“+”和“-”所作用的类型是数值，因此下面的表达式运算将导致一次类型转换，其效果相当于 `parseInt('123')`：

```
+ '123'
```

然而有些运算符存在一定的歧义，同样是“+”运算，即可以是数值求和，也可以是字符串连接，也可以是上面所说的一元运算符^①。那么这时就需要反过来，通过运算数的数据类型来推定运算符含义。仅对两个运算元的“+”运算符来说，JavaScript 设定的规则是：当运算元中有任意一个是字符串时，运算符视为“字符串连接”。而接下来的推断是：因为字符串连接运算将操作两个字符串，所以另一个运算元需要被隐式地转换为字符串。

所以，JavaScript 运算过程中的隐式类型转换，即受到基于运算符的“类型推定”的影响，又受到基于运算元类型的“运算符推定”的影响。这些特性完整地体现了“JavaScript 是一种类型动态化的语言”这一特点。

^① 是否是一元运算符是在语法分析期根据顺序解析的上下文来识别的，所以在运行期不会做这个识别。

关于对这样的类型推定过程的分析，任何人都可能出现错误。例如^①：

```
0 + '' == 0
'' + 0 == 0
1 + '' == 1
'' + 1 == 1
0 + 'a' == '0a'
'a' + 0 == 'a0'
```

上面这些表达，其实仅有最后两项是正确的。前四项中，数值跟字符串的“+”运算总是返回字符串，尽管这些字符串能被转换为右侧的数值，但那已经是又一次类型转换的结果了。更加复杂的类型推定包括下面这样的例子：

```
var x = {};
alert( x[1,2,3,4] );
```

这个例子中的“[]”是对象成员存取运算，因此“1,2,3,4”被作为表达式理解，并隐式地要求将结果值转换为字符串。因此这里最终的运算是：

```
x['4']
```

除了这种概念性的错误之外，有些则是疏于对复杂环境的了解。例如下面这个运算：

```
var x = y = {};
x + y == ? // <-- 结果将是一个字符串
```

一般人会认为既然 x、y 都是对象而非字符串，那么“+”运算符应该视为数值求和运算，于是对象直接量“{ }”应被转换为 NaN，结果值应该是“NaN+NaN=NaN”。但真实的情况是：由于 x、y 是对象，因此引擎先检测其 valueOf() 返回值的类型，并随后调用了 toString() 方法来转换为字符串类型。因此事实上该运算变成了：

```
"[object Object]" + "[object Object]"
```

对这一过程的详细分析参见：

- 👉 1.7.5 从引用到值：深入探究 valueOf() 方法
- 👉 1.7.6 到字符串类型的显式转换”

最后，作为一种补充：一些函数在调用时可能执行一些隐含的强制转换。例如 String.Search() 方法，会转换第一个参数为正则表达式。这看起来很是多余，但它是 ECMA 规范中的一部分。因此类似于这种识别 / 强制参数类型的函数，在调用中也会产生一些隐式的类型转换（并可能因此导致异常）。

^① 这些例子来自 Kris Kowal(<http://cixar.com/~kris.kowal/>)的“JavaScript Tips for Novices, Acolytes, and Gurus”(<http://arstechnica.com/journals/linux.ars/2007/08/27/javascript-for-all-ages>)。

1.7.3.2. 语句(语义)导致的类型转换

一些语句在语义分析时也会做一些强制转换的操作。一个非常明显的例子是“if 语句”必然会把表达式的结果转换为布尔值。与此相同的还有循环中的 while() 语法。这些语法元素需要一个布尔值，因此它会将括号中的表达式运算的结果转换为布尔值。

在对象声明时，也存在标识符到字符串的转换（这种情况是非常罕见的）：

```
obj = {  
  length: 100  // <-- 这里存在一个标识符到字符串的转换  
}
```

这里强调的是“标识符到字符串”的转换（而不是反过来）的原因，是因为在 JavaScript 引擎内部，对象成员名是以字符串形式存在并访问的（参见“1.6 关联数组：对象与数组的动态特性”）。同样的道理，你也可以认为“obj.prototype”这种存取运算也存在一个标识符向字符串转换的过程。

with 语句总是试图打开一个对象的对象闭包，因此如果它作用的表达式返回一个值类型数据，那么 with 语句会通过包装类（参见“1.5 包装类：面向对象的妥协”），将值包装为对象并打开它的闭包。与此类似的，for..in 语句也会有这样的过程。由于包装类的实质是将值类型转换为同类的对象类型，因此这里也可以称为一个隐式的类型转换过程。相同的原因，也可以认为“(2).prototype”这个存取运算中也存在一个包装类的类型转换过程。

最后对 switch 语句作一些补充。switch() 语句试图对表达式求值，并用该值与 case 分支中的值进行比较运算。在比较中采用的是类似于“===”操作符的运算，亦即是说是优先进行类型检测的，因此这里不会发生类型转换过程。因此下面的示例不会进入到 case 分支：

```
var obj = new Number(2);  
switch ( obj ) {  
  case 2: alert('hello');  
}
```

1.7.4. 值类型之间的转换

我们知道把一个值转换成它自己类型的值是没有意义的，例如说字符串“abc”转换成“abc”，这种运算根本就不必要。除此之外，下表列出几种值类型数据之间的转换：

转换到	undefined	number	boolean	string
undefined		1.7.4.1	1.7.4.1	1.7.4.1
number			1.7.4.2	1.7.4.2
boolean		1.7.4.3		1.7.4.3
string		1.7.4.4	1.7.4.4	

1.7.4.1. undefined 的转换

任何值都不能“转换为 undefined”，但反过来却不是。因为 undefined 实际上也需要参与运算，例如你试图显示一个 undefined 值。

undefined 能转换为特殊数字值 NaN，因此它与数字值的运算结果将会是 NaN，而不会导致异常。例如：

```
// 示例：undefined 的转换
var value = undefined;
// 显示 NaN
alert(10 + value);
```

undefined 能转换为字符串 'undefined' 与布尔值 false。它总是恒定的得到这两个值。下面的例子说明这一点：

```
// 示例：undefined 进行的两种转换
var value = undefined;

if (!value) {
    alert( '' + value);
}
```

对于单个的 undefined 值，某些宿主的输出函数可能会显示为字符串 'undefined'，例如在浏览器中 window.alert() 和 document.writeln() 方法；而另一些则有可能什么也不显示，例如在 WScript 环境中的 WScript.Echo()。但这并不一定是类型转换规则的问题，而可能取决于输出函数是否处理 undefined 值。

1.7.4.2. number 的转换

任何值都可以被转换到 number 类型的值。如果转换得不到一个有效的数值，那么结果会是一个 NaN。而 NaN 其实是一个可以参与数值运算的值，这样处理的目的，是使得表达式可以尽量求值，而不是弹出一个异常。

Number 值转换到布尔值时，非零值都转为 true，零与 NaN 都转为 false。

与上述规则相容的情况是，以下五个特殊的数值都会被转换为 **true**：

- 👉 `Number.NEGATIVE_INFINITY`：比 `(-Number.MAX_VALUE)` 更小的值
- 👉 `Number.POSITIVE_INFINITY`：比 `(Number.MAX_VALUE)` 更大的值
- 👉 `Number.MIN_VALUE`：最接近零的正数值
- 👉 `Number.MAX_VALUE`：最大的正数值
- 👉 `Global.Infinity`：`Number.POSITIVE_INFINITY` 的初始值

Number 值在转换到字符串时有极其复杂的内部规则。因此除非使用显式转换（参见“1.7.6.2 从数值到字符串的显式转换”），否则很难保证 **Number** 到字符串这一转换的输出格式。但总的来说，以下特殊值被转换为非数值含义的字符串：

- 👉 `Number.NEGATIVE_INFINITY`：转换为 `'-Infinity'`
- 👉 `Number.POSITIVE_INFINITY`：转换为 `'Infinity'`
- 👉 `Global.Infinity`：转换为 `'Infinity'`
- 👉 `Global.NaN`：转换为 `'NaN'`

除此之外，其它数值都能被转换为一个有数值含义的字符串，例如 `'5e-324'`。

1.7.4.3. **boolean** 的转换

boolean 值的 **true** 和 **false** 总是被转换为数值 1 和 0。因此你也总是可以将布尔值参与到数值运算过程中。例如下面两个示例都将显示数值 1：

```
var checked = true;

// 例 1
var i = false;
if (checked) {
    i++;
}
alert(i);

// 例 2
var i = false;
i += checked;
alert(i);
```

由于例 2 中的加号可能被理解为字符串连接符，因此更加安全的代码应该

是下面的例 3（当然也远比例 1、2 难于理解）：

```
// 例 3
i += !!checked;
```

另一个典型的例子是：

```
// 例 4
alert(true + false);
```

它将显示值 1——注意不是字符串 “turefalse”。因为这里的 “+” 号运算被理解为数值的求和运算。

boolean 值的 true 和 false 总是被转换为字符串的 "true" 和 "false"。

1.7.4.4. string 的转换

如果一个字符串由数值和不多于一个的小数点构成，那么它总是可以被转换为数值的。这种情况下，它被当作十进制数处理，并且类似于下面这样的字符串也被识别为合法：

```
// 显示 21
alert('0022' - 1);

// 显示 2.2
alert('00.22' * 10);

// 显示 2.2
alert('.22' * '100.')
```

如果字符串由 '0x'（零和 x|X）开始作为前缀，且由 0~9、A~F、a~f 这些字符（不包括小数点）构成，则它总是可以被作为十六进制数转换为数值的。也许一些人会因此推论出“以 0 为前缀则表示八进制”，但事实上下面的代码证明这种推论是错误的：

```
/**
 * 例 1
 * 前缀 0 可以用在直接量中，以表示 8 进制
 * 结果值输出： 12
 */
alert(033 - 15);

/**
 * 例 2
 * 前缀 0 用在字符串中，在 (隐式) 转换将被忽略
```

```
* 结果值输出: 18
*/
alert('033' - 15);
```

尽管你能将 `boolean` 值转换为字符串 “true” 和 “false”，但是反过来你却不能把这两个字符串转换对应的 `boolean` 值。在字符串中，有且仅有空字符串能被转换为布尔值 `false`，其它的（任何有值的）字符串在转换后都会得到 `true`。

1.7.4.5. 值类型数据的显式转换

我们先讨论 `Global.parseInt()` 和 `Global.parseFloat()` 这两个方法，它们用于将字符串显式地转换为数值。由于这两个方法是全局对象 (`Global`) 的方法，所以我们通常省去 `Global`，直接调用 `parseInt()` 和 `parseFloat()`。

`parseInt()` 支持传入一个值在 2~36 之间的 “radix” 参数来指定所使用的进制。语法如下：

```
parseInt(aString, [radix])
```

调用 `parseInt()` 时，如果转换不能成功，则返回 `NaN`。如果 `radix` 为 16，无论 `aString` 是否有前缀 “0x”，该字符串都将按 16 进制处理；如果 `radix` 为 8，无论 `aString` 是否有前缀 “0”，则该字符串都将按 16 进制处理。

如果不指定 `radix`，那么将采用数值直接量声明中使用的规则：将有前缀 “0x” 的处理成 16 进制字符串，有前缀 “0” 的处理为 8 进制字符串。然而令人大跌眼镜的是，前缀 “0” 在隐式转换中却并不被识别为 8 进制字符串。所以下面的两个例子的结果并不一致：

```
/**
 * 例 3
 * (同上, 隐式转换)
 * 结果值输出: 18
 */
alert( parseInt('033' - 15) );

/**
 * 例 4
 * 显式地转换中, parseInt() 将前缀 “0” 识别为 8 进制
 * 结果值输出: 12
 */
alert( parseInt('033') - 15 );
```

`parseInt()`和 `parseFloat()`另一项特性在于总是尽可能地得到转换结果。即使字符串中只有(前缀的)部分能被转换,那么该转换也将成功进行。有些时候这项特性很好用,例如在 WEB 开发中我们遇到 CSS 的属性值带单位(pt/px 等)时,我们可以简单地用 `parseFloat()`去清除它,以方便参与其它的运算。在下面这个例子中,我们设定 INPUT 元素的宽为 10em(字符宽),当我们要将它增大一倍时,我们使用了算术运算,这时 `parseFloat()`和 `parseInt()`可以忽略掉单位“em”,使我们不必做一些无谓的字符串截取运算:

```
<!-- HTML 代码 -->
<input id="UserName" style="width: 10em">

<script>
var el = document.getElementById('UserName');
el.style.width = parseFloat(el.style.width) * 2 + 'em';
</script>
```

但有些时候,我们又会因为 `parseFloat()`和 `parseInt()`的这项特性遇到麻烦。在下面这个例子中:

```
var v = '4F';

// 显示 4
alert(parseInt(v));
```

结果值输出 4。但事实上我们可能只是想转换 16 进制字符串,而又忘掉了传入 `radix` 参数(或增加“0x”前缀)而已。

通过显式转换,我们也可以将不包括 `undefined` 在内的任意值、数据等等转换为字符串。不过,这将会涉及到更多的问题,例如如何从对象转换到值。因此,接下来我们先讨论一些基础的话题,而在更靠后的章节中去探讨转换到字符串的细节。

1.7.5. 从引用到值:深入探究 `valueOf()`方法

在元类型系统中的,三种值类型数据到“对象(object)”的转换,是通过包装类来实现的。这在前面的“1.5 包装类:面向对象的妥协”章节中已经讨论过了。而本章节则主要讨论逆转过来的情况:对象(以及其它引用类型)如何参与值运算。

首先,如果该一个数据是引用类型,且该数据需要进行值运算,那么引擎将先调用它(或它经过包装后的对象)的 `valueOf()`方法求值,然后以该值参与

值运算。对于 JavaScript 中所已知的对象来说，下表中列出的前面几种的 `valueOf()` 值都会是有效的值类型，因此可以满足上述规则进行运算。

对象实例	<code>valueOf()</code> 返回值	备注
Boolean	布尔值。	
Number	数字值。	
String	字符串值。	
Array	字符串值。与 <code>Array.toString</code> 和 <code>Array.join</code> 方法相同。	
Date	数字值。从 1970 年 1 月 1 日午夜开始计的毫秒数。	
Function	函数本身。	引用类型
Object	对象本身。	引用类型
RegExp	正则对象本身。	引用类型
Error	错误对象本身。	引用类型

在下例中，我们先通过重写 `valueOf()` 方法捕获这一运算过程，然后显示运算后的值：

```
1 // 1. 备份旧的方法，并替换成新方法
2 void function() {
3   _valueOf = Boolean.prototype.valueOf;
4   Boolean.prototype.valueOf = function() {
5     alert(typeof this);
6     return _valueOf.call(this);
7   }
8 }();
9
10 // 2. 创建一个布尔对象，并声明一个布尔值
11 var obj = new Boolean(true);
12 var v = false;
13
14 // 3. 尝试数值运算
15 alert( v + obj );
```

执行第 15 行后将有两个输出，先是在第 5 行捕获到对象 `obj`，并输出它的类型字符串 `'object'`，然后是在第 15 行输出运算结果值 `1`。这证明：当一个引用类型（例如对象）参与值运算时，会先调用它的 `valueOf()` 方法取值。

不过这个例子构造得有点巧妙——你应该留意到 15 行是 `“+”` 运算，而不是某种布尔运算，例如 `“&&”` 或 `“||”`。然而当你改变这行代码为：

```
15 alert( v && obj );
```

又会发现输出中不会包括第 5 行的结果，也就是说 `valueOf()` 没有参与运算。但这并不是 `valueOf()` 的问题，而是因为变量 `v` 的值为 `false`，因此在做 `“&&”` 运

算时导致“短路”，从而优化掉了后面的运算。当然，你也可以改成下面的代码来避免“短路”：

```
15 alert( v || obj );
```

但你还是会发现 `valueOf()` 仍然不参与运算。而这又是另外的规则导致的了：在布尔运算中，无值的（`0`, `undefined`, `null` 和空字符串）被作为 `false`，而其它（有值的数据）直接作为 `true` 而无需类型转换。

所以为了避开这些复杂的、有关布尔运算自身的特性，在前面的例子中我们使用了“+”来做求和运算——这比较便于我们展示 `valueOf()` 的性质。

但是仍然有一部分引用类型数据，通过它的 `valueOf()` 方法还是不能得到一个有效的值。例如表格中的 `Object`、`Error`、`RegExp` 对象，以及 `Function` 对象。在这种“`valueOf()` 返回引用类型”的情况下，引擎会再次调用 `toString()` 方法以取得一个字符串值。字符串总是可以以值类型的形式参与运算的，这保证了 JavaScript 内部的任意数据总是可以直接参与值运算。

为了使这个过程展示得更加清楚，我们修改上面的例子，使 `valueOf()` 返回一个引用，然后我们再观察 `toString()` 的效果：

```
1 // 1. 备份旧的方法，并替换成新方法
2 void function() {
3     _valueOf = Boolean.prototype.valueOf;
4     _toString = Boolean.prototype.toString;
5     Boolean.prototype.valueOf = function() {
6         alert('do valueOf...');
7         return {}; // <-- 返回一个引用类型数据
8     }
9     Boolean.prototype.toString = function() {
10        alert('do toString...');
11        return toString.call(this);
12    }
13 }();
14
15 // 2. 创建一个布尔对象，并声明一个布尔值
16 var obj = new Boolean(true);
17 var v = false;
18
19 // 3. 尝试数值运算
20 alert( v + obj );
```

这个测试将返回三条信息：

```
do valueOf...
do toString...
falsetrue
```

前两条信息证明在 `valueOf()` 返回一个引用（本例中是一个空的对象）时，将会调用 `toString()` 方法。第三条信息是由 `false`、`true` 两个字符串拼起来的。这是因为运算过程中先调用了 `obj.toString()` 并返回了一个字符串，而按照值运算的规则，当“+”的运算元中有出现字符串值时，该运算符被理解为“字符串连接运算符”，并将全部运算元转为字符串参与运算。因此变量 `v` 也被转换为字符串 `'false'`，连接之后的值就成了 `'falsetrue'`。

这个例子也反映了另外一个问题：（元类型中的）值类型数据在类型转换中并不会并调用 `toString()` 或 `valueOf()` 方法，也不会为此进行包装类的操作。上例中，变量 `v` 最终需要以字符串形式参与运算，就直接使用了 `'false'` 值，而没有调用 `Boolean.prototype.toString()` 方法。

这样的 `valueOf()` 与 `toString()` 规则在 JScript、SpiderMonkey JavaScript 中是一致的。但是，上例的 `valueOf()` 方法在 SpiderMonkey JavaScript 中被调用的次数会多一些——第 11 行代码中 `call()` 方法就会导致这样的一次运算。这也因此会使得上例在 SpiderMonkey JavaScript 中出现异常——需要强调的是，类型转换规则并没有因此改变，而只是 `valueOf()` 的使用场合变多了^①。

1.7.6. 到字符串类型的显式转换

现在我们知道：

- 👉 如果数据是值类型，则直接参与值运算；否则，
- 👉 如果 `valueOf()` 返回一个值类型数据，则以该数据参与值运算；否则，
- 👉 使用 `toString()` 返回的字符串参与运算。

那么可以想见的是：有很多的数据都会以字符串形式参与运算。的确如此，JavaScript 除了展现出一种“完整的”、“有趣的”和“基础的”值运算规则之外，在实用中它的绝大多数运算都会是字符串运算——因为它一直以来的主要应用环境就是浏览器。

^① SpiderMonkey JavaScript 中，当调用 `call` 与 `apply` 方法时，引擎会先尝试对第一个参数 `"thisObject"` 作预处理 (`js_ValueToObject`)，其目的是通过包装类使一些值数据变成对象。在这个强制调用的预处理中，会首先检测 `thisObject` 是值，或已经是对象——于是 `Boolean.prototype.valueOf()` 就被第二次调用了。

因此在 JavaScript 中的任何一种数据类型，都可以（显式或隐式地）转换到字符串类型。提供这种能力的前提是：JavaScript 约定一切都是对象（`undefined` 值除外），因而必然存在 `toString()` 方法，也就可以取得当前对象的字符串表示值。事实上，隐式转换其实也是通过调用 `toString()` 方法达到转换数据的目的，只不过是 JavaScript 引擎来负责调用这个方法而已。

在 ECMAScript v3 标准中，对象还有一个 `toLocaleString()` 方法。它的含义是根据本地规范来显示某种格式化的字符串，它被应用于 `Array`、`Date`、`Number`、`Object` 等对象，它的效果与引擎的、操作系统环境的设置有关。在缺省情况下它与 `toString()` 的效果是一致的，且隐式转换使用 `toString()` 而不是 `toLocaleString()`，因此在本书中不详细讨论它。

1.7.6.1. 重写 `toString()` 方法

重写 `toString()` 方法很明显地会影响到某种类型到字符串的显式转换，因为这正是 `toString()` 作为方法的价值所在。然而我们上面也说到它同样会影响到隐式转换过程。这在上一节关于 `valueOf()` 的示例中便已经讲到了——示例中 `Boolean.prototype.toString()` 的重写影响到了一个布尔对象在运算过程中隐式地转换到字符串值。由于在“某种类型到字符串类型”的转换过程中，显式转换与隐式转换是同一的，所以那行代码效果完全等同于：

```
20 alert( v + obj.toString() );
```

但 `toString()` 对类型系统的影响是非常复杂的——当然也同时是灵活的。例如 `String` 对象也有 `toString()` 方法，也能被重写。那么，到底 `String` 对象值本身还是 `toString()` 值存在运算意义呢？

根据前面的规则，由于 `String` 对象的 `valueOf()` 值返回一个值类型的字符串值，因此我们应该明确运算过程中并不需要调用 `toString()` 方法。下例说明这一点：

```
1  /**
2   * 示例：重写 String 对象的 toString() 对隐式转换的影响
3   */
4  String.prototype.toString = function() {
5      return 'hi, ';
6  }
7
8  // 显示结果值为 'hello, world.'
9  var str = new String('hello, ');
```

```
10 alert(str + 'world.');
```

可见在第 10 行中的隐式转换并没有用到字符串对象的 `toString()` 方法。我们要让 `toString()` 方法生效，比较正确的方法就是让 `valueOf()` 返回引用而不是值：

```
1  /**
2   * 示例：重写 String 对象的 valueOf() 和 toString() 对隐式转换的影响
3   */
4  String.prototype.valueOf = function() {
5      return this;    // this 对象是一个引用
6  }
7  String.prototype.toString = function() {
8      return 'hi, ';
9  }
10
11 // 显示结果值为 hi, world.
12 var str = new String('hello, ');
13 alert(str + 'world.');
```

1.7.6.2. 从数值到字符串的显式转换

除了使用数值对象的显式方式——标准的 `toString()` 方法来转换之外，我们还可以使用 `toString()` 的一种扩展形式：

```
aNumber.toString([radix])
```

其中 `radix` 表示进制。例如将一个数值转换为十六进制或八进制：

```
var value = 1234567;

// 显示 12d687
alert( value.toString(16) );

// 显示 4553207
alert( value.toString(8) );
```

尽管我们平常不会用到 22 进制 (或其它进制)，但是你仍然可以在 JavaScript 中进行这种转换。对于超过 10 进制的表示值，JavaScript 会用 “a~z” 的小字 ASCII 字符来表示。因此，`radix` 的值的边界是 “2~36” 例如：

```
var value = 1234567;

// 显示 100101101011010000111
alert( value.toString(2) );

// 显示 qglj
alert( value.toString(36) );
```

`toString()`转换的结果不会根据你的意愿来补零前缀，而且默认总是小写的字符。因此如果你希望将数字 1234567 转换成十六进制的“0012D687”，就得多费点手脚。例如使用下面这样的代码：

```
Number.prototype.toString = function() {
    var _toString = Number.prototype.toString;

    return function(radix, length) {
        var result = _toString.apply(this, arguments).toUpperCase();
        var length = length || 0;
        return (result.length >= length ? result
            : (new Array(length - result.length)).concat(result).join('0')
        );
    };
}();

var value = 1234567;

// 显示 0012D687
alert( value.toString(16, 8) );
```

在进行 `Number` 到字符串的转换时，还存在两个问题：指定小数点的位置与如何启用指数记数法。这需要使用到 `Number` 对象的另外三个方法，分别是：

- 👉 `toFixed(digits)`：使用定点计数法，返回串的小数点后有 `digits` 位数字；
- 👉 `toExponential(digits)`：使用指数计数法，使返回串的小数点后有 `digits` 位数字；
- 👉 `toPrecision(precision)`：使用定点计数法，返回串包括 `precision` 个有效数字（如果整数部分少于 `precision` 位，则在小数部分用 0 补齐）。如果整数部分多于 `precision` 位，则使用指数计数法，并使小数点后有 `precision-1` 个数字。

1.7.6.3. 其它类型的显式转换

在 JavaScript 各种对象中，存在一组以“to”为前缀的方法，这些方法无一例外的都用于转换为字符串^①：

- 👉 `Date.toGMTString`：格林威治标准时间 (GMT) 表示的日期字符串；
- 👉 `Date.toUTCString`：全球标准时间 (GMT) 表示的日期字符串；
- 👉 `Date.toString`、`Date.toTimeString`：引擎默认格式的日期字符串（`Date` 部分、`Time` 部分）；

^① 在 ECMA 标准之外可能会有有一些特别的实现。例如 JScript 中存在 `VBArray.toArray()` 方法，转换的结果就是数组对象，而不是字符串值。

- 👉 `Date.toLocaleDateString`、`Date.toLocaleTimeString`：宿主环境的当前时区、当前系统配置 的日期字符串（`Date` 部分、`Time` 部分）；
- 👉 `<对象>.toLocaleString`：`Array`、`Date`、`Number`、`Object` 等类（的实例）具有该方法。除 `Object` 之外，其它类的该方法都有独自的实现，而不是直接指向 `toString` 方法。
- 👉 `String.toUpperCase`、`String.toLowerCase`：字符串转换为大、小写。
- 👉 `String.toLocaleUpperCase`、`String.toLocaleLowerCase`：字符串转换为大、小写，并使用本地字符串格式。在多语言环境、不同的内码中可能会有不同表现。
- 👉 `Number.toFixed`、`Number.toExponential`、`Number.toPrecision`：参见上一小节。

1.7.6.4. 序列化

通过字符串作为中介，你可以拥有两个相同功能、但不存在引用关系的函数。例如下面的代码：

```
func1 = function() {  
    // ...  
}  
  
// 得到与 func1 相同功能的函数 func2  
func2 = eval( func1.toString() );
```

类似的这种技术被称为“序列化”与“反序列化”。很多开发人员希望语言能够将代码（而不仅是数据）也序列化并存储，而这在绝大多数通用语言中并不提供。但是在 JavaScript 中，由于任何对象都有 `toString()` 方法，因此天生就支持将函数和对象方法序列化为字符串。

不过也许一切并不那么理想。`toString()` 方法涉及到很多的问题，其中之一就是“`<对象>.toString()`”本质上并不等义于序列化——仅有少数对象能用 `toString()` 来序列化。这其中比较常见的包括 `Array` 与 `Function`，然而即使对于这些对象，`toString()` 的序列化效果也并不令人满意。例如 `Array` 的 `toString()` 不能支持多维数组，也不能有效的处理元素中的引用类型；而 `Function` 的 `toString()` 则可能“意外地”附上函数名^①——但某些情况下留着这个名字也是有用的。下面说明这两种情况：

```
var arr = [[1,2],[3,4]];  
function foo() {
```

^① 可见的函数名给代码带来了一定的风险。例如被调用函数可以通过 `arguments.callee.caller` 来获得调用者函数，并进而得到调用者函数的名称。但 JavaScript 允许重写 `toString()` 方法，因此事实上我们也可以通过这样的技巧来避免用户代码知道调用者的名字——不过重写效果能被 `delete` 运算清除，因此并不是绝对安全的。

```
// ...
}

// 示例 1: 返回字符串中不包括多维数组的信息
alert( arr.toString() );

// 示例 2: 返回字符串中包含了“不必要的”函数名
alert( foo.toString() );
```

所以大多数情况下我们并不能用 `toString()` 来实现序列化。在 SpiderMonkey JavaScript 中，是用函数 `uneval()` 来做这件事。对于上例上来说，如果使用 `uneval()` 而不是 `toString()` 来处理：

```
// (续上例，适用于 SpiderMonkey JavaScript 引擎)

alert( uneval(arr) );
alert( uneval(foo) );
```

那么返回的结果将是：

```
[[1,2], [3,4]]
function foo() {}
```

请留意函数 `foo` 的名字仍然未被过滤——如果需要，你能够用正则表达式来过滤掉它。在 SpiderMonkey JavaScript 中，为了实现 `uneval()` 函数，引擎为每个对象定义了一个名为 `toSource()` 的方法。它类同于 `toString()`，但只用于 `uneval()` 函数^①。因此上述代码的效果完全等义于：

```
// (续上例，适用于 SpiderMonkey JavaScript 引擎，等义于调用 uneval() 函数)
alert( arr.toSource() );
alert( foo.toSource() );
```

同样的原因，我们也可以通过重写 `toSource()` 方法来改变 `uneval()` 的效果：

```
// (续上例，适用于 SpiderMonkey JavaScript 引擎，重写 toSource() 以影响 uneval() 函数)
foo.toSource = function() {
    return 'hide the function.'
}
alert( uneval(foo) );
```

在 JScript 中没有与此类似的机制，但一些第三方的代码尝试实现兼容的 `uneval()` 函数。例如 Esteban AlCapousta 在 JScript 环境下实现类似于 JavaScript 中 `uneval()` 函数的代码。Esteban 的方案能识别 `Number`、`String`、`Array`、`Function`、`Boolean`、`RegExp`、`Object`、`Date` 这几种类型并将它们序列化为字符串。其基

^① `toSource()` 与 `toString()` 的不同处不仅于此。事实上 `toSource()` 还有格式化代码的能力：当它传入一个数值参数时，可以表示进行代码格式化，该数值表示格式化时在左侧缩进空格数。

本的思路是通过：

- 👉 列举对象（Object）成员
- 👉 列举数组（Array）元素
- 👉 返回<对象>.toString()值

这三种的方法来处理上述类型到字符串的转换。这种方案与 SpiderMonkey JavaScript 中使用 toSource()的机制并没有基本的相似性。

将序列化的结果字符串转换为原始对象的技术称为反序列化。如同函数名 uneval()所展现的一样，它的逆过程就是 eval()。JavaScript 中，eval()用于执行字符串并返回执行结果，而如果字符串是“直接量单值表达式”，那么自然就返回该直接量——这与字符串的反序列化的语义是完全一致的。

不过一般来说，序列化时处理的是值本身，而如果将这个序列化结果直接用于 eval()，会存在语法问题——这在“1.2.2 动态执行过程中的语句、表达式与值”中已经讲述过了。因此 JSON^①采用在字符串外面套上一对“()”表明强制表达式运算，并交由 eval()处理；Esteban 的方案似乎疏漏了这一点。

1.8. 综述

特性、由字符串而非标识符进行成员存取，使对象系统可以任意扩展

特性、数据结构的解绑（例如字符串、数组、结构与对象等不需要连续存储：结构化编程建立在一个基本假设之上：存取连续数据比存取离散数据高效。但动态语言中并没有在内存中实际连续的数据结构。）

特性、动态绑定与动态类型转换（动态语言）

特性、由 eval()带来的运算、语句与代码块

特性、object.apply()与 object.call()方法带来的动态方法调用

特性、弱类型识别：例如用对象模拟数组，以及 arguments 的数组特性

特性、编译器只解释声明，不执行运算，即使运算是可预测的（例如立即值或立即值的运算）——强调：只有函数是声明即有值的，也是唯一在立即值声明的同时可以声明标识符的。

特性、valueOf()会影响到比较运算，其根源在于类型转换。Date()能参与序列检测的根源。

特性、语法解释与表达式执行是分阶段的。试以此观点解释“var i=100”和“i=100”的不同。

特性、重写构造器带来的问题，尤其是重写 Object 和 call()方法带来的问题。

特性、eval 对函数调用栈的破坏：重要的是指出 eval()中的代码在列举栈时，ff 与 ie 呈现的不同效果。

^① JSON(JavaScript Object Notation)，是一种轻量的数据交换格式，它最初是基于 JavaScript 实现的。得益于 ajax 的迅速发展，JSON 已经扩展为在几十种语言 / 框架中通用的一种数据交换（序列化与反序列化）协议。

第三部分 最佳实践

（关于语言是什么的问题，）我的观点是，一个通用程序设计语言必须同时是所有的这些东西，这样才能服务于它缤纷繁杂的用户集合。但也有惟一的一种东西，语言绝不能是——这也将使它无法生存——它不能仅仅是一些“精巧”特征的汇集。

我始终不渝的信念是，所有成功的语言都是逐渐成长起来的，而不是仅根据某个第一原则设计出来的。原则是第一个设计的基础，也指导着语言的进一步演化。但无论如何，即使是原则本身也同样是会发展的。

——C++之父，ACM 院士 Bjarne Stroustrup

引自：《C++语言的设计和演化》

第六章 Qomo 框架的核心技术与实现

Qomo 的全称是 Qomolangma OpenProject。它是一个真正的、完整的 OOP 框架，它支持接口 (Interface)、命名空间 (namespace)、面向切面编程 (AOP)、异步 javascript 和 XML 框架 (ajax)、模板编程等多种编程框架和技术体系。

1.1.Qomo 框架的技术发展与基本特性

Qomo 是一个偏向技术研究的 JavaScript 扩展框架。如第一章中所讲述的，本书讲述 Qomo 的目的只是让读者看到 JavaScript 的语言特性的应用，而非刻意地推荐某种框架^①。

Qomo（以及其它开源项目）总被提及到的问题是：为什么是这样。然而框架“被设计成什么样子”，总是源于某些其历史原因或具体应用环境限制的，因此我总是无法用简单的语言给出一个完美的答案。本章节将全面地叙述 Qomo 在技术上的历史沿革，以及基本特性、限制和原则。这或许有利读者对 Qomo 框架进行剪裁，或有针对性的研究。

1.1.1. Qomo 框架的技术发展

1.1.1.1. 两种技术方案的选择

在开发 Qomo 的前身 WEUI 之前，我就对 JavaScript 下的 OOP 编程展开了研究。JavaScript 被称为“基于对象”而不是“面向对象”的原因，在于它的对象系统中的封装和继承特性都实现得并不完整。而我们知道，在通用语言的世界里完整的实现 OOP 特性的包括有 Delphi、C++或 Java 等等，于是许多的开发人员基于这些语言的基本思想，展开了对 JavaScript 在框架级别上的扩展。

这通常有两种做法。一种是创建与通用语言相似的语法结构，并用单独编写的 Syntax Parser 来将这种语法的代码译成 JavaScript 风格的——无论是静态的转译，还是在执行期动态地翻译——这事实上是实现了一种语言的转换工具。在第一章中提及的开源项目 SuperClass 就是在这种方案下，以 Java 语法为基础的语言扩展。SuperClass 把自己当成一个语法解释器，并在这个前提下

^① 如果读者对我在书中写了某种特定框架而耿耿于怀，那么我想我写任何一种框架都总会有人耿耿于怀——固而这不是一种技术探索者应有的心态。

约定了语法规则，要求通过装载机（一个指定的函数 / 方法）载入程序，然后根据语法解析脚本、生成新的类和继承关系，从而完成了翻译一套“面向对象的脚本语法”的功能。

在 WEUI 项目开发的早期，我曾设想过这样的方案。更具体地说，我希望开发人员写下这样的代码：

```
// 1. 声明
function MyClass() {
  protect: {
    // ...
  }
  private: {
    // ...
  }
  init: {
    //...
  }
}

// 2. 注册
WEUI.Classes.Extends(MyClass);

// 3. 创建实例
var obj = new MyClass();
```

这个方案的取巧之处在于：我们在函数内部使用标签 'protect'、'private'、'init' 等等来替代语法关键字。在编译时，这种“<标签>:”的结构非常便于语法分析；在开发时，这种块结构的语法也非常便于开发人员理解和书写代码。

然而后来我终于意识到这样做所带来的问题：

- 👉 实际执行代码与开发人员原始编写代码并不一致；
- 👉 用脚本语言 (JavaScript) 做一种新语言的解释器效率过低；
- 👉 解释器做了多少工作，对开发人员来说是黑箱；
- 👉 过度的扩展语法会使得“新的 OOP JavaScript”的学习门槛过高。

于是 WEUI 的进入正式开发阶段就放弃了这种做法。

不过这种方案并没有被 JavaScript 的爱好者们抛弃。当人们希望创建一种新的语法风格时，便会很自然地应用这种方案：从一个独立的 parser 开始，一直到 compiler、executor 等等逐一实现。其中最有名的应当是 JavaScript 之父

Brendan Eich 在 2004 年完成的一套 Narcissus，他的原意是“在 JS 中实现 JS(JS implemented in JS)”——大多数情况下，“用语言实现该语言自身”被用于证明语言的完备性。在不依赖引擎特性的情况下，Narcissus 完整地实现了 SpiderMonkey JavaScript 1.5 的解释与执行。在 Brendan 完成 Narcissus 的两年多之后，Neil Mix 对 Narcissus 做了进一步修改：

- 👉 使它可以在 JScript 引擎上使用
- 👉 修改了编译与执行模块
- 👉 添加了 C 语言上著名的 Yielding 语法符号 “->”
- 👉 添加了专用的运行期库(Runtime Library)

从而发布了另一个著名的开源项目“Narrative JavaScript”。这套开源项目最令人瞩目的特性是为 JavaScript 添加了真正的多线程支持——事实上 Narrative 实现了一个独立的执行器并管理了代码执行指针^①。

JavaScript 框架级别扩展的另一种做法，是基于 JavaScript 自身的语言风格，在构建的基类上添加大量的方法——这些基类上的方法相当于通用语言中的语法关键字——这种方法相当于扩展了一套 JavaScript 的基础库或基础类库。这是后来 Bindows 采用的方法。与此相似的，Yahoo UI、Microsoft Atlas 等著名的开源项目也无一例外地采用了这种方案。

这种方案规避了前一种方案带来的问题。但是由于这种新框架的特性都依附于基类方法或属性，因此可定制与剪裁的能力很弱——基本上开发人员在脱离这种框架下的任何行为都可能是灾难。而且，事实上让情况更糟糕的是：如果开发人员试图混用两种以上的框架，那么框架对公共变量和对象成员名的争用将会导致两个框架都不能正常运行。

于是不同的框架都提供了“命名空间”的特性来使自己被隔离在一个独立的根命名空间之下。这的确很快解决了问题，但是当用户代码“必须”运行在一个命名空间中，并且类创建、继承等都必须使用空间全名的时候，大多数开发人员就开始拒绝将这种框架应用在一些“并不那么庞大”的浏览器应用中了。

WEUI 基本上是采用了后面这种方案，但更多的考虑了性能与第三方代码兼容问题。

^① 在 Narrative JavaScript 1.7 中，效率问题随 Narcissus 与生俱来：分析 150k 代码大约需要 8 分钟——不包括执行时间。后来 Qomo 项目组向 Narcissus 提交了一份正则表达式的优化方案，使得该时间可以缩至 8 秒。但这样的效率仍然离实际应用很远。

1.1.1.2. WEUI 所奠定的基本风格

自 2004 年初开始的一年多时间里，WEUI 项目组完成 OOP 框架、数据库和 UI 表现层。其中 OOP 框架采用了“类抄写”的技术，基本思想是：

- 👉 在注册类的时候遍历类的所有属性，然后构建类信息结构将它们放在“<构造器>.constructor”成员中；
- 👉 在实例化对象时，列举对象全部成员，将根据“<构造器>.constructor”中的类信息初始化这些对象成员。

WEUI 的出发点是：应当通过框架的执行过程来构造整个对象系统，而不是通过语法分析。因此上述的两个“列举”都是基于 JavaScript 的“for..in”语法对对象实例的考察而进行的。这样做的好处是效率比较高，坏处则是如果加载第三方框架，则会变得不可收拾——第三方框架可能通过原型对 WEUI 的对象系统进行未知的任何修改。

WEUI 的另一个问题是，由于类信息存储于“<构造器>.constructor”，因此事实上它是可见、可存取和可修改的。这些类信息除了“在 JScript 下不可列举^①”之外，并没有任何安全性可言。

但是 WEUI 约定了这种框架的基本风格。准确地说，它约定了在这种框架上书写 OOP 代码的基本风格。主要包括：

```
// 1. 类声明
function MyObject() {
    // ...
}

// 2. 类注册
TMyObject = Class(TObject, 'MyObject');

// 3. 实例创建
var obj1 = new MyObject();
```

此外，它也允许使用一种“类似 Delphi 的”语法来创建对象实例：

```
// 4. 类似 Delphi 的实例创建语法
var obj2 = TMyObject.Create();
```

^① JavaScript 中对象的内部成员是会不会在 for..in 列举中出现的。对这一特性，JScript 引擎与 SpiderMonkey 有着不同的实现：对于前者，重写这些成员不会有任何影响；对于后者，重写内部成员会导致他们在 for..in 列举中变得可见。从本质上来说，前者的“成员可见性”是基于成员名称的，而后者则基于成员在内部属性表中的性质。

这与上一种语法所创建的对象实例没有任何的区别。

WEUI 也约定，在类声明中的代码主要用于描述“类的形式”——也就是类的成员、事件等等，以及包括一个实例构造周期的声明——它完全等义于 JavaScript 中的构造器函数。

WEUI 另外一项有关类声明的关键约定是针对“特性(Attribute)”^①的——所谓特性，是指可以定制“读写器(getter/setter)”的属性。这项约定是：特性是属于每个对象实例的性质，但它在 properties 列表中不可见——也就是不能用 for..in 来列举，不过你能够使用“<实例>.set()”和“<实例>.get()”方法来存取这些特性。

好的，这就是 WEUI 最基础的一些东西了，我们接下来开始讲 Qomo。

1.1.1.3. Qomo 完善了整个内核框架

Qomo 项目于 2005 年末启动，它自一开始便立意于继承和发展 WEUI 框架。它面临的主要问题是：

- ☞ WEUI 的类信息放在“<构造器>.constructor”中是不安全的；
- ☞ WEUI 所使用的“类抄写”开销太大；
- ☞ WEUI 支持的语言、框架特性不够丰富。

第一个问题在上一小节已经讲述过。第二个问题的实际状况是：WEUI 将对象的实例信息暂存起来，然后通过 for..in 列举逐一检查，并确定在新构建的实例上如何初始化成员（属性、方法或多投事件）。无论是哪个类或者该类处理怎样的继承层次，每个实例的创建过程都是这样：

```
// _this 是一个构造函数创建的一般对象，从构造器原型中获得的 ClassInfo 成员中包括类信息
// _constructor 是开发人员所声明的原始的构造器
// _attr 是类注册时创建的特性列表
var AClass = _this.ClassInfo;
var _constructor = AClass.constructor._constructor;
var _attr = AClass.constructor.attributes;

// 执行流程(1)，为对象实例抄写特性(Attribute)的初值
_this._attributes = {};
for (var i in _attr) _this._attributes[i] = _attr[i];
```

^① 这个名词来自于.NET中的用户可定制特性——尽管它成 DOM 中的 Attribute 冲突，但我们实在找不出一个更好的名词了。

```
// 执行流程(2)，为对象实例绑定事件派发器对象
_CLASSBIND.call(_this);

// 执行流程(3)：对象重载的构造函数
_constructor.call(_this, arguments);
```

类抄写使得 JavaScript 的原型继承完全失去了意义。因为每一个实例都是经过复制、绑定的过程，从一个空白的对象复制而来，因此当类继承层次变得更深，或成员过多的时候，实例创建就将会是一场灾难。更深层面的问题是：因为类继承层次之间是通过外在的 `ClassInfo` 成员来维护继承性，因此 JavaScript 原生的 `instanceof` 运算失效了。

WEUI 采用的类抄写方案起源于我对 Delphi 的类继承体系的理解，也是对这种类继承体系的生硬抄袭。后来我看到的另一些方案也采用了这种技术，即：

```
// 1. 构造器作为普通函数，用于声明类成员或相关信息
function _constructor() {
    this.XXX = ...;
    this.YYY = ...;
}

// 2. 使用类抄写来实现继承
_constructor.call(_this, ...);
```

通过这样的方案来实现的继承无一例外的带有上述的问题。

因此在类继承体系方面，Qomo 的技术方案基于我对 JavaScript 的原型继承与函数式特性的重新认识而设计。新方案用函数闭包来“封装”原来由“<构造器>.constructor”保存的类信息；并且将所有公共成员放在构造器原型中，这样实例创建便可以充分利用原型特性，而 `instanceof` 也可以有效使用。

除此之外，Qomo 对基于类继承的面向对象体系做出了两项重要约定：

- 👉 明确、严格地区分“类构造周期”与“实例构造周期”，两个阶段具有各自特有的代码书写方法和上下文环境；
- 👉 在使用带读写器的“特性(attribute)”时，只允许使用“<实例>.get()”和“<实例>.set()”方法存取，不再支持“setXXX”或“getXXX”这样的方法调用。

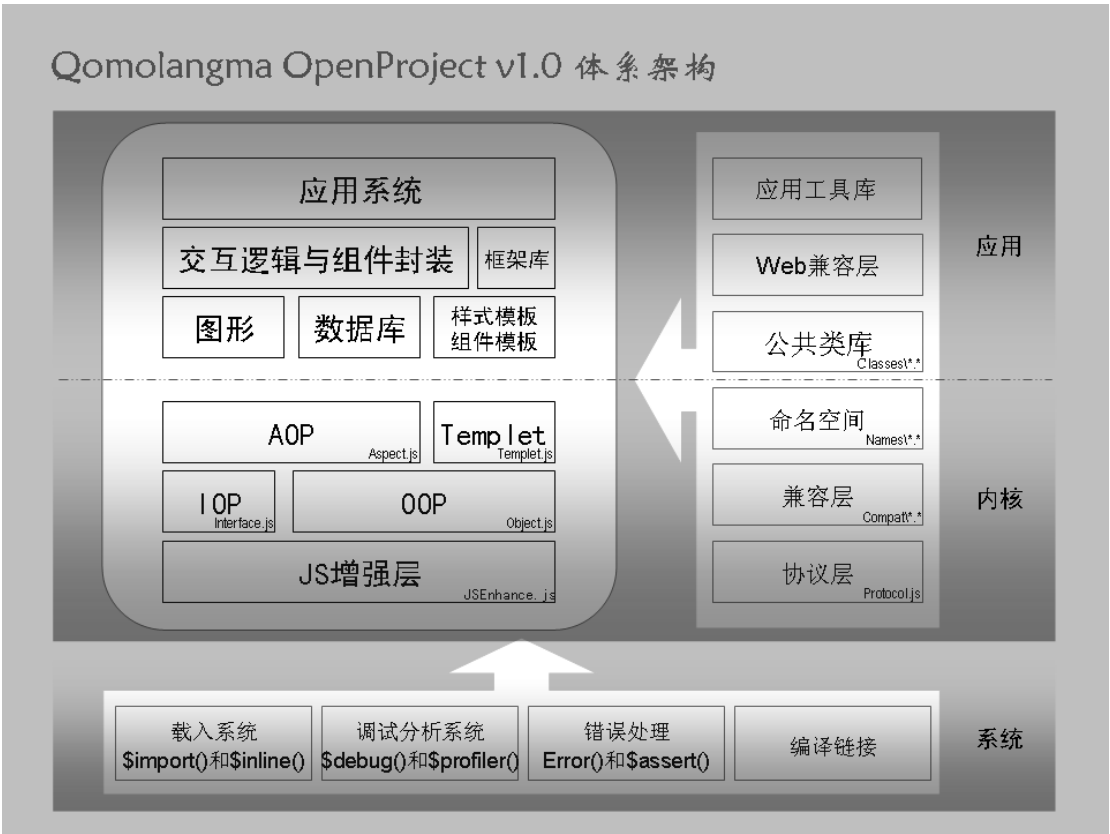
因为有了这两项设定，Qomo 的继承体系变得非常清楚，而且同时兼容了 JavaScript 标准的原型继承模式——不仅仅是能用 `instanceof` 运算那么简单。

最后，Qomo 重新构建了兼容层、命名空间、接口、面向切面等等在 WEUI

不友好或根本不存在的框架设计，Qomo 在一年多之后终于发布了它的第一个正式版本。

1.1.2. Qomo 的体系结构

下图是 Qomo V1.0 正式版本的体系架构，V2.0 版本目前正基于该架构持续发展：



在这个架构中，Qomo V1 主要实现了“内核”层与“系统”层，以及“应用”层的部分工具。上图中每个模块右下标有源代码所在目录或源代码的关键函数^①，没有做这种明确标注的模块一般都是在 Qomo V2 阶段实现或正式发布的。

在本章中，我们主要讲述内核层中的大部分内容，下一章则对应用层和系统（工具）层做一些介绍。在内核层中，主要包括以下特性的实现：

- 👉 多投事件系统
- 👉 自维护的命名空间

^① 随书光盘中带有 Qomo V1 的正式发布版本，以及 Qomo V2 的当前版本。

- 👉 基于类继承的面向对象系统(OOP)
- 👉 接口系统(IOP)
- 👉 切面系统(AOP)
- 👉 基本的模板系统(TOP)

1.1.3. Qomo 框架设计的基本原则

Qomo 的内核层框架在本质上就是一个语言扩展层，Qomo 团队在这个部分实现了包括 OOP、IOP、AOP 等等在内的众多特性。在这些系统 / 模块的设计中，我们始终确保如下原则：

- 👉 允许开发人员尽可能自由地拼装语言特性
- 👉 尽可能不创建（和分析）新的语法，在运行过程中实现系统
- 👉 系统尽可能减少对引擎全局的影响
- 👉 系统尽可能在标准 JavaScript 规范下实现

其中，对于规则一，Qomo 允许用户（开发人员）独立地使用除 AOP 和 TOP 之外的任意部分，而 AOP 与 TOP 是在 OOP 和 IOP 的基础上实现的，因此仅对他们具有依赖性；对于规则二，Qomo 的整个系统都没有要求某种特殊的 JavaScript 语法，也没有对代码做特殊的语法分析（例如 NarrativeJS 项目使用大量的代码为 JavaScript 扩展了 “->” 运算符和相关语法）。

Qomo 能兼容 safari 2.0、mozilla(firefox) 1.8 和 IE5.x 以上的全部版本，为此 Qomo 做出了一些性能和代码可读性方面的牺牲——其中绝大多数牺牲是为 Internet Explorer 5.0 付出的。除此之外，Qomo 对 Opera 也提供了一定的兼容性。为了避免这些牺牲影响到一个“通常的”开发环境^①，Qomo 将所有关于兼容性的代码隔离在系统之外，有且仅有在用户选择集成(Build)一个兼容的代码包时，它才会被加入到系统中。

Qomo 的集成器(Builder)用于将 Qomo 代码树（目录）中的 .js 文件拼成一个单独文件，并做一些简单的混淆。这个集成器也是用 Qomo 开发、在 Qomo 环境中运行的。

^① 这里指纯粹基于 IE5.5 以上，或 Mozilla Firefox 开发的环境。

1.2. 基于类继承的对象系统

在对象系统方面，Qomo 向开发人员提供了四个重要特性：

- 👉 一种基于类类型的继承体系，便于清晰地、层次化地组织大型对象系统；
- 👉 一个通过读写器存取的“特性(Attribute)”系统，用于隐式地每个对象实例存储专有值，以避免大型对象系统中对象成员列表臃肿；
- 👉 一个便捷地调用“父类方法”的工具，用户不必独立维护父类或父类实例（以自行实现类似功能，参见“1.5.2 多态”）；
- 👉 一种支持多投的事件系统，能够将一个事件投送到多个处理例程中，并允许有选择性地中断投送过程。

除此之外，Qomo 的 OOP 扩展对命名空间(Namespace)和接口(Interfaces)提供“智能化的”支持——当你不需要这些特性时，可以通过简单的配置项移除它们而不会造成任何负面的影响。

1.2.1. Qomo 类继承的基本特性

首先，Qomo 支持一种特殊的类型：类类型。它的命名习惯是：

<T>对象构造器名

例如：

```
TObject    // 表明是 Object 构造器的类类型
TMyObject  // 表明是 MyObject 构造器的类类型
```

类类型具有一些特别的属性，（以 TObject 为例，）这些属性包括：

```
TObject.Create()    // 指向它的构造器，即 Object()
TObject.ClassName    // 类名字符串，例如'TObject'
TObject.ClassInfo    // 指向 TObject 类类型自身
TObject.ClassParent  // 指向 TObject 类的父类类型
```

通过 Qomo 的类继承系统创建的对象实例，会具有一些特殊的方法和属性。Qomo 的类继承系统是通过特定的属性来维护继承关系的。仍以上面的 TObject 为例，它的实例总会包括如下成员：

```
obj.ClassInfo    // 指向 TObject 类类型
obj.get()        // 方法，特征(Attribute) 读取器
obj.set()        // 方法，特征(Attribute) 改写器
obj.inherited()  // 方法，用于继承调用父类方法
```

另外，如果用户的类声明中存在实例构造周期，则该对象会有一个 Create()

方法，但它与 `TMyObject.Create` 并不是同一个方法。如下例：

```
// 类声明
function MyObject() {
    // ...

    this.Create = function() {
        // 实例构造周期
        // (这里是真实的 obj.Create 代码)
    }
}

// 类注册
TMyObject = Class(TObject, 'MyObject');
// 创建对象实例
obj = new MyObject();
// 显示值：false，表明对象实例的 Create 方法，与 TMyObject.Create 不同
alert( obj.Create == TMyObject.Create )
// 显示值：true，表明 TMyObject.Create 实际指向构造器
alert( MyObject == TMyObject.Create );
```

如同前面所强调的，在 Qomo 中类继承系统显式地通过：

```
obj.ClassInfo
TObject.ClassInfo
TObject.ClassParent
```

来维护继承关系。用户代码一般不需要关注这个过程，因为类注册函数 `Class()` 将会帮你填好所有的值。

因为 Qomo 一些内部系统依赖于这些属性来实现，因此用户代码不应该修改这些属性和方法。事实上它也是你辨识对象是 Qomo 类、Qomo 对象或普通的 JavaScript 对象实例的方法。例如在 `SysUtils.js` 单元中的公共函数：

```
/**
 * check qomo classes.
 */
function IsClass(cls) {
    return (cls instanceof Function) && cls.Create &&
        cls.ClassParent;
}

function IsObject(obj) {
    return obj.ClassInfo && IsClass(obj.ClassInfo) &&
        (obj instanceof obj.ClassInfo.Create);
}
```

1.2.2. Qomo 类继承的语法

1.2.2.1. 基本的类类型声明与使用

我们要记得：Qomo 所有在语法上的扩展都是基于 JavaScript 自身的语法的。也就是说，任何语法都是可由引擎识别与执行的。从这样的视角来看，“类声明”其实就是一个普通的构造器函数：

```
// 类声明
function MyObject() {
    this.aMethod = function() {
    }
    this.aProperty = 100;
}
```

“类注册”则是随后的一个函数调用。函数名为 `Class()`，返回值则是一个类类型（你可以看成标识符或变量）：

```
// (续上例)
// 类注册
TMyObject = Class(TObject, 'MyObject');
```

这其中，`TObject` 是所有类类型的基类，它对应于 `Object` 构造器，是后者的一个封装。`'MyObject'` 是构造器的函数名，它必须保证与类声明中的构造器名一致。

当一个类被注册后，你就可以创建该类的实例：

```
// (续上例)
// 创建实例
var obj1 = TMyObject.Create();
var obj2 = new MyObject();

// 使用，方法或成员来自类声明
alert( obj1.aProperty );
obj1.aMethod();
```

这两种创建方法没有任何本质的不同。只是前一种在语义上更加符合“类继承对象系统”的定义^①，后一种则沿用了 JavaScript 的标准语法。只有在通过 `Class()` 调用完成类注册后，我们创建的对象实例将才会具有以下方法和属性：

👉 `obj.ClassInfo`：指向类类型的一个引用

^① 一点点的私念在于：这是一个 Delphi 面向对象的语法。

- 👉 `obj.get()`: 取特性值
- 👉 `obj.set()`: 置特性值
- 👉 `obj.inherited()`: 继承 (调用) 父类方法

1.2.2.2. 特性、默认读写器与限制符

请留意上面的过程，除了多了一个“类注册”的动作之外，所有的代码与我们此前所讲的 JavaScript 的对象系统没有任何区别。但是，在类声明中，我们还可以有较为复杂的类定义：

```
// 多投事件类型声明
TOnInit = function(count) {};

// 类声明
function MyObject() {
    // 特性声明
    Attribute(this, 'Name', 'MyName', 'r');
    Attribute(this, 'Index');

    // 私有的类静态成员
    var count = 0;

    // 多投事件声明
    this.OnInit = TOnInit;

    // 实例构造周期
    this.Create = function() {
        this.set('Index', count++);
        this.OnInit(count);
    }
}
```

我想这已经比较复杂了。下面我们创建它的实例并使用（当然还要先注册）：

```
// (续上例)
TMyObject = Class(TObject, 'MyObject');

// 创建第一个实例，显示索引值为 0
obj = new MyObject();
alert(obj.get('Index'));

// 创建第二个实例，显示索引值为 1
obj = new MyObject();
```

```
alert(obj.get('Index'));
```

这个例子示范了如何使用特性 'Index'。不过你会注意到，`MyObject()` 还声明过一个名为 'Name' 的特性，声明时初值为 'MyName'，后面还有一个参数 'r'，表明只读（你还可以用 'w' 来表明只写，'p' 来表明受保护的特性）。这意味着你不能给特性 'Name' 置值，例如下面的代码就会导致异常：

```
// (续上例)
// 如下代码导致异常，显示 'The "Name" attribute cant write for TMyObject.'
obj.set('Name', 'NewName');
```

当然，对上例中的 'Index' 就没有这项限制——你可以写这个特性。

1.2.2.3. 多投事件与继承父类方法

我们还应该留意上面类声明中有一个名为 `OnInit()` 的多投事件句柄——在 Qomo 中，所有以 'On' 开始的方法都被系统理解为事件句柄^①，并初始化为多投事件。这个 `OnInit()` 是在实例构造周期 “`this.Create = ...`” 中被触发的——如同我们前面讲述的过：JavaScript 没有明确的事件系统，事件是一个外部（例如 DOM）加载与触发的方法，只是这里在 “`this.Create`” 中被触发罢了。但如果是在实例构造周期就触发了，那么谁能够向该事件中添加事件响应函数呢？——在这个周期中，`new` 运算还没有向 `obj` 变量返回表达式值呢。

这就与“类类型系统用于构建复杂的大型对象系统”这样的设定有关了。我们上例这样设计 `MyObject` 类，在于为“所有子类”使用同一个 `count` 计数器。下面声明这样的子类（的一个示例）并示范如何使用这个 `OnInit()`：

```
// (续上例)

// 子类声明
function MyObjectEx() {
    // 一个私有的函数，用作事件响应
    var doInit = function(count) {
        alert('Created Instance: ' + count);
    }

    // 子类的实例构造周期
    this.Create = function() {
        // 添加事件响应句柄
        this.OnInit.add(doInit);
    }
}
```

^① 在以浏览器为宿主的环境中，'on' 开始的通常是 DOM 的事件句柄（注意这里的 'on' 是小写字符）。

```

        // 调用父类(同名)方法
        this.inherited();
    }
}

// 注册为 TMyObject 的子类
TMyObjectEx = Class(TMyObject, 'MyObjectEx');

// 测试, 显示 'Created Instance: 3'.
// ( 本例前面部分曾创建了两个父代类的实例 )
obj = new MyObjectEx();

```

在这里，子类的实例构造周期中就有了 `OnInit()` 事件，我们给它添加了一个事件响应函数，然后用 `Inherited()` 调用父类的同名方法，也就是父类中的 “`this.Create`”。当父类中的 `OnInit()` 事件被激活时，子类中的那个 `doInit()` 就被调用了。所以上例会显示测试结果 “Created Instance: 3”。

一般来说，`Inherited()` 在类继承系统中的使用是很广泛的。因为类继承系统对行为（方法）的解释就是：子类与父类行为相似，并继承、细化或修改父类的行为。例如在一个简单描述的动物世界里：

```

function Animal() {
    this.leg = function() {
        alert('跑');
    }

    this.run = function() {
        this.leg();
    }
}

function Cat() {
    this.jump = function() {
        alert('跳');
    }

    this.run = function() {
        this.jump();
        this.jump();
        this.inherited();
    }
}

```



```
TAnimal = Class(TObject, 'Animal');
TCat = Class(TAnimal, 'Cat');

obj = new Cat();
obj.run();
```

我们看到：

- 👉 “动物” 有一种 `run` 的行为，这个行为的内容是(不停地) “跑(leg)”；
- 👉 “猫” 是一种 “动物”，并从 “动物” 那里继承了 `run` 的行为；
- 👉 “猫” 通常是 “跳两下”，然后再继续 `run` 的行为^①。

所以，猫最终的 `run` 的行为描述是：

```
this.run = function() {
    this.jump();
    this.jump();
    this.inherited();
}
```

这种用继承性来构建对象系统的方法，与 JavaScript 中使用原型的方法有明显的思想差异。

1.2.3. Qomo 类继承系统的实现

本小节分析的代码，若未特殊标注，则来自于 Qomo 代码表中的 `Object.js` 单元。

我们需要这样的特性：

- 👉 对象可以从 “类” 中构造实例；
- 👉 所有的对象实例访问该类的同一份类成员；
- 👉 对象继承类全部的成员与方法，但有机会重写它们。

1.2.3.1. 类继承系统的基本思想

在 JavaScript 中，我们可以通过原型继承和类抄写两个手段之间的互补，来构造复杂的类继承体系。这样的代码一般写作如下：

^① 这种行为方式的猫大概只能在卡通片里，或者我们这样用程序来构建的 “原始动物世界” 中能看到。无论如何，请不要认为我脚下这只小猫具有这样的怪异行为。

```
// 构造器声明
function MyObject() {
    this.XXX = ...
    this.YYY = ...
}
// 修改原型
MyObject.prototype.ZZZ = 100;
MyObject.prototype.MMM = 'a test string';
MyObject.prototype.aMethod = function() {...};
```

这种方法中，通过原型修改添加的成员应该是一些“值类型”数据；如果是“引用类型”数据，则在该构造器的所有实例中共享同一个引用。我们留意到“所有实例中共享同一个引用”正是类继承系统中的一个重要特性，也就是说这具有“类公开成员”的基本特性。因此，在 Qomo 中把这些修改原型的代码折叠到了构造器内部，并将 MyObject 变成了独立的“对象构建周期”：

```
1  function Qomo_MyObject() {
2      // 类构造周期，this 引用相当于上例中的 MyObject.prototype
3      this.ZZZ = 100;
4      this.MMM = 'a test string';
5      this.aMethod = ...
6
7      // 实例构造周期，this.Create 相当于上例中的 MyObject 构造器
8      this.Create = function() {
9          this.XXX = ...
10         this.YYY = ...
11     }
12 }
```

所以，事实上 Qomo 的语法不过是 JavaScript 中一种复杂的、基于原型继承的对象系统的变形而已。这种变形在语法上仍然是完全 JavaScript 的，而在语义上则符合“类类型对象系统”的基本设定：

- 👉 类构造周期用于定义类类型系统中在所有子类 / 实例中不发生变化的部分，以及基本不变的逻辑；
- 👉 除非特别说明，类构造对“this”所声明成员是类成员，所有对象实例共享同一引用或值；
- 👉 类构造周期所声明的“类私有成员”是所有对象实例共享的；
- 👉 实例构造周期用于对每一个用 new 运算创建的对象实例进行改写，所有的改写只影响当前实例。

所以在上面的例子中，代码行 3~8 中的 this 引用是类类型的引用，而 9~10

行中的 `this` 引用，仅是某个确定的对象实例的引用。换言之，如果仅是上面的声明代码而没有任何的 `new` 运算^①来创建实例，则 9~10 行代码永远不会被执行到。

`Qomo` 使用这样的类声明与构造的形式，表明 `Qomo` 既承认“类类型声明”的不可变性——即所有子类实例具有完全相同的基类特性，也承认“（原型的）构造器”可以重新改写任何对象实例而不影响它的同类实例。`Qomo` 通过前者来保证今后将讨论到的“接口（Interfaces）”能安全地与类绑定，通过后者来维持 JavaScript 对象系统的灵活性。

不单如此，用户代码还可以改写 `Qomo_MyObject.prototype`，用原型方式来影响所有的实例，这与原先 JavaScript 的定义完全一致。不过，你必须在类注册完成之后，才能修改这个原型。

在 `Class()` 函数内部进行类注册时，`Qomo` 修改了你所见到的这个 `Qomo_MyObject` 函数。也就是说：

```
13 // (续上例)
14
15 // 备份一个引用
16 var foo = Qomo_MyObject;
17 // 类注册
18 TQomo_MyObject = Class(TObject, 'Qomo_MyObject');
19 // 复核 Qomo_MyObject() 函数。显示 false，表明调用 Class() 中修改了该函数
20 alert( foo == Qomo_MyObject );
```

如果用户只是使用 `Qomo` 的类类型机制来构造对象系统，那么不必了解 `Class()` 内部的细节。然而如果用户想要 `Qomo` 如何充分利用函数式、动态语言以及 JavaScript 的其它语言特性来实现它，那么请随我进入下面的章节。

1.2.3.2. 类继承系统的实现框架

在 `Qomo` 的 `Object.js` 文件中，通过实现 `Class()`，封装了“类继承”体系的绝大部分细节。它内部实现的框架代码如下：

```
1 Class = function() {
2     // ...
3     // 基本的类数据、公共变量等
4
5     // 构建类数据块
```

^① 当然还包括注册后的 `TQomo_MyObject.Create()` 方法，再次强调它与 `new` 运算创建实例是完全等义的。

```

6     function ClassDataBlock() {
7         var cls = function(Constructor) {
8             // ...
9         }
10        return cls;
11    }
12
13    // 真正的 Class() 函数
14    function _Class(Parent, Name) {
15        var cls = new ClassDataBlock(Parent, Name);
16        cls.OnClassInitializtion(Constructor);
17
18        // 构建对象(实例)数据块
19        function InstanceDataBlock() {
20            // ..
21        }
22
23        // 真正的对象构造器函数
24        cls.Create = function () {
25            // ...
26            var Data = new InstanceDataBlock();
27            this.get = Data.get;
28            this.set = Data.set;
29            this.inherited = Data.inherited;
30            // ...
31
32            if (this.Create) this.Create.apply(this, arguments);
33            // ...
34        }
35        cls(Constructor);
36        cls.OnClassInitialized(InstanceDataBlock);
37        // ...
38
39        // 替换构造器
40        eval(Name + '= cls.Create');
41        return cls;
42    }
43
44    return _Class;
45 }();

```

Class()函数采用了声明即执行(t001)的技术，从 1 ~ 4 5 行是一个完整的

匿名函数，在声明后立即执行。返回值 `_Class` 引用被赋给全局函数 `Class`。根据闭包的基本规则，行 2 ~ 11 中所有的变量、构造器（`ClassDataBlock`）对于每次 `Class()`调用来说，都将会是公用的。

接下来留意 `_Class()`函数，它返回 `cls`。而 `cls` “看起来”是一个 `ClassDataBlock()`的一个实例，但事实上每次 `new ClassDataBlock()`调用都不是直接返回新构建的对象，而是返回一个函数对象：行 7 ~ 9。这段代码的含义是：类类型其实是一个函数对象。例如：

```
function MyObject() {  
}  
  
TMyObject = Class(TObject, 'MyObject');  
  
// 显示“类类型”的元类型名: function  
alert( typeof TMyObject );
```

代码 16 ~ 36 行是一个完整的类初始化过程。`Class()`中的类初始化过程，就是用于处理用户代码的类声明中的代码。这其中，从 19 ~ 34 行是两个函数声明：`InstanceDataBlock()`构造器与 `cls.Create()`方法。由于这两个声明被包含在一个调用 `Class()`的上下文中（闭包），因此对于注册时的该类(变量 `cls`)、以及该类的所有实例来说，这些数据只有一份——我们用闭包对数据的封装来实现了“类公共数据在所有对象实例中共享”的特性。第 35 行传入 `Constructor` 参数以调用 `cls()`函数，完成类的整个初始化。在这个调用过程中，`Constructor` 中的所有代码被执行一次——这里的 `Constructor` 就是用户代码中的类声明（例如 `MyObject()`构造器）。最后，第 36 行简单的调用一下结束化过程，以清理掉“类类型（函数对象）”上的一些多余的成员。

接下来的第 40 行则通过动态调执行来实现一个对全局变量的重写。这里的 `Name` 变量即是我们传入的构造器名参数，所以，（以构造器 `MyObject()`为例，）该行代码实际执行的是：

```
eval('MyObject = cls.Create');
```

第 41 行则将 `cls` 返回到 `Class()`调用之外，在该调用后，它通常被赋值给一个名为“`T<构造器名>`”的全局变量——例如这里的 `TMyObject`。所以我们在前文中一再强调下面两行代码并没有本质的不同，就是源于上面这一行对全局变量的重写：

```
new MyObject()  
TMyObject.Create()
```

1.2.3.3. 类数据块(CDB)与实例数据块(IDB)

`ClassDataBlock()` 是一个构造器函数，它是 `_Class()` 的一个 `upvalue` 值，因此它只有唯一一个函数实例。但是这个构造器函数的实例却被创建于 `_Class()` 调用过程中，因此对于每一个 `Qomo` 的类类型来说，都拥有一个 `ClassDataBlock()` 的实例，该实例被称为“类数据块(CDB)”。与此类似的，每一个类类型所在的闭包中也有一个 `InstanceDataBlock()` 构造器函数，但它的实例却在 `cls.Create()` 中被创建——亦即是调用 `new` 运算时发生，因此对于所有“类类型的实例”来说，都拥有一个 `InstanceDataBlock()` 的实例，被称为“实例数据块(IDB)”。上一小节的实现框架清晰地展现了 `Qomo` 通过闭包来分离构造器与实例的方法，其中 CDB 创建于第 15 行，而 IDB 创建于第 26 行。

但 CDB 并不象上一节的实现框架所展示的那样简单。CDB 也通过闭包的形式封装了类创建过程所需的全部信息，其基本框架如下：

```
1 // 构建类数据块
2 function ClassDataBlock() {
3     var Attr = function() {}; // 特性的构造器和默认值，基于原型继承的
4     var _events = []; // 事件列表，用于多投事件系统
5     var _maps = {}; // "父类同名方法链表"的缓存，用于 inherited
6
7     var cls = function(Constructor) {
8         // ...
9     }
10
11     cls.OnClassInitializtion = ...
12     cls.OnClassInitialized = ...
13
14     // TypeInfo of the class. Don't change anything!!!
15     cls.ClassName = 'T' + Name;
16     cls.ClassInfo = cls;
17     cls.ClassParent = Parent;
18
19     return cls;
20 }
```

可以看到，第 15 ~ 17 行代码反应了“类类型的基本属性”（参见“1.2.1 `Qomo` 类继承的基本特性”），而第 3 ~ 5 行代码，则体现了 `Qomo` 实现类类型特性扩展时的基本技术。更多的、更复杂的秘密则暗藏在 `cls()` 这个函数中——也就是我们的类类型“`T<构造器名>`”。这个函数内部是这样的：

```

1  var cls = function (Constructor) {
2      if (cls.ClassParent) {
3          var clsInfo = getClassTypeInfo(cls.ClassParent);
4          Attr.prototype = clsInfo.$Attr_;
5      }
6      Attr = new Attr();
7
8      var base = new Constructor();
9      for (var i in base) {
10         if (base[i] instanceof Function) {
11             if (_r_event.test(i)) _events.push(i);
12             if (_r_attribute.exec(i)) {
13                 Attr[i] = base[i];
14                 delete base[i];
15
16                 if (!(RegExp.$2 in Attr)) Attr[RegExp.$2] = _undefined_;
17             }
18         }
19     }
20
21     setClassTypeInfo(cls, base, Attr);
22 }

```

代码分为两部分。第一部分是行 2 ~ 6，这段代码通过 `getClassTypeInfo()` 在一个内部的表中查找当前类的父类的类信息（你可以理解为 RTTI, RunTime Type Infomaction），根据类型信息我们可以查找到父类的特性表（的一个实例），然后在第 4 行中我们将父类特性表用作了子类特性表的原型。第 6 行使用了一个简单的技巧来创建一个单例的对象 `Attr(t002)`，它被用来存放当前类的所有特性，由于它使用原型继承，所以它的开销很小，而且也满足我们对特性的一些性质设定（例如特性是可以从父类继承的）。

代码的第二部分是第 8 ~ 19 行。其中第 8 行反应了类构造周期的一个事实：类构造器函数将被执行一次，并产生一个实例。以“1.2.3.1 类继承系统的基本思想”的代码为例，`Qomo_MyObject()`总是要作为构造器函数执行一次的，而且会产生一个实例：`base`，至少具有如下属性：

```

base.ZZZ
base.MMM
base.aMethod
base.Create

```

这里说“至少具有”是因为该构造器使用之前，在 `cls.OnClassInitializtion()`

中会有一行代码设置该构造器的原型：

```
if (Parent) Constructor.prototype = getPrototype(Parent);
```

由于 **Constructor** 以父类的 **base** 为原型，所以上面产生的实例会带有从父类（通过原型继承）得来的属性——也就是说，在任意多层次的继承之后，**base** 总是当前子类的一个实例，具有该实例的全部属性。

接下来，从第 9 行开始我们列举全部这些属性，并逐一考察。这里的考察也很简单：

- 👉 如果属性名以 'On' 为前缀，则识别为多投事件，将该事件名放到 **_events** 列表中；
- 👉 如果属性名以 'get' 或 'set' 为前缀，则识别为特性读写器，将该读写器函数置为 **Attr** 对象的成员、尝试在 **Attr** 中创建一个特性名，以及从 **base** 原型中移除该读写器方法。

应当留意我们将特性名、初值和读写器都写在了 **Attr** 这个对象实例上。也就是说，如果类声明一个名为 'Name' 的特性，则 **Attr** 上可能会有下列属性：

```
Attr.Name = <初始值>或<_undefined_>
Attr.getName = 特性读取器函数
Attr.setName = 特性置值器函数
```

其中只有 **Attr.Name** 是必然会存在的，而后面两个是否存在，就取决于用户是否声明过读写器函数，以及是否在 **Attribute()** 函数中声明过存取限制。

关于特性系统如何利用这里的 **Attr** 对象，我们放在后文中再讲。这里只强调 **_undefined_** 这个值，它是一个对象引用，因为新创建的任意两个对象都不可能重复，因此它被作为一个独立的值来指示系统中的 **undefined**——我们需要指示该值的原因是：显然用户可能声明某个特性，而该特性初始值为 **undefined**。如果具有这种特性，那么在较低版本的 **JScript** 中无法检测^①，这里使用 **_undefined_** 就是为了兼容这种情况。

这些就是 **CDB** 的主要作用：创建特性表、事件表，以及基于原型继承来维护一套类类型系统所需要的成员列表。相比之下，**IDB** 则相对要简单清晰得多。它的基本框架是：

```
1 function InstanceDataBlock() {
2     var data_ = this;
3     var cache = []; // cached call map.
4
5     this.get = function (n) { ... }
```

^① 低版本 **JScript** 不支持 **in** 运算，因此不能有效检测成员是否具有某个成员：对于值为 **undefined** 的成员会存在误判。


```

6     this.set = function (n, v) { ... }
7     this.inherited = function(method) { ... }
8 }

```

在 IDB 中为每个对象初始化了两个结构，一个是 “data_”，用来存储当前实例重写过的特性值；另一个是 cache，用来存储当前实例在调用父类方法时所使用的 “父类同名方法链表”——后文中通常称为 “方法的继承链” 或 “继承链”。这两个结构初始时都是空的——对于 data_ 来说是 “空白对象”，对 cache 来说是 “空数组”。当用户代码执行：

```
obj = new MyObject()
```

时，实际将会调用到 cls.Create，从而进入实例构造周期位于 _Class() 中的代码（因此比真正的实例构造周期更加靠前）：

```

cls.Create = function () {
    // ...
    // 检测是 new XXXX()
    if (this && this.constructor===cls.Create) {
        // 1. 构建 IDB
        var Data = new InstanceDataBlock();
        // 2. 读写器
        this.get = Data.get;
        this.set = Data.set;
        // 3. 继承方法调用
        this.inherited = Data.inherited;

        // 4. 抄写和初始化多投事件
        var all = $all('event');
        for (var i=0, imax=all.length; i<imax; i++) this[all[i]] = new MuEvent();

        // 5. 真正地进入实例构造周期
        if (this.Create) this.Create.apply(this, arguments);
    }
}

```

步骤 1 ~ 3 用于构建 IDB 并套取 IDB 中的 get、set、inherited 三个方法，它们是为每个对象初始化的。之所以这三个方法都包含在 IDB 中（即是说隶属于具体的对象实例），是因为 get、set 所操作的 “特性(Attribute)” 是用于为每个实例保存私有数据的，而 inherited 的 “父类同名方法链表” 中的，除了包括 “父类方法”，也可能会包括当前实例 “在实例构造周期重写的同名方法”。

下一小节我们将讲述步骤 2 中的读写器（Data.set 和 Data.get）如何实现。

至于步骤 3 ~ 5：

👉 步骤 3，由“1.2.3.5 细节 2：调用父类方法”讲述；

👉 步骤 4，由“1.2.3.6 细节 3：对多投事件系统的支持”讲述；

步骤 5，由“0 注意到这里多声明了一个 \$all，它是为了存取更多的 CDB 数据而预留的——不过 Qomo 目前只是在实例构造周期（之前）用它来获得 _events[] 数组：

```
cls.Create = function () {  
    // ...  
  
    // 4. 抄写和初始化多投事件  
    var all = $all('event');  
    for (var i=0, imax=all.length; i<imax; i++) this[all[i]] = new MuEvent();  
  
    // ...  
}
```

这个 _events[] 数组只保存了事件名，而没有任何的事件响应例程。所有的例程都必须通过 obj.aEvent.add(...) 这样的形式添加到事件投送列表中。但一个多投事件对象是如何实现的并不是本小节的话题，这些内容留在“1.3 多投事件系统”中讲述。

由于 _events[] 数组不能保存事件响应例程，因此如果想要在类构造周期中“为所有的实例声明一个共用的事件响应”，下面的代码是错误的方法：

```
function MyObject() {  
    this.OnNotify = function() {  
        ...  
    }  
}
```

这是因为 Qomo 类构造周期中所有的“this.OnXXXX”都只有名称上的含义。所以你必须实例构造周期来做这件事，并且如果子类覆盖了 this.Create()，那么一定要调用 inherited() 来保证该事件响应被添加到事件对象中。例如：

```
function MyObject() {  
    var doNotify = function() {  
        ...  
    }  
    this.OnNotify = NullFunction;  
  
    this.Create = function() {  
        this.OnNotify.add(doNotify);  
    }  
}
```

```

    }
}

function MyObjectEx() {
    // ...

    // 若子类覆盖了 this.Create, 则应调用 inherited() 来添加 doNotify
    this.Create = function() {
        // ...
        this.inherited();
    }
}

TMyObject = Class(TObject, 'MyObject');
TMyObjectEx = Class(TMyObject, 'MyObjectEx');

```

👉 细节 4：“实例构造周期的特殊性”讲述。

1.2.3.4. 细节 1：特性、读写器和存取限制符

这里首先给读者一个有趣的提示：Qomo 对特性表（前面的 Attr 变量）的实现，与 JavaScript 实现属性表（以及其原型继承）的方法是完全一致的。不单如此，事实上我们在实现特性表的读写时，也采用了与属性表类似的写复制的机制；而在实现读写器及其存取限制时，所采用的方法也与 SpiderMonkey JavaScript 为属性实现读写器^①时的方法基本一致。

我们接下来讲这些性质的具体实现。

特性是在类构造周期声明的，但只对对象实例有意义。因此 Class() 函数在 CDB 中构建它的信息以及赋初值，而在 IDB 中才实现它的读写方法。下面的代码为例：

```

1  function MyObject() {
2      Attribute(this, 'Name', 'Normal');
3
4      this.setValue = function(v) {
5          v = v || 'default value';
6          this.set(v);
7      }
8

```

^① SpiderMonkey JavaScript 支持一种特殊的语法扩展，使开发人员可以声明指定属性（properties）的读写器函数、可见性和存取权限等。

```

9      this.getName = function() {
10         return 'My name is ' + this.get();
11     }
12
13     this.Create = function(name) {
14         if (name) this.set('Name', name);
15     }
16 }
17 TMyObject = Class(TObject, 'MyObject');
18 obj = new MyObject();

```

在这段示例中，行 2 ~ 11 用于声明 'Name' 与 'Value' 这两个特性以及它们的读写器。行 2 显式地声明了 'Name' 特性，由于没有指定存取限制，表明使用缺省的存取器函数；行 9 ~ 11 则声明了一个 `getName()` 方法来覆盖了 'Name' 的读取器。因此 'Name' 事实上是由一个缺省置值器和一个自定义的读取器构成。在第 4 ~ 7 行中，声明了一个 `getValue()` 方法，由于没有显式地用 `Attribute()` 声明过特性，因此这个 `getValue()` 就隐式地声明一个 'Value' 特性——这与 JavaScript 的变量声明规则一致：如果访问变量不存在，则隐式声明该变量。

从第 4 ~ 7 行代码可知：如果一个 `getter/setter` 所指示的特性名不存在，那么 Qomo 系统将内部创建该特性。由此可知，事实上上例的第 2 行代码是多余的，因为行 9 ~ 11 也会导致一个名为 'Name' 的特性被创建。不过，即使开发人员多写这行声明，也并没有任何负面的影响。而且，我们在“0 由于对于用户代码来说，`this.Create` 相当于一个实际上是一个标准的构造器函数（传入的 `this` 引用为 `new` 创建的对象实例），所以用户可以按自己的习惯在这里书写任意的 JavaScript 代码，但是与“类声明相关的”一些语法不能在这里使用：

- 👉 `Attribute()`, `_set()`, `_get()`, `_cls()` 不能使用；
- 👉 `this.getValue()` 等不能声明成特性。

不过 `this.Create` 与类声明在同一个闭包中，因此可以访问类的私有成员和方法。

除了给实例添加新的成员之外，用户代码也可以重写任何从类类型继承的方法，以及向多投事件添加事件响应句柄——大多数情况下你不必重写多投事件对象。由于在实例构造周期已经可以使用 `this.inherited()`，所以当用户代码重写了继承自类类型的方法后，用户代码可以用它来调用父类同名方法，而无需使用特殊的 `hook` 过程来做这件事。例如：

```

function MyObject() {
    this.aMethod = function() { ... }
}

```

```

this.Create = function(x, y) {
    // 重写从类类型继承的方法 aMethod
    this.aMethod = function() {
        // 用 inherited() 调用父类的 aMethod 方法
        return this.inherited();
    }
}
}

```

由于这里是实例构造周期，所以对于实例引用 `this` 来说它的父类 `aMethod`，其就是在类构造周期中声明的那个方法。用户不必象下面这样写代码：

```

// 用 hook 的方法来实现上例的功能
this.Create = function(x, y) {
    var func = this.aMethod;
    this.aMethod = function() {
        return func.apply(this, arguments);
    }
}
}

```

不过应该留意的是，`this.get`、`this.set` 和 `this.inherited` 与前面所提到的 `this.value` 一样是在实例方法，而不是来自于类类型的方法。只不过这些实例方法是在 `this.Create` 调用之前，由 Qomo 的内核在 `this` 引用上添加的罢了。正是由于它们并没有在类类型中声明过，因此也不可能在 IDB 中的这三个方法内部再调用 `this.inherited()`。更进一步地说，如果我们要重写三个方法，我们其实只能使用上面的 `hook` 方法（参见“1.2.4.2 实例构造周期重写读写器 (getter/setter)”）。

同样的原因，`this.OnXXXX` 这些事件名对于类类型来说是没有意义的，准确地说，它们已经从类类型的原型中被清除了。因此 `this.OnXXXX` 在继承链中并没有同名方法。从语义上来讲，调用一个事件亦即意味着触发它，而在事件响应例程中再次调用 `this.inherited()` 就会导致事件执行中再次触发自身，这显然是不合理的。综上所述，用户代码不可能在一个事件中使用 `this.inherited()`。

Qomo 类继承系统的高级话题”中的高级话题中还会讲到 `Attribute()` 的更多应用。

在类构造周期，行 2、4、9 是有意义的。行 2 会调用一个公共函数来操作 CDB，并直接向前面提到过的“类特性列表”Attr 置值。相关代码如下：

```

1  Attribute = function() {
2      // ...
3      return function(base, name, value, tag) {

```

```

4      var i, argn = arguments.length;
5      var Constructor = Attribute.caller;
6      var cls = Constructor.caller;
7
8      // ...
9
10     return (cls && cls.set && cls.set(name, value));
11 }
12 }();

```

回顾上一小节所说的，我们是通过如下的代码来启动类构造周期的：

```

var cls = function (Constructor) { // <-- cls 其实会是将来的 TMyObject
    // ...

    Attr = new Attr();
    var base = new Constructor(); // <-- 这里启动类构造周期
    // 接下来，从base向Attr抄写特性名和读写器
    // ...
}

```

我们留意这个调用过程，会发现行 5 ~ 6 就是通过函数调用栈来查找 Constructor 和 cls（参见“1.5.5 可遍历的调用栈”），亦即是 MyObject 和 TMyObject。第 10 行代码是一行关键代码，代码中的 cls.set 并不是我们今后会用到的 obj.set，而是来自类类型初始化中的代码：

```

// getAttribute 与 setAttribute 用于存取 Attr
function getAttribute(n) {
    return Attr[n] != _undefined_ ? Attr[n] : undefined
}
function setAttribute(n, v) {
    Attr[n] = v!=undefined ? v : _undefined_
}

// 类类型初始化时为类实例(cls)添加的 get 与 set 方法
cls.OnClassInitializtion = function(Constructor) {
    // ...
    this.get = getAttribute;
    this.set = setAttribute;
}

```

因此第 10 行代码：

```

10     return (cls && cls.set && cls.set(name, value));

```

中的 `cls.set(...)` 其实就是直接向 `Attr` 中置初值。所以本小节开始的例子中的：

```
Attribute(this, 'Name', 'MyName');
```

其实的真正效果就是：

```
Attr['Name'] = 'MyName';
```

`Attribute()` 函数的最后一个形式参数是 `tag`，亦即是“读取限制符”。它是一个由单个字符组成的字符串。这些单个字符包括：`r`：可读；`w`：可写；`p`：保护。这些字符的可能组合还包括：

- 👉 `'rw'`：可读写
- 👉 `'pr'`：可读，但写时保护
- 👉 `'pw'`：可写，但读时保护
- 👉 `'[xx]p'`：当字符串以 `'p'` 作为最后一个字符，无论前面的字符是什么（或没有）都将表示保护读写。

这里所谓保护（`'p'` 限制符），是指只能在实例的方法中调用 `setter` 或 `getter`，而不能在外部（例如全局或其它对象的方法中）调用该特性的读写器。一般来说，“保护”限制是很少见的，而且它的实现效率也非常低，因此并不建议使用。如果在代码中提供这种特性，那么它通常是限制开发人员使用 `setter/getter` 的环境，而并非试图对最终用户环境的使用造成影响。所以在面向最终用户的发布包中，有关 `'p'` 特性的相关限制代码是可以清除掉的。

除了这种差异之外，`'r'`、`'w'`、`'p'` 之间的实现方案并没有本质的不同：实现专有的读写器。但这样的说法可能稍有些不妥，本质上来说它们是实现各自的“限制器”。这种实现方案所基于的前提是：如果用户没有声明读写器，则当“`obj.get`”或“`obj.set`”调用时将直接访问 `Attr` 中的值。换言之，如果一个特性是只读的，事实上只需要实现“一个限制写的 `setter`，而 `getter` 使用缺省值”（只写或保护标志类似）。这在 `Attribute()` 中的代码是这样：

```
// 1. 先置读写器为各自的限制器 cantRead() 和 cantWrite()
base['get'+name] = cantRead;
base['set'+name] = cantWrite;
...

// 2. 如果标志中存在'r'则清除掉 cantRead() 以使用缺省值，'w'标志类此
for (var i=0; i<tag.length; i++) {
    switch (tag.charAt(i)) {
        case 'r': delete base['get'+name]; break;
```

```

    case 'w': delete base['set'+name]; break;
  }
}

```

这样来看，如果以 'rw' 为标志，则 `cantRead()` 与 `cantWrite()` 都被清除，这与不设标志位的效果是一致的。也就是说，下面两种声明完全等效：

```

Attribute(this, 'Name', 'MyName', 'rw');
Attribute(this, 'Name', 'MyName');

```

这些是工具函数 `Attribute()` 所做的全部工作，那么本小节示例中的 `this.setValue` 和 `this.getName` 这些用户自定义的读写器方法又是怎样处理的呢？这些代码其实隐藏在上一小节所讲述的 CDB 中。更确切地说，是在：

```

var base = new Constructor(); // <-- 这里启动类构造周期
// 接下来，从 base 向 Attr 抄写特性名和读写器

```

这行代码之后。由于这里的 `base` 就是类构造周期中的 `this`。所以本小节开始的示例中，当类构造周期结束，`base` 对象包括的方法有：

```

base.getName
base.setValue
base.Create

```

由于有一行 `Attribute()` 声明，因此 `Attr` 对象被置了一个值：

```

Attr.Name = 'Normal';

```

接下来把 `base` 中的读写器移到 `Attr`，因此 `base` 中只存在了 “`base.Create`”，而 `Attr` 则有了：

```

Attr.Name = 'Normal';
Attr.getName = <自定义的读取函数>
Attr.setValue = <自定义的置值函数>

```

但是，大家应该记住这是一个位于 CDB 中的类属性——现在是在类构造周期。我们最终是希望对一个实例调用 `obj.set()` 和 `obj.get()`，它们在 IDB 中实现。

在讲述实例构造周期中如何使用 IDB 来访问 `Attr` 之前，我们先留意一下示例代码中 `getName()` 和 `setValue()` 的写法：

```

4   this.setValue = function(v) {
5     v = v || 'default value';
6     this.set(v);
7   }
8
9   this.getName = function() {
10    return 'My name is ' + this.get();
11  }

```


其中的 `this.set(v)` 与 `this.get()` 是 “<对象>.set” 和 “<对象>.get” 的特殊用法，表明调用缺省的读写器，访问 “当前” 特性。更细节地来说，对于代码：

```
obj.get('Name');
```

将调用这里 9 ~ 11 行代码中的 `this.getName()` 方法；而当 `this.getName()` 中出现了这行代码：

```
obj.get();
```

时，就将调用系统缺省的读写器方法。所谓的 “当前” 特性，就是指这里的 “Name” 特性——既然位于 `getName()` 方法中，当前的特性名当然是 “Name”。与此类似的，第 6 行代码也使用了 “<对象>.set” 的特殊用法，被省略掉的特性名则是 “Value”。

所以，IDB 中的 `getter` 与 `setter` 的实现代码如下：

```
1  this.get = function (n) {
2    if (arguments.length==0) {
3      // call from custom-built getter/setter
4      // custom-built func call from real(and for outside) this.get/set()
      only!
5      var args = this.get.caller.arguments;
6      n = args.length == 1 ? args[0] : args[1];
7      if (this.get.caller!=$attr((args.length==1 ? 'get':'set') + n))
      return;
8    }
9    else {
10     // get custom-built getter from cls's $Attr.getXXXXXX
11     // in ClassDataBlock, the ref. equ data_['get'+n]
12     var f = $attr('get'+n);
13     if (f) return f.call(this, n);
14   }
15
16   if (data_[n] === _undefined_) return undefined;
17   return data_[n]; // a value
18 }
19
20 // 与 this.get 类似，略.
21 this.set = function (n, v) {
22   // ...
23 }
```

其中的 `$attr()` 是一个从 CDB 中取 `Attr` 成员的函数，它直接指向我们前面讲过的 `getAttribute()`。当用户使用 `obj.get('Value')` 这样的调用时，直接进入第 9 行，并

通过 `$attr('get+n')` 尝试取 `'Attr.getValue'` 这个自定义的读写器。如前所述, `Attr` 没有 `getValue` 这个成员, 因此进入第 16 ~ 17 行代码, 返回 `undefined` 或 `data_[n]` 中的值。

而对于 `obj.get('Name')`, 由于存在用户定制的读取器, 因此在第 13 行代码开始调用 `"Attr.getName()"`, 而在 `"getName()"` 方法的内部, 则会再通过省略特性名地调用 `"this.get()"` 的形式, 再次进入上述方法内。(如果在用户定制的 `getName` 中显式调用 `this.get()`, 则) 第二次进入该方法时, 由于传入参数个数为 0, 因此进入第 3 ~ 7 行代码。这些代码将从调用栈中找出特性名 `n`, 然后进入第 16 ~ 17 行代码取值并返回。

应该注意到我们没有为 `data_` 这个用来为每个实例缓存的 `Attr` 值的对象做任何初始化。那么当第一个 `obj.get('Value')` 调用时, `data_` 中该有什么值呢? 在这里 `Qomo` 又一次充分的利用了原型继承的特性。在类构造周期结束时, `Class()` 中调用了下面的代码传入 `IDB`:

```
cls.OnClassInitialized(InstanceDataBlock);
```

然后在 `OnClassInitialized()` 中将 `IDB` 的原型设为 `Attr` 对象:

```
if (Parent) IDB.prototype = getAttrPrototype(cls);
```

因此当我们为每个实例创建 `InstanceDataBlock()` 的实例时, 该实例将有且仅有一个最小化的属性表, 而所有的属性都继承自原型 `Attr`。该实例就被赋给 `data_` 变量, 用来访问特性值了。根据原型继承的特点, 它初始的值总是来自原型 `Attr`, 而置值时就会重写在自己的属性表中而不会影响到原型 `Attr`。而这, 正是我们对 `Qomo` 对象的“特性(Attribute)”的存取设定。

1.2.3.5. 细节 2：调用父类方法

有很多种情况需要调用父类方法, 其中既包括同名的, 又包括不同名的。但总的来说, 都是会在一个对象方法 (包括普通方法和事件) 内来调用。它的调用格式为:

```
this.inherited([aMethodName [, args]]);
```

- `aMethodName`: 指定调用父类的方法名 (也可以直接指定方法)
- `args`: 像函数调用一样传入一组参数, 或传入 `CustomArguments()` 构造器的一个实例

当 `aMethodName` 省略时, `inherited()` 方法会自动查找当前所在方法名; 当 `aMethodName` 指定时, 用户代码可以指定或不指定参数列表 `args`; 当不指定该列表时, 则当前函数调用时的 `arguments` 被传给父类方法调用。但是当用户代

码既不愿传当前函数的 **arguments**，又没有任何参数要传给父类方法时：

```
// 下面的代码并不表明没有参数传给父类方法 aName，而是表明传给当前调用中的 arguments
// (这看起来是这个调用存在了歧义)
this.inherited('aName');
```

就似乎有了歧义。那么用户可以代码可以用下面的代码来实现：

```
// 创建 CustomArguments 实例时参数表为空，这样父类的 aName 方法可以得到这个空参数表
this.inherited('aName', new CustomArguments());
```

包括这一强调的特例在内，以及下面的调用示例都是合法的：

```
function MyObjectEx() {
    this.aMethod = function() {
        this.inherited(); // <-- 调用父类 TMyObject 的同名方法，并传入当前参数
    }

    this.OnNotify = function() {
        this.inherited(); // <-- 激活父类同名的事件并传入当前参数
    }

    // 注意本例不会调用到当前类所声明的 aMethod
    // (调用当前类所声明的 aMethod，应直接使用 this.aMethod() )
    this.aMethod2 = function() {
        this.inherited('aMethod'); // <-- 调用父类的 aMethod() 方法，并传入当前参数
    }

    // 注意本例可以传入任何定制参数
    this.aMethod3 = function() {
        this.inherited('aMethod', new CustomArguments(1, 'test'));
    }

    // 注意本例可以直接传入一个参数表
    this.aMethod4 = function(x,y,z) {
        this.inherited('aMethod', z, y, 100); // <-- 参数[z, y, 100]被传给父类方法
    }

    // 注意本例直接使用对象方法来指示需要调用的父类方法名，其效果与 aMethod2() 没有任何不同
    this.aMethod5 = function(x,y,z) {
        this.inherited(this.aMethod);
    }
}

// 注册为 TMyObject 的子类，TMyObject 的声明略
TMyObjectEx = Class(TMyObject, 'MyObjectEx')
```

由于调用父类同名方法的情况非常复杂，因此 Qomo 在这项功能上花费了非常多的代码：一方面，我们要考虑父类同名方法被重写的情况，另一方面还要考虑动态查找父类同名方法时的效率问题。这在 Qomo 中用两套方案来处理：为前一种情况动态地构建了一个继承链，为后者则构造了一个面向当前对象的缓存。

先说明一下“面向当前对象的缓存”。因为在类继承体系中，所有的实例的方法列表应当都是一样的，父类的、祖先类的也一样。因此只需要构造一个面向父类的缓存就可以了。但是在 Qomo 中，除了类继承之外，还会有原型继承和实例构造阶段的改写。例如下面的代码：

```
function MyObject() {
    this.aMethod = ...

    var count = 0;
    this.Create = function() {
        if (count++ < 5) {
            this.aMethod = function() {
                // do something..

                // 注意这里会调用到实例的父类同名方法，也就是上面类构造周期声明的 aMethod
                this.inherited();
            }
        }
    }
}

TMyObject = Class(TObject, 'MyObject');
```

在这个例子中 MyObject 类创建的前 5 个实例都会重写 aMethod 方法，而其后创建的实例则不会重写。因此 this.inherited()调用时，这两种实例所使用的继承链就不一样。这是 Qomo 中必须使用“面向当前对象的缓存”的原因。

我们接下来详细讲述继承链的创建过程。其实上继承链是每个类所独有的，对于每个类的每一个方法都有可能创建这个链表。所谓“有可能”是因为这个创建过程是完全动态的：如果在子类实例中的某个方法中，从来没有调用过 inherited()方法，则与该方法相关的链表将不会被创建。

创建这个链表的函数名为 \$map，它实际上来自于 cls.map，该方法在 cls.OnClassInitializion 阶段被指向 getInheritedMap()函数。getInheritedMap()函

数的声明位于 CDB 块中，也就是说每一个类拥有一个 `getInheritedMap()` 用于构建自己的继承链——这里被称为 `InheritedMap` 的原因是每一个方法都（可能）有一个自己继承链，因而对于类类型来说就形成了一个 `Map`。`getInheritedMap()` 函数的基本代码如下（注意该函数调用时将传入当前对象实例作为 `this` 引用）：

```
1 function getInheritedMap(method) {
2   if (!(method instanceof Function)) return [method, $inherited_invalid];
3
4   // 从 _maps[] 中搜索指定方法 method，如果找到则返回继承链
5   for (var i=0, len=_maps.length; i<len; i++)
6     if (_maps[i][0] === method) return _maps[i];
7
8   // 如果在 obj.get 和 obj.set 中调用 inherited，则通过 Attr 查找继承链并返回到变量 a
9   var a = inheritedAttribute(method);
10  if (!a) {
11    var p, n, a=[method], $cls=cls;
12    // ...
13    // create inherited stack
14    while ($cls) {
15      p = getPrototype($cls);
16      if (p.hasOwnProperty(n)) a.push(p[n]);
17      $cls = $cls.ClassParent;
18    }
19  }
20  }
21  a.push($inherited_invalid);
22  return (_maps[len] = a);
23 }
```

注意实现代码中的第 11 与第 21 行。这两行表明，对于一个方法来说，它的继承链格式是：

```
map = [method, ..., $inherited_invalid];
```

这个 `map` 是一个数组，`map[0]` 是方法名以用于 4 ~ 6 行的检索，这样以后就不必重复创建过程了——对于一个类来说，它的每一个方法的继承链总是确定的，所以可以放在 `_maps` 中缓存。数组最后的一个元素则总是 `$inherited_invalid()` 函数，该函数简单地抛出异常，它的作用我们稍后会讲到。

现在 `getInheritedMap()` 向 “<对象>.inherited()” 返回了一个数组，该数组立即被缓存到当前实例的 IDB 中的 `cache[]` 数组。缓存及其使用过程代码如下：

```
this.inherited = function(method) {
  // ...
```

```

var p = cache[cache.length] = $map.call(this, f).slice(1);
// begin call inherited. at after, we will delete it from cache.
try {
    var v = p[0].apply(this, args);
}
finally {
    cache.remove(p);
}
return v;
}

```

由于在加入缓存时我们用 `slice(1)` 删除了 `map` 头部的 `method`，所以 `p[0]` 就是当前方法在父类中的同名方法了，而接下来 `p[0].apply()` 调用就完成了“调用父类同名方法”的功能。

然而接下来的问题是：如果在 `p[0].apply()` 的调用中，还有 `inherited()` 的话又该怎么办呢？例如下面的代码的效果：

```

function MyObject1() {
    this.show = function() {
        alert('hi');
    }
}

function MyObject2() {
    this.show = function() { this.inherited(); }
}

function MyObject3() {
    this.show = function() { this.inherited(); }
}

TMyObject1 = Class(TObject, 'MyObject1');
TMyObject2 = Class(TMyObject1, 'MyObject2');
TMyObject3 = Class(TMyObject2, 'MyObject3');

var obj = new MyObject3();
obj.show();

```

当我们执行 `obj.show()` 时，`MyObject3.show()` 中的代码的确会通过 `inherited()` 找到上面的 `p[0]`，并成功调用到 `MyObject2.show()`。然而我们发现 `MyObject2.show()` 中又是一个 `inherited()`，这又怎么处理呢？

记得现在即使调用到 `MyObject2.show()`，当前实例 `this` 仍然是最外层的 `obj`，所以当再次调用 `this.inherited()` 的时候，仍然会进入到此前的那个 `IDB` 中。也就是说，`cache[]` 仍然是刚才的建立的那个缓存。现在调用者函数是 `p[0]`，它总是位于 `cache[]` 中的某个元素——例如 `cache[x]`。因此只要查找到 `cache[x]`，那么我们就可以找到上次创建的那个继承链。这段代码很简单，因为此时的调用者函数正是 `p[0]`，也就是 `cache[x][0]`，我们只需要在数组中检测一下就行了：

```
this.inherited = function(method) {
    var f = this.inherited.caller, args = f.arguments;
    // ...

    // find f() in cache, and get inherited method p()
    for (var p, i=0; i<cache.length; i++) {
        if (f === cache[i][0]) {
            p = cache[i];
            p.shift(); // <-- 弹出队列的第一个元素，亦即是当前方法 f()
            return p[0].apply(this, args);
        }
    }

    // ...
}
```

所以这段代码就是为了应付“在 `inherited()` 中的 `inherited()` 调用”的，通过 `cache[]` 数组，我们不必总在 `CDB` 中去查找继承链或创建它。

最后一步工作是为了处理用户代码意外地调用 `inherited()` 的情况。这是一个有趣的问题，因为用户代码随时都可能调用 `this.inherited()`——可能在对象方法之外，也或者用户忘了父类有没有同名方法而错误地调用了它。那么，这怎么办呢？

记得我们前面创建继承链时添加的最后一个元素 `$inherited_invalid` 吗？它总是位于继承链顶端。所以，即使是对于一个根本没有父类同名方法的方法，那么也会具有如下的继承链：

```
map = [method, $inherited_invalid];
```

因此当我们去掉 `map[0]` 或调用 `p.shift()` 之后，如果链表中没有父类方法，那么就会得到 `$inherited_invalid` 函数并立即被 `apply()` 执行——结果是：中断继承调用过程，提示用户代码出错。一般来讲，这是在开发周期中就会出现的情况了。

上面的讨论中我们没有讲述关于“`this.inherited(aMethodName)`”这种情况。

对于指定方法名，在 `this.inherited()` 中只是简单地分析了参数，并用新的方法来替换当前方法名。除此之外，其它的执行过程没有任何差异。

1.2.3.6. 细节 3：对多投事件系统的支持

一个对象 / 类类型具有哪些多投事件，是在 CDB 中就收集好了的。这被放在 CDB 中的一个名为 `_events` 的数组中。IDB 与 CDB 在两个不同的闭包中，因此在 IDB 中访问 CDB 的数据，需要通 IDB 的 `upvalue` 值来实现。这些 `upvalue` 值在 `cls.OnClassInitializion()` 之后，通过创建 `cls` 的成员引用保留给 IDB 的一个更外层的闭包。我们上一小节使用到的 `$map()` 和 `$attr()` 都来自于这种机制：

```
// some member reference for class
var $all = cls.all;
var $map = cls.map;
var $attr = cls.attrAdapter;

// 对于 InstanceDataBlock () 来说，上面的变量声明是一些 upvalue 值
function InstanceDataBlock () {
    ...
}
```

注意到这里多声明了一个 `$all`，它是为了存取更多的 CDB 数据而预留的——不过 Qomo 目前只是在实例构造周期（之前）用它来获得 `_events[]` 数组：

```
cls.Create = function () {
    // ...

    // 4. 抄写和初始化多投事件
    var all = $all('event');
    for (var i=0, imax=all.length; i<imax; i++) this[all[i]] = new MuEvent();

    // ...
}
```

这个 `_events[]` 数组只保存了事件名，而没有任何的事件响应例程。所有的例程都必须通过 `obj.aEvent.add(...)` 这样的形式添加到事件投送列表中。但一个多投事件对象是如何实现的并不是本小节的话题，这些内容留在“1.3 多投事件系统”中讲述。

由于 `_events[]` 数组不能保存事件响应例程，因此如果想要在类构造周期中“为所有的实例声明一个共用的事件响应”，下面的代码是错误的方法：

```
function MyObject () {
```



```

this.OnNotify = function() {
    ...
}
}

```

这是因为 Qomo 类构造周期中所有的 “this.OnXXXX” 都只有名称上的含义^①。

^① 但必须给该成员赋一个函数，你可以用 Qomo 全局变量中的 NullFunction 来表明这是一个没有明确参数传入的事件，或者可以自定义一个有形式参数的空函数来指示语法。这些细节我们在后面的 “0 由于对于用户代码来说，this.Create 相当于一个实际上是一个标准的构造器函数（传入的 this 引用为 new 创建的对象实例），所以用户可以按自己的习惯在这里书写任意的 JavaScript 代码，但是与 “类声明相关的” 一些语法不能在这里使用：

👉 Attribute(), _set(), _get(), _cls() 不能使用；

👉 this.getValue() 等不能声明成特性。

不过 this.Create 与类声明在同一个闭包中，因此可以访问类的私有成员和方法。

除了给实例添加新的成员之外，用户代码也可以重写任何从类类型继承的方法，以及向多投事件添加事件响应句柄——大多数情况下你不必重写多投事件对象。由于在实例构造周期已经可以使用 this.inherited()，所以当用户代码重写了继承自类类型的方法后，用户代码可以用它来调用父类同名方法，而无需使用特殊的 hook 过程来做这件事。例如：

```

function MyObject() {
    this.aMethod = function() { ... }

    this.Create = function(x, y) {
        // 重写从类类型继承的方法 aMethod
        this.aMethod = function() {
            // 用 inherited() 调用父类的 aMethod 方法
            return this.inherited();
        }
    }
}

```

由于这里是实例构造周期，所以对于实例引用 this 来说它的父类 aMethod，其实就是在类构造周期中声明的那个方法。用户不必象下面这样写代码：

```

// 用 hook 的方法来实现上例的功能
this.Create = function(x, y) {
    var func = this.aMethod;
    this.aMethod = function() {
        return func.apply(this, arguments);
    }
}

```

所以你必须实例构造周期来做这件事，并且如果子类覆盖了 `this.Create()`，那么一定要调用 `inherited()` 来保证该事件响应被添加到事件对象中。例如：

```
function MyObject() {
    var doNotify = function() {
        ...
    }
    this.OnNotify = NullFunction;

    this.Create = function() {
        this.OnNotify.add(doNotify);
    }
}

function MyObjectEx() {
    // ...

    // 若子类覆盖了 this.Create, 则应调用 inherited() 来添加 doNotify
    this.Create = function() {
        // ...
        this.inherited();
    }
}

TMyObject = Class(TObject, 'MyObject');
```

```
}
```

不过应该留意的是，`this.get`、`this.set` 和 `this.inherited` 与前面所提到的 `this.value` 一样是在实例方法，而不是来自于类类型的方法。只不过这些实例方法是在 `this.Create` 调用之前，由 Qomo 的内核在 `this` 引用上添加的罢了。正是由于它们并没有在类类型中声明过，因此也不可能在 IDB 中的这三个方法内部再调用 `this.inherited()`。更进一步地说，如果我们要重写三个方法，我们其实只能使用上面的 `hook` 方法（参见“1.2.4.2 实例构造周期重写读写器 (getter/setter)”）。

同样的原因，`this.OnXXXX` 这些事件名对于类类型来说是没有意义的，准确地说，它们已经从类类型的原型中被清除了。因此 `this.OnXXXX` 在继承链中并没有同名方法。从语义上来讲，调用一个事件亦即意味着触发它，而在事件响应例程中再次调用 `this.inherited()` 就会导致事件执行中再次触发自身，这显然是不合理的。综上所述，用户代码不可能在一个事件中使用 `this.inherited()`。Qomo 类继承系统的高级话题”中将会再次提及。

```
TMyObjectEx = Class(TMyObject, 'MyObjectEx');
```

1.2.3.7. 细节 4：实例构造周期的特殊性

如前所述，在真正进入用户代码中 `this.Create` 所定义的实例构造周期之前，Qomo 已经为该实例：

- 👉 创建了 IDB，并通过 IDB 初始化了特性与读写器；
- 👉 通过 IDB 初始化了 `this.inherited()` 方法以及相关的缓存数组；
- 👉 通过 CDB 初始化了所有的多投事件对象。

这意味着对象实例总是存在 `this.get`、`this.set` 和 `this.inherited` 方法，并且如果有事件，则 `this.OnXXXX` 成员总是有效的。接下来，Qomo 中的 `cls.Create` 通过一行代码进入 `this.Create()` 以启动用户的实例构造周期：

```
cls.Create = function () {  
    // ...  
  
    // 5. 真正地进入实例构造周期  
    if (this.Create) this.Create.apply(this, arguments);  
}
```

由于这里的 `this` 是一个实例，基于原型继承的特点，在当前子类没有重写 `this.Create` 时，上面代码中的 `this.Create` 将指向父类的 `this.Create`。只有所有父类都没有声明 `this.Create` 时，该方法才不会被执行。

该行代码的调用方式表明了两件有趣的事实：

- 👉 因为这里没有用 `return` 返回值给外部代码的 `new` 创建过程，所以用户代码在 `this.Create` 中不能通过 `return` 来返回一个新的对象；
- 👉 这里将 `arguments` 传给了 `this.Create` 过程，意味着用户代码可以在 `this.Create` 中针对参数做灵活处理，甚至返回“同类但不相似的”对象实例。

如下代码说明这些特性：

```
function MyObject() {  
    // ...  
    this.Create = function(x, y) {  
        // 这里可以根据参数来对实例做修改  
        if (arguments.length>1) {  
            this.value = x * y;  
        }  
    }  
}
```

```

    // (尽管可以任意修改实例 this, 但)你不可能通过 return 来返回新对象
    return {};
}
}
TMyObject = Class(TObject, 'MyObject');

var obj1 = new MyObject(10, 5);
var obj2 = new MyObject();

```

由于对于用户代码来说，`this.Create` 相当于一个实际上是一个标准的构造函数函数（传入的 `this` 引用为 `new` 创建的对象实例），所以用户可以按自己的习惯在这里书写任意的 JavaScript 代码，但是与“类声明相关的”一些语法不能在这里使用：

- 👉 `Attribute()`, `_set()`, `_get()`, `_cls()` 不能使用；
- 👉 `this.getValue()` 等不能声明成特性。

不过 `this.Create` 与类声明在同一个闭包中，因此可以访问类的私有成员和方法。

除了给实例添加新的成员之外，用户代码也可以重写任何从类类型继承的方法，以及向多投事件添加事件响应句柄——大多数情况下你不必重写多投事件对象。由于在实例构造周期已经可以使用 `this.inherited()`，所以当用户代码重写了继承自类类型的方法后，用户代码可以用它来调用父类同名方法，而无需使用特殊的 `hook` 过程来做这件事。例如：

```

function MyObject() {
    this.aMethod = function() { ... }

    this.Create = function(x, y) {
        // 重写从类类型继承的方法 aMethod
        this.aMethod = function() {
            // 用 inherited() 调用父类的 aMethod 方法
            return this.inherited();
        }
    }
}

```

由于这里是实例构造周期，所以对于实例引用 `this` 来说它的父类 `aMethod`，其就是在类构造周期中声明的那个方法。用户不必象下面这样写代码：

```

// 用 hook 的方法来实现上例的功能
this.Create = function(x, y) {
    var func = this.aMethod;
    this.aMethod = function() {

```

```
    return func.apply(this, arguments);  
  }  
}  
}
```

不过应该留意的是，`this.get`、`this.set` 和 `this.inherited` 与前面所提到的 `this.value` 一样是在实例方法，而不是来自于类类型的方法。只不过这些实例方法是在 `this.Create` 调用之前，由 Qomo 的内核在 `this` 引用上添加的罢了。正是由于它们并没有在类类型中声明过，因此也不可能在 IDB 中的这三个方法内部再调用 `this.inherited()`。更进一步地说，如果我们要重写三个方法，我们其实只能使用上面的 `hook` 方法（参见“1.2.4.2 实例构造周期重写读写器 (getter/setter)”）。

同样的原因，`this.OnXXXX` 这些事件名对于类类型来说是没有意义的，准确地说，它们已经从类类型的原型中被清除了。因此 `this.OnXXXX` 在继承链中并没有同名方法。从语义上来讲，调用一个事件亦即意味着触发它，而在事件响应例程中再次调用 `this.inherited()` 就会导致事件执行中再次触发自身，这显然是不合理的。综上所述，用户代码不可能在一个事件中使用 `this.inherited()`。

1.2.4.Qomo 类继承系统的高级话题

在 JavaScript 中，如果需要做对象系统的继承性检测，那么构造器通常被作为 `Class` 或 `ClassName` 来使用。例如：

```
<obj> instanceof <class>
```

这个语法中 `class` 就由一个构造器函数来承担。所以一般都说 JavaScript 中没有严格的“类”的概念。

在 Qomo 中，类、类类型、构造器是不同概念。我们此前严格的区别过类类型与构造器，例如下面的代码中：

```
// 1. MyObject 是构造器  
function MyObject() {  
}  
  
// 2. TMyObject 是类类型  
TMyObject = Class(TObject, 'MyObject');
```

简单的区别是，类类型通常表示为“T<构造器名>”。但有些时候类类型（的引用）却需要通过特殊方法得到。例如本小节会讲到的“类构造周期中的 `_cls()` 工具函数”，或在匿名类类型中通过 `ClassInfo` 来得到。

但还是有一个问题：Qomo 中的“类”又是什么呢？在 Qomo 中，“类”是对象的原型。在内核代码中，它可能表示为“proto（原型）”或“base（基类）”。这用在不同的环境中。一般来说，Qomo 中的“类”对用户代码不可见，也无需关注。但是它并不神秘，因为我们此前已经在经常性地使用它了：

```
function MyObject() {  
    this.name = 'MyName'; // <- 类构造周期的 this，就是“类”  
}  
TMyObject = Class(TObject, 'MyObject');
```

那么我们说“（Qomo 中的）类是对象的原型”又是什么意思呢？下面的例子说明这一点：

```
var _this;  
  
function MyObject() {  
    _this = this; // <--创建一个全局变量引用  
}  
TMyObject = Class(TObject, 'MyObject');
```

// 显示 true，表明 MyObject() 构造器的原型，其实指向“类”
alert(**_this === MyObject.prototype**);

所以，从本质上来说，Qomo 的类继承系统仍然是构建在 JavaScript 的原型继承的基础之上的。一种比较拗口的说法是：“类”实例是对象实例的原型。

1.2.4.1. 类构造周期的完整特性与工具函数

在 Qomo 中用户代码所编写，是一个用于注册的“类声明函数”（尽管它在 Class() 函数中也被作为构造器使用），因此它所有的特性都表现为一种“类声明语法”。所谓类声明语法，就是指“仅于一个对象的表现有关”的语法，而不应该加入过多的逻辑代码。基本上来说，这些声明可以包括如下代码：

```
// 类声明  
function MyObject() {  
    Attribute(this, 'Value', 0); // 快速特性声明  
    var v = _get('Data'); // 取(当前类的)父类的特性值  
    _set('Data', v); // 置特性值，但不覆盖父类  
  
    var cls = _cls(); // 取类引用  
    var count = 10; // 声明类(私有)静态成员  
    var foo = function() {}; // 声明类私有函数  
    function foo2() {}; // 同上
```

```

this.data = 123; // 公开属性

this.getValue = function(){ // 特性读取器
    this.set(v); // 内部的读方法
    this.get(); // 内部的写方法
}

this.setValue = function(){ // 特性置值器
    this.set(v); // 内部的读方法
    this.get(); // 内部的写方法
}

this.method1 = function() { // 类方法
    this.inherited(); // 继承(调用)父类方法
}

// (other code...)
}

// 类注册
TMyObject = Class(TObject, 'MyObject');

```

上面的示例包括了“类声明周期”的绝大多数代码及其语法。当然，在“other code”中用户也可以加入自己的代码——例如初始化类的一些特性，或者将类加入全局的 **monitor** 等，但用户需要注意的是，在“类声明周期”：

- 👉 **this** 指向所有对象实例的原型；
- 👉 不能直接操作当前类声明的特性值(可能会存在一些限制)；
- 👉 不能给(类声明时的)构造器加入入口参数^①；
- 👉 这个构造器函数只被执行一次。

在这些代码中有四个工具函数是需要特别说明的，分别是：

- 👉 **_cls()**：取类类型的一个引用，也就是 **TObject**、**T<构造器名>**等；
- 👉 **_get()**：从 CDB 的 **Attr** 中取特性初值，该特性名、值可以是父类继承的；
- 👉 **_set()**：向 CDB 的 **Attr** 中置特性初值，该特性名可以是父类继承的；
- 👉 **Attribute()**：显式声明一个特性并置初值，以及设定特性存取标识符。

这几个工具函数是“调用的上下文受限的”：它们只能在类构造周期中使用。

^① 对于用户代码来说，类声明时的构造器不能加入入口参数很大程度上的限制了使用。但在“类继承体系”中，这是很合理的——你不能用同一个类声明去描述两个不同的类及实例。如果需要构建不同的实例，你应该在 **this.Create** 中使用参数。

从它们的实现代码中可以看到:

```
_get = function(n) { return _get.caller.caller.get(n) }
_set = function(n,v) { return _set.caller.caller.set(n, v) }
_cls = function() { return _cls.caller.caller }

Attribute = function() {
  // ...
  var i, argn=arguments.length;
  var Constructor = Attribute.caller;
  var cls = Constructor.caller;
}
```

这些函数都将通过“函数被调用时的上下文”来查找到“类引用(cls)”和“构造函数(Constructor)”——这里是指类声明时的 `MyObject()`等, 而非后来被替换的 `cls.Create`。能够这样做的原因, 是因为在 `Class()`函数的实现代码中, `Constructor` 总是在下面这样的环境中被调用:

```
var Constructor = eval(Name); // eval('MyObject'), etc.
// ...

var cls = function (Constructor) { // <-- Constructor.caller
  // ...
  base = new Constructor(); // <-- Attribute.caller
}
```

在这个类的构造周期中, `Attribute()`调用中的 `caller` 就是 `Constructor`, 而 `Constructor.caller` 就指向 `cls()`这个函数, 这就是 `Attribute()`中下面代码的来源(其它几个函数类同):

```
var Constructor = Attribute.caller;
var cls = Constructor.caller;
```

而 `cls()` 调用发生在一个类初始化 (`OnClassInitializtion`) 和结束化 (`OnClassInitialized`)之间:

```
cls.OnClassInitializtion(Constructor);
cls(Constructor);
cls.OnClassInitialized(InstanceDataBlock);
```

Qomo 在初始化中写着:

```
cls.OnClassInitializtion = function(Constructor) {
  // ...
  this.get = getAttribute;
  this.set = setAttribute;
  // ...
}
```



```
}
```

这样，`cls.get()`、`cls.set()`等特性在“类构造周期”就可用了。而接下来：

```
cls.OnClassInitializtion = function(Constructor) {  
    delete this.all;  
    // more ...  
}
```

又把这些属性和方法给清除掉。使得“类构造周期”之外不可能再通过这些方法来访问 CDB。

那么`_cls()`、`_get()`、`_set()`这些工具函数又有有什么用呢？

首先，如果父类设置了一个特性的初值，而子类需要知道或改写这个初值，应该怎么办呢？例如下面的代码是一个父类：

```
function MyObject() {  
    Attribute(this, 'Name', 'MyObject');  
}  
TMyObject = Class(TObject, 'MyObject');
```

很显然这里的'Name'指示了构造器名。但是子类的构造器名就不一样了，于是子类需要修改这个值。最简单的方法，它可以这么办：

```
function MyObjectEx() {  
    _set('Name', 'MyObjectEx');  
}  
TMyObjectEx = Class(TMyObject, 'MyObjectEx');
```

如果它需要检测父类中的'Name'值，那当然也能用`_get()`，原理上是一致的。

而如果通过`_cls()`，用户代码有机会取得一些系统内部函数的引用，包括`cls.all()`、`cls.map()`、`cls.get()`、`cls.set()`、`cls.attrAdapter`等。尽管用户代码可以用`hook`的方式来替换它们，但这种替换并不会影响到实例的 IDB 中对它们的使用——IDB 创建过程中所需的引用已经建立过了。除此之外，通过`_cls()`，还可以访问一些类成员，例如 `ClassName` 等。

在后面章节要讲述到的接口（`Interfaces`）的一些机制中，也还会使用到`_cls()`。不过现阶段我们基本不需要关心`_cls()`，它看起来功能很强，但对于用户来说意义却不甚巨大。

1.2.4.2. 实例构造周期重写读写器(getter/setter)

由于 `this.Create` 中可以做任何事，因此你事实上也可以重写 `this.get` 和 `this.set` 以定制新的 `getter/setter` 方法——尽管这是不被推荐的：

```
function MyHtmlController() {
    Attribute(this, 'Element');

    this.Create = function() {
        var _get = this.get;
        this.get = function(n) {
            var el, v = _get.apply(this, arguments);
            if (v == undefined && (el = _get.call(this, 'Element'))) {
                v = el.getAttribute(n) || el[n];
            }
            return v;
        }
    }
}

TMyHtmlController = Class(TObject, 'MyHtmlController');
```

这个例子是 `Qomo` 代码包的组件系统中的 `THtmlController` 类的一个简化版本。它的作用是：如果 `'Element'` 被置为一个 HTML 元素，那么当 `this.get()` 不能从当前对象的 `Attributes` 中取值时——特性名不存在或无有效值，就会尝试通过调用元素的 `getAttribute()` 来读取 HTML 标签属性，或从元素的对象属性中取值。这样一来，用户无须任何代码就可以做到 HTML 元素与 `Qomo` 对象的绑定：

```
<!-- HTML 代码 -->
<div id="myId" style="border: 1px solid red">hello</div>

<script>
var myId = document.getElementById('myId');
var obj = new MyHtmlController();
obj.set('Element', myId);

alert( obj.get('id') ); // <- 实际取值 myId.getAttribute('id')
alert( obj.get('style') ); // <- 实际取值 myId.getAttribute('style')
alert( obj.get('tagName') ); // <- 实际取值 myId.tagName
alert( obj.get('innerHTML') ); // <- 实际取值 myId.innerHTML
</script>
```

1.2.4.3. 类注册函数 **Class()** 的特殊语法与应用

Qomo 对象系统中的 **Class()** 是一个关键性的函数，它被设计得跟 JavaScript 的保留字 **class** 只存在字符大小写的区别——这也意味着在今后的版本（例如 Qomo for JS 2.0 中）有可能删除掉它。**Class()** 总是返回一个注册成功的类类型。它的入口参数形式非常复杂，最简单的情况是只有一个参数，即 **<Constructor Name>**。这有两种情况：

```
// 语法一：值为 'Object'，仅仅保留给 TObject 的声明（该声明位于 Object.js 中）
function Object(){
}
TObject = Class('Object');

// 语法二：值不为 'Object'，相当于调用 Class(TObject, '<Constructor Name>');
function MyObject(){
}
TMyObject = Class('MyObject');
```

Class() 标准的用法是两个参数，即：

```
// 语法三：一般性的语法，可以用类类型的祖先类 TObject 作为参数 AParentClassType
AClassType = Class(AParentClassType, ConstructorName);
```

这些用法还有两种可能的变形：

```
// 语法四：Class() 在类声明前调用
TMyObject = Class(TObject, 'MyObject');
function MyObject(){
}

// 语法五：只注册但忽略返回的类类型
function MyObject(){
}
Class(TObject, 'MyObject');
```

语法四使用了 JavaScript 编译期的特性：由于 **function** 直接量声明将在编译期被接受，因此 **TMyObject** 对 **'MyObject'** 的使用可以出现在它的声明之前。但是下面这样的用法就不行：

```
// 不正确的用法
TMyObject = Class('MyObject');
MyObject = function(){
}
```

语法五利用了 Qomo 中不强制使用类类型的特性，只将 **MyObject()** 注册成

为类，而忽略了返回的类类型。但类类型还是存在的，与它完全等义的引用可以在以下两个地方找到：

```
// 用 new 来创建对象实例
// （ 因为没有 TMyObject 标识符，所以不能用 obj = TMyObject.Create() 这样的语法 ）
obj = new MyObject();

// 1. 在 obj.ClassInfo 中存在类引用
cls = obj.ClassInfo;
alert(cls.ClassName);

// 2. 在命名空间上存在该类引用
var cls = eval(obj.ClassInfo.SpaceName + '.TMyObject');
alert(cls.ClassName);
```

我们的确有一些地方会试图忽略类类型。换言之，我们希望使用 Qomo 的机制，但我们不想为注册一个“T<构造器名>”这样的标符识来占用全局变量名。另外的一些情况，我们根本就不需要全局的“T<构造器名>”标识符，例如我们使用类工厂模式时，实例创建的工作已经交给了类工厂。那么我们可以用下面这样的代码来实现：

```
// 一个简单的类厂
function ClassFactory() {
    Attribute(this, 'ProductClasses');

    this.register = function(name, cls) {
        this.get('ProductClasses')[name] = cls;
    }

    this.newInstance = function(name) {
        var cls = this.get('ProductClasses')[name];
        return cls &&
            cls.Create.apply(cls, Array.prototype.slice.call(arguments, 1));
    }

    this.Create = function() {
        this.set('ProductClasses', new Object());
    }
}

// 待注册到类工厂的类
function Class_1() { }
function Class_2() { }
```

```

// 注册类
TClassFactory = Class(TObject, 'ClassFactory');

var clsFactory = new ClassFactory();
clsFactory.register('bird', Class(TObject, 'Class_1'));
clsFactory.register('chicken', Class(TObject, 'Class_2'));

// 在不使用类类型 (例如 TClass_1) 的情况下得到实例
bird = clsFactory.newInstance('bird');
chicken = clsFactory.newInstance('chicken');

// 检测实例, 分别显示'TClass_1'和'TClass_2'
alert( bird.ClassInfo.ClassName );
alert( chicken.ClassInfo.ClassName );

```

但是我们也发现一个问题, 上例中的'**Class_1**'和'**Class_2**'这些构造器的名字也没有任何意义——由于类厂的存在, 我们不会使用“**new Class_1()**”这种形式来构建一个 **bird**。而且由于这些构造器必须是声明为全局变量, 因此造成了命名的污染。那么, 我们可不可以使用匿名函数, 或让他声明在 **ClassFactory** 函数内的局部呢?

在 **Qomo** 中允许在 **Class()**的第二个参数中传入一个构造器引用, 而不是一个构造器的名称。由于构造器引用也可以是匿名函数, 因此这会使得注册后的类既不能用“**T<构造器名>.Create()**”来创建, 也不能用“**new <构造器名>()**”来创建。但这种特性正好可以用在上面的例子中:

```

function ClassFactory2() {
    // 待注册的类
    function Class_1() { }
    function Class_2() { }

    // 用类私有成员来初始化类厂中的子类列表
    var ProductClasses = {
        'bird': Class(TObject, Class_1),
        'chicken': Class(TObject, Class_2)
    }

    // 实例化
    this.newInstance = function(name) {
        var cls = ProductClasses[name];
        return cls &&
            cls.Create.apply(cls, Array.prototype.slice.call(arguments, 1));
    }
}

```

```

    }
}
TClassFactory2 = Class(TObject, 'ClassFactory2');

// 测试, 显示匿名的类类型 'TAnonymous'
var clsFactory2 = new ClassFactory2();
bird = clsFactory2.newInstance('bird');
alert( bird.ClassInfo.ClassName );

```

这样一来, `Class_1` 与 `Class_2` 就只是局部变量, 而不会影响到全局的标识符系统了。不过由于没有使用上例中的 `ClassFactory.register()` 的机制, 因此也不能动态添加类。有趣的是, 如果你接受更加复杂的语法, 那么也可以在 `ClassFactory.register()` 中使用匿名的构造器:

```

// 参见上例有关 ClassFactory () 的声明与注册
var clsFactory = new ClassFactory();
clsFactory.register('ostrich', Class(TObject, function() {
    // 类 3, 使用匿名函数作为构造器.
    // ( 可以使用 Qomo 类声明中的全部语法与特性 )
}));

// 显示匿名的类类型 'TAnonymous'
ostrich = clsFactory.newInstance('ostrich');
alert( ostrich.ClassInfo.ClassName );

```

最后, `Class()` 的实现与使用中还有一部分是关于接口系统与命名空间系统的扩展的。这些内容我们放在相关的章节中去讲述, 这里就先不讨论了。

1.2.4.4. Qomo 对象仍然是基于原型构造的

与其它的一些 OOP 框架不同的是: Qomo 对象仍然是基于原型构造的。这表明开发人员仍然可以使用 `prototype` 来修改实例和构造器的属性。重要的是, 这种修改对 Qomo 的类构造系统不会造成任何负面的影响。例如:

```

function MyObject() {
    Attribute(this, 'Value', 200, 'r');
}
TMyObject = Class(TObject, 'MyObject');

MyObject.prototype.Value = 100;

var obj = new MyObject();

```

```
alert(obj.Value);
alert(obj.get('Value'));
```

Qomo 框架采用这种“对原型继承透明”的设计，使得它可以更容易地嵌入到其它的框架或者系统中。大多数情况下，第三方的框架感觉不到 Qomo 的存在，也不会受到 Qomo 语法的任何影响。

对于匿名类类型来说，我们可以找到这个类的引用（它可能存在于命名空间、某个变量或者对象成员中），然后使用 `cls.Create` 来操作原型。例如：

```
var Classes = [];
Classes.push(Class(TObject, function() {
    // 匿名类类型的声明
    Attribute(this, 'Value', 200, 'r');
}));

var cls = Classes[0];
var proto = cls.Create.prototype;
proto.Value = 100;

var obj = Classes[0].Create();
alert(obj.Value);
alert(obj.get('Value'));
```

1.2.4.5. Qomo 对象构造的第二种语法

首先是作为一种习惯，我在 Qomo 中实现了一个“与 Delphi 完全一致”的创建对象实例的语法：

```
function MyObject() {
}
TMyObject = Class(TObject, 'MyObject');

var obj = TMyObject.Create();
```

其实“对象=类类型.Create()”在语义上更符合“类继承体系”，这一点我们在前面已经讲过。但是 Qomo 不希望将任何与 JavaScript 无关的语法或者语义“强加”给开发者。所以 Qomo 中仍然推荐使用 JavaScript 中标准的构造器语法：

```
对象 = new 构造器();
```

需要说明的是，尽管这两种方法得到的对象实例没有任何的区别，但是使用

Create()的语法在效率上会稍逊于使用 new 运算的语法。如果用户原意，可以使用 Create()语法的一个变种：

```
obj = new TMyObject.Create();
```

返回的对象实例仍然没有区别，但是效率却与直接使用“new 构造器()”完全一致。

还有什么原因使得我们必须实现一个“类类型.Create()”这样的语法呢？

其一，是因为在有些情况下，我们需要使用定制参数来调用构造器。这在上一小节的类工厂 ClassFactory2()中有一个示例：

```
function ClassFactory2() {  
    // ...  
    this.newInstance = function(name) {  
        // ...  
        cls.Create.apply(cls, Array.prototype.slice.call(arguments, 1));  
    }  
}
```

我们需要使用 newInstance()调用时的参数中的一部分（或全部）来创建对象实例，但是 newInstance()并没有明确的形式参数表。因此只能用函数的 apply()方法动态调用，并传一个用 Array.prototype.slice()动态获取的数组作为参数表。但是，在“new <构造器>()”这个语法中，我们并不能使用 apply()，也不能使用定制的参数表，类似于下面这样的代码是非法的：

```
new <构造函数>.apply(this, arguments);
```

其二，是因为在命名空间中，我们存储的是类的引用，而非构造器的引用。也就是说，在命名空间中找到的总是一个类引用：

```
cls = <a_name_space>.TMyObject;
```

此外，对于匿名类类型来说我们也只能保存 Class()返回的类类型。这些情况下，由于找不到“构造器”的引用，就只能使用“类类型.Create()”这样的语法了：

```
cls = <a_name_space>.TMyObject;  
  
// 使用'<类类型>.Create()'的语法  
var obj1 = cls.Creae();  
var obj2 = new cls.Create();  
  
// 错误的方法  
// var obj3 = new cls();
```

最后，不要试图对一个类使用 new 运算。在 Qomo 中，“new cls()”这样的

语义不被理解，因而总是触发一个异常。

1.2.4.6. 读写器声明的隐式效果

读写器方法必须跟 `Attribute()` 的特性声明在同一个 `MyObject()` 声明函数中，并且名称为 “`this.get<特性名>`” 或 “`this.set<特性名>`”。你可以只声明二者之一，或者全部。没有这种显式声明的，Qomo 会使用一个缺省的 `getter/setter`。

子类如果使用 `Attribute()` 声明了一个与父类同名的特性，那么子类的声明覆盖父类的，这主要是指初始值的覆盖。因此这种覆盖与用 `_set()` 工具函数来重写特性的初始值并没有本质区别。

子类可以通过 `Attribute()` 来重写读写权限标识符。但声明读写器方法也具有相似的隐式效果，因此你不必总是使用 `Attribute()`。例如：

```
function MyObject() {
    Attribute(this, 'Name', '', 'r'); // <-- 只读
}

function MyObjectEx() {
    // 声明写方法后，该子类的 'Name' 特性将是可写的
    // （声明读写器方法时，会隐式地打开读/写存取权限）
    this.setName = function() {
        // ...
    }
}

TMyObject = Class(TObject, 'MyObject');
TMyObjectEx = Class(TMyObject, 'MyObjectEx');
```

不单单是在子类中，即使是在当前类中，在 `Attribute()` 声明之后的读写方法声明，也具有修改存取权限的隐式作用。然而这里存在一个矛盾：如果 `Attribute()` 声明在某个读写器方法之后，那么也能禁止掉该读写器方法。例如：

```
function MyObject() {
    // 1. 声明 'Name' 特性的写方法
    this.setName = function() {
        // ...
    }

    // 2. 声明 'Name' 特性是只读的
    Attribute(this, 'Name', '', 'r');
}
```

```
TMyObject = Class(TObject, 'MyObject');
```

这种声明的结果将导致 `this.setName` 失效，'Name'特性最终是只读的。也就是说，`Attribute()`声明与特性读写器方法声明（的隐式效果）是相互覆盖的。

1.2.4.7. 通用读写器

如果某个类的许多特性的方法都基本一致，或者适合放在同一个函数中来实现，那么你可以为它们定义一个通用的读写器方法。例如：

```
// 不使用通用读写器的版本
function MyObject() {
    this.setName = function(v) {
        if (!v) v = 'normal';
        this.set(v);
    }

    this.setValue = function(v) {
        if (!v) v = 'normal';
        this.set(v);
    }
}
```

那么你可以用下面的代码来完成相同的功能：

```
// 使用通用读写器的版本
function MyObject() {
    this.setValue =
    this.setName = function(v) {
        if (!v) v = 'normal';
        this.set(v);
    }
}
```

通用读写器的作用，是让 `setValue()`与 `setName()`以及更多的读写器方法使用同一个特性存取函数。为了让你在代码中能够区分调用来自于哪一个方法，**Qomo** 在调用这个特性存取函数时，会多传入一个参数。根据这项特性，你可以写出如下的代码：

```
function MyObject() {
    this.setValue =
    this.setName = function(v, n) {
        if (!v) v = 'normal';
        this.set(v);
    }
}
```

```

    ExtObject['OnChange' + n]();
  }
}

```

这个例子中 `ExtObject` 是一个假设的外部对象。而在上面的特性存取函数被调用时，例如：

```

obj.set('Name', ' - MyName');
obj.set('Value', ' - MyValue');

```

则下面的外部对象方法也将被调用：

```

ExtObject.OnChangeName();
ExtObject.OnChangeValue();

```

当然，你也可以在这里传入参数 `v`，或者用 `switch()` 语句来识别变量 `n` 并加以处理。

最后，请注意一个细节，对于通常的 `set/getter` 来说，其声明方法是：

```

this.getName = function() { ...
this.setName = function(v) { ...

```

而对通用的特性 `get/setter` 来说，传入的特性名是追加在其后的：

```

this.getName = function(n) { ...
this.setName = function(v,n) { ...

```

不过两种方法实现的读取器的使用方法完全一致：

```

obj.get(n);
obj.set(n,v);

```

通用读写器并不是有意而为之的一个功能，而是在实现 `obj.set()` 与 `obj.get()` 方法时的一项“意外所得”。以下面的代码为例：

```

1  function MyObject() {
2      this.getName = function() {
3          return this.get();
4      }
5  }
6  TMyObject = Class(TObject, 'MyObject');
7
8  obj = new MyObject();
9  obj.get('name');

```

在调用“`obj.get()`”时，有关于 IDB 数据块中的 `get()` 方法的调用栈总是：

```
行 3: obj.get()
行 2: obj.getName()
行 9: obj.get('Name')
```

由于行 3 与行 9 处都是同一个函数，所以用户代码不能从行 9 处找到参数 'Name'（参见“ ”）。然而在行 2 处的 getName() 又不需要这个参数，因此当执行到栈顶的行 3 时，Qomo 实际上已经找不到“当前存取的特性名”了。为了解决这个问题，在 IDB 中实现 get() 方法时，我们在调用 obj.getName() 时就会多传入一个 Name 参数：

```
function InstanceDataBlock() {
    // ...
    this.get = function (n) {
        // ...

        // 从$Attr 中取读写器 (例如 this.getName)
        var f = $attr('get'+n);
        // 调用 this.getName, 并传入特性名 n
        if (f) return f.call(this, n);

        // ...
    }
}
```

所以我们在声明通用读写器时，读写器的第一个形式参数为 name，并且会在运行中得到来自于“obj.get('Name')”这行代码中的参数。

在 Qomo 内部，这种技巧也多有应用，例如 Attribute() 中的读写限制器——函数 cantRead()、cantWrite() 等等。

1.2.4.8. this.Create() 的作用与重写

在类声明中的 this.Create 会最终被用作实例构造周期，但当实例构造周期结束后，实例中的 this.Create 还有什么意义呢？

这是一个有趣的话题。对于曾经用过 Delphi 的开发人员来说，理解“<类>.Create”与“<对象>.Create”应该会是一个恶梦。而 Qomo 中的这项特性也正是从 Delphi 中借鉴来的，因此我并不期望开发人员一开始就能理解它。

在 Qomo 中，“<类>.Create”被用作构造器，对于 TMyObject 类类型来说，“MyObject”其实是“TMyObject.Create”的一个引用。所以下面两行代码是同义的：

```
obj = new MyObject();  
obj = new TMyObject.Create();
```

而 Qomo 特别的处理了 TMyObject.Create 方法，使得它可以通过下面的调用产生一个与上两种方法效果的对象实例：

```
obj = TMyObject.Create();
```

但上面三种方法生成的实例 obj 也（可能）具有 Create 方法。该方法总是指向类声明中的 this.Create——如果声明过该方法的话：

```
function MyObject() {  
    this.Create = function() { // <-- obj.Create 指向这里  
        //...  
    }  
}  
TMyObject = Class(TObject, 'MyObject');  
obj = new MyObject();
```

对于 Qomo 系统来说，所谓“实例构造周期”其实包括 this.Create 之前的一些工作：用 new 运算构建产生对象实例（this），并完成了一些 Qomo 内核的初始化工作（例如 IDB 和多投事件相关的代码）。因此对于用户来说，“实例构造周期”中的 this.Create 中的代码，事实上是对该实例——this——的初始化代码，而所谓的 obj.Create，也不过是指向这一初始化过程的方法而已。

也许读者已经意识到了：如果用户代码调用 obj.Create()，只不过是重新初始化对象实例 obj 自身而已。按照 Delphi 对“<对象>.Create”的设定，如果你调用这个方法，既不会产生新对象，也不会重写对象的内部特性。但是所有在 this.Create 中的初始化代码会被重新执行一次——如果它触发了事件，那么事件将再次触发；如果它调用了父类方法，那么将再次调用……

用户创建的对象实例的确有可能需要“重新初始化”，但重新初始化也可能带来灾难。如果开发人员不希望某个实例在运行期意外地调用一次 obj.Create，那么的确可以重写它。这并不复杂：

```
function MyObject() {  
    this.Create = function() {  
        //...  
        this.Create = NullFunction;  
    }  
}
```

这种重写可以保证 obj.Create 仅在初始化时被执行一次，但你不能用 delete 运

算来删除这个成员，`obj.Create` 删除后可能会暴露其父类原型中的 `Create()` 方法，因此并不能达到你预想的效果。

然而重写 `this.Create` 是非常少到的技巧，它给用户（我的意思是使用你的代码的二次开发者）带来的困惑远远大于它带来的好处。在整个 Qomo 项目中，目前只有一处使用了这种重写：在 `THtmlController` 类中，当一个对象被实例化后，它的 `obj.Create()` 方法将被重新指向 `obj.assign()`。

1.1.多投事件系统

Qomo 中的多投事件系统并不是一个复杂的系统，它之所以被独立为一个章节来讲述，原因在于它在 Qomo 系统中完全独立的：它与 Qomo 的 OOP 框架完全地脱离开，不利用任何的 OOP 特性、框架特性。

而且，最重要的一点是：Qomo 的多投事件系统对任何框架来说是“完全透明”的！因此，它可以在其它任何框架中，象一个普通的事件函数（响应句柄）一样地使用^①。

1.1.1.多投事件系统的基本特性与语法

我们说过在 JavaScript 中没有真正的事件系统，在 DOM 中的事件是由宿主提供的，对于 JavaScript 来说，它只是一个方法调用而已。

在 JavaScript 引擎中没有任何对“事件”的约定，但 Qomo 的对象系统中扩展了这一点。Qomo 约定，在类构造周期所遇见的任何一个以 'On' 为前缀的方法，都是事件。而且 Qomo 缺省支持事件多投，因此该事件将会由一个多投事件对象接管并处理。

所谓多投事件，是指当一个事件被触发时，事件消息会被投送到多个响应例程，响应例程自行决定是否、以及如何处理该事件。基本的结构如下：

```
// 初始多投事件，并添加响应例程
var e = <多投事件对象>;
e.add(func_1);
e.add(func_2);
e.add(func_3);
```

^① 多投事件系统——以及今后讲述到的接口与命名空间系统——的设计思想完整地体现了 Qomo 的目标与宗旨，以及我们对框架系统的认知。我们期望一个框架系统的各个组件之间是低耦合的，甚至是零耦合的，我们也愿意用户将经过裁剪的 Qomo 的某些局部用于第三方的系统或框架中。对于第三方框架，我们希望 Qomo 尽可能是无侵扰、无强制规则的。

```
// 触发事件
function fireEvent() {
    e(v1,v2,v3);
}
fireEvent();
```

当事件被触发时，func_1~3 都将依次被激活，并收到参数 v1~3。

在 Qomo 中，多投事件对象由一个名为 MuEvent()的构造器负责创建，它的语法如下：

```
e = new MuEvent([handle1, [handle2, [...]]]);
```

请留意构造器允许初始化任意多个的事件响应。因此上面的示例中的代码可以写作：

```
// 初始多投事件，并添加响应例程
var e = new MuEvent(func_1, func_2, func_3);

// (略...)
```

由于 MuEvent()创建的多投事件对象具有以下方法：

```
/* 以下方法的形式参数中
   - foo: 一个用作事件响应例程的函数
   - obj: 指示调用 foo() 函数时传入的 this 引用
*/
event.add(foo)           // 添加一个函数作为事件响应，激活函数时传入持有 event 的对象
event.addMethod(obj, foo) // 添加一个方法作为事件响应，将以 obj.foo() 形式激活该方法
event.clear()            // 清空投送列表
event.close()            // 关闭事件对象(的投送列表)，调用该方法后投送列表不能再修改
```

我们可以用 add()和 addMethod()方法来维护一张事件激活时的投送列表，clear()方法用于清空列表。这个列表是只入的、有序的，先添加的事件响应总是先被激活。

除了具有这些方法之外，这个多投事件对象本身也是一个函数。也就是说，可以对上例中的 event 做函数调用。因此，激活一个事件的方法其实就是把它作为函数调用——因为 JavaScript 本身没有事件，因此它无论如何看起来还是很象一个方法。

多投事件对象本身也是一个函数的性质，可以解决投送列表的维护问题。下例可以在一个“既有的”投送列表前添加一个新的事件响应例程：

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

```
// 1. 对象的 OnNotify 事件将投送 func_1~3 这个列表
var obj.OnNotify = new MuEvent(func_1, func_2, func_3);
// 2. 重写 OnNotify, 新的事件将选投送到 func_4, 然后再投送到 func_1~3
obj.OnNotify = new MuEvent(func_4, obj.OnNotify);
```

多投事件对象的 `addMethod()` 方法允许调整响应例程执行时的 `this` 引用。这是一项有用的特性，例如 `setTimeout()` 这样的函数在执行期只允许传入函数，而不能传入对象方法。这使得定时执行一个对象方法的代码只能这样写：

```
function doTimer() {
    obj1.call();
    obj2.call();
}
setTimeout(doTimer, 1000);
```

在使用 `MuEvent()` 的情况下，我们可以把对象与它的方法“包裹到”一个多投事件对象的内部列表中，而将多投事件对象作为函数来使用。因此下面的代码就可以很简单了：

```
var e = new MuEvent();
e.addMethod(obj1, obj1.call);
e.addMethod(obj2, obj2.call);
// 注意事件对象 e 是一个函数
setTimeout(e, 1000);
```

多投事件对象的 `close()` 方法是一个有趣的性质，它用于关闭投送列表。事实上它是删除 `add()`、`addMethod()` 等等这个方法，使多投对象彻底地回复到它原来的状态：一个普通的函数。当一个多投事件对象被 `close()` 之后，将没有任何办法恢复“变更投送列表”的能力，也没有办法检测这个函数是否是一个 `MuEvent()` 创建的多投事件对象。

最后，由于多投事件对象本身也是一个函数（不管它是否 `close`），因此它自己也可以被添加到自己的投送列表中。对此 `Qomo` 没有做任何限制——因为我们无法证明递归地投送事件是多余的特性。

1.1.2. 多投事件系统的实现

多投事件的实现其实可以做得非常简单。一般性的实现方法是这样：

```
function MuEvent() {
    // this is a new obj instance
    var all = this;
    all.length = 0;
```



```

function add(foo) { /* ... */ }
function addMethod(obj, foo) { /* ... */ }
function clear() { /* ... */ }
function close() { /* ... */ }
function run() { /* ... */ }

var e = function() { return run.call(this, arguments) }
e.add = add;
e.addMethod = addMethod;
e.clear = clear;
e.close = close;

return e;
}

```

这种方法很自然，也容易被理解。而且由 `new()` 关键字构造的对象实例 `this` 已经被内部变量 `all` 执有了一个引用，用以建立事件列表，避免了不必要的开销。看起来是不错的，因而大多数其它框架的多投事件系统都采用这种方式来实现。但是这种情况下，我们对比多个事件对象，会发现一个不可接受的事实：

```

var e1 = new MuEvent();
var e2 = new MuEvent();
// 显示 false, 表明两个方法不是同一个函数实例
alert(e1.add === e2.add)

```

这就是说：有多少个事件对象，就会有多少个 `add`、`clear` 等等方法。其开销极其巨大： $n * 5$ 。

Qomo 试图让这些方法变成完全相同函数引用，而不是每个对象执有的不同函数实例。然而由于 `new MuEvent()` 返回的是函数，所以不能利用“原型共用属性”的特性。因此，Qomo 让每一个 `MuEvent()` 持有一个唯一的标识 `Handle`：

```

MuEvent = function() {
    var all = {
        length : 0,
        search : function(ME) { ... }
    }

    function _MuEvent() {
        // get a handle and init MuEvent Object
        var handle = all.length++;
        var e = function() { ... } // <--多投事件对象，本身是一个函数
        // ...
    }
}

```

```

    return e;
}
return _MuEvent;
}();

```

这个 `handle` 本质上是一个索引，用来在 `all` 对象中查找指定的投送列表，例如 `all[10]` 就是索引为 10 的多投事件对象的投送列表。

但是接下来的问题就变得非常复杂了：我们如何让某个多投事件对象执有这个句柄呢？最简单的方法可以是这样：

```

MuEvent = function() {
    function add(foo) {
        var list = all[this.handle];
        list.push(foo);
    }

    function _MuEvent() {
        // (略, 参见上例)
        e.handle = handle;    // 使用 handle 值为公开属性
        e.add = add; // 这样赋值 add() 只有一个引用
        return e;
    }
    return _MuEvent;
}();

```

也就是我们通过对象 `e` 的公开属性 `'handle'` 来保存句柄值。但是你知道，这是一个外部属性，如果用户删除这个成员，或者修改它从 10 变成 20 会怎样呢？——系统全乱套了是不是？

所以在 `all` 对象中我们保留了 `search()` 方法，这个方法用于查找当前的多投事件对象的 `handle` 到底是多少^①。Qomo 找到了一个非常简单的方法来“查找”这个 `handle`，它利用了两条特性：

- 👉 事件对象本身是一个函数，因此也是可以被调用的；
- 👉 在 JavaScript 中对象实例是全局唯一的（它只等值于它自身的引用）。

这段代码是这样写的：

```

1  MuEvent = function() {
2      var GetHandle = {};
3

```

^① 这里用 `search()` 这个名字，是因为在 Qomo 的早期版本中这里是一个“对象/handle 值”的对照表。

```

4     var all = {
5         length : 0,
6         search : function(ME) {
7             var i = ME(GetHandle), me = all[i];
8             if (me && me.event==ME) return me;
9         }
10    }
11
12    function _MuEvent() {
13        // get a handle and init MuEvent Object
14        var handle = all.length++;
15        var ME = function($E) {
16            if ($E==GetHandle) return handle; //<-- 这里可以访问 upvalue 值 handle
17            if (all[handle].length > 0) return run.call(this, handle, arguments)
18        }
19        // ...
20
21        return ME;
22    }
23    return _MuEvent;
24 }();

```

注意第 2 行声明了一个对象 `GetHandle`，它其实什么作用也没有^①，而只是一个标识符：唯一的，`MuEvent()`函数外部无法访问也无法伪造的标识。这个标识会在 `all.search()`和 `ME()`这两个函数中使用。在 `ME()`，当形式参数 `$E` 为对象 `GetHandle` 时，函数就返回它的 `handle`。

`ME()`这个函数其实就是我们用 `MuEvent()`来构造的多投事件对象——注意在 21 行代码上它被作为该构造器的返回值。因此，在用户代码中：

```

var obj = new Object();
obj.event = new MuEvent();
obj.event(); <-- 激活多投事件

```

这里激活多投事件时所执行的函数调用，就是 `ME()`。但由于它位于 `MuEvent()`构造闭包之外，因此永远不可能访问到 `GetHandle` 这个对象，也不可能变成这样的调用：

```

obj.event(GetHandle); <-- 在 MuEvent() 之外无法行到 GetHandle 引用

```

这样一来，`handle` 被保护起来，而又能在 `all.search()`中快速地取得。然后通过 `all[i]`立即得到了多投事件列表。也因此，`add()`、`addMethod()`等方法的实

^① 换言之，你也可以省略掉这个声明，而使用 `all` 或者其它任何的局部对象变量来替代它。

现代码都写成如下形式：

```
function addMethod(obj, foo) {  
    var e = all.search(this); // 取多投事件列表  
    if (e) e.push(foo, obj);  
}
```

1.1.3.多投事件的中断与返回值

在许多系统框架中，事件是没有返回值的，因为事件只是一个行为的通知，而不是行为执行的逻辑。但有些时候事件也需要返回值，因为事件自身具有逻辑，也需要被外部系统检测。

这是一个两难的选择，Qomo 在这其中建立了一条规则：

- 👉 如果事件不返回值（即 `undefined`）则忽略，否则认为它有一个有效返回值；
- 👉 当投送列表中有多个有效返回值时，向外部返回最末一个。

例如以下多投事件：

```
function f1() {  
    // 无返回值，即 undefined  
}  
function f2() {  
    return 'test';  
}  
function f3() {  
    return undefined;  
}  
  
// 建立多投事件，激活，并显示事件执行结果值'test'  
e = new MuEvent(f1, f2, f3);  
alert( e() );
```

投送列表中，`f1` 与 `f3` 的返回值都被忽略，所以最终输出结果是'test'。如果在上面代码中再添加新的、返回有效值的事件响应例程，那么结果值会受到影响：

```
(续上例)  
function f4() {  
    return 100;  
}  
  
// 添加新的事件响应例程，激活，并显示事件执行结果值 100  
e.add(f4);  
alert( e() );
```

这个例子也说明：如果你确定了一个多投事件响应的流程，那么你应当 `close` 它，以避免第三方代码干扰你的既定规则。

另外一个问题，则是一个多投事件列表能否被中断。尽管这个问题也存在争议，但当我们把系统隔离成内、外两个部分时，我们就会注意到：框架系统的开发人员为某个事件缺省的、内在的投送列表负责，而应用系统则为外在的投送列表负责。因此，如果我们提供给用户中断多投事件投送的能力，那么最低限度是：用户代码不能中断系统的投送列表。

在 Qomo 中这样的限制实现为如下的规则：

☞ 响应例程能返回一个中断信息，要求系统停止向其后续添加的例程投送事件。

例程不可能改变投送顺序或者删除列表中的某个部分，因此它的影响只能是针对当前投送序列的，也只能对当前位置之后产生影响——用户例程不可能早于对象系统或内核系统向多投事件添加响应例程，因此它不可能更早地中断投送。下例说明例程返回中断的方法：

```
// (部分响应例程声明参见上例)
function f5() {
    return new BreakEventCast('hello');
}
function f6() {
    return 'test';
}

// 创建多投事件对象并初始化投送列表 f1~6
e = new MuEvent(f1, f2, f3, f4, f5, f6);
// 触发事件，返回值 'hello'
alert( e() );
```

我们用到了一个特殊的构造器 `BreakEventCast()`：如果响应例程返回它的一个实例，则多投事件系统会中断投送过程。它接一个任意形式的参数，如果该参数有值，则会认为是该例程（上例上的 `f5()`）返回的有效性；如果该参数无值，则该例程没有有效的返回值。也就是说 `BreakEventCast()` 对象同时有返回值和中断投送两个作用。由于返回值会参与到前面所说的规则运算中去，因此，如果 `f5()` 的中构造 `BreakEventCast()` 实例时使用了 `'hello'` 参数，那么它就覆盖了 `f4()` 的返回值 100；如果没有使用任何参数，那么这次事件触发就会返回 100。

1.1.4.多投事件系统的安全性

前面讲到，我们通过隐藏 `handle` 来提高 Qomo 中的多投事件系统的安全性。但是这仍然是不够的。下面我们仔细评估这些代码的安全性：

```
var all = {
  // 注：多投事件的外部方法都会调用 search 方法来获取内部列表，因此我们应检测它的安全性
  search : function(ME) {
    // 步骤 0：获取句柄
    var i = ME(GetHandle), me = all[i];
    // 步骤 2：检测事件对象
    if (me && me.event==ME) return me;
  }
}

function _MuEvent() {
  var ME = function($E) {
    // 1. 当参数$E是 GetHandle 对象时，返回句柄
    if ($E == GetHandle) return handle;
    ...
  }
}
```

这三处代码中有两个问题：

- 👉 问题 1：当我们从步骤 0 开始获取句柄时，我们需要考虑步骤 1 过程是否安全；
- 👉 问题 2：当我们回到步骤 2 时，我们需要考虑为什么要复核这个事件对象。

对于问题 1，因为 `GetHandle` 是一个对象，所以我们通常只需要使用 `'=='` 运算就可以检测，但为什么要使用 `'==='` 运算呢？对于这里的检测，我们是否设想过一个问题：如果 `$E` 不是对象呢？下面的例子说明这种情况：

```
hack = {}.toString();
e = new MuEvent();
// 显示 handle 值!
alert( e(hack) );
```

看看，由于 `hack` 是字符串，因此在进行 “`$E==GetHandle`” 比较时，后者会先被转换为字符串，而这显然与 `hack` 是等值的。所以我们被欺骗了。

但这又有什么用呢？因为这个 `handle` 值毕竟是一个内部的索引，我们拿不到 `all` 对象，也不能（在运行期）进入 `MuEvent()` 函数的闭包去创建一个子函数，所以我们仍然操作不到具体的多投事件列表呀？

然而我们还可以更进一步地欺骗这个框架：创建一个假的事件对象！下面的例子说明这种情况：

```
// 0. 我们关闭掉事件 e 的投递列表
e = new MuEvent(f1, f2, f3);
e.close();

// 1. 通过 hack 取得事件 e 的索引
idx = e(hack);

// 2. 构造一个假的事件对象
fake = function() {
    return idx; // 若经过以上步聚取 idx 值，也可指定任意索引
}

// 3. 从真的事件对象上取方法引用
evt = new MuEvent();
fake.add = evt.add;
fake.clear = evt.clear;

// 4. 添加或清除列表的操作将会作用在事件 e 上——然而此前它已经被 close() 了
fake.add(myFunc);
fake.clear();
```

对于这个例子，如果我们在处理前面的问题 2 时，不使用 `me.event` 对 `me` 进行复核，那么 `fake.add` 和 `fake.clear` 就会作用在事件 `e` 上。因此 Qomo 中实现了这样的机制：添加 `me.event` 属性，使它在多投事件对象被创建时指向该对象的一个引用；而等到调用 `all.search` 方法时，多投事件对象（函数）作为 `this` 引用传入，变成 `search()` 方法的形式参数 `ME`：

```
// 0. 创建多投事件对象时为它保持一个引用在 all[handle].event 中
function _MuEvent() {
    // ...
    all[handle] = this;
    this.event = ME;
    // ...
}

// 1. addMethod 方法中的 this 引用即是多投事件对象
function addMethod(obj, foo) {
    var e = all.search(this);
    // ...
}

// 2. search 方法中的形式参数 ME，并以 me.event 与它复核
```

```

search : function(ME) {
    var i = ME(GetHandle), me = all[i];
    if (me && me.event==ME) return me;
}

```

综上所述，对于前面所列的两个问题：在 `GetHandle` 检测时使用“全等运算符（`===`）”，以避免类型转换带来的影响^①；而为了解决问题二可能产生的伪造，Qomo 在投送列表的维护中进行了复核。通过这些（看起来复杂而冗余的）机制，Qomo 以尽可能少的代价提高了内核的安全性。

1.2. 接口系统

有两种接口。一种是纯粹描述性的，例如 UML 中的接口，以及 C++ 中的接口都属于这种。纯描述性的接口自身不必实现，也不从接口中派生子系统（或子系统功能），接口作为一个标识，用来表明其它的某种系统^②是否实现过它。另一种接口是描述并实现性的，例如 COM 框架中的接口既是描述也是实现。描述并实现的接口中，接口可以用于编程，而不仅仅是标识用来另外的编程系统。

在 JavaScript 的其它框架中也实现过接口，但一般是用来描述的。例如微软的 Atlas 框架，框架系统使用下面的代码来实现接口：

```

// 1. 声明接口
Web.IArray = function() {
    this.get_length = Function.abstractMethod;
    this.getItem = Function.abstractMethod;
}

// 2. 注册接口，以及为类注册接口
Type.registerInterface("Web.IArray");
Type.registerClass('Web.UI.Data.DataControl', Web.UI.Control, Web.IArray);

// 3. 接口操作(方法)
Function.prototype.implementsInterface = function(interfaceType) { ... }
Function.prototype.isImplementedBy = function(instance) { ... }

```

^① 在这里列举这个例子的目的，也是想提醒读者在应用开发过程中留意等值与全等检测，以及其它运算符带来的隐式类型转换效果。

^② 例如对象或结构。一般来说是通过对象系统来实现接口，但在一些语言（例如 C#）中也可以通过结构或其它数据类型来实现接口。在 JavaScript 中，由于“一切都是对象”，所以一般情况下，实现接口的总是对象——即使实现者是一个函数，也是以“函数对象”的形式实现接口。

因此用户代码要么是用 `implementsInterface()` 来查询类是否实现过接口，要么用 `isImplementedBy()` 来查询接口被谁实现。例如在属性检测 (类似反射) 时，有这样的代码：

```
Web.TypeDescriptor.getProperty = function(instance, propertyName, key) {
    if (Web.ICustomTypeDescriptor.isImplementedBy(instance)) {
        return instance.getProperty(propertyName, key);
    }
    // ...
}
```

由于这种接口只起到“检查”的作用，因此这样的代码与下例并没有本质区别：

```
Web.TypeDescriptor.getProperty = function(instance, propertyName, key) {
    if ('getProperty' in instance) {
        return instance.getProperty(propertyName, key);
    }
    // ...
}
```

在 Qomo 中，接口既是描述，也是实现。实现的接口 (的方法) 可以被调用，这与 Win32 中的 COM 机制是类同的。不但如此，接口在 Qomo 中还可以作为一种数据类型来使用：“实现的接口”与“接口声明”之间存在实例化的关系。

Qomo 在实现了更加完整的接口特性，并在这样的基础之上，完整地实现了 AOP 的种种特性。尽管接口是 Qomo 中非常关键的内核机制，但与多投事件一样，它也可以脱离 Qomo 项目运行。

1.2.1. 基本概念与语法

在上面的例子中，Atlas 使用下面的代码来“声明接口”：

```
Web.IArray = function() {
    this.get_length = Function.abstractMethod;
    this.getItem = Function.abstractMethod;
}
```

接口声明用户展示接口定义的功能。这个例子表明：

👉 如果 **“某个功能系统”** 实现了 `Web.IArray` 这个接口，则该功能系统必存在方法：`get_length()` 和 `getItem()`

上面的陈述存在一个问题，既然是“某个功能系统”，那么在实际使用中，如何指明“是哪个功能系统”呢？这个问题的解决方案就是“注册接口”：

```
Type.registerClass('Web.UI.Data.DataControl', Web.IArray);
```

这行代码表明：

👉 类 `Web.UI.Data.DataControl` 实现了 `Web.IArray` 接口

这样一来，根据上一个推论，类 `Web.UI.Data.DataControl` 的实例(instance)将必然具有如下方法：`get_length()`和 `getItem()`。

但在接口声明时，类 `Web.UI.Data.DataContro` 是否存在、如何实现以及如何使用，都是接口声明时所不关注的。而“接口注册”是（使用接口的）客户的行为，客户可以是类 `Web.UI.Data.DataContro` 的缔造者，因此可以确知该类是否实现接口，并正确的决定可否进行注册。

取了与 `Atlas` 相似的“接口声明”与“接口注册”之外，`Qomo` 还提供“接口查询（或获取）”。通过“接口查询”，客户可以得“被实现的接口”的一个引用。这些基本概念在 `Qomo` 中的语法如下：

```
// 1. 接口声明
IInterface = function() {
    this.QueryInterface = Abstract;
}

// 2. 接口注册
RegisterInterface(aImplementer, IInterface);

// 3. 接口查询
intf = QueryInterface(aImplementer, IInterface);
```

下面，我们将分别叙述这些概念中的三个关键函数：`Abstract()`、`RegisterInterface()`与 `QueryInterface()`。

1.2.1.1. 接口的声明与继承性

函数 `Abstract()`总是被用在接口声明中，它表明指定的方法是抽象的：`Qomo` 中的所谓接口声明，就是将一个构造器函数中所有的方法都声明成抽象的。

在 `Qomo` 中，接口本身是没有继承性的。接口是一个平面的、对象行为的描述系统，因此如果让接口本身实现出一种继承性，事实上是增大了复杂性。但接口在声明时，却是可以使用继承的技巧的。不过 `Qomo` 希望用户能明确地在代码中指出这一点。下例讲述这种技巧：

```
IMyIntf = function() {
    this.method_1 = Abstract;
    this.method_2 = Abstract;
```

```

}

IMyIntfEx = function() {
    IMyIntf.call(this);
    this.method_3 = Abstract;
}

```

在这个例子中，IMyIntfEx 对 IMyIntf 是有继承关系的：当 IMyIntf 发生更改时，IMyIntfEx 也会被修改。维护这种关系的方法，是显式地调用下面这行代码：

```
IMyIntf.call(this);
```

这种接口声明上的继承，是组织大型的接口系统的一种便捷方式，它并不表明与之对应的对象系统是否具有这种继承关系。接口只是通过这种方式来简化了声明，并没有任何其它的约定。

1.2.1.2. 接口的注册

可以用以下三种方式之一来注册接口，它们是完全等效的（因为它们指向同一个函数）：

```

// 下面的 obj 指一个对象实例或一个构造器函数，"..."号表明被注册的接口列表
Interface(aImplementer, ...);
RegisterInterface(aImplementer, ...);
Interface.RegisterInterface(aImplementer, ...);

```

它们表明将一批接口（列表）注册给 aImplementer，这里的 aImplementer 可以是除了 undefined 和 null 之外任意的东西^①，当 RegisterInterface()注册一批接口时，表明 aImplementer 是这些接口的公共实现者。在 Qomo 内部使用第三种格式来注册，它是一种最安全的调用方式——被覆盖的机会最小，而且被覆盖后也通常能释出异常。

在 Qomo 中使用 Class()注册类时，也可以隐式地注册接口。如下：

```

function MyObject() {
}

TMyObject = Class(TObject, 'MyObject'[, Interface1, Interface2, ...]);

```

这时后面列表中的接口也将被注册，它的 aImplementer 就是构造器 MyObject()。这种形式是在 Qomo 对象系统时最经常使用的方法。不过，即使你在 Class()

^① Qomo 不限制您将接口注册给值类型数据，但这没有什么特别的意义。如果你非要将接口注册给 undefined 或 null，那么你会得到一个返回值-1，除此之外没有任何效果。

注册类时没有同时注册接口，也可以后期通过 `RegisterInterface()` 来注册：

```
TMyObject = Class(TObject, 'MyObject');  
RegisterInterface(MyObject, Interface1, Interface2, ...);  
// 或  
RegisterInterface(TMyObject.Create, Interface1, Interface2, ...);
```

如果你将接口注册到一个构造器，则该构造器的所有实例都将继承该接口。所谓“实例继承该接口”的含义是指：如果父类实现了某种接口，其对象和子类都实现了该接口。例如父类实现了 `method_1`，当然子类也会有该方法，与之对应的接口也就自然地实现了。

除此之外，Qomo 也支持一种复杂的、隐式的接口注册机制：`Aggregate`。与 `Class()` 一样，`Aggregate()` 在内部也是通过调用 `RegisterInterface` 来注册接口的。有关 `Aggregate()` 的内容将在“1.4.3 Qomo 接口系统的高级话题”中讲述。

1.2.1.3. 接口的查询

接口查询（`QueryInterface`）的含义是指询问被查询者（`aImplementer`）是否实现了某个接口。如果函数调用成功，则返回一个“实现的接口”——我们称为“接口实例”，一个包含很多方法的普通对象。

我们可以对对象实例或函数进行接口查询：

```
var intf1 = Interface.QueryInterface(a Implementer, AInterface);  
var intf2 = QueryInterface(a Implementer, AInterface);
```

根据被查询者（`aImplementer`）的不同，有两种情况：

- ☞ 被查询者是函数，而函数内部聚合（`Aggregate`）了某些接口的实现；
- ☞ 被查询者是对象（这也包括函数直接量与函数对象，但这里作为对象处理），则按对象、对象构造器、类类型 `.Create` 方法的顺序查询接口的实现。

类似于第一种情况，对象构造器（以及与之相同的“类类型 `.Create` 方法”）所注册的接口，也是被注册在函数上。但是不能直接查询它们，因为它们的接口不是为自己，而是为实例注册的^①。

由于 `QueryInterface()` 总是根据接口的注册信息来查询，所以即使被查询者并没有真实地实现该接口，也可能会成功地返回接口实例。但由于被查询者没有实现或仅部分实现该接口，所以接口实例可能部分或者全部的方法仍然指向 `Abstract()`。很不幸，出于效率的考虑，Qomo 不能为您检测接口实例的每个方

^① Qomo 有多种注册接口的方法，而不仅仅是通过 `RegisterInterface()` 来显式地注册。

法的有效性。但它确保你（或实现者）在调用前给 `aImplementer` 注册过该接口。

如果被查询者（及其类、构造器等）从未注册该接口，则 `QueryInterface()` 会返回 `undefined`。

接口查询可能触发由参数不合法导致的异常。包括两种情况：

- 👉 参数 `aImplementer` 是 `null` 或 `undefined`；
- 👉 参数 `AInterface` 并不是有一个有效的接口。

对于第二种情况的解释是：如果一个接口只声明过，但从未被注册过，则该声明是无效的——这时它只是一个普通函数，而非接口。

1.2.2.接口实现

接口的声明、注册与查询这整个过程，只是对接口使用方法的阐述。其中，声明与注册是接口提供者的行为，查询（以及使用）是接口使用者的行为。但在这个描述中少了一个关键元素：接口由谁来实现。

这取决于使用哪种方式注册接口。例如：

```
Interface(aImplementer, ...);
RegisterInterface(aImplementer, ...);
Interface.RegisterInterface(aImplementer, ...);
```

就表明由 `aImplementer` 实现接口，而

```
function MyObject() {
}
TMyObject = Class(TObject, 'MyObject'[, Interface1, Interface2, ...]);
```

则表明由 `MyObject()`实现接口。后面将分别讲述它们具体的实现方法。在此之前我们先声明一个接口以用于后面的示例：

```
IMyIntf = function() {
    this.method_1 = Abstract;
    this.method_2 = Abstract;
    this.method_3 = Abstract;
}
```

1.2.2.1. 类实现接口

如果声明为一个类（构造器）实现接口，则表明它的所有实例都实现了该接口。这个实现是在类构造周期，或实例构造周期来实现的，也可以通过原型

来实现。例如：

```
function MyObject() {  
    // 1. 类构造周期实现  
    this.method_1 = function() { ... }  
  
    // 2. 实例构造周期实现  
    this.Create = function() {  
        this.method_2 = function() { ... }  
    }  
}  
// 注册类(和接口)  
TMyObject = Class(TObject, 'MyObject', IMyIntf);  
// 3. 通过原型实现  
MyObject.prototype.method_3 = function() {  
    // ...  
}
```

对于接口来说，三种实现方法没有本质的区别。因此如果用下面的代码往从该类的一个实例查询到接口后，对三个接口方法调用并没有任何区别：

```
obj = new MyObject();  
intf = QueryInterface(obj, IMyIntf);  
// 调用接口方法  
intf.method_1();  
...
```

1.2.2.2. 对象实现接口

除了使用 Qomo 的类，以及 JavaScript 的构造器之外，我们也可以用直接量声明一个对象并使用接口机制。当然，你得通过 RegisterInterface() 显式地注册它。例如：

```
// 声明对象与注册接口  
obj = {  
    method_1: function() { ... },  
    method_2: function() { ... },  
    method_3: function() { ... }  
}  
RegisterInterface(obj, IMyIntf);  
  
// 查询接口与调用接口方法  
intf = QueryInterface(obj, IMyIntf);
```

```
intf.method_1();  
...
```

由于函数也是对象，所以同样的方法也能用在函数中：

```
function IMyFunc() {  
    this.call = Abstract;  
    this.apply = Abstract;  
}  
  
function myFunc() {  
    alert('hi');  
}  
  
RegisterInterface(myFunc, IMyFunc);  
intf = QueryInterface(myFunc, IMyFunc);  
intf.call();
```

请注意这种情况下函数只是作为对象被注册，而不是象上一小节中那样作为构造器注册。这两种形式在 **Qomo** 中并没有任何的区别，只是接口实现方法上的不同：上一小节是类构造、实例构造周期与原型实现，此处是 **myFunc** 自身的方法实现了接口。

理论上说，你也可以直接从宿主环境或 **DOM** 中得到一个对象，并使用接口。例如：

```
IWriter = function() {  
    this.write = Abstract;  
}  
  
// (以下在浏览器环境中测试)  
RegisterInterface(document, IWriter);  
intf = QueryInterface(document, IWriter);  
intf.write('hi');
```

但在应用中这样的代码却可能存在问题。仅以本例来说，这不是 **Qomo** 接口系统的限制，而是因为 **document.write()** 并不是一个真正的函数，关于这一点，在上一章“1.7.1 宿主环境下的特殊类型系统”中已经讨论过了。所以在宿主环境或其它环境中使用接口，可能会存在一些问题。

1.2.2.3. 用户实现接口

一些非常罕见的情况下，用户可能无法创建任何对象，而只想使用接口。

由于接口本身也是对象，因此这事实上是可行的。在 Qomo 的内部实现上就的确存在这样的一个实例。

在 HtmlCanvas.js 中，由于绘图机制的设计，我们需要让 BeginPaint 到 EndPaint 之前的所有方法临转向到一个 Buffer 中。而 EndPaint 之后，我们一次性地从 Buffer 中调用所有的绘制过程，然后 Render 到输出对象（画布）上。但是所以的绘制方法我们是通过 IBaseCanvas 接口公布到系统外部的，因此如果我们“临时替换”它，也必须实现为一个接口。

如果我们实现一个 TCachedRender 类，那么这个类必须与 IBaseCanvas 接口同时维护——接口改动则类也必须修改。尽管这只需要模式化地添加一个转向 Buffer 输出的方法，但显然在接口与类系统中同时维护一个完全相同的机制是多余的和易出错的。那么，我们可否通过一个简单的方法，在 HtmlCanvas.js 中，在无需声明类或对象的前提下“实现”一个接口呢？

下面的示例中，接口完成了其自身的实现：

```
1 // fast made a faked interface.
2 function CachedRender(canvas) {
3     var _this = new IBaseCanvas();
4     for (var i in _this) {
5         if (_this[i] == Abstract) {
6             _this[i] = function(name) { //<-函数 func_1
7                 return function() { //<-函数 func_2
8                     canvas.RenderBuffer.push(name, arguments); //<-缓存所有方法名与参数
9                 }
10            }
11        } (i);
12    }
13    _this.QueryInterface = function(intf) {
14        return Interface.QueryInterface({}, intf)
15    }
16    return _this;
17 }
```

在第三行代码中我们创建了“接口的一个实例”，4~11 行则列举所有虚方法并重写为一个函数^①。除了重写所有的虚方法之外，该实例还具有一个名为 QueryInterface() 的方法。这是因为 Qomo 的接口系统具有另外一项特性：任何接口实例都具有 QueryInterface() 方法，以便查询接口中是否包括其它接口。这

^① 重写为函数 func_2。在该函数外层，匿名函数 func_1 通过形式参数 name 将接口方法名“i”保存在了 func_2 可访问的 upvalue 值中。

项特性借鉴自 COM 接口系统的实现。

最后，在第 16 行代码，我们返回接口实例。这样用户代码就可以直接使用
该接口实例来替换其它真实的接口实例。在任何外部系统看来，这与一个通
过 QueryInterface() 从对象、类或函数（内部聚合）中取得的接口都没有区别。

1.2.3.Qomo 接口系统的高级话题

1.2.3.1. 将接口注册到构造器、类与类类型

构造器、类与类类型都是函数，而且都与其个实例化的对象相关。正因为
它们与实例是相关的，所以实例是否、以及如何继承这些函数注册的接口，就
成了必须被解释的问题。

而在前面我们还讲到过，函数自身可以作为对象来注册接口。那么哪些是
注册给函数自身，哪些又是注册给实例化的对象的呢？

第一条基本规则是：

- 👉 如果一个接口被注册给一个构造器函数，那么该接口对所有实例有效，这意味
着“注册”是被实例继承的。

在 Qomo 对象中，“类类型.Create”其实指向构造器，因此与普通构造器使用
同一规则。除了这种显式的注册之外，在使用 Class() 进行类注册的同时隐式地
注册接口也使用本规则。因为 Class() 函数将用如下格式调用接口注册：

```
// from RTL\Object(1).js
Interface.RegisterInterface.apply(null, [cls.Create].concat(
    Array.prototype.slice.call(arguments, 2)
))
```

其中代码加粗部分将拼接出一个有效的 RegisterInterface() 函数调用的参数列
表，它的第一个参数（注册到的对象）指向构造器 cls.Create()。

第二条基本规则是：

- 👉 如果一个接口被注册给“类类型”，则该接口只对类类型（例如 TObject）及
其子类（例如 TMyObject）有效，对类的实例是没有意义的。

但为什么注册到“类类型”时，就不具有（对实例的）继承性呢？这根本
上说是因为某些接口只依赖于“类”的特性，而不必非得创建一个实例。例如，
我们声明如下类：

```
function MyObject() {
    Attribute(this, 'Name');
    Attribute(this, 'Value');
}
TMyObject = Class(TObject, 'MyObject')
```

对于 TMyObject 这个类来说，该类总是有 'Name' 与 'Value' 这两个特性，尽管这两个特性的“值”是对象的性质，但“具有这两个特性”却是该类的性质。因此下面这个接口：

```
IClass = function() {
    this.hasAttribute =
    this.hasOwnAttribute = Abstract;
}
```

是被注册给 TMyObject 类的。即使我们不创建该类的实例，仍然可以用下面的代码来获取该接口，以及调用接口方法：

```
intf = QueryInterface(TMyObject, IClass);

// 显示值 true
alert( intf.hasAttribute('Name') );
```

在类类型上注册接口，对类的实例没什么影响。例如我们认为类的 Create() 方法可以创建实例，因此 ICreator 接口被注册给 TMyObject：

```
ICreator = function() {
    this.Create = Abstract;
}
RegisterInterface(TMyObject, ICreator);
```

如果我们从 TMyObject() 中获取该接口，那么是可以成功调用接口方法并获取 obj 实例的；而同样的代码，用在对象上就没有意义：

```
// 查询 TMyObject 的 ICreator，并获取一个接口实例
intf = QueryInterface(TMyObject, ICreator);
// 从类的 ICreator 接口可以创建对象实例
obj = intf.Create();
// (尝试)从对象中查询 ICreator 实例
intf = QueryInterface(obj, ICreator);
// 因为对象不能继承类类型(TMyObject)的接口，因此下面的代码导致异常
obj2 = intf.Create();
```

很显然，obj.Create 方法通常是不存在的——即使有该方法，那么它也只是用于实例构造周期的一个普通方法，不能用于创建新的实例。所以在 Qomo 中，注册给“类类型”的接口，与前面的（显式或隐式地）注册给构造器的接口并

不相同，不能混用。

最后，我们的确有可能给类构造周期中的“类”（类实例的一个引用）注册接口。例如：

```
function MyObject() {  
    RegisterInterface(this, IMyIntf); // <-- 这里的 this, 即是类实例  
}  
TMyObject = Class(TObject, 'MyObject');
```

但这不见得有什么意义。因为在外部的代码中，我们只能通过构造器原型：

```
MyObject.prototype  
// 或  
TMyObject.Create.prototype
```

来得到这个“类”。而构造器原型上注册的接口，一般并不会直接使用，也并没有任何的继承性。

1.2.3.2. 函数作委托者的二义性问题

类类型、构造器作委托者时，既表示自身注册，也表示子类或子类实例注册。在 Qomo 中，这两种情况是不区分的。例如：

```
IFunction = function() {  
    this.apply =  
    this.call = Abstract;  
}  
  
// 为 MyObject 注册 IFunction 接口  
function MyObject() {  
}  
RegisterInterface(MyObject, IFunction);  
  
// obj 是 MyObject() 的一个实例  
var obj = new MyObject();  
  
// intf1 与 intf2 都是有效的 IFunction 接口实例。  
intf1 = QueryInterface(MyObject, IFunction);  
intf2 = QueryInterface(obj, IFunction);
```

也就是说，如果用户注册一个接口给函数，其本意只是自身的注册（上例中 IFunction 是 MyObject 的注册）。然而它的子类或实例却都可以查询到这样一个接口：无意义的、根本没被实现的——上例中 intf2 有效，但无法调用任

何接口方法。Qomo 无法识别这种极端的情况，这需要用户代码自身解决。

1.2.3.3. 接口的接口

我们可以给接口注册接口。因为接口本身是一个对象实例（是从接口中实例化的对象），而无论接口本身，还是实例化的对象自身都是没有继承性的，因此“接口的接口”也没有继承性。

Qomo 系统中，所有的接口都被注册了一个接口：IInterface^①。这表明我们可以通过下面的代码来查询任何接口：

```
function IMyIntf() {
    this.method_1 = Abstract;
}

function IMyIntf2() {
    this.method_2 = Abstract;
}

function MyObject() {
    this.method_1 = function() { alert('hi, 1') }
    this.method_2 = function() { alert('hi, 2') }
}

TMyObject = Class(TObject, 'MyObject', IMyIntf, IMyIntf2);

aObject = new MyObject();
intf = QueryInterface(aObject, IInterface);
myIntf = intf.QueryInterface(IMyIntf);
myIntf2 = intf.QueryInterface(IMyIntf2);
```

也可以不通过中介的 intf，而直接用下面的代码：

```
myIntf = QueryInterface(aObject, IMyIntf);
myIntf2 = myIntf.QueryInterface(IMyIntf2);
```

接口实例与对象实例都注册了接口，其好处之一是我们有些时候可以混用它。例如下面这样的代码：

```
(续上例)
// 一个用作缓存的数组(或 list 列表)
var arr = [aObject, myIntf, intf];
// 下面将通过接口调用三次 method_2() 方法，但这些接口是通过查询不同的实现者得到的
for (var i=0; i<arr.length; i++) {
```

^① 更确切地说，除 undefined 和 null 之外的所有东西都被注册了该接口。

```
var intf = QueryInterface(arr[i], IMyIntf2);
intf.method_2();
}
```

1.2.3.4. 接口内聚(内部聚合)

我们通常说的接口，是展示一个实现者的外部表现，例如一个对象有哪些方法。但是有些时候，一个实现者的内部如果具有某些能力，但又不能通过其外部表现展示出来，又能否通过接口来说明它呢？

例如一个函数，其内部可能隐藏了一些功能（内嵌函数，或提供查询内部结构的能力），如果我们非要将它展示出来，那么可以用如下的代码：

```
function myFunc() {
    function memberQuery() {
    }

    // 展示成 myFunc 的外部方法
    myFunc.memberQuery = memberQuery;
}
```

这种情况下我们也可以为 **myFunc** 注册一个接口并使用它：

```
function IMyFunc() {
    this.memberQuery = Abstract;
}
RegisterInterface(myFunc, IMyFunc);
```

但如果我们不想让 **myFunc()** 函数的外部成员有什么特别的变化，而仅仅是想在接口系统中使用它，那么又能怎么办呢？这种情况下，我们就需要声明“函数内部聚合了一些接口”。**Qomo** 中的“接口内聚”就是一个批量的“函数内部实现接口”的声明方法。这需要用到 **Aggregate()** 这个工具函数：

```
Intfs = Aggregate(aFunction, [intf1, [intf2, [...]]]);
```

注意返回值 **Intfs** 是一个“聚合接口对象”，这个对象只有一个方法：

```
Intfs.getInterface = function (intf) { ... }
```

通过 **getInterface()** 方法，用户代码可以用 **Intfs** 中取得一个未实现的接口：

```
intf = Intfs.getInterface(IMyFunc);
```

然后再去实现它。

仍以下例的 **myFunc()** 为例，可以写成如下的代码：

```
function myFunc() {
```

```

function memberQuery() {
}

// 1. 声明(注册)聚合
var intfs = Aggregate(myFunc, IMyIntf);
// 2. 取指定接口的一个引用
var intf = intfs.getInterface(IMyIntf);
// 3. 实现接口方法
intf.memberQuery = memberQuery;
}

```

现在我们就可以用 QueryInterface 来查询 myFunc() 内部聚合的接口了：

```

// 4. 执行函数
myFunc();
// 5. 查询接口
intf = QueryInterface(myFunc, IMyIntf);
// 6. 使用接口方法
intf.memberQuery();

```

但这里仍然有一个小问题。在步骤 4 中，我们需要执行一次函数，但每次执行函数的目的并不见得都是“执行步骤 1~3 以聚合接口”，而是调用 myFunc() 自身的功能。如何分别执行“内聚接口”与“函数功能”这个两部分呢？这可以使用如下的代码，以保证 myFunc() 在函数声明的同时就进行了接口聚合，而在函数执行期不会再重复这一（不必要）的过程：

```

myFunc = function() {
    function memberQuery() {
    }

    function _myFunc() {
        // 函数自身的功能代码
    }

    // 1. 声明(注册)聚合
    var intfs = Aggregate(_myFunc, IMyIntf);
    // 2. 取指定接口的一个引用
    var intf = intfs.getInterface(IMyIntf);
    // 3. 实现接口方法
    intf.memberQuery = memberQuery;

    // 传出函数引用
    return _myFunc;
}(); // <-- 这里只执行一次（相当于步骤 4），以完成接口内聚的相关代码步骤 1~3

```

最后需要说明的是，内部聚合 `Aggregate()` 这个工具函数只能针对函数使用，它的第一个参数必须是函数，也必然与后面使用 `QueryInterface()` 查询时使用的参数一样。内部聚合一批接口时，这些接口如何实现，以及由函数内部的哪些内嵌函数或对象方法来实现，都取决于用户自己的实现代码（上例中的步骤 3），对此 Qomo 的接口系统没有任何限制。

1.2.3.5. 接口与特性读写器

如果我们声明这样的接口：

```
INamedEnumerator = function() {  
    this.getLength =  
    this.items =  
    this.names = Abstract;  
}
```

那么它的实现就会与 Qomo 类的机制冲突，因为 Qomo 对名为 "getXXXXXXXXX" 的方法有限制。在 Qomo 的早期版本中，你必须这样写代码才能使它用于上面的 `INamedEnumerator` 接口：

```
function MyObject() {  
    this.getLength = function() {  
        // 这里的代码是为 Attribute 写的  
    }  
  
    this.Create = function() {  
        // 这里的代码才提供给接口，并通过访问 Attribute 才得到值  
        this.getLength = function() {  
            return this.get('Length');  
        }  
    }  
}
```

为此 Qomo 添加为接口系统添加了一些特性，使得这个过程变得非常简单。现在，对于接口来说，名字以 "get/set" 开始方法名将会被直接映射到 "对象.get()" 或 "对象.set()"。因此用户不需要为此多写任意一行代码。如下例：

```
IUserInfo = function() {  
    this.getName = Abstract;  
    this.getAge = Abstract;  
}  
  
function MyObject() {
```

```

    Attribute(this, 'Name', 'MyName');
    Attribute(this, 'Age', 30);
}
TMyObject = Class(TObject, 'MyObject', IUserInfo);

var obj = new MyObjet();
var intf = QueryInterface(obj, IUserInfo);

alert(intf.getName());
alert(intf.getAge());

```

但仍然有些用户希望能够使用自己实现的 `getLength()` 而不是 `get('Length')` 方法调用。那么用户在实例构造周期声明一下该方法即可。例如：

```

function MyObject() {
    //...
    this.Create = function() {
        this.getLength = function() { ... }
    }
}

```

或许用户的对象根本不是 Qomo 对象，而是普通的 JavaScript 对象——因此并没有 `getter/setter` 方法。接口系统能够自己识别这种情况，并将方法映射到一般对象方法上。因此你可也可给下面这样的对象注册这类接口：

```

// (部分代码略)
var obj = {
    getName: function() { ... },
    getAge: function() { ... }
}
Interface.RegisterInterface(obj, IUserInfo);

// 测试
var intf = QueryInterface(obj, IUserInfo);
alert(intf.getName());
alert(intf.getAge());

```

1.2.3.6. 为特定的对象注册接口

在 Qomo 的(目前的)发布版本中，对一些 JavaScript 原生对象未提供接口支持。例如对 `Array()` 对象。

这只是因为 Qomo 没有觉察到这样做的必要，因而没有加入相关的代码。

这在技术实现上其实非常简单。下面参考对 `MuEvent()` 的实现，介绍一下 `Array()` 的接口实现方法：

```
// 以下代码填写在 interface.js 中

// 1. 接口声明
IArray = function() {
    this.push = Abstract;
    this.pop = Abstract;
    // ...
}

// 2. Interface() 的实现代码中，wrapInterface() 函数声明之后加入代码
wrapInterface(IArray);

// 3. 在 isImplemented() 的实现代码中，修改 (添加) 如下代码
switch (intf) {
    // ...
    case IArray: return this instanceof Array;
}
```

除了这种直接修改 `Interface.js` 的方法之外，也可以利用 `Qomo` 的注册函数 `RegisterInterface()` 来为构造器注册接口——但在这种实现方法下，`QueryInterface()` 的效率会略低于上面的方法：

```
// 为 Array 注册接口 (IArray 的接口声明参见上例)
Interface.RegisterInterface(Array, IArray)

// 使用接口
arr = new Array();
intf = QueryInterface(arr, IArray);
```

1.2.4. 接口委托

现在来到 `Qomo` 接口系统中最复杂的一个部分：接口委托，它是在 `Qomo V2` 的早期版本才开始实现和应用的一项技术。因为接口委托的功能实在强大，当实现完了这个功能之后，原来的 `Aggregate()` 的实现代码就被废弃了：新的 `Aggregate()` 是直接接口委托的基础上实现的。

接口委托的含义是指：委托者已经注册了接口，但是它自身没有实现该接口的能力，因此委托给第三方来实现该接口（的方法）。因此委托总是发生在注册之后，并通过一个函数来实现。这个工具函数就是 `Delegate()`，调用格式如下：

```
Delegate(consigner, confer)
```

其中，consigner 是一个委托者，而 confer 是指一份委托协议。confer 我们稍后再讲，这里先讲讲委托者 consigner。

先来回顾一下，我们在下表中列出了所有 Qomo 系统中能显式或隐式地调用 RegisterInterface 的注册者：

类型	注册者	示例	注册示例代码	
函数	类类型	TObject	RegisterInterface(TObject, ...); RegisterInterface(_cls(), ...);	
	Qomo 类(构造器)	MyObject TMyObject.Create	Class(TObject, 'MyObject', ...); RegisterInterface(MyObject, ...); RegisterInterface(TMyObject.Create, ...); RegisterInterface(_cls().Create, ...);	*
	JavaScript 构造器	Object	RegisterInterface(Object, ...);	
	一般函数	foo = function() { ... }	Aggregate(foo, ...);	**
对象	Qomo 对象	obj = TMyObject.Create(); obj = new MyObject();	RegisterInterface(obj, ...);	
	JavaScript 对象	obj = new Object(); obj = {};	RegisterInterface(obj, ...);	

(*)指 TObject 及其子类。对构造器注册的接口，其子类 and 子类实例都将继承。

(**)一般函数也可以以普通对象身份注册接口，这里强调 Aggregate 在“内部聚合”上的特殊性。

这些注册者全部都可以用来做委托者，只要你能确保：

- 👉 在声明委托之前，该接口已经被注册到注册者；
- 👉 在声明委托时，你能持有注册者的一个引用。

当委托者是函数时，它可能存在二义性：一方面可以查询这个函数以得到接口实例；另一方面，如果函数是构造器或类类型，那么它的实例、子类以及子类实例都可以查询到该接口。当委托者是一个对象实例时，委托关系不具有上述的继承性。

1.2.4.1. 接口委托的基本方法

委托者总是提交出一份协议：confer。这份协议包括两个要素：受委托的第三方，与委托者接受的协议规则。这里的第三方——被委托者——可以是一

个对象、函数或者一批对象或函数；协议规则是一个字符串数组，数组中每一笔记录被称为一个协议项（**rule**）。

接下来我们先来看一个简单的接口委托示例，该示例中的委托者与被委托者都是对象，因此是一种比较简单的委托关系：

```
IMyIntf = function() {
  this.doAction1 =
  this.doAction2 = Abstract;
}

proxy = {
  doAction1: function() {},
  doAction2: function() {
    alert('hi, doAction2.');
```

在本示例中，委托者 **obj** 是一个空的对象，什么也没有实现，因此当完成步骤 1 之后，接口虽然被注册成功（也能被查询到接口实例），但不能使用。在步骤 2 中，声明了一个委托关系，被委托者 **proxy** 是一个普通对象，但它实现了方法 **doAction1()** 与 **doAction2()**。因此，第 3~4 步骤就可以正常使用了。需要留意的是第 4 步调用的 **doAction2()** 是 **proxy** 的方法。

但是这里仍然有问题，如果 **proxy** 只实现了一个方法，另一个方法由别的 **proxy** 实现了，又该怎么办呢？这需要用 **Delegate** 声明一份复杂一些的协议：

```
// (部分代码参见上例)
proxy2 = {
  doAction1: function() {
    alert('hi, doAction1.');
```

```

    }
}
// 2. 声明委托
Delegate(obj, [ // <- confer
    [proxy, ['IMyIntf.doAction2']],
    [proxy2, ['IMyIntf.doAction1']]
])
// 测试, 显示'hi, doAction1.'
intf = QueryInterface(obj, IMyIntf);
intf.doAction1();

```

这样，这里的 `doAction1()` 方法就由 `proxy2` 实现了，而方法 `doAction2()` 则仍由 `proxy` 来实现。

1.2.4.2. 接口委托中的代理函数

但是在实用中我们可能还会面临另一个问题：如果我们不能总是预先知道 `proxy` 对象的话，如何做这个委托呢？这需要使用到另外一种被委托者类型：函数。当被委托者是一个函数时，该函数总是接受一个入口参数，该参数指示当前的委托者对象（如果原始的委托者是构造器，那么这里参数指示的会是它的一个实例）；该函数总是返回一个对象，该对象是真实的受委托者。下例说明这种复杂的关系：

```

// (部分代码参见上例)
function MyProxyObj() {
    this.doAction1 = function() {
        alert('doAction1, by proxy_func');
    }
}

function proxy_func(obj) {
    return new MyProxyObj();
}

// 2. 声明委托
Delegate(obj, [ // <- confer
    [proxy, ['IMyIntf.doAction2']],
    [proxy_func, ['IMyIntf.doAction1']]
])
// 测试, 显示'doAction1, by proxy_func'
intf = QueryInterface(obj, IMyIntf);

```

```
intf.doAction1();
```

这样，doAction1()方法就是由 proxy_func()在运行期动态创建的一个对象实例来实现的了。在实际应用中，proxy_func()可以查询预先建立的表格，或者直接返回某些对象的成员等等，总之它只需要返回一个带有 doAction1()方法的对象即可。而且，事实上它也可以返回一个函数或者直接量，总之这个返回值只要能实现在委托中指定的方法名就行了。

1.2.4.3. 普通函数作委托者

上面我们讲述了 proxy 是函数和对象这两种情况，但是使用的委托者都是同样的：对象实例 obj。而我们在前面的表格所列出的委托者中，还包括函数。下面我们来讲述委托者是函数的情况。

当委托者是函数时，它首先总是表明自己是接口的委托者——因此查询时也应该使用该函数。例如：

```
function IMyFunc() {
    this.getArgsCount = Abstract;
}

function myFunc(v1, v2, v3) {
}

proxy = {
    getArgsCount: function() {
        return 0;
    }
}

// 1. 注册接口
RegisterInterface(myFunc, IMyFunc);

// 2. 声明委托
Delegate(myFunc, [
    [proxy, ['*IMyFunc']]
]);

// 测试，显示 0
intf = QueryInterface(myFunc, IMyFunc);
alert( intf.getArgsCount() );
```

这里应注意 `proxy` 是一个对象，因此调用该对象的 `getArgsCount()` 方法时，方法内能够访问的 `this` 引用将会是 `proxy` 对象本身。然而这与我们注册 `IMyFunc` 给 `myFunc()` 时的本意有些出入：我们是希望通过 `getArgsCount()` 来查询 `myFunc()` 声明时的形式参数数目的。如果我们要准确地实现这个功能，那么我们应该使用函数形式的 `proxy`，而这会使代码看起来变得复杂一些：

```
function proxy(obj) {
  return ({
    getArgsCount: function() {
      return obj.length; // <-- Function.length
    }
  });
}
// ...
// (声明、注册接口略)

// 测试，显示 3
intf = QueryInterface(myFunc, IMyFunc);
alert( intf.getArgsCount() );
```

在函数 `proxy` 中，返回的对象（直接量）持有了该函数形式参数 `obj` 的一个引用，而这个 `obj` 其实就是 `QueryInterface()` 时传入的 `myFunc()`，因此 `getArgsCount()` 就成了取该函数的形式参数个数。

1.2.4.4. 构造器函数作委托者

在委托者（例子中的 `myFunc()`）作一个构造器使用时，它具有另外一层含义：为它所有实例、子类以及子类实例注册该接口。这里用到了两个概念，一个是 JavaScript 中的构造器与其实例，另一个是 Qomo 类继承系统中的子类与其实例。`Delegate()` 的委托关系对这二者都是有效的。

这种情况的典型表现是：接口注册者与查询者不相同。如下例：

```
function MyObject() {
}
RegisterInterface(MyObject, IMyIntf);

obj = new MyObject();
intf = QueryInterface(obj, IMyIntf);
```

需要注意的是，通过 `Class()` 注册的接口都被隐式地注册到 “<类类型>.Create()”

这个构造器函数上，这与下例使用 `RegisterInterface()` 注册的效果是一致的^①：

```
RegisterInterface(TMyObject.Create, IMyIntf);
```

在这些情况下，构造器也可以声明一份委托，表明它所有子类及子类实例中的某个方法是由一个被委托者实现的。这使情况变得非常复杂：一个对象实例的接口中，一部分由该对象实现，一部分由其父类或构造器声明的委托者实现。然而，这正是我们需要的效果。下例说明这种复杂的技术：

```
function IMyIntf() {
    this.doAction = Abstract;
    this.getCount = Abstract;
}

var count = 0;
function MyObject() {
    count++; // 针对 getCount() 方法，测试用

    this.doAction = function() {
        alert('doAction.');
```

```
    }
}
```

```
TMyObject = Class(TObject, 'MyObject', IMyIntf);
```

```
// 一个返回对象实例的通用受委托者
```

```
function proxy_all(obj) {
    return obj;
}
```

```
// 一个受托者对象
```

```
proxy = {
    getCount: function() { return count }
}
```

```
// 1. 声明委托
```

```
Delegate(MyObject, [
    [proxy_all, ['*']],
    [proxy, ['IMyIntf.getCount']]
]);
```

```
// 声明子类，并注册类继承关系
```

```
function MyObjectEx() {
```

^① 这与注册在 `MyObject()` 构造器上也没有区别——如果该构造器已通过 `Class()` 注册给 Qomo 的类系统的话。

```

}
TMyObjectEx = Class(TMyObject, 'MyObjectEx');

// 2. 创建子类实例
obj = new MyObjectEx();
// 3. 查询接口
intf = QueryInterface(obj, IMyIntf);
// 4. 测试, 调用接口方法.
intf.doAction();
alert( intf.getCount() );

```

这在示例主要强调：在步骤 1 中，委托者是 `MyObject()` 构造器，而在步骤 3 中，查询者却是子类的实例 `obj`。

需要留意的一个细节是步骤 4 中我们调用接口的两个方法。其中，`doAction()` 是由对象实例自身实现的，根据协议，除 `'IMyIntf.getCount'` 之外的其它所有 (标识符 `"*"`) 方法都由代理函数 `proxy_all()` 实现，而该函数总是返回对象实例本身——也就是查询时传入的参数 `obj`。步骤 4 中的另外一个方法 `getCount()`，则总是由代理对象 `proxy` 来实现的。

1.2.4.5. 类类型作委托者

在 Qomo 中，类类型是不负责创建实例的，例如你不能用 `"new TObject()"` 来得到一个对象实例。因此类类型不为对象的继承性负责。但是，类类型自身是具有继承性的，例如 `TMyObject` 就继承自 `TObject`。基于这样的原因，如果你为某个基类类型注册了接口，那么它的所有子类类型也可以查询到该接口，但基类的或子类的实例都查询不到这个接口。

当类类型作委托者时，委托关系也对这种继承关系起作用。因此如果你已经为基类类型委托实现了一个接口，那么就可以在子类中使用。

在“1.2.4.3 类注册函数 `Class()` 的特殊语法与应用”中，我们讲过一种用注册 (`register`) 机制和匿名类类型的技术来实现的类工厂。但是从一个类工厂的理论模型（这里说的是纯理论概念上的 GoF 基本模式）上来讲，我们不必要实现得那么复杂。因为我们的类类型其实就是类工厂的创建者。

关于这种概念上的映射就不再细述了。对模式基础了解深刻的读者，可以接下来看看理论工厂模型在 Qomo 中如何实现：

```

// 声明创建者接口, 工厂方法名为 Create()
ICreator = function() {

```



```

    this.Create = Abstract;
}
// 注册给基类类型
RegisterInterface(TObject, ICreator);

// 产品接口声明
IMyProduct = function() {
    this.show = Abstract;
}

// 1. 类声明
function MyCreator() {
    this.show = function() {
        alert('a Product.');
```

这里最重要的是 TMyCreator 类被注册了哪些接口的问题。由于它继承自 TObject，所以也继承了 ICreator 接口。而 Class()函数又注册了 IMyProduct 接口，需要注意的是，IMyProduct 并没有注册在 TMyCreator 上，而是注册在 TMyCreator.Create 这个构造器方法上。所以，ICreator 是从类类型上查询的（步骤 3），而 IMyProduct 则是从类实例上查询的（步骤 5）。

对于本示例来说，用户代码是使用接口方法调用（步骤 5）还是使用对象方法调用（步骤 4），并没有明显区别。这仅取决于用户希望在什么样的体系下编写代码而已。

1.2.4.6. 委托协议

调用 `Delegate()` 函数时需要同时提交一份委托协议（`confer`）。该协议总是一个数组，结构如下：

```
[
  [proxy, [confer_item, ...]],
  [proxy, [confer_item, ...]],
  ...
];
```

如前所述，其中 `proxy` 可以是代理对象或代理对象。而 `confer_item` 总是一个字符串，呈如下格式：

```
[<*|+|->]InterfaceName[.MethodName[:AliasName]]
```

字符串应当以一个修饰字符为前缀，含义如下（如果缺省，则以“+”为修饰字符）：

- 👉 “*”: 实现指定接口，或所有接口的所有方法
- 👉 “+”: 实现指定接口，或接口的指定方法
- 👉 “-”: 不实现指定接口，或接口的指定方法

这三个修饰字符构成一个包含和排除的规则集。此外，修饰字符有两条优先规则，以表明委托关系中出现重复条目时的覆盖方法：

- 👉 排除匹配“-”优先于包含匹配“+”或“*”
- 👉 指定匹配“InterfaceName.MethodName”优先于通用匹配“*”

这个字符串其它的三个部分用于指定接口。包括：

- 👉 `InterfaceName`: 接口名。例如 `IInterface`
- 👉 `MethodName`: 方法名。例如 `IInterface.QueryInterface`
- 👉 `AliasName`: 别名，这是指实现者(proxy)中用来实现 `MethodName` 的名字

一个实现者 `proxy` 可以实现多条协议，也可以对实现者进行重复指定，系统会按优先级合并出一个最终的协议集。例如：

```
[
  [proxy, ['*IMyIntf', '*IJoPoints']],
  [proxy, ['-IMyIntf.method_2']]
];
```

这个协议中 `proxy` 就会实现 `IMyIntf` 中除开“`method_2`”之外的所有方法。但

这种排除也会带来一个问题：没有在协议中的方法由谁实现呢？答案是，默认由 **consigner** 自己来实现。不过你也可以在协议中显式地指定。这种情况下，你需要将一条 “*” 修饰符的规则放在尽可能靠前的协议项中。如同上例中的 “*IMyIntf”，它表明所有后续协议项中未描述的 IMyIntf 接口方法，都交由 proxy 实现。

但是如果 **consigner** 注册了三个接口，而协议只描述了其中两个接口（的部分或全部），那么如为其它的接口声明协议呢？这可以用到单个修饰符的 “*” 号，它表明所有的接口方法都由该 proxy 实现：

```
[
  [proxy, ['*']]
];
```

当修饰符作用于接口名（而不确指方法名时），它表明该协议项针对整个接口（的全部方法）。例如我们在上例中加一点小小的更改：

```
[
  [proxy, ['*', '-IMyIntf']]
];
```

这就表明是 “proxy 实现除 IMyIntf 之外的所有接口（的方法）”。

委托协议支持一个由 “:” 引导的别名。这项特性用户允许 proxy 通过一个不同的方法名来实现指定的接口方法。例如：

```
[
  [proxy, ['+IMyIntf.method_1:doAction']]
];
```

这项协议就明表明最终是由 proxy.doAction 来实现 IMyIntf.method_1；如果 proxy 是一个函数，则由函数返回对象的 doAction() 方法来实现它。

最后，对于支持继承的委托者（包括类类型与构造器）来说，协议也是有继承性的。因此我们可以在子类，或子类构造器中重订协议中的一部分协议项。例如：

```
18 // 接口声明
19 IMyIntf = function() {
20   this.doAction1 =
21   this.doAction2 = Abstract;
22 }
23 // 父类
24 function MyObject() {
25   this.doAction1 = function() { alert('doAction1') };

```

```

26     this.doAction2 = function() { alert('doAction2') };
27     proxy_all = function(obj) { return obj };
28
29     Delegate(_cls().Create, [
30         [proxy_all, ['IMyIntf']]
31     ]);
32 }
33 // 子类
34 function MyObjectEx(){
35     Delegate(_cls().Create, [
36         [{
37             doAction2: function() { alert('doAction2 - MyObjectEx') }
38         }, ['IMyIntf.doAction2']]
39     ]);
40 }
41 // 注册。子类会继承父类的接口，因此不必为 MyObjectEx 重复注册
42 TMyObject = Class(TObject, 'MyObject', IMyIntf);
43 TMyObjectEx = Class(TMyObject, 'MyObjectEx');
44
45 // 测试代码
46 obj1 = new MyObject();
47 obj2 = new MyObjectEx();
48
49 intf1 = QueryInterface(obj1, IMyIntf);
50 intf2 = QueryInterface(obj2, IMyIntf);
51 // 显示'doAction2'
52 intf1.doAction2();
53 // 'doAction2 使用子类的协议，显示'doAction2 - MyObjectEx'
54 intf2.doAction2();
55 // doAction1 仍然使用父类的协议，显示'doAction1'
56 intf2.doAction1();

```

在这个例子中，父类 `MyObject` 委托了一份协议，表明所有“`IMyIntf`”的方法都由 `proxy_all()`函数返回的对象实例来实现。而 `proxy_all()`总是返回传入的 `obj` 参数，这表明在运行期是由一个具体的实例来实现的。这里使用这个技巧的原因是：在类声明周期，我们无法为某个确切的实例来声明委托。不过，在本例中使用 `proxy_all()`只是为了让读者明白这个机制，在实用中，我们可以应用另一条规则：未被委托协议的接口方法由对象实例 `consigner` 自己（或它的实例）来实现。因此将行 10~14 删除，并不会影响本例的执行效果。

在子类 `MyObjectEx` 中，18~22 行委托了一份新的协议，该协议只声明了

一个协议项：由一个直接量对象来实现 `IMyIntf` 接口的 `doAction2` 方法。这项协议覆盖了父类协议的一个部分。因此在第 37 行的输出中，就调用了这个新协议中的方法；而 39 行则仍然调用旧协议中的（未被覆盖的）方法。

1.2.4.7. 委托方法中的 `this` 引用

在下面的示例代码中：

```
obj = {}
proxy = {
  doAction1: function() { ... }
}

RegisterInterface(obj, IMyIntf);
Delegate(obj, [
  [proxy, ['+IMyIntf.method_1:doAction']]
]);

// 测试
intf = QueryInterface(obj, IMyIntf);
intf.doAction1();
```

我们有一个问题：在 `doAction1()` 方法被调用时，传入的 `this` 引用到底会是 `obj` 呢？还是 `proxy`？

对于委托方法 `proxy.doAction1()` 来说，Qomo 总是为它传入 `proxy` 作为 `this` 引用。这是因为在委托过程中，我们并不知道 `obj` 对 `doAction1()` 的执行过程是否有意义，也不能预测 `doAction1()` 在不以 `proxy` 为 `this` 引用时的执行效果。但正是由于这一项策略，对于使用对象类型的 `proxy` 来说，是无法知道 `obj` 对象的——没有任何位置供我们通知 `proxy.doAction1()`，让它知道它被 `obj` 对象委托执行。

对于函数类型的 `proxy` 来说，函数总是返回一个对象（例如 `proxy_obj`），该对象的方法在被调用时，传入的 `this` 引用仍然是 `proxy_obj`。这与上面的规则一致。但是，函数类型的 `proxy` 会接受一个传入的参数，它是运行期的委托者对象 `obj`。因此，`proxy` 函数有机会为 `proxy_obj` 指定 `this` 引用，或者直接返回 `obj` 来替代 `proxy_obj`——我们在“1.4.4.3 普通函数作委托者”的 `getArgsCount()` 方法中使用了前一种技巧，上一小节中的 `proxy_all()` 则使用了后一种技巧，读者可以再自行回顾一下。

1.2.5.Qomo 接口系统的实现

本小节分析的代码，若未特殊标注，则来自于 Qomo 代码表中的 Interface.js 单元。

1.2.5.1. 接口系统的实现框架

接口系统基本上由三个函数构成。其中函数 Interface()占了代码的 2/3，其它两个是工具函数 Delegate()与 Aggregate()。

函数 Interface()的主要代码结构如下：

```
Interface = function() {
    var handle = '_INTFHANDLE_';
    var $Intfs = { length:1 }; // skip zero
    var $Deles = [];
    var $Delei = [];

    function _Interface(obj) { /* Intf1 .. Intfn */
        // ...
    }

    _Interface.QueryInterface = function(obj, intf) {
        // ...
    }

    _Interface.RegisterInterface = _Interface;
    _Interface.IsInterface = isInterface;
    return _Interface;
}();
RegisterInterface = Interface.RegisterInterface;
```

所以 RegisterInterface()与 QueryInterface()都实现在同一个匿名函数的内部，该匿名函数返回 _Interface()给外部的全局变量 Interface()。在系统全局，RegisterInterface()与 Interface()其实指向同一个函数引用。

除了这两个函数之外，Interface()外公布给外部系统一个方法 IsInterface()，用以验证接口声明。在 Qomo 系统的 SysUtils.js 单元中，该方法被赋给全局函数 IsInterface()，可以作为公共的工具函数使用。

接口系统中的另外两个工具函数 Delegate()与 Aggregate()都依赖接口系统

的 RegisterInterface()实现。其中 Delegate()的基本框架代码是：

```
Delegate = function() {  
    // ...  
  
    // interface delegation.  
    function _Delegate(consigner, confer) {  
        Interface.RegisterInterface.call(consigner, new Tbl(confer));  
    }  
    return _Delegate;  
}();
```

这表明 Delegate()的实质是向接口系统中注册一份协议（confer）。而 Aggregate()的基本框架代码是：

```
Aggregate = function() {  
    // ...  
  
    // interface aggregation.  
    function _Aggregate(foo) { /* Intfl .. Intfn */  
        if (foo instanceof Function) {  
            Interface.RegisterInterface.apply(_Aggregate, arguments);  
  
            var confer = [], ff = new Interfaces(arguments, confer);  
            Delegate(foo, confer);  
            return ff;  
        }  
    }  
    return _Aggregate;  
}();
```

这表明 Aggregate()的实质是将接口注册给指定函数（foo），并通过一份协议（confer）将这些接口委托(Delegate)给一个内部创建的 Interfaces 对象实现。

1.2.5.2. 接口注册的实现

Qomo 中的接口注册实现得比较简单，因为从本质上来说，所谓接口注册，就是构造一张对照表，表明“某某对象 / 函数实现了某某接口”。这个过程由 _Interface()完成，其内部调用的关键函数是 warpInterface()。

warpInterface()是一个重要的函数，它的作用是把一个接口声明函数变成一个真正的接口，并注册到一张内部的查询表中。这张查询表在上面的实现框

架中，声明为一个局部变量：**\$Intfs** 对象。该对象模拟了数组，因此具有 **length** 属性——为避免成员 0 被布尔运算视为 **false**，**length** 属性是从 1 开始计数的。

所谓“将接口声明函数变成真正的接口”，是指接口声明只是一个普通的构造器函数，而 **warpInterface()** 为它分配了一个惟一标识句柄。该句柄是由 **\$Intfs.length** 来作递增序分配的。**warpInterface()** 保证接口声明得到惟一句柄，并添加为接口的一个成员。这个成员的名字也声明在上面的实现框架中：

```
var handle = '_INTFHANDLE_';
```

以 **IInterface** 为例，在调用 **warpInterface()** 之前，它只是一个普通的函数声明。而在调用 **warpInterface()** 之后，下面的代码将显示 **true**：

```
alert( '_INTFHANDLE_' in IInterface );
```

而且“**IInterface._INTFHANDLE_**”一定返回一个大于 0 的数值，即是 **warpInterface()** 分配给该接口的句柄。

\$Intfs 通过句柄构建起一个类似于数组的表，元素 **\$Intfs[handle]** 总是指向一个接口本身。这行代码在 **warpInterface()** 中：

```
// intf 是当前接口，h 是该接口的句柄。第一次注册该接口时，将创建一个只包含元素 intf 的数组。  
$Intfs[h] = [intf];
```

而所谓注册，即是指将实现者对象添加到该数组的末端。这段代码在 **_Interface()** 中：

```
// 通过 warpInterface() 封装接口，并返回所有接口注册者的表  
var all = warpInterface(args[i]); // all[0] equ intf  
// 在表的末端添加当前注册者 obj  
if (indexOf.apply(all, [obj]) == -1) all.push(obj);。
```

缺省情况下 **Qomo** 的接口系统注册了四个基本接口：

```
warpInterface(IInterface);  
warpInterface(IJoPoint);  
warpInterface(IMuEvent);  
warpInterface(IJoPoints);
```

这些接口没有注册任何的实现者，而是由接口系统自动识别的。那么为什么要单独注册它们呢？对于 **IInterface**，该接口是接口系统固有的，并实现在 **QueryInterface()** 内部，所以应当由本单元自注册。而其它三个接口，则是因为它们被实现在 **JSEnhance.js** 中，该单元被设计为不依赖于其它 **Qomo** 模块，因此不适宜于在 **JSEnhance.js** 中调用 **RegisterInterface()**。

_Interface() 中有一部分代码是留给外部工具函数扩展的。目前它只用于

Delegate()函数，因此将留到“1.4.5.4 聚合与委托的实现”中去讲述。

1.2.5.3. 接口查询的实现

接口查询的实现非常复杂，这是因为它概括了接口的所有能力，包括继承、委托等等复杂机制，都依赖于接口查询来完成。

接口的实现代码中共分为 6 个关键步骤。我们先讲述前 5 个步骤的实现，对于第 6 个步骤，我们先简化到没有“接口委托”机制的情况下来讲述。其框架代码是：

```
_Interface.QueryInterface = function(obj, intf) {  
    // 1. 检查参数  
    if (obj===undefined || obj===null) throw new Error(EQueryObjectInvalid);  
    if (!isInterface(intf)) throw new Error(EInterfaceNotExist);  
  
    if (typeof obj == 'object') {  
        // 2. 接口实现自身  
        if (obj instanceof intf) return obj;  
        // 3. 查询对象obj是一个接口实例(a vtbl)  
        if (isVtbl(obj)) return obj.QueryInterface(intf);  
    }  
  
    // 4. IInterface 接口总是被实现的  
    if (intf===IInterface) {  
        return ({  
            constructor: IInterface,  
            QueryInterface: function(f) { return _Interface.QueryInterface(obj, f) }  
        })  
    }  
  
    // 5. 检测如果查询对象obj没有实现指定接口，则返回undefined  
    if (!isImplemented.call(obj, intf)) {  
        return void null;  
    }  
  
    // 6. 创建一个接口实例，并将接口方法指向对象方法  
    var c = [], f = new intf();  
    var confers = ...; // 获取委托协议  
    if (confers) { // delegated interfaces  
        // ...  
    }  
}
```

```

    }
    else { // implemented by obj
        for (var n in f) c.push(bySelf(n, obj))
    }
    eval(c.join('\n'));

    // 接口总是存在 QueryInterface() 接口
    f.QueryInterface = function(intf) {
        return Interface.QueryInterface(obj, intf)
    };

    // 返回接口实例
    return f;
}

```

在步骤 1 中，除了保证 `undefined` 和 `null` 没有接口注册之外，还检测被查询的 `intf` 是否是一个真正的接口——是否被 `warpInterface()` 分配过有效的句柄。

步骤 2 和 3 用于分析查询对象 `obj` 本身是一个接口实例的情况。当它是被查询的 `intf` 的一个实例时，它返回自身；否则将调用它的 `QueryInterface()` 方法，重新向接口内部的“原始对象”查询——由于一个接口总是步骤 4 或 6 创建并返回的，因此它总有一个 `QueryInterface()` 方法，并且总能够该问最初查询并获取接口时的原始对象 `obj`。

步骤 4 快速地返回一个（伪造的）`IInterface` 接口实例。它使用了一个小技巧：它模拟“`new IInterface()`”用直接量方式声明了一个接口实例——这种声明法只能欺骗内部的 `isVtbl()` 函数。

步骤 5 中的 `isImplemented()` 是一个关键函数。整个接口查询依赖于它来检测查询者、注册者、委托者和实现者等等多个系统元素之间建立的关系。它的确是一个复杂的系统，但主要是识别查询者是函数亦或普通对象。对于前者，`isImplemented()` 会检测它是否是类类型——因为类类型可以继承父类类型的接口；对于后者，`isImplemented()` 直接检测实例的构造器、父类的构造器等等来查询继承的接口。

但是步骤 5 只负责检测有或没有实现接口而并不具体实现它，当查询对象 `obj` 没有注册或通过继承得到指定接口，那么步骤 5 就返回 `undefined`。否则进入步骤 6。

在不讨论聚合与委托这样的复杂关键的情况下，步骤 6 是非常简单的，它只是简单的列举接口对象的所有成员——就是那些被声明为 `Astract()` 的方法，

然而调用 `bySelf()` 为该方法的名字生成一个字符串并随后用 `eval()` 动态执行。一般来说，这个字符串的格式很简单（生成字符串的方法参见“1.7.2.1 连接槽：`joinSlot()`”）：

```
f["<methodName>"] = function() {  
    return obj["<methodName>"].apply(obj, arguments)  
}
```

由于变量 `f` 总是指向该接口实例。因此上面的代码就使得这些接口实例的方法直接调用一个匿名函数，该函数直接调用 `obj` 对象的指定方法——并使用当前传入的参数。注意这里没有使用委托，因此这些方法调用时，通过 `apply()` 传入的 `this` 引用仍然是 `obj` 自身^①。

在函数 `bySelf()` 中会识别当前的接口方法是否是一个“`getXXX`”或“`setXXX`”形式的读写器方法。如果是，那么对这种接口方法（参见“1.4.3.5 接口与特性读写器”），`bySelf()` 返回的字符串会略为复杂一些：

```
f["<methodName>"] = function() {  
    return obj.get("xxx") // 或 obj.set("xxx", arguments[0])  
}
```

`bySelf()` 会识别这里的方法名并填写合适的代码。

1.2.5.4. 聚合与委托的实现

Qomo 中存在两种特殊的接口注册方式：`Aggregate` 和 `Delegate`。从根本来说，两种方式可以归为 `Delegate`（委托）这一种。而接口委托依赖于一份“委托协议(`confer`)”，它声明时的结构如下：

```
[  
    [proxy, [confer_item, ...]],  
    [proxy, [confer_item, ...]],  
    ...  
];
```

在 `Delegate()` 函数中，它被转换为一份内部的委托协议，然后交给 `RegisterInterface()` 去注册：

```
function _Delegate(consigner, confer) {  
    Interface.RegisterInterface.call(consigner, new Tbl(confer));  
}
```

^① 应该时时记得“闭包”在函数式语言中对作用域的影响：这段代码是通过 `eval()` 执行得到的，它总是位于 `QueryInterface()` 这个函数的闭包中，因此也总是可以访问函数参数表中的 `obj` 变量。

构造器函数 `Tbl()` 用于实现这个转换过程，它在内部调用 `cnvTbl()` 函数将转换后的结果存放在对象实例（`this`）的各个成员中。这些成员包括：

```
this['@'] = tbl;
this['#'] = { length:1 }; // skip zero
this['*'] = { '*': [] };
this['+'] = { };
this['-'] = { };
```

其中 '@' 成员存放原始协议 `tbl` 的一个备份，而 '#' 成员则创建一个类似数组的对象——这种技巧在讲述接口注册中的 `$Intfs` 时就讲述过了，用以存放所有的实现者 `proxy`。之所以用 '@'、'#' 等等这些特殊的成员名，并没有什么特别的原因，只是方便后面使用：例如依赖协议项前面（`confer_item`）的修饰字符来存取指定成员。

函数 `cnvTbl()` 的转换过程比较复杂，最终会使得 `Tbl()` 的实例（`this` 引用）中的各个成员具有如下结构：

```
thisRef = {
  '@': tbl,
  '#': { ... },

  '*': {
    '*': [],
    '<interface_handle>': {
      '*': []
    },
    ...
  },

  '+': {
    '<interface_handle>': {
      '+': [],
      '<method_name>': [],
      ...
    },
    ...
  },

  '-': {
    // 与成员 '+' 类似
  }
}
```

这样的结构便于我们将来在 `QueryInterface()` 中做检索。试做如下的存取：

```
// 所有接口的公共实现者
thisRef['*']

// 指定 interface_handle 接口的实现者
thisRef['*'][interface_handle]['*']

// 指定不实现 interface_handle 接口的
thisRef['-'][interface_handle]['-']

// 指定实现 interface_handle.method_name 的
thisRef['+'][interface_handle][method_name]
```

这些存取都返回一个数组，数组中的成员是实现者 `proxy` 在 `thisRef['#']` 中的索引，如果 `proxy` 使用别名来实现指定方法，则在下一个成员位置补写了一个方法名(方法名为数值的问题)。

`Delegate()` 所做的工作仅仅是这些，随后它将内部格式的协议交给 `RegisterInterface()`。`RegisterInterface()` 为外部工具函数留备了一种扩展的方法：

```
function _Interface(obj) { /* Intf1 .. Intfn */
    // ...
    switch (_Interface.caller) {
        case Delegate: if (this!==Delegate) {
            ...
        }
    }
}
```

在这里，`RegisterInterface()` 检测调用栈，以及复杂 `this` 引用是否由外部函数用 `apply()` 传入，以确定是否是来自工具函数的调用——这里使用 `switch` 语句以便于扩展其它的外部函数。当确定是来自 `Delegate()` 的调用后，`RegisterInterface()` 检测内部的委托表 `$Deles` 与 `$Delei`，并决定是创建一份新协议，亦或是将协议合并到既有协议之上。

现在，当用户代码使用 `QueryInterface()` 查询对象 `obj` 是否具有接口 `intf` 时。`QueryInterface()` 就查询该对象（也可能是函数）以及它的父类（构造器或类类型）在 `$Deles` 中登记过的所有协议。如果有协议，则使用协议化的方式来实现接口实例，否则按上一小节中所述的简单方式实现它。

由于检索协议时将包括父类中注册的协议——可能不止一份，因此下面的代码返回的是一份协议清单：

```
var confers = (obj instanceof Function ?
    (IsClass(obj) ? getConfersClass : getConfers) :
    getConfersWithClass)(obj);
```

这里用到了三个函数：

getConfers () ：检查对象自己注册的协议

getConfersClass () ：当对象是类类型，检查它以及其父代类类型上注册的协议

getConfersWithClass () ：检查对象自己，以及其父代类类型上注册的协议

这些函数以运算元的形式参与表达式运算并返回，最后执行函数调用运算“()”并传入参数 obj^①。

QueryInterface()中实现的另一个关键函数是 idByName()^②，它按照指定接口、指定方法在协议中检索是否有协议化的实现者 proxy。当它找到 proxy 时，将会检查 proxy 的类型：function 或 object（或其它），并分别调用 byProxyFoo() 与 byProxyObj() 来获取一个动态执行的字符串；当没有 proxy 时，该动态字符串仍缺省使用 bySelf() 来获取。

当 proxy 类型不是 function 时，该字符串格式为：

```
f["<methodName>"] = function() {  
    var p2=confers["<idxConfers>"]["#"]["<idxProxy>"];  
    return p2["<methodName>"].apply(p2, arguments)  
}
```

当 proxy 类型是 function 时，该字符串格式为：

```
f["<methodName>"] = function() {  
    var p2=confers["<idxConfers>"]["#"]["<idxProxy>"](obj);  
    return p2["<methodName>"].apply(p2, arguments)  
}
```

最后，当 methodName 为“getXXX”或“setXXX”形式的读写器方法时，该字符串的格式也会复杂得多：

```
f["<methodName>"] = function() {  
    var p2=confers["<idxConfers>"]["#"]["<idxProxy>"](obj);  
    return (!("<methodName>" in p2) && ("get" in p2) ? // 或("set" in p2)  
        p2.get("XXX") : // 或 p2.set("XXX", arguments[0])  
        p2.<methodName>.apply(p2, arguments))  
}
```

可见，除了为接口方法的实现代码所做的包装的不同之外，在委托协议上的查询与一般的接口查询的实现过程（或结果）并没有多大的差异。

最后我们简述一下接口聚合的实现。在 Aggregate()的内部存在一个名为

^① 你也许会把它看成一种复杂的技巧，但事实上它只是一种典型的函数式语法的应用而已。

^② 这个是一个非常复杂的、使用了大量连续表达式运算的函数。为了用简洁的代码实现复杂的委托协议，我牺牲了代码的可读性，这是一个并不友好的选择。

`Interfaces()`的构造器，我们将 `Aggregate()`的关系委托给该构造器的实例，并注册到接口系统中：对 `Aggregate()`所聚合的接口列表构造了一个代理对象，并为该代理对象（自动地）创建了一份协议，最后注册它。

不过，我们也为这稍稍地扩展了一下 `Delegate()`：允许在 `confer_item` 中直接使用 `InterfaceHandle`——也就是接口的内部句柄来替代 `InterfaceName`。因此，在委托协议的协议项（`confer_item`）中：

```
[<*|+|->]InterfaceHandle[.MethodName[:AliasName]]
```

也是合法的——不过目前这只被用在 `Interface.js` 内部，用于 `Aggregate` 的实现。

1.3. 命名空间

尽管 `Qomo` 花费了异常巨大的精力用于实现命名空间系统，但 `Qomo` 并不认可命名空间在以浏览器为主体的 `JavaScript` 环境中的应用价值。反过来说，正是因为这种不认可，所以 `Qomo` 竭尽全力想要抹平有与没有命名空间时期的编程差异，以使得用户在不开销更多代码的情况下在两种系统中自然过渡。

1.3.1. `Qomo` 的命名空间的复杂性

在其它的编程系统中，命名空间是系统或编译器一级实现的功能，一般来说会同时实现一个“缺省命名空间”，以及命名空间域的“标识符覆盖规则”。因此它不必总是使用命名空间的全名——至少在当前单元中的标识符引用不需要。但是在 `JavaScript` 中，由于引擎缺省并不支持命名空间，因此需要用对象与对象成员来模拟。一般性的代码如下：

```
// 根空间
Qomo = {}
// 更多的命名空间
Qomo.RTL = {}
Qomo.RTL.System = {};
Qomo.Classes = {};
```

用这种方法创建出复杂的命名空间系统来之后，你就会看到 `JavaScript` 代码中有大量的、无可忍受的重复代码了：

```
Qomo.Classes.List.HashedList.TStrongHashList = Qomo.System.Class(
  Qomo.Classes.TObject,
  'StrongHashList',
  Qomo.Interface.List.IStrongHashList,
```

```
Qomo.Interface.List.IList);
```

而这行代码的本意不过是：

```
TStrongHashList = Class(TObject, 'StrongHashList', IStrongHashList, IList);
```

命名空间带来了对不同子系统的隔离，然而由于 JavaScript 的语言限制，这些有限的隔离性也带来了诸多的不便。Qomo 对此的建议是：推荐在小型系统中不使用命名空间，在大型系统中有限度地使用命名空间。

为此，Qomo 的命名空间与装载系统配合起来，使用路径识别技术将单元文件的位置与它的命名空间联系起来，自动地构建整个命名空间系统。例如：

```
{Qomo}\
├─Framework
│   └─RTL
│   └─Compat
│   └─Common
...
```

当我们把 Framework 注册为“Qomo.System”空间时，根据目录结构关系，我们可以自动推断出如下命名空间：

```
Qomo.System.RTL
Qomo.System.Compat
Qomo.System.Common
```

如果 Common 目录中的某些单元使用 Class()注册了类类型，我们也就可以认为这些类注册在了 Qomo.System.Common 命名空间上，例如：

```
Qomo.System.Common.THttpGetMachine
Qomo.System.Common.TTimeline
...
```

但是这些命名空间上的类是用如下的方法：

```
THttpGetMachine = Class(TObject, 'HttpGetMachine');
TTimeline = Class(TTimeMachine, 'Timeline');
..
```

来注册的，因此我们任意选用下面的方式之一来使用它：

```
obj = Qomo.System.Common.THttpGetMachine.Create();
obj = THttpGetMachine.Create();
obj = new HttpGetMachine();
...
```

对于用户代码来说，上述三种（或更多的）使用方式之间没有差异。用户使用哪一种形式，只取决于他现在在构建怎样的系统，以及需要对哪些子系统进行

隔离。

然而除了要建立一个命名空间系统之外，我们也必须关注其应用环境的一些限制，例如在浏览器环境中的路径问题。具体的说，在浏览器环境中 Qomo 必须面临四种不同的路径系统，包括：

```
docBase : 这是指在网页自身的 url 地址，可能是一个本地地址(例如 file:///c:/...)
srcBase : 这是指在网页中用<script src=...>的形式来装载 Qomo 时的"src"属性值
curScript : 当 Qomo 通过$import() 装载一个单元之后，curScript 表明该单元的路径
activeJS : 在多个<script>标签装载脚本时，activeJS 指向当前正在装载的脚本路径
```

在这些路径中，通常情况是用户使用下面的代码来装载 Qomo.js，以加载 Qomo 系统：

```
<!-- 示例: 当前网页中使用 Qomo -->

<!-- 装入 Qomo 系统 -->
<script src="js/qomo/Qomo.js"></script>

<!-- 用户的页面内脚本代码 -->
<script>
    $import("js/3rd/a Script.js");
</script>
```

在这个例子中，Qomo.js 加载所有 Qomo 中的单元并将它们初始化到预定义的命名空间系统中去。在 Qomo 中存在的问题会是这样：

```
// 1. in Qomo.js
$import('Framework/system.js')

// 2. in system.js
$import('RTL/Object.js')
```

当我们在 Qomo.js 中装载 Framework/system.js 时，使用了相对路径，而加载系统必须使用基于当前网页地址（docBase）的路径来获取文件，也就是说路径应当被这样计算出来：

```
docBase + srcBase + 'Framework/system.js'
docBase ==> <root>/
srcBase ==> "js/qomo/"
```

因此应当是：

```
<root>/js/qomo/Framework/system.js
```

接下来进行对 Object.js 文件的相对路径计算时，地址计算要依赖于“当前正在装载的脚本的路径（curScript）”，该值是 system.js 的路径：

```
<root>/js/qomo/Framework/
```

因此重新的计算结果是：

```
<root>/js/qomo/Framework/RTL/system.js
```

这只是 Qomo 处理的路径系统中极少的一部分。更加细节的问题包括：

- 👉 用户使用更加复杂的相对定位，例如：`../../abc/../../myScript.js`；
- 👉 用户可能在网页中使用 `$import()` 装载文件，例如上例中的 `js/3rd/aScript.js`；
- 👉 用户指定的路径名可能跨域，例如 IE 中的本地文件系统 (`file:///` 协议)；
- 👉 需要考虑不同浏览器环境对 url 的限制，例如 opera 的 `file://localhost/` 协议；
- 👉 一些浏览器无法侦测当前正在用 `<script>` 加载的脚本，`activeJS` 路径无效；
- 👉

Qomo 将这些问题屏蔽在命名空间系统之后，当用户装载一个脚本之后，可以以该脚本为相对路径创建它的命名空间，而所有在该脚本中加载的类都会自动的、按照路径系统的关系添加到命名空间。所有的不同路径关系最终被简化为以两套：

- 👉 以 `location.href` 为基点的、由 `$import()` 使用的装载路径；
- 👉 以 Qomo 根命名空间，或用户指定命名空间为起点的虚拟路径。

前者由加载系统使用，后者由命名空间维护——用户代码无需关注。

1.3.2.命名空间的使用

Qomo 提供一组工具函数让用户创建与维护命名空间系统。这些工具函数包括：

- 👉 `isNamespace(n)`：检测名字 `n` 是否是一个有效的命名空间；
- 👉 `$map(n, p, r)`：映射一个命名空间 (`name`) 到一个物理路径 (`path`)，如果传入参数 `r`，则 `path` 是基于 `r` 运算的相对路径；
- 👉 `$mapx(n)`：将命名空间字符串 (`name`) 展开成完整的命名空间；
- 👉 `$n2p(n)`：取命名空间所映射到的物理路径；
- 👉 `$p2n(p)`：取路径所对应的命名空间 (`name`)；

1.3.2.1. 空间创建

Qomo 的空间创建需要使用到两个函数 `$map()` 和 `$mapx()`。

`$map()`用于映射命名空间 (`name`)与物理路径 (`path`)的关系。当参数 0 为字符串时，它表示创建命名空间：

```
// 创建一个虚的命名空间
$map('Qomo.System', '');

// 创建一个命名空间并映射路径
$map('Qomo.System', '/qomo/system/');

// 创建一个命名空间并映射路径(使用当前脚本所在路径的相对路径)
$map('Qomo.System', 'system/');

// 创建一个命名空间，命名空间的中间部分(Core.Mem.MemProf)将是虚的
$map('Qomo.System.Core.Mem.MemProf.FieldTest', 'system/');
```

当参数 0 为命名空间时，它用于重新指定 `path`，即重置命名空间：

```
// 重设命名空间的物理路径
$map(Qomo.System, 'system/');
```

当参数 2 为路径（字符串）时，表明在创建或重置命名空间时使用相对路径：

```
// 重设命名空间的物理路径
$map(Qomo.System, 'system/', $n2p(Qomo.Component));
```

`$mapx()`则用于将命名空间字符串(`n`)展开成完整的命名空间，与`$map()`不同的是，用`$mapx()`展开的每一个命名空间(或子空间)都不是虚的：他们指向一个经过运算得到的路径。`$mapx()`只接受一个字符串参数：前半部分是一个既已存在的命名空间（不能是虚的），后半部分是待扩展的。例如：

```
$mapx('Qomo.System.Core.Mem.MemProf.Field');
```

假设 `Qomo.System` 指向真实路径 `“/qomo/system/”`，那么`$mapx()`将扩展出以下空间（后面表面它映射的真实路径）：

```
Qomo.System.Core --> /qomo/system/Core/
Qomo.System.Core.Mem --> /qomo/system/Core/Mem/
Qomo.System.Core.Mem.MemProf --> /qomo/system/Core/Mem/MemProf/
Qomo.System.Core.Mem.MemProf.Field--> /qomo/system/Core/Mem/MemProf/Field/
```

`Qomo` 内部系统创建空间的具体过程如下。

首先，使用使用 `$map()`函数，`Qomo` 在系统中注册了一个根空间 `Qomo`，并将空间映射到 `Qomo.js` 装载的物理路径上：

```
$map('Qomo', './', function() {
    return $import.get('docBase') + $import.get('srcBase');
```

```
}());
```

其中的 `docBase` 是网页的路径，而 `srcBase` 则是 `Qomo.js` 相对于网页的路径。因此这里为 `Qomo` 注册的根空间的路径其实就是：

```
<docBase>/<srcBase>./
```

这个位置是指向 `Qomo.js` 以网页地址（URL 或 URI）为基础运算的相对路径。

一般来说，`Qomo` 的代码表在安装位置上的目录结构为：

```
Qomo\  
├─Components  
│   ├──Graphics  
│   ├──LocalDB  
│   ├──Controls  
│   └─DomCompat  
└─Framework  
    ├──RTL  
    └─Common
```

如果直接按照目录来“自动”创建空间的话，RTL 中的 `TObject` 将被注册到下面这样的命名空间上：

```
Qomo.Framework.RTL.TObject
```

但这并不是我们想要的结果。因此 `Qomo` 并没有在注册根空间后直接使用“自动”创建来构建整个命名空间，而是添加了一些次级根空间。例如：

```
$map('Qomo.System', 'Framework/', $n2p(Qomo));  
$map('Qomo.UI', 'Components/', $n2p(Qomo));
```

应当注意这里的 `$n2p()` 调用，它的作用是取指定命名空间的路径。因此以 `$n2p()` 为参考的相对路径“`Framework/`”与“`Components/`”就变成了：

```
<docBase>/<srcBase>./Framework/  
<docBase>/<srcBase>./Components/
```

这与我们上述的目录结构就一致了。

接下来 `Qomo` 使用 `$mapx()` 函数将命名空间展开，完成了对所有子目录的命名空间注册：

```
$mapx('Qomo.System.Common');  
$mapx('Qomo.System.RTL');  
$mapx('Qomo.UI.Graphics');  
$mapx('Qomo.UI.Controls');
```

这些空间的路径都自动映射到了 `Framework` 和 `Components` 的各个子目录中。

然而还是有些小问题。按照这样的映射，LocalDB 将会被注册到：

```
Qomo.UI.LocalDB
```

这样的空间上，这也不是我们想要的。我们试图将它注册在下面的空间中：

```
Qomo.DB.LocalDB
```

注意这里的 Components 目录已经被注册给 Qomo.UI 了，我们不可能把它注册给另一个名字。因此我们给了 Qomo.DB 一个“虚空间”——不能映射为任何有效的物理路径的空间：

```
$map('Qomo.DB', '');
```

然后再把 LocalDB 注册给它：

```
$map('Qomo.DB.LocalDB', 'Components/LocalDB/', $n2p(Qomo));
```

现在 Qomo 的命名空间就创建完了，它与路径系统的对应关系如下：

命名空间	路径系统	备注
Qomo	Qomo/	
Qomo.System	Qomo/Framework/	
Qomo.System.RTL	Qomo/Framework/RTL/	
Qomo.System.Common	Qomo/Framework/Common/	
Qomo.UI	Qomo/Components/	
Qomo.UI.Graphics	Qomo/Components/Graphics/	
Qomo.UI.Controls	Qomo/Components/Controls/	
Qomo.DB	<虚空间>	
Qomo.DB.LocalDB	Qomo/Components/LocalDB/	
Qomo.Thirdparty	<虚空间>	(*)

(*)该命名空间是 Qomo 系统内部注册的，任何不能通过路径影射注册的类，都将被注册在这里。

1.3.2.2. 将类注册到命名空间

当命名空间被创建之后，用户代码可以将任何类注册到任意空间下，Qomo 没有对此做任何限制。因此，你事实上可以将 TObject 注册到根空间下——或者别的任何空间下，如：

```
// 将 TObject 注册在 Qomo 根空间下
Qomo.TObject = TObject;
```

但是大多数情况下，用户不需要手工地将类类型注册到命名空间。在使用 Class()注册类时，Class()会将类自动地注册到当前所在的命名空间上——当前命名空间(activeSpc)是通过分析装入该代码的脚本文件对路径而得到的，这也是 Qomo 映射路径与命名空间的关键。这段代码如下：

```
// code in Framework/RTL/Object(6).js
var activeSpc = $import_getter('activeSpc');
if (activeSpc) {
    var spc = activeSpc();
    cls.SpaceName = spc.toString();
    spc[cls.ClassName] = cls;
}
```

所以在使用命名空间系统之后，类类型会有一个 `SpaceName` 成员存放所在空间的字符串；而该空间也会存放该类的一个引用。

1.3.2.3. 查询与转换

`$n2p()`与`$p2n()`两个函数可以用于命名空间与路径的互查与转换。例如：

```
// 返回 (字符串): '/qomo/system/Core/Mem/MemProf/FieldTest/'
$n2p(Qomo.System.Core.Mem.MemProf.FieldTest);

// 返回 (命名空间对象): Qomo.System.Core.Mem.MemProf.Field
$p2n('/qomo/system/Core/Mem/MemProf/Field/');
```

对于`$n2p()`来说，如果命名空间是虚空间，则返回空字符串，否则返回该空间映射的路径。对于`$p2n()`说，如果查询空字符串，且 `Qomo` 根空间被映射为空地址^①，那么就返回 `Qomo` 根空间，否则返回能查找到的第一个虚空间。

1.3.3.命名空间的实现

在 JavaScript 中实现命名空间是很简单的技术，一般直接使用对象声明就可以了。但是由于 `Qomo` 设定了更高的目标——让命名空间与宿主路径系统之间无缝衔接，因此实现较为复杂。

如前所述，由于`$import()`牵涉到具体宿主环境对文件系统（以及网络地址系统）的理解，因此我们不具体讲述它的机制。下面的章节，我们主要讨论当一个路径系统已经被建立起来，那么它如何被映射到命名空间系统上呢？关于路径系统的建立，在 `Qomo` 中是由以下模块来完成的：

```
// 1. 地址协议分析模块
Framework\RTL\Protocol.js

// 2. 路径转换方法

// 以下声明于 Framework\RTL\system.js，并在多个模块中被重写
```

^① 当 `Qomo.js` 与装载它的网页位于同一目录时，`Qomo` 根空间的地址为空字符串。

```
transitionUrl ()
parseRelativeURL ()
```

其转换结果，是将所有加载的模块路径转换为如下标准形式：

```
../Framework/system.js
../Framework/Debug/Debug.js
../Framework/Common/ParserLite.js
../Components/Controls/HtmlUtils.js
...

// (当系统配置为支持 IE 的绝对路径时，存在以下路径格式)
/Qomo/Framework/Compat/private_ie2.js
/Qomo/Framework/Compat/common_js16.js
/Qomo/Framework/RTL/Interface.js
...
```

实现命名空间与路径的简单映射是使用对象成员来实现对照表。这里涉及到两个方法，其一是用：

```
aObject[<fullPath>] = aNamespace;
```

来构建“路径—命名空间”的映射；其二是用：

```
aArray[index] = [aNamespace, fullPath];
```

来构建从“命名空间—路径”的映射。前者的问题在于对象的内部成员（例如 `constructor` 等）与 `fullPath` 可能存在冲突；后者的复杂性在于 `aNamespace` 本身是一个对象，在 JavaScript 中对象无法用作（数组或对象的）成员名检索，因此只能使用数组索引检索，而这是非常低效的方法。

Qomo 使用 `fullPath` 的长度来建立一个快速的 `hash` 表，以解决第一个问题。这个表被称为路点（`signpost`），内部对象名称为 `map`。对象的数字成员存放对应长度的路径与空间的对照表。例如下面两个路径：

```
/Qomo/Framework/Compat/
/Qomo/Component/Graphi/
```

其它都保存在 `map[23]` 所指向对象中。该对象有两个数组类型的成员：

```
$map$[23] = {
  paths: [fullPath_1, fullPath_2, ...],
  names: [namespace_1, namespace_2, ...]
}
```

当我们试图查找的 `fullPath` 并不在指定长度的 `map[fullPath.length].paths` 中时，Qomo 会尝试缩短路径，例如将路径从：

```
/Qomo/Component/Graphi/Vector/VML/
```

变为：

```
/Qomo/Component/Graphi/Vector/
```

并再次在 `map` 中查找。假设在缩短到：

```
/Qomo/Component/
```

这个路径时，能够查找到空间名 `Qomo.UI`，那么 `Qomo` 也就能根据系统中“路径与空间对应”这样的关系，确定最初查找的路径所对应的命名空间是：

```
Qomo.UI.Graphi.Vector.VML
```

因此类似 `Qomo.UI` 这种“最短的、可映射到路径系统的命名空间”就被称为路点（signpost），这些路点极大减少了在复杂系统中需要建立大型的对照表的麻烦。也正是通过 signpost，以及使用 `fullPath.length` 为 hash-key 的检索技术，`Qomo` 得以实现相对高效的“路径—命名空间”检索系统。

另一方面，`Qomo` 在每个命名空间对象的 `constructor` 成员中保存了它对应的路径^①，同时重写了 `toString()` 使之返回命名空间的字符串形式。通过这两种简单的技巧，`Qomo` 对“命名空间—路径”以及“命名空间—命名空间字符串”的转换也非常快捷。

1.4.AOP

“面向切面编程（AOP，Aspect Oriented Programming）程序设计框架的应用是有一定争议的，它是否需要在 JavaScript 中实现一直以来也是个问题。因此我们需要先知道，滥用 AOP 的特性将导致系统效率下降、性能不稳定等后果。因此 `Qomo` 拥有了强大的 AOP 框架，但如果你不足够了解它，那么还是慎用之。

`Qomo` 早期的 `Interface` 框架其实是很简单的，后来因为两次技术需求而得以发展。一次是在包含第三方框架代码是所需要的接口委托，另一次就是因为 AOP 框架所需要的内部聚合。`Qomo` 正是因此而发展出强大的 `Interface` 机制。但这并不是说用户需要定义很多接口，才能使用 AOP——接口是在 `Qomo` 实现 AOP 中的“定制切面”时使用到的关键技术，而不是用户使用 AOP 时所必须的技术。

`Qomo` 的 AOP 框架依赖于 `Qomo` 中提供的如下特性：

^① 我并不建议重写对象（以及其它数据）的 `constructor` 属性，或系统内部成员。但这里的命名空间是一个完全独立的、不参与运算的子系统，命名空间在理论上只有根路径可能会对其它子系统造成影响——`Qomo` 在重写 `constructor` 时已经考虑到了这些负面因素。不过最终在这里使用了不推荐技巧的原因，还是效率的问题。

- 👉 接口机制: Interface.js
- 👉 JSEnhance 中的事件多投: MuEvent()
- 👉 Qomo 的 OOP 框架: Object.js

1.4.1.基本概念与语法

一个关键的名词是"切面(Aspect)", Aspect 被译作“方面”、“切面”和“剖面”都是有的, 请不要追究这个用词。它首先是基于 OOP 体系的一个概念, 切面描述的是对“一组”对象实例的共同行为能力的“一个关注”。也就是说: 如果你希望了解一些对象(无论它们是否是同一父类 / 基类)的一些相类同的行为, 那么你可以将这些行为发生的“位置”理解成一个“切面”。而 AOP 就是一套针对这个“切面”进行编程的框架。

另一个比较学术的名词是“不知觉性(obliviousness)”, 这是 AOP 的特性之一。它要求加入一段 AOP 的代码对原有系统不会产生可察觉的影响。

其它一些 AOP 中的几个关键术语包括:

- 👉 联接点(join point): 程序的结构或者执行流中定义好的“位置”。Qomo 中简写为 JoPoint。
- 👉 通知(advice): 在联接点上会发生的一种行为, 这种行为能力是 AOP 框架来提供的。
- 👉 编织(weaving): 将核心功能与方面组合在一起, 以“产生一个(基于 AOP 的)工作系统的过程”。
- 👉 周围、之前与之后(around, before and after): 联接点上(常见的)三种"通知(advice)"能力。需要强调的是: around 通知可能改变原有系统的行为, 这可能使得“不知觉性”被破坏或者产生歧义。

Qomo 中用到的几个名词 / 术语:

- 👉 观察者与被观察者(observer/observable): 一个切面中, 观察者是切面(aspect), 被观察者是切面(当前)拦截到的对象。
- 👉 切点(pointcut): 与“联接点(join point)”对应, 切点是对这个“联接点位置”的一个描述。AspectJ 中使用“切点原语(一种表达式)”来描述 pointcut, 而 Qomo 使用一个表示名字的字符串。
- 👉 元数据(metadata): 在处理切面或执行切面代码时所需要的一些数据。这可以是用户在建立切面时初始的任何数据, 甚至是用于获取数据的函数回调。
- 👉 引导(Introduction): Qomo 中的一种切面事件, 发生在 before 通知之前, 可以决定切面的行为是否需要发生——是否需要拦截并发出通知。

一个经常被提到的“切面”是“(记录一些对象行为的)日志系统”。而在 Qomo 中，AOP 被用来作为实现 JavaScript Profiler 的基础技术。

1.4.1.1. 切面对象与切点

Qomo 中，Aspect 是一个标准的 Qomo 对象。也就是说，Qomo 中存在 TAspect 类及其子类。这包括：

```
TAspect
- TFunctionAspect
- TClassAspect
- TObjectAspect
- TCustomAspect
```

其中 TAspect 是一个抽象基类，因此你不应当创建它的实例^①。可以使用标准 JavaScript 语法，或 Qomo 的“类类型.Create()”语法来创建切面，例如：

```
var asp = new ObjectAspect();
```

这样的一个切面对象总是具有如下接口：

```
IAAspect = function() {
  this.supported = Abstract;
  this.assign = Abstract;
  this.unassign = Abstract;
  this.merge = Abstract;
  this.unmerge = Abstract;
  this.combine = Abstract;
  this.uncombine = Abstract;

  this.OnIntroduction = Abstract;
  this.OnBefore = Abstract;
  this.OnAfter = Abstract;
  this.OnAround = Abstract;
}
```

其中 supported() 方法用于检测切面能否支持 (support) 某种切点 (pointcut)。切点名是一个字符串，例如 “Function” 或 “Method”。Qomo 对切面与切点之间的支持关系约定如下：

```
support pointcut:
  for TFunctionAspect : 'Function'
  for TClassAspect   : 'Method'
```

^① 不过，并不绝对是这样。有些情况下你仍然可以这样做，例如后文的“1.6.2.5 匿名函数的切面”。

```
for TObjectAspect : 'Method', 'Event', 'AttrGetter', 'AttrSettter'
for TCustomAspect : <可以通过用户代码为被观察者定制切点>
```

下面的示例用于检测一个切面是否能切入某种切点：

```
var asp = new ObjectAspect();

// 显示 true, 对象切面支持'AttrGetter' 切点
alert( asp.supported('AttrGetter') );
// 显示 false, 对象切面不支持'Function' 切点
alert( asp.supported('Function') );
```

1.4.1.2. 将切面关联到被观察者

切面要被关联到一个或一些具体的被观察者(observable)才会有意义。这通过 assign() 方法来实现：

```
a_Aspect.assign(aObservable, '<name>', '<pointcut>');
```

不同的切面（a_Aaspect）能观察的目标并不一样——在内部实现上，一个具体的被观察者被称为切面的宿主（Host）——对于不同的被观察对象，host、name 和 pointcut 的含义不尽相同。详情如下：

切面	<host>	<name>	<pointcut>
TFunctionAspect	函数	函数名	'Function'
TClassAspect	类类型	该类实例(原型)的方法名	'Method'
TObjectAspect	对象实例	方法 / 事件 / 特性名	'Method', 'Event', 'AttrGetter', 'AttrSettter'

下面的示例创建一个对象切面，它关联到对象的一个特性读方法上：

```
function MyObject() {
    Attribute(this, 'Value', 10);
}
TMyObject = Class(TObject, 'MyObject');

// 创建切面并关联到被观察者
var obj = new MyObject();
var asp = new ObjectAspect();
asp.assign(obj, 'Value', 'AttrGetter');
```

切面也可以在创建时就关联到被观察者，这些切面的构造器的参数与 assign() 是一样的。因此上面最后两行代码其实等效于下面这一行：

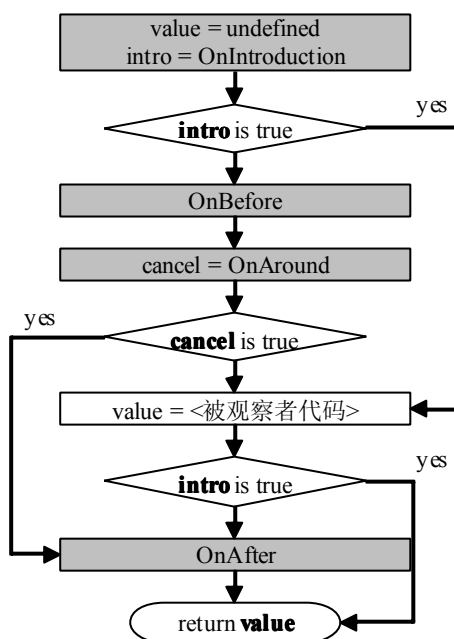
```
var asp = new ObjectAspect(obj, 'Value', 'AttrGetter');
```

1.4.1.3. 切面行为

Qomo 在切面上有四个行为，都是通过事件来响应的，包括：

```
this.OnIntroduction = Abstract;  
this.OnBefore = Abstract;  
this.OnAfter = Abstract;  
this.OnAround = Abstract;
```

其中 OnAround、OnBefore 与 OnAfter 是三种通知（advice），而 OnIntroduction 是一种框架决策。OnIntroduction 表明本切面可以决定是否响应此次拦截，如果 OnIntroduction 返回 false 值，那么所关注的被观察者仍然执行原有的代码，但切面上的三种通知都不被触发。OnAround 通知可以改变切面上的行为，如果 OnAround 返回 false，那么被观察者的代码不被执行。OnAfter 与 OnAfter 发生在被观察者代码执行的前后，但既不能改变传入给被观察者代码执行的参数，也不能改变被观察者代码返回的结果值。这一过程的基本流程如下图所示：



（图：切面上的基本流程与逻辑）

由于切面上的四个行为都被定义为 Qomo 中的多投事件，因此用户代码不应当通过赋值，而应当通过多投事件的 add()、addMethod()方法来响应事件。仍以上一节的代码为例，如果用户代码需要观察读取到的 obj 的特性值，那么应当使用如下代码：

```
// (续上例)

// 添加切面响应句柄(切面上的行为)
asp.OnAfter.add(function(o, n, p, a, v) {
    alert('In aspect, ' + n + ': ' + v);
});

// 示例, 尝试读取特性, 以激活该切面
obj.get('Value');
```

切面方法上的响应事件句柄的形式参数 (o,n,p,a,v) 所表示的实际含义是:
(observable, aspectname, pointcut, args, value)

一般简写成 (o,n,p,a,v) 这样的形式——这些参数用户也不一定会全部用到。另外, 除了 OnAfter() 之外, 其它的事件句柄中当不包括参数 value。很显然, 在那些切面行为中, 用户代码还没有执行, 因此返回值 value 是无意义的。

1.4.2. 高级切面特性

切面是用来观察对象行为了, JavaScript 中具有行为能力的主要是函数和方法。所以你会看到上述的切点都是面向(类或对象的)函数或方法的。另外, 由于 Qomo 中还存在特性读写器, 因此在对象的切点上, 还存在 AttrGetter 和 AttrSetter。

需要注意的是, 不能在切面例程中, 调用受影响的被切方法/特性。例如在一个名为 'Name' 的 'attrGetter' 切面中, 调用 observable.get('Name'), 或者在函数 'run' 的 'Method' 切面中, 调用 observable.run()——很显然, 这些都将导致锁死的递归。

Qomo 的 AOP 系统可用于 Qomo 的 OOP 系统之外的其它对象与函数。尽管 Qomo 的 AOP 依赖 OOP 和 Interface, 但对第三方系统来说, 仍然不难从中分离出一个非 Qomo 的 OOP 实现的继承关系的 AOP——当然, 如果要实现定制切面 (CustomAspect), 仍然是需要完整的 Interface 特性的。

下面讲述切面的高级特性, 以及应用中的一些问题和应对方案。

1.4.2.1. 定制连结点与定制切面

首先我们要注意到的是: 在 JavaScript 中, 具有行为能力的不单单是函数

和方法。JavaScript 的函数式语言特性使得它存在一种“函数内嵌函数”这样的行为能力。例如在下面的代码中：

```
MyFunc = function() {  
    // 1. 一些 MyFunc() 中的逻辑代码  
    var func_01 = function() {  
        alert('hi, func_01');  
    }  
  
    var func_02 = function() {  
        alert('hi, func_01');  
    }  
  
    // 2. 实现 MyFunc()  
    function _MyFunc() {  
        func_01();  
        func_02();  
    }  
  
    // 3. 返回 MyFunc()  
    return _MyFunc;  
}();
```

我们如何观察内嵌函数 func_01() 和 func_02() 的行为呢？

相较于其它 AOP 系统，Qomo 在这方面提供了更强大的特性。Qomo 允许开发人员在“当前函数中”为外部系统定制连结点。这看起来与 AOP 系统的“不知不觉性”有些背离，但也可能是实现这种机制的唯一方法——除非 JavaScript 解释器内部提供等同的功能，或者单独编写外部的语法分析器。

Qomo 中这个“定制连结点”的机制要求“observable 有能力告知外部系统自己可提供的连接点(Join Point)”，但是当这些连接点被 AOP 系统接入(或切入)时，observable 却是“不知不觉”的。这种能为被 Qomo 分解成两个部分：

- 👉 Qomo 提供一组工具，来使 observable 可以产生连结点，并在连结点上产生通知；
- 👉 observable 应当将这些连结点通过一个 IJoPoints 接口向外抛出。

因此 Qomo 要求 MyFunc() 在实现中添加一些代码来暴露它的连结点。这用到了三种技术：

- 👉 连接点 (Join Points)：产生可供外部使用的连结点。对外部代码它表现为 pointcut；

- 👉 编织(weaving): 使连结点与目标的内部的“位置(或位置上的方法)”发生关系;
- 👉 聚合(Aggregate): Qomo 使用“(内部的)聚合”来暴露一个实体内部的接口。

下面的代码演示如何在上面的 MyFunc()中定制连结点:

```
MyFunc = function() {  
    // CustomAOP_1: 创建连接点  
    var _joinpoints_ = new JoPoints();  
    _joinpoints_.add('step1'); // 'step1'切点(pointcut)  
    _joinpoints_.add('step2'); // 'step2'切点(pointcut)  
  
    // CustomAOP_2: 编织(或织入)  
    // 1. 对 MyFunc() 中的逻辑代码  
    var func_01 = _joinpoints_.weaving('step1', function() {  
        alert('hi, func_01');  
    });  
  
    var func_02 = _joinpoints_.weaving('step2', function() {  
        alert('hi, func_02');  
    });  
  
    // 2. 实现 MyFunc()  
    function _MyFunc() {  
        func_01();  
        func_02();  
    }  
  
    // CustomAOP_3: 聚合 IJoPoints 接口  
    var _Intfs = Aggregate(_MyFunc, IJoPoints);  
    var intf = _Intfs.GetInterface(IJoPoints);  
    intf.getLength = function() { return _joinpoints_.length }  
    intf.items = function(i) { return _joinpoints_.items(i) }  
    intf.names = function(i) { if (!isNaN(i)) return _joinpoints_[i] }  
  
    // 3. 返回 MyFunc()  
    return _MyFunc;  
}();
```

我们看到,在这个例子中,步骤 CustomAOP_1~3 对 MyFunc()的程序原有结构并没有太大的变化。最关键的地方,是 _MyFunc()、func_01()和 func_02()内部实现代码并没有变化。

接下来,我们来创建切面,并书写有关切面上的行为的代码。亦即是测试

MyFunc():

```
// 创建切面, 这里的'test_aspect'可以是任意字符串
var asp = new CustomAspect(MyFunc, 'test_aspect', 'step1');
asp.OnAfter.add(function() {
    alert('do OnAfter');
});

// 测试
MyFunc();
```

测试结果输出为:

```
hi, func_01
do OnAfter
hi, func_02
```

这表现切面 asp 已经成功地切入 func_01, 并在它执行完之后、func_02()执行之前调用到了 asp.OnAfter()。

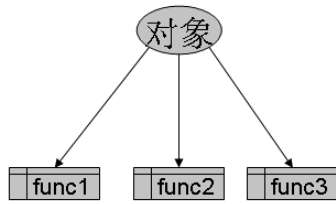
1.4.2.2. 切面的合并(merge)和联合(combine)

Qomo 中的切面有四种被关注者: 类、对象、函数和定制连接点的函数, 但是要知道 AOP 的本意是不区分这些被关注者的类型的。那么, 如何才能使一个切面能够处理更复杂的 observable 呢?

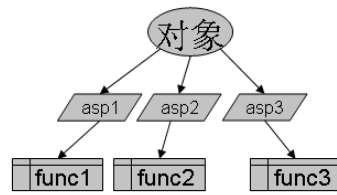
Qomo 为此提出了切面的合并和联合这两个概念。其中, 合并是指切面 A 将切面 B 的行为加到自身, 使 A 拥有 B 的关注能力, 但不改变 B 的能力。而联合, 则是指切面 A 和其它切面(B,C,D, ...)的行为联合在一起, 作为 A~D(或者更多)共有的关注能力。

下图说明这两种技术的不同:

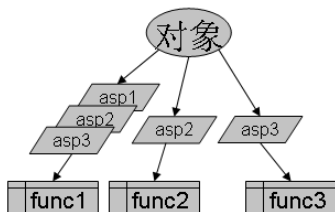
图一：对象对方法的调用



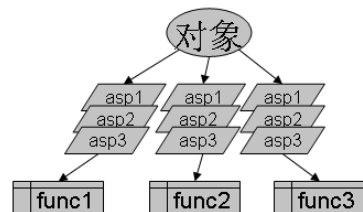
图二：对三个方法创建切面



图三：asp1.merge(asp2,asp3)



图四：asp1.combine(asp2,asp3)



如果我们要记录一批目标的执行(例如做 log 系统)，那么下面的 Aspect() 代码可能是一个不错的示例：

```

function MyObjectEx() { }
function MyObject () {
  this.getValue = function () {
    return 100;
  }
  this.run = function() {
    alert(this.get('Value'));
  }
}
TMyObject = Class(TObject, 'MyObject');

var obj = new MyObject();
var A1 = new ObjectAspect(obj, 'Value', 'AttrGetter');
var A2 = new ClassAspect(TMyObject, 'run', 'Method');
var A3 = new CustomAspect(Class, 'a_custom_aspect', 'Initializtion');
var A4 = new FunctionAspect(Ajax, 'Ajax', 'Function');

A1.OnBefore.add(function(o, n, p, a) {
  alert('Before: ' + n);
});

A2.OnAfter.add(function(o, n, p, a, v) {
  alert('After: ' + n);
});
  
```

```
});

// 测试
A1.combine(A2, A3, A4);
TMyObjectEx = Class(TMyObject, 'MyObjectEx');
obj.run();
```

1.4.2.3. 切面行为间的数据传递

切面的四个行为的响应句柄之间，可能存在自身的一套逻辑，例如我们用切面来做 profiler，就需要在 `OnBefore` 中记录开始时间，在 `OnAfter` 中用结束时间减去开始时间。那么这个“开始时间”如何传递给 `OnAfter` 以用于计算呢？

Qomo 的 AOP 中允许切面创建或关联切面到被观察者时传出一些元数据（meta data），这些元数据可以在同一个切面的不同方法与事件之间传递。对于上面这个问题，可以将“开始时间”暂存在元数据中。元数据总是一个数组，可能为零个至仍然多个元素，这取决于切面创建或关联切面到被观察者时传入的数据个数。元数据是切面对象的一个特性，因此你可以用 `aAspect.get()` 去取得元数据数组并操作它，也可以重写该特性。下面以 profiler 为例^①，说明如何使用这项功能：

```
function myFunc() {
    // ...
}

// 创建切面，并传入第一个元数据
asp = new FunctionAspect(myFunc, 'myFunc', 'Function', new Date());
asp.OnBefore.add(function(o, n, p, a) {
    // 向 MetaData 数组中压入当前时间值
    this.get('MetaData').push(new Date());
});
asp.OnAfter.add(function(o, n, p, a, v) {
    // 取 MetaData 数组
    var data = this.get('MetaData'), now = new Date();
    var str = 'aspect: %sms; exec: %sms';
    // 显示切面时间，以及 myFunc() 执行的时间
    alert(str.format(now-data[0], now-data[1]));
});
```

^① Qomo 中有单独的 Profiler 工具类，这里的 profiler 只是表明实现原理的一个简单示例。

```
// 测试, 执行 myFunc();  
myFunc();
```

我们在创建 `FunctionAspect` 的同时创建了一个日期对象作为第四个参数——如果有, 包括其后的任意多个参数——作为元数据, 在 `OnBefore` 中我们取出元数据并添加当前时间, 因此到 `OnAfter` 时, 数组中会有两个元素: 切面创建时间与被观察开始执行的时间。

有些时候, 我们可能不愿意记住创建时到底有多少个元数据被初始化, 也或者不愿意在代码中用 `get()` 方法来访问 `MetaData`。那么也可以直接操作事件响应句柄中的传入参数 (`o, n, p, a`), 这其中比较理想的是使用第四个参数 `args`, 它是用户代码调用时传入的参数——亦即是上例中 `myFunc()` 中的 `arguments` 对象。`arguments` 是一个普通对象, 因此也可以为它添加方法、定制成员, 而且它在四个切面行为中都被传递, 所以也可以用来携带数据。

更加不错的还有两点。其一, AOP 框架在调用被观察者——例如 `myFunc()` 时会使用这样的代码:

```
myFunc.apply(this, arguments);
```

这个时候 `arguments` 被传递到 `myFunc()` 内部时的只是一份拷贝, 而且不再带有多余的成员——也就是说, 四个切面行为对 `arguments` 进行任何修改都是 `myFunc()` 不可见的。其二, 由于 `arguments` 对象是由 JavaScript 在函数内部管理的一个对象, 因此它的生存周期是由 JavaScript 内部机制决定的。换言之, 不用担心它的创建、销毁等生存周期问题。

所以, 在不使用 `MetaData` 的情况下 (当然也不考虑 `Aspect` 创建的时间), 上例可以改成:

```
// (参见上例, 部分代码略)  
asp.OnBefore.add(function(o, n, p, args) {  
    args['exec'] = new Date();  
});  
asp.OnAfter.add(function(o, n, p, args, v) {  
    var now = new Date();  
    alert(now - args['exec']);  
});
```

1.4.2.4. 切面对目标系统的影响

在四个切面行为中传递 `arguments` 也带了一个问题, 那就是: 参数可修改。

举例来说，下面的代码中：

```
function myFunc(value) {
    alert(value);
}

var asp = new FunctionAspect(myFunc, 'myFunc', 'Function');
asp.OnBefore.add(function(o, n, p, args) {
    args[0] = 'aspect changed!';
});

// 下面代码将输出在切面中被修改后的值'aspect changed!'
myFunc(100);
```

同样的问题也会出现在返回值上。如果返回值不是值类型数据，而是一个对象，那么它就能被切面修改——对象在函数参数中是以引用形式传递的。例如：

```
function myFunc() {
    return [1,2];
}

var asp = new FunctionAspect(myFunc, 'myFunc', 'Function');
asp.OnAfter.add(function(o, n, p, a, value) {
    value.pop();
});

// myFunc() 返回值被切面修改，因此下面输出只显示数组的一个成员：数字值 1
alert( myFunc() );
```

切面对可能目标系统的输入输出产生影响，这游离于切面的“不知不觉性 (obliviousness)”的边缘：一方面它的确构成了对目标系统的影响，另一方面目标系统的确对这种影响没有反应。然而如果从更泛义的目标系统——就是包括调用被观察对象的整个系统来说，参数或返回值的不确定变化正是灾难之源。因此，Qomo 建议用户代码尽可能规避这些问题。如果用户真的希望利用这种特性来构建新系统或影响既有系统，那么应当对 AOP 概念，以及 Qomo 中的切面系统有充分的认识才可以。

1.4.2.5. 匿名函数的切面

Qomo 的切面需要知道被观察者的标识符或名字，例如函数名、方法名或

特性名。对于方法、特性、事件来说，这当然不成问题，但对于函数来说却很困难：因为有的函数是匿名的。

当一个函数匿名，但显式地存在一个标识符引用时，它仍然是可以直接使用 `FunctionAspect()`切面的^①。例如：

```
// 下面的代码声明了一个匿名函数，并赋值给一个变量(标识符)
myFunc = function() { ... }

// 仍然可以使用函数切面对象来观察它
asp = new FunctionAspect(myFunc, 'myFunc', 'Function');
```

但另外一些情况下，匿名函数声明之后就立即被使用了。例如：

```
e = new MuEvent();
e.add(function(){ //<-这里有一个声明后立即使用的匿名函数
    ...
});
```

那么我们如何对这个匿名函数使用切面呢？

我们可以使用特殊的 `Aspect()`对象来观察这个函数。这可能是在 `Qomo` 中的一种最小化构造切面的方法，这个过程的关键在于 `TApect.supported()`是一个抽象方法，因此我们不能使用创建时关联被观察者的方法，而只能使用 `assign()`方法——并在使用前重写 `supported()`。示例如下：

```
// 创建切面对象，并重写 supported() 使它支持任意切点
asp = new Aspect();
asp.supported = function() { return true };
asp.OnBefore.add(function(o, n, p, a) {
    alert('in aspect...');
});

// 示例：在多投事件列表中对匿名函数添加切面
e = new MuEvent();
e.add(asp.assign(function() {
    // ...
}));

// 测试输出 'in aspect...'
e();
```

^① 不过这个切面只能针对于指定的标识符，而不能应用到一个函数的多个引用上。这其中的原因，在于切面系统事实上会重写该变量，而重写变量标识符是不会影响到变量引用的。关于这一点，请参见“1.4.3 对象成员的重写”。

应当留意，在 `asp.assign()` 方法中，我们只传入了第一个参数，也就是 `host`。这是 `Aspect()` 对象特殊之处：它并不强制检测 `name` 参数，而我们通过重写 `supported()` 方法使得 `pointcut` 参数可以是任意值。所以这里就不必传入这两个参数了。而 `assign()` 方法会返回被切面系统封装过的函数^①，所以添加到事件投递列表中的就将是一个被切面观察的、匿名的函数了。

1.4.2.6. 函数名对切面系统的影响

AOP 的基础是函数或方法重写，但重写依赖于函数名，并且与对象引用是有关系的。而切面系统并不动态地检测函数或方法名，因此当这些名字上出现了重写，那么切面就对新系统没有效力了。举例来说^②：

```
var foo = function() {  
  var t = new Date();  
  foo = function() { //<- 重写全局变量，或父函数闭包中的名字 foo  
    return t;  
  }  
  return foo();  
}
```

这个模式类似于创建了一个单例 `t`。它与我们常用的：

```
var foo2 = function() {  
  // ...  
  function foo2 () { //<-- 声明的 foo2 是局部变量  
    // ...  
  }  
  return foo2;  
}();
```

区别在于后者是声明后立即执行，而前者是在第一次执行时产生重写效果。

但正是因为有了“第一次执行重写”这样的概念，所以 AOP 系统不能有效地观察到后面的代码执行过程。仍以上面的 `foo()` 函数为例：

```
// (续上例)  
asp = new FunctionAspect(foo, 'foo', 'Function');  
asp.OnBefore.add(function(o, n, p, a) {  
  alert('in aspect...');
```

^① 这看起来有点象定制切面中的 `weaving`。的确如此，因为在早期的切面系统中的 `assign()` 是没有返回值的，后来考虑到处理匿名函数的切面，才参考 `weaving` 添加了这项功能。

^② 这个例子来自于 <http://peter.michaux.ca/>，作者称之为“Lazy Function Definition Pattern”。其实就是在第一次运行函数时重写函数自身的技术。

```
});

// 以下两次执行，只能观察到第一次
foo();
foo();
```

如果的确试图观察这样的函数，那么可以在 `OnAfter` 再重新 `assign()` 一次。不过这不见得是绝对有效的法子，因为用户代码并不知道 `foo()` 函数第几次执行后会重写，或者是否会重写。

另外一个与函数名相关的问题是递归。我们通常的递归写法是：

```
function myFunc() {
  // ...
  return myFunc(); // <-- 注意这里以函数名来递归调用
}
```

由于这里是用函数名来递归调用的，因此如果我们为它添加一个切面：

```
asp = new FunctionAspect(myFunc, 'myFunc', 'Function');
```

那么在递归中的每次调用都会被切面观察到。解决这个问题方法，是在递归中不使用名称，而使用函数引用调用。例如：

```
function myFunc() {
  // ...
  return arguments.callee(); // <-- 注意这里以函数引用来递归调用
}
```

1.4.3.Qomo 中切面系统的实现

尽管 AOP 非常复杂、强大，但是核心技术却非常简单。AOP 在核心层面主要就是一个通知与响应的逻辑（参见“1.6.1.3 切面行为”）：

```
var intro = OnIntroduction() !== false;
if (intro) OnBefore();

var cancel = intro && OnAround() === false;
var value = cancel ? undefined : call_observable_method_or_more();

if (intro) OnAfter();
```

这样的核心逻辑被实现在工具函数 `JoPoints()` 和 `Aspect.js` 的 `$Aspect()` 函数中：

```
function $Aspect(pointcut, foo) {
```

```

var _aspect = this;
var host = _aspect.get('AspectHost');
var name = _aspect.get('AspectName');

// AOP 的核心
return function($A) {
    if ($A===GetHandle) return foo;

    // (略, 大致与上述的 AOP 基本逻辑相同)
    return _value;
}
}

```

\$Aspect()中暂存了 `_aspect`, `host`, `name` 等引用, 供核心部分安全地调用。另外核心部分的代码也可以通过 `upvalue` 访问到 `$Aspect()` 的参数 `foo`, 亦即是 `Aspect()` 对象所关注的方法。这既用于核心逻辑部分调用, 也用于 `unassign()` 中还原被关注者。

Qomo 的 `Aspect()` 对象实现的另一处关键在于 `assign()` 和 `unassign()` 方法。`assign()` 方法内部调用 `$Aspect()` 并用返回的函数改写 `AspectHost[AspectName]`。因此, 当 `unassign()` 时就只需要通过下面的代码:

```
host[name] = host[name](GetHandle);
```

来还原就可以了——通过向函数传入 `GetHandle` 对象, 来获取内部的 `foo()` 函数的引用, 这个技巧的细节请参考“1.3.2 多投事件系统的实现”。

Qomo 较新的版本简化了 `assign()` 与 `unassign()` 的逻辑, 将大量与切面类型相关的具体代码放在了各个子类的 `assign()` 与 `unassign()` 方法中。在子类覆基类 `Aspect()` 的 `assign()` 时, 子类应当留意: 子类所接受的 `host` 与 `name` 参数, 并不一定是 `Aspect()` 所能理解的。因此可能要在 `inherited()` 调用父类方法前重写 `arguments` 参数。例如在 `ClassAspect()` 中, `assign()` 传入的 `host` 是类类型或构造器, 而真正应当被重写的方法名却只它们的原型成员, 因此这里就发生了参数的重写:

```

this.assign = function(host, name, pointcut) {
    var hh = (host['ClassInfo'] ? host.Create : host.constructor).prototype;
    var f, h = host; // backup
    host = hh; // rewrite arguments
    f = this.inherited(); //<--这里使用 host 值重写过的 arguments 调用父类同名方法
    // ...
}

```


类似于 'AttrGetter' 和 'AttrSetter' 等切面/切点也使用到了这样的技术。应用这样的技术之后，可能 `unassign()` 也需要相应的处理，而且切面中被 `Aspect()` 基类填值的一些特性也需要修正。

1.5. 其它

Qomo 具有大量的工具函数和工具类，象多投事件之类的部分内核机制也是以工具的形式提供的。作为工具，这些函数与类具有很高的通用性，或面向通用功能的设计与实现。

通常的工具都具有语言兼容性限制，Qomo 也不例外。如果在跨宿主或跨脚本版本中使用 Qomo 的工具函数与类，可能不得不背负上 Qomo 的兼容层这样的包袱。不过，Qomo 的工具在绝大多数情况下考虑了这些问题，以使得这些代码可以脱离 Qomo 使用。

本节也讲述一些应用框架级别的机制，例如缓存、池化处理等。

1.5.1. 装载、内联与导入

Qomo 使用自己的装载系统来载入各个 JavaScript 脚本单元。考虑到下述三个方面的因素，Qomo 的装载系统实现得比较复杂：

- 👉 Qomo 希望整个框架可以跨引擎，因此对具体宿主环境存有依赖的装载系统必须被实现为可拆卸、可重写的；
- 👉 Qomo 希望外部系统不要依赖装载系统的内部信息，因此装载需要在使用结束后对自身做清理；
- 👉 Qomo 希望装载系统理解复杂的路径系统，而且整体框架在其它路径或位置系统中是可被复用的。

`$import()` 用于实现脚本装载的功能^①。除此之外，还有两个关键函数以实现类似 / 相关的功能：

- 👉 内联函数 `$inline()`，读取脚本并返回一个字符串，但不执行该字符串；
- 👉 导入函数 `$import2()`，读取脚本并在一个函数闭包内执行脚本代码。

^① 在较长远的设计中，它还应当实现命名空间与包的装载，但目前 Qomo 项目并没有这样的需求与实现。

1.5.1.1. 装载: \$import()

在具体实现上, Qomo 将装载系统独立在整个系统框架之外(位于 Qomo.js 中), 对于不同的引擎或宿主环境, Qomo 只需要替换这个独立的模块, 并保证关键接口具有一致性即可。

在外部系统看来, 这个关键接口包括:

- 👉 \$import.get(), 类似于 TObject.get(), 用于读取 \$import() 函数内部的数据;
- 👉 \$import.set(), 类似于 TObject.set(), 用于修改 \$import() 函数内部的数据;
- 👉 \$import.setActiveUrl(), 在系统不能识别当前装载的模块(.js 文件)时设置路径;
- 👉 \$import.OnSysInitialized(), 在系统装载完成之后清理 \$import() 的外部信息。

\$import.get() 与 \$import.set() 目前的可以处理的内部属性中大部分都是留给路径系统和(浏览器)兼容系统的。这些内部属性在 \$import() 中通过如下方式进行封装:

```
$import = function() {  
    var _sys = {  
        'scripts': {},  
        'curScript': '',  
        'activeJS' : $JS,  
        'parseRelativeURL': $RURL,  
        'bodyDecode': ...,  
        'docBase' : ...,  
        'srcBase': ...,  
        'pathBase' : ...,  
        'transitionUrl': ...  
    }  
  
    // 真实的$import() 装载程序  
    function _import (target, condition) {  
        // 可被替换为第三方装载代码  
        // _load_and_exec(target);  
    }  
  
    // 内部属性的读写器  
    _import.get = function(n) {  
        return eval('_sys[n]');  
    }  
  
    _import.set = function(n, v) {  
        eval('_sys[n] = v');  
    }  
}
```

```

}
return _import;
}();

```

因为 Qomo 的命名空间基于路径系统，因此对上述内部属性中的：

```

'curScript': '',
'activeJS' : $JS,
'parseRelativeURL': $RURL,
'docBase' : ...,
'pathBase' : ...,
'transitionUrl': ...

```

都会关注，这一定程度上也体现了 Namespace.js 内部实现的复杂性。但是在其它引擎或宿主环境中，可能 Namespace.js 并不需要这样复杂的实现，因此读者可以先忽略它们。

命名空间系统使用一个特殊的 \$import() 内部属性来反映“当前活动的命名空间”，该属性是 activeSpc，它指向一个返回活动空间的函数：

```

$import.set('activeSpc', function() {
    // 分析当前活动空间并返回
});

```

这是外部系统——例如 OOP 框架对命名空间的唯一依赖。因此用户可以简单地替换命名空间系统——只要保证重写该属性。

\$import() 提供一个特别的内部属性用于对装载系统进行性能分析，该属性名为 'perf_exec_stub'。当 \$import.set() 检测到名为 'perf_exec_stub' 的属性时，会重写内部函数 _load_and_exec()，以保证用户可以向 \$import() 内部挂接自己的分析代码。

除了这些之外，\$import() 只是一个用来从指定位置读取脚本内容并调用宿主环境的 execScript() 方法执行脚本的简单函数^①。

1.5.1.2. 内联: \$inline()

\$inline() 并没有执行脚本的能力，因此它一般都与 eval() 函数连用。它是一项用于弥补 \$import() 总是在全局闭包中执行代码的设计。例如下面的代码：

```

function myFunc() {
    eval($inline('test.js'));
}

```

^① 并不是所有的宿主都提供 execScript() 方法，但一般都具有类似函数以提供在全局闭包中执行脚本的能力，用户只需要简单地封装一个名字即可。

```
}
```

这样 `test.js` 中的脚本文件就会在 `myFunc()` 函数的闭包内执行。

但是 `$inline()` 是基于 `$import()` 的路径分析的，也就是使用与 `$import()` 相同的相对路径系统。这在系统装载时没有问题，而在系统执行的实时过程中却不能有效地识别路径。因此如果你不能保证 `$inline()` 是在第一次装载时执行的话，就应当使用下面的方法来内联代码：

```
// 在装载时使用$inline()来读取代码
var str = $inline('test.js');

function myFunc() {
    eval(str);
}
```

也可以不声明变量 `str`，而藉由 `$inline()` 内部缓存系统来实现内联代码的预载入：

```
// 在装载时使用$inline()来预载入
$inline('test.js');

function myFunc() {
    eval($inline('test.js')); // <-- 使用与预载入相同的文件名
}
```

注意这里的文件名是缓存的关键。`Qomo` 在 `$inline()` 中并不以它被转换后的真实路径来缓存，而仅仅使用这里的传入参数值。因此如果在整个系统中存在完全相同的文件名（以及在装载时相同的相对路径名），那么缓存系统会紊乱。

1.5.1.3. 导入：\$import2()

最后被提及的是函数 `$import2()`，它是在 `$inline()` 的基础上实现的——使用 `$inline()` 来读取代码，因此也会受到上述缓存的影响。`$import2()` 的作用是装载第三方代码，它可以装载一个大型的包，或者一些类、代码片断等等。它总是装第三方代码放在一个函数闭包中运行。这样一来，第三方代码便很少有机会能影响到 `Qomo` 的系统环境。

`$import2()` 提供两个参数来向第三方代码传入和传出数据，声明如下（其中 `src` 和 `condition` 参数与 `$import()` 是一致的）：

```
function $import2(src, prepare, patch, condition) { ... }
- prepare: 在装入代码前面添加的代码，例如变量初始化等
- patch: 在装入代码尾部添加的代码，用于向外部（例如 Qomo）传出数据
```

使用 `prepare` 在第三方代码前添加的代码有两个作用。其一是向第三方传入数据。下面的代码演示如何传入值类型和引用类型数据：

```
var v1 = 100, v2 = {};  
var csPrepare = 'var value_1 = ' + v1 + '; var value_2 = v2;';  
  
// test.js 中的代码中可以通过 value_1 和 value_2 访问到外部的数据 (值或引用)  
$import2('test.js', csPrepare, '', true);
```

其二是可以初始化局部变量，以避免第三方代码中的变量泄露出来，与 Qomo 的全局变量冲突。例如：

```
// 第三方代码文本 (文件名: test.js)  
value1 = 100;  
name = 'mySystem';
```

如果我们直接执行它，那么 Qomo 的全局变量中就会多出 `value1` 和 `name` 两个变量名。如果我们用下面的方法来装载：

```
$import2('test.js', 'var value1,name;', ...);
```

那么就不会有问题了。

`$import2()` 中的另一个参数 `patch` 也具有特别的作用。一方面你可以为 `patch` 参数传入一个字符串做代码块，这个代码块中可以直接使用 `this`。另一方面，你也可以为 `patch` 参数传入一个函数，该函数在执行时的第一个参数就是这个 `this` 引用。例如：

```
var obj = $import2('test.js', 'var value1,name;', function(_this) {  
    _this.value = value1;  
    _this.name = name;  
});
```

这样一来，我们就可以在外部系统中得到这个对象 `obj`，它的 `value`、`name` 则来自于第三方代码。当然，其它的对象方法或类都可以用这种技术来从第三方代码中获取^①。

1.5.2. 四种模板技术

JavaScript 的字符串处理效率是非常差的，而且由于只提供了为数不多的一些内置方法，因此灵活性也不够。针对不同场合下的实际需要，Qomo 在系

^① Qomo 代码包中有一个更加完整的示例，位于 `Common\ParserLite.js`。它从 `3rd\jsparse.js` 中获取了 `parse()` 和 `tokens()` 两个方法，用作 `ParserLite.prototype` 原型方法。通过这种技术，`ParserLite()` 既具有了 Qomo 类的特性，又在对 Qomo 完全无扰的情况下，使用了第三方代码。

统的不同层次实现了四种模板技术^①：

- 👉 `joinSlot()`：一个小型、快捷的基于数组的模板；
- 👉 `String.prototype.format()`：一个小型的字符串格式输出的函数；
- 👉 `Pattern()`对象：一个快速的、有应用限制的模板对象；
- 👉 `TTemplet` 类：一个特性完整的 `Qomo` 类。

1.5.2.1. 连接槽：joinSlot()

`Qomo` 中的连接槽是一个简单的小技术，甚至可以说成是小技巧。它应用在接口实现单元 `Interface.js` 中。它主要解决一种特殊的字符串拼接问题。例如：

```
str = '<div id="' + v1 + '" style="' + v2 + '">' + v3 + '</div>';
```

首先这个拼接的代码看起非常啰嗦，也十分不便于维护；其次，如果拼接频繁则效率极低。这种情况下，就比较适合使用 `joinSlot()` 技术。它首先要求使用一种特殊的格式来定义字符串，然后切分成数组：

```
var slot = '<div id="$ $" style="$ $">$ $</div>'.split('$');
```

注意这里字符串中使用 `"$ $"` 来分隔，而 `split()` 切分时则使用 `"$"`——事实上你也可以使用任意其它的字符或字符串来分隔。这样一来我们就得到了一个如下形式的、带“空槽”的数组：

```
['<div id="', <槽 1>, '" style="', <槽 2>, '">', <槽 3>, '</div>'];
```

接下来我们可以用 `joinSlot()` 函数来快速拼接它：

```
function joinSlot(slot) { // 1..n
  for (var i=j=1,args=arguments,argn=args.length; i<argn; i++,j++) {
    slot[j++] = args[i];
  }
  return slot.join('');
}

// 示例
str = joinSlot(slot, v1, v2, v3);
```

看起来这是一个很简单的技巧，但在很多复杂的环境中都可以很好的应用。一方面我们可以任意定制分隔字符串 `"$ $"`，另一方面原始的模板看起来也比较规则，所以很方便维护。而且，由于 `slot()` 是一个可复用的数组，所以在

^① 在对这四种模板技术的讨论中，我们将使用长度 400 的 10 万个字符串为测试数据进行效率分析。

大量的、频繁的字符串拼接中，至少可以用以下两种方法来提高效率^①。第一种方法不改写上述的 `joinSlot()`：

```
var arr = [
    [v1, v2, v3],
    [v1, v2, v3],
    // ...
    [v1, v2, v3]
];

var ss=[slot], slots=[];
for (var i=0; i<arr.length; i++) {
    slots.push( joinSlot.apply(this, ss.concat(arr[i])) );
}
str = slots.join('');
```

第二种方法改写 `joinSlot()`以减少一些不必要的连接：

```
// arr[] 的声明参见上例
function joinSlot(slot, args) { // 1..n
    for (var i=0,j=1,argn=args.length; i<argn; i++,j++) {
        slot[j++] = args[i];
    }
    return slot.join('');
}

var slots=[];
for (var i=0; i<arr.length; i++) {
    slots.push( joinSlot(slot, arr[i]) );
}
str = slots.join('');
```

由于 `joinSlot()` 的应用比较灵活，因此 Qomo 没有单独将它抽取为工具函数，读者可以参考上述模式在自己的代码中应用这一技术。

1.5.2.2. 格式化字符串：format()

Qomo 用非常简洁的代码实现了一个与 Delphi 风格一致的 `format()` 函数^②：在字符串中使用 “%s” 和 “%n”（这里的 n 指整数索引）作为匹配符——或称

^① 优化方法涉及到不同引擎对数据、字符串的处理方法，因此结果并不一致。在 JScript 中，方法 2 比方法 1 约快 20%，整体性能良好；但在 SpiderMonkey JavaScript 中，两种方法性能基本一致，但远比 JScript 效率差。

^② `format()` 比较简洁和高效，但由于匹配依赖于正则表达式对字符串的扫描，因此效率约比 `joinSlot()` 低 45% 左右。

为占位符。由于 JavaScript 中各种类型都可以转换为字符串型，因此 `format()` 没有实现复杂的类型转换或识别——例如 Delphi 中的 “%d” 等匹配符。

`format()` 实现为 `String()` 对象原型上的一个方法，但作为习惯，Qomo 也提供了一个全局函数的版本：

```
String.prototype.format = function() { ... }  
String.format = function(str, arr) { ... }  
format = String.format;
```

`format()` 在使用时可以简单地传入数据（将转换为字符串处理），例如：

```
pat = 'value: %s, %s, %s';  
alert( pat.format(1, 2, 3) ); // 输出'value: 1, 2, 3'
```

也可以使用 `format()` 函数并传入数组：

```
alert( format(pat, [1, 2, 3]) );
```

在模式字符串中，除了使用 “%s”，也可以使用参数索引。例如：

```
pat2 = 'value: %0, %2, %1, %s, %s, %s';  
alert( format(pat2, [1, 2, 3]) ); // 输出'value: 1, 3, 2, 1, 2, 3'
```

参数索引从 0 开始计数。“%s” 匹配符会消耗一个参数计数，而索引匹配时却不消耗。并且当参数计数被 “%s” 消耗完之后，索引匹配仍然是可用的。例如：

```
pat3 = 'value: %s, %s, %s, %2, %1, %0';  
alert( format(pat3, [1, 2, 3]) ); // 输出'value: 1, 2, 3, 3, 2, 1'
```

但当参数计数耗完之后，“%s” 就将不再被替换了。例如：

```
pat4 = 'value: %s, %s, %s, %s';  
alert( format(pat4, [1, 2, 3]) ); // 输出'value: 1, 2, 3, %s'
```

关于 `format()` 的其它规则包括：

- 👉 仅有 “%s” 与 “%S” 是消耗匹配计数的；
- 👉 “%%” 输出为单个 “%”，且不再参与匹配；
- 👉 当索引超出参数个数时，指定位置被替换为空字符串^①；
- 👉 除上述外，“%” 后的任意字符都将与 “%” 符号一同输出。

1.5.2.3. 模板对象：Pattern()

Qomo 中的 `Pattern()` 对象用于解决更大批量的字符串的一次性拼接。

^① 这取决于引擎在使用正则对象模式替换的 `String.replace()` 方法中如何处理 `undefined` 值。一些引擎中可能在这里替换为字符串 `'undefined'`。

对于不太多的字符串来说，`format()`更加方便；对于总是使用相同模板的大批量、分散的字符串连接，`joinSlot()`提供了更好的性能。而 `Pattern()`主要用于解决在更大批量下、一次性的字符串连接^①。

`Pattern()`的核心思想是这样。如果一个字符串能被这样替换：

```
pat = 'value: %s, %s, %s, %s';

// 返回 'value: 1, 2, 3, 4'
str = format(pat, [1,2,3,4]);
```

那么它也可以反过来看成是下面的替换模式：

```
// 替换的目标字符串模式
src_pattern = 'value: $1, $2, $3, $4';
// 提供源数据的数据格式
data_pattern = /(.*?), (.*?), (.*?), ([^,]*)/, ?/g;
// 源数据
arr = [1,2,3,4];
data = arr.join();

// 返回 'value: 1, 2, 3, 4'
str = data.replace(data_pattern, src_pattern);
```

也就是把字符串中的“%s”替换，看作是对一个有格式的数据（`data_pattern`）中的数据值的替换。

这个方法的特点是：整个替换过程没有用户的脚本代码参与，因此效率最高。在上面的例子中，两个关键方法 `Array.join()` 和 `String.replace()` 都是 JavaScript 的内部方法。而且由于 `arr.join()` 会拼合多维数组，因此上述代码也可以用于大批量的、规则化的字符串替换。例如：

```
// (参见上例)
arr = [
  [1,2,3,4],
  [3,4,5,6]
];
data = arr.join();

// 返回 'value: 1, 2, 3, 4value: 3, 4, 5, 6'
str = data.replace(data_pattern, src_pattern);
```

`Qomo` 中的 `Pattern()` 对象封装了上述技术。它使用两个内部函数实现了

^① 它的性能比 `joinSlot()` 高大约五倍。在 `Qomo` 中，它用于在图形层中批量生成数据，以实现在 `beginPaint()` 之后缓存，至 `endPaint()` 时一次渲染的绘制风格。

src_pattern 与 data_pattern 的基本生成算法:

```
function fmt_Pattern(patt) {
    return ('$'+patt.join('|',$')).split('|,');
}

function fmt_NumberRow(count) {
    return (count < 2 ? (count == 1 ? '(.*)|$\n' : '')
        : (new Array(count).join('(.*)|,') + '([^\n]*)|,?\n'));
}
```

这两个内部函数可以这样使用:

```
// 1. 通过标准的 format() 字符串生成源字符串模板
pat = 'value: %s, %s, %s, %s';
src_pattern = fmt_Pattern(pat);

// 2. 生成指定元素个数(示例中是 4 个)的数据模板
data_pattern = new RegExp(fmt_NumberRow(4), 'g')
```

这些过程被封闭在 Pattern.pattern() 方法中。我们在外部代码中使用这个模板时非常简单:

```
// 1. 标准的 format() 格式字符串
pat = 'value: %s, %s, %s, %s\r';

// 2. 一维或多维数据
arr = [
    [1,2,3,4],
    [4,6,7,9]
];

// 3. Pattern() 对象的使用
aPat = new Pattern(pat);
aPat.format = [1,2,3,4];
str = aPat.pattern(arr);
alert( str );
```

注意上面的这个 format 参数。它表明数据源(数组 arr)提供数据的格式,上例表明按 1~4 的顺序提供。因此输出结果为:

```
value: 1, 2, 3, 4
value: 4, 6, 7, 9
```

而如果数据格式发生了变化,例如上面的数据源格式变成了下面这样:

```
arr = [
```

```
[1,2,3, 'b','c','d' ,4],  
[4,6,7, 'H','H','H' ,9]  
];
```

只需要修改参数:

```
aPat.format = [1,2,3,7];
```

就可以了。其它代码，以及输出结果都不会受到影响。

1.5.2.4. 模板类: TTemplet

Qomo 将 Templet.js 放入 RTL，以其在 RTL 一级支持一种全局性的模板语言，用于组织框架中的重复性元素，包括脚本、样式和 HTML 等，甚至可以通过 Templet 来生成类——如同 C++ 中的模板。但与 C++ 不同的是，javascript 不需要为了类型问题而实现一种模板或者范型系统，因此 Qomo 的模板与 C++ 的模板并不完全等义^①。

Qomo 的 TTemplet 是基于类继承系统的——这也是它被称为模板类，而 Pattern() 被称为模板对象的原因。TTemplet 除基本逻辑外，不包括任何实现代码，因此你也不应当直接用它作模板，而应当派生它的子类。例如：

```
function MyTemplet() {  
    _set('TempletContext', '<font color="%Color%">%Info%</font>');  
  
    Attribute(this, 'Color', 'red');  
    Attribute(this, 'Info');  
}  
TMyTemplet = Class(TTemplet, 'MyTemplet');  
  
var ctx = new MyTemplet();  
ctx.set('Info', 'hello, world!');  
alert(ctx);
```

如该例所示，TTemplet 支持一个用 "%..%" 格式描述的模板上下文 (Templet Context) ——写成 "%%"，则表明应该替换成单个的 "%"。每一对 "%..%" 中间包含一个标签。对于标签中的被替换内容，Templet 默认认为它来自于模板实例的 Attribute。因此，上例的实际输出会是一段 HTML 代码：

```
<font color="red">hello, world!</font>
```

^① 这也是 Qomo 使用 Templet 这个词，而不是 Template 的原因之一。这尽管是一个小小的差异，但也希望引起你的注意。

Templet 的构造器还可以接受一个参数，它可以是从 Qomo 的 Object 继承下来的任意对象。这时，Templet 将使用该对象的，而不是模板实例的 Attribute。例如与上面的例子等效的代码是：

```
function MyObject() {
    Attribute(this, 'Info', 'hello, world!');
    Attribute(this, 'Color', 'red');
}
TMyObject = Class(TObject, 'MyObject');

function MyTemplet() {
    _set('TempletContext', '<font color="%Color%">%Info%</font>');
}
TMyTemplet = Class(TTemplet, 'MyTemplet');

var ctx = new MyTemplet(new MyObject());
alert(ctx);
```

通过这样的技术，我们就可以将模板系统与提供数据或内容的业务系统分离开来。事实上，在 Qomo 的组件系统中，就采用这样的技术来分离 HTML 内容框架、业务内容与组件样式。不过，TTemplet 在设计上与 HTML 标签或者其它任何特定的内容都没有关系。因此你既可以用它来生成 HTML 的模板或者 CSS 的模板，也可以用它来生成 javascript 的模板并用 eval() 来执行——我的意思是说这可以是一种实现元类的基本技术^①。

模板类在实现上依赖于接口系统。它需要使用 IAttrProvider 接口中从类中查询 Attribute 信息，并决定替换哪些标签。从这个角度上来说，模板在创建时所传入对象只需要实现该接口即可，并不要求必须是 Qomo 对象。举例来说，在 Qomo 的 ObjUtils.js 单元中提供了一个虚拟 Qomo 对象的构造器 _ObjectInstance()^②，这个构造器就实现了 IAttrProvider 接口，因此可以传入它的一个实例。例如：

```
// (续上例)

// 特性对象可以是一个从远程(例如用 ajax)读取的字符串或序列化对象
var attrs = '({Info: "aimingoo", Color: "blue"})';
// 构造虚拟对象
var vObj = new _ObjectInstance();
// 虚拟对象可以添加显式的特性和特性对象
```

^① 元类与类的区别在于：前者用于构建类（例如 Qomo 中的类类型），后者用于构建对象。

^② 虚拟对象能快速创建“类似于 Qomo 对象”的对象实例并使用一些基本的 Qomo 对象特性，可在一些性能要求较高的环境中用于局部优化。

```
vObj.data = eval(attrs);

var ctx2 = new MyTemplet(vObj);
alert(ctx2);
```

模板类 Qomo 的组件系统中有更加复杂的应用。因为从原理上来说，子类的接口实现是可以覆盖父类的，因此子类（例如界面组件）可以重写 IAttrProvider 接口，动态地判断从类属性、成员或者相关联的 DOM 对象成员或属性中取值。关于这些复杂的技术细节，可以参阅 Qomo 代码包中的文档。

1.5.3. 出错处理

JavaScript 中的异常是比较难于管理的，例如异常的编号，或者显示信息时大多使用直接声明的字符串^①。为此，Qomo 约定了一种使用两个成员的数组表示的异常。例如：

```
EAccessInvaildClass = [8109, 'Class invaild: lost typeinfo!'];
```

在第二个成员(字符串)中允许使用 format() 中的通配符 "%s" 和 "%n"。例如：

```
EAttributeCantRead = [8112, 'The "%s" attribute can\'t read for %s.'];
```

这种异常记录/对象的结构定义是非常取巧的：

- 👉 如果不使用 Qomo 的异常框架，那么标准 JS 中将会把数组转换成字符串，这时得到的信息是可以阅读的。
- 👉 如果使用 Qomo 异常框架，那么由于 Qomo 替换了 Error() 类，因此将生成更友好的信息。
- 👉 在 JSEnhance.js 中，Qomo 再次替换了 Error() 类，这使得 %s 可以被处理，从而使动态的组织异常信息变得更加便利。

Qomo 通过重写 Error() 类，以支持更加友好地显示错误信息。这个过程分为两步。其一，在 Error.js 单元中，重写了 Error() 类以支持如下构造形式：

```
e = new Error();
e = new Error(number);
e = new Error(number, description);
e = new Error(number, description, instanceObj);
e = new Error(a_qomo_exp);
e = new Error(a_qomo_exp, instanceObj);
```

其中，instanceObj 表明一个关联到该异常的对象实例。但 Qomo 并不处理 instanceObj，只是通过异常对象来传递它。这样可以使 try .. catch 捕获到的

^① SpiderMonkey JavaScript 中能够方便地构造异常子类，但 JScript 没有这样的特性。

异常对象有一个 `instanceObj` 属性，指向触发异常时送过来的一个“参考实例”。

而 `a_qomo_exp` 表示一个按 `qomo` 的规则声明的异常数组（参阅前面的内容）。这样我们就可以用下面的代码来简单的触发一个异常：

```
throw new Error(EAccessInvaildClass);
```

如果系统未装载过 `Error.js`，那么显示的信息将是：

```
错误: 8109,Class invaild: lost typeinfo!
```

如果系统已经装载过 `Error.js`，那么显示的信息将是：

```
错误: Class invaild: lost typeinfo!
```

其二，在 `JSEnhance.js` 单元中，通过重写 `Error()` 类以支持 `format()` 的匹配符。例如这种使用带通配符的异常：

```
throw new Error(EAttributeCantRead.concat('get', 'Enumerator'));
```

如果我们装载过 `JSEnhance.js`，那么显示的信息将是：

```
错误: The "get" attribute can't read for Enumerator.
```

这样，无论如何，我们都能给用户“相对友好”的错误信息。

最后，`Qomo` 也实现了一个简单的断言。它其实就是一个 `Qomo` 的异常：

```
var
    EAssertFail = [8001, 'assert is failed.\n\n%s'];

$assert = function (isTrue, info) {
    if (!isTrue) throw new Error(EAssertFail.concat([info]));
}
```

由于 `Qomo` 有自己的异常实现，因此断言的显示也相对友好。

1.5.4.其它功能模块

`Qomo` 框架一直处在不断地完善之中，目前^①除语言特性方面的扩展相对稳定之外，其它部分（包括框架类库、组件库、图形层、开发工具等）都在持续发展的阶段。下面简要介绍一些相对完善的功能模块（需要注意的是，大部分都需要运行在 `Web` 浏览器中）。

^① 这里指的是本书当前版次结稿的时间：2007 年 9 月。

1.5.4.1. 性能分析工具

Qomo 在 Debug.js 单元中载入了一个 Profiler 工具，是分析代码性能的利器。它使用起来非常方便，也可以使用多组的 profiler。系统中单独初始化了一个全局的 \$profilers，以方便使用：

```
<!-- 载入 Profiler 类 -->
<script src='Framework/Debug/Profilers.js'></script>

<script>
// 1. 在用户代码中插入分析语句，然后执行
function myFunc() {
    $profilers('myFunc').begin()

    // your code ...

    $profilers('myFunc').end()
}
myFunc();

// 2. 输出分析结果
document.writeln($profilers);
</script>
```

你可以通过一组工具来优化第二步的输出结果（以及定制输出），这需要装载 Dbg.Utills.js 单元：

```
<!-- 载入 Profiler 类和调试用工具单元 -->
<script src='Framework/Debug/Profilers.js'></script>
<script src='Framework/Debug/Dbg.Utills.js'></script>

<script>
// 在用户代码中插入分析语句，然后执行
// （同上，略...）

// 重写 $debug() 函数
$debug.resetTo(function() {
    arguments.join = Array.prototype.join;
    document.writeln(arguments.join(''));
});

// 显示 profiler 信息
```

```
showProfiler($profilers);  
</script>
```

注意这里用到了 `$debug.resetTo()` 和 `showProfiler()`。前者是因为在 Qomo 装载过程中的输出都直接面向 `document`，但这可能不是你想要的，因此在 `Dbg.Utills.js` 中重写了 `$debug`，将输出重新重向到了一个内部的缓存中：

```
$debug = function() {  
    // ...  
    arguments.callee['$cached$'] += arguments.join('');  
}
```

这里的 `resetTo()` 方法用于传入一个函数，新函数要能够输出这些被缓存的信息——它在被 `resetTo()` 时将被自动调用一次，然后该新函数将替代原 `$debug()` 在系统中的作用。如果你不打算让新的 `$debug()` 处理被缓存的那些信息，那么可以用“1.6.2.6 函数名对切面系统的影响”中提及到的运行期重写的方法来忽略第一次调用的信息。例如：

```
// 重写 $debug() 函数，使信息被输出在网页中一个 id 为 'DBGINFO' 的标签中  
$debug.resetTo(function() {  
    // 这里先忽略第一次调用，然后重写 $debug  
    $debug = function() {  
        arguments.join = Array.prototype.join;  
        document.getElementById('DBGINFO').insertAdjacentHTML(  
            'beforeEnd', arguments.join(''));  
    }  
});
```

而 `showProfiler()` 函数则用于格式化输出一个 `profiler` 的结果。它使用的是 `HTML` 标签来模拟表格显示，用户也可以考虑该函数来重写自己的工具函数。

在处理复杂分析过程时，你可以为分析对象指定精确的标签。例如在上面的示例里，我们用一对：

```
$profilers('myFunc').begin()  
$profilers('myFunc').end()
```

来开始和结束分析^①。这里的 `myFunc` 可以是任意字符，也可以是任意多的参数。这样，你可以指定：

```
$profilers('myFunc', 'build').begin();  
$profilers('myFunc', 'build').end();
```

^① 大概要为每一个函数去写 `begin()` 和 `end()` 会是一件让人痛苦的事，而且频繁地改动原先的函数也具有风险。因此事实上在 Qomo 代码包中有关使用 `Profilers` 的示例，都是通过 AOP 来捕获一批函数或方法，并在 `OnBefore` 和 `OnAfter` 中调用 `Profilers.begin()` 与 `Profilers.end()` 来实现的。


```
$profilers('myFunc', 'execute').begin();
$profilers('myFunc', 'execute').end();
```

这样对一个函数的多组分析。也可以指定：

```
$profilers('myFunc', '1').begin();
$profilers('myFunc', '1').end();

$profilers('myFunc', '2').begin();
$profilers('myFunc', '2').end();
```

这样来表明步骤。

此外，**profiler** 也会返回一标签用于一个分析会话过程的处理。例如：

```
function calc(n) {
  // 1. 开始会话
  var tag = $profilers('calc').begin();

  // 用户代码
  var v = n;
  if (v > 0) {
    v = n + calc(n - 1);
  }

  // 2. 结束会话
  $profilers('calc').end(tag);

  return v;
}

// 测试
calc(10);
```

这种情况将在 **profiler** 中产生同一记录项的多个记录值，在用户重写的 **showProfiler()**中需要对这种情况加以识别。请参见 **Dbg.Utls.js** 单元。

最后，只要你愿意，你可以创建多个 **Profilers()**对象，他们之间是没有干扰的——当然，你也可以只创建一个，并用不同的标签来分组显示它们。这一切只取决于你的选择。创建新 **profilers** 对象的方法也可以用在清除 **profiler** 数据上——因为 **Profilers()**对象并没有提供 **clear()**方法，所以你只有创建对象并重写变量。例如：

```
// 重新创建全局的分析器
$profilers = new Profilers();
```

1.5.4.2. 兼容与增强模块

Qomo 有一个框架相关的兼容层，该兼容层中有较多的代码是围绕 Qomo 的 \$import() 来写的，因此通用性并不高。另外，兼容层还为 Qomo 的集成工具也准备了一个框架，这使得兼容层显得比较冗肿。

但事实上兼容层自身具有很高的可定制和可扩展的能力。它能适应于不同的浏览器环境，以及引擎环境。这些适应能力来自于一个可扩展、定制的导入文件 CompatLayer.js，它在集成工具环境中被使用的版本是 CompatInline.js。

通过兼容层，Qomo 使大多数 SpiderMonkey JavaScript 1.6 语言特性都能被不同浏览器中的引擎所支持。

这样带来的好处之一，就是在兼容层框架上可以使用统一的语言增强。这些语言增强功能被实现在 JSEnhance.js 单元中，主要是扩展内部对象的原型方法（例如最常用的 String.format() 方法），以及支持多投事件 (MuEvent) 这样的基础特性。该单元是可以脱离 Qomo 装载和使用的。

1.5.4.3. 框架类

框架库主要实现一些与具体应用无关公共类或工具类。在目前的版本中，框架库中主要包括：

- 👉 时间线与时间处理器
- 👉 处理池与处理机
- 👉 脚本语法分析工具
- 👉 一些常用的工具函数 / 对象

其中，一般性的框架类包括 ParseLite 对象、Enum 对象、Dict 对象等等，本小节就不再详述了。下面主要概述一下“时间线与时间处理器”、“处理池与处理机”的设计思想与应用，它们的详细文档与示例请参见 Qomo 的代码包。

Qomo 框架库同时实现时间线与帧，但对二者不提供有偏好的推荐。Qomo 中，二者的区别表现在：时间线并不精确，而且并不按标准时间间隔提供；帧是纯序列化的渲染空间和数据供应，但与时间并不精确的重叠。在使用时间线与时间处理器机制时，用户可以应当把一个阶段性的时序动作（例如动画效果）分解为“时间变化”和“数据变化”两个部分——无论时间变化还是数据变化，

都可以导致时序动作效果的变化^①。Qomo 也认可时间与数据同时发生的变化，也就是“非固定间隔的时间线”下的数据变化。尽管在实用中这种变化很难处理，但可以产生独特的效果。

Qomo 把具备理解这种逻辑的能力对象称为“时间处理器(TimeMachine)”。这个类接口描述为：

```
ITimeMachine = function() {  
    this.start = Abstract;    // function(data, time) {}  
    this.OnTimer = Abstract;  // function(step, data) {}  
    this.stop = Abstract;  
}
```

其中，ITimeMachine.start()方法的入口有 time 与 data。它们可以单纯的值，而也以是一个 TSteper 类型的对象——这种对象用于产生每个单位间隔可释出的值（新的数据，或者新的时间间隔）。三者的关系主要建立在 OnTimer 事件的 step 参数上。因为 TSteper 类的事件 OnStep 的类型声明为

```
TOnStep = function(nStep, nLast) {}
```

其中 nStep 表明第几步，nLast 表明上一步产生的数据，作为此次产生数据使用的参考。因此，整个 Qomo 的时间处理体系表达的逻辑就是：

- 👉 TimeMachine.OnTime, time.OnStep 和 data.OnStep 在相同的 step 值时产生一组数据；
- 👉 在相同的 step 下，OnTime 针对于 data.OnStep 产生的数据 data 所进行的处理；
- 👉 TimeMachine 通过 time.OnStep 来得到下一次发生处理的延时。

举例来说，我们可以用下面的代码来产生一个飞行效果：

```
// 0. 飞行的方法与基本数据  
var x0, x1;  
var doFly = function(step, data) {  
    // ...  
}  
  
// 1. 构造一个时钟及其处理程序  
var T2 = TTimeline.Create(doFly);  
  
// 2. 构建一个数据发生器，用于向时钟提供数据  
provide = TYuiSteper.Create();  
provide.set('From', x0);
```

^① 这种变化既可以看着时间线，也可以看着序列帧，这取决于你使用哪种数据发生器(Trigger)。Qomo 中提供了一个序列帧抽象的 Steper 类，以及用于帧数据发生的 SteperTrigger 类。

```
provide.set('To', x1);

// 3. 启动时钟(100ms 间隔)
T2.start(provide, 100);
```

处理池与处理机是 Qomo 框架类中另一个有趣的机制。严格地说，Pool 被译成“池”即可，只是在 Qomo 的 TPool 类中，所有放在“池”中的对象都是处理机(Machine)，因此这个 TPool 被称为“处理池”类。目前，Machine 只实现了 THttpGetMachine 类，用于处理 POST 与 GET 类型的 HTTP 请求，所以，严格地说，它应该名为 THttpRequestMachine^①。

Pool 提供四种能力：

- 👉 注册处理机类(Machine Class)
- 👉 管理处理机实例(Machine Instance)
- 👉 调度处理机(包括状态：resume, sleep, free)
- 👉 管理数据(Data)对象

处理机类提供处理数据(Data)的能力，数据(Data)的结构对 Pool 来说是完全透明，Pool 只在调度过程中，将数据传给处理器，处理器如何识别、使用、修改这个数据，Pool 完全不知。

处理机应当具体的能力有：

- 👉 提供 OnStateChange 事件，响应来自 Pool 的调度状态(resume, free)
- 👉 在必要的时候，激活 OnStateChange 事件，以通知 Pool 进入调度状态(sleep)

事实上，Pool 和 Machine 都可以响应三种状态，架构中没有约定哪种状态由谁发出，或者是由谁处理（并终结）。但在目前的设计中，

- 👉 Machine 处理完一段事务后，（类似于线程一样，）应触发 sleep 状态。
- 👉 Pool 在调用 push() 方法填入新的数据时，如果有空闲的 Machine，则触发该 Machine 的 result；否则，（在 Pool 未满的情况下，将）尝试创建新的 Machine，会触发它 Machine 的 resume。
- 👉 Machine 处理完自己的事务并进入 sleep 状态时，如果 Pool 有未处理数据，则为该 Machine 触发一个新的 resume。

其它的规则包括：

^① Pool 是一种机制，并不局限在使用 HttpGetMachine 上（后者只是 Machine 的一个实现实例）。我们先澄清这一点，以避免开发人员把它当成了 Ajax 技术的另一个名词。

- 👉 **Pool** 和 **Machine** 可以根据需要重新设定触发方式，但不能改变对各状态值的含义的约定；
- 👉 **Pool** 和 **Machine** 都可以添加新的状态；
- 👉 **Machine** 的启动时间，以及 **Data** 处理的结束时间，是没有时序和关联的。

尽管定义与规则如此复杂，但处理池和处理机的使用上却非常简单。首先我们应当创建处理池，并声明它接受的处理机类（同一个处理池中，只能使用相同的处理机）：

```
// 步骤 1: 声明一个处理 THttpGetMachine 的池，池的大小(深度)为 10
var pool = new Pool(THttpGetMachine, 10);
```

这里，池的大小为 10 表明池中容纳 THttpGetMachine 实例的上限为 10 个。当 Machine 在忙 (Busy) 时，如果有新的数据请求处理，则会尝试创建新的 Machine。当上限设定数的 Machine 都在忙时，数据被缓存到队列中等待处理。从原理上来说，无论是创建大小为 1 个还是 100 个的池，最终所有的数据都能得到处理。只是处理的并发量不一样，因此速度和性能也就有差异——池的大小通常是经验值。

池创建后立即开始工作。但它只响应一个方法：

```
// 步骤 1: 填入处理数据
pool.push( _your_data );
```

这里的 `_your_data` 是用户自己决定的一种结构，它对于 Pool 来说是透明的。当它被处理时，会被传递到某个 Machine 实例，而这个实例确切地知道如何处理这种数据。对于 THttpGetMachine 来说，它能处理一个对象，该对象有一个名为 `src` 的属性，即：

```
// _your_data 并不限制必须是变量或直接量声明
pool.push({ src: "http://www.doany.net/" });
```

1.5.4.4. 集成: \$builder

Qomo 有一个专用的 Build 工具，仅仅是出于对 Qomo 自身能力的考验，我们使用基于 Qomo 框架的一些脚本代码来实现这个工具。这是一个可以定制，而且对使用者来说公开、透明的工具，具体特性有：

- 👉 使用 Qomo 自身框架与标准 JavaScript 实现，不依赖第三方应用程序
- 👉 对既有的和将来的 Qomo 项目透明，无影响
- 👉 深入到 Qomo 内核的、分层的、可分解的模块定制

👉 编译的结果代码，与使用中的调试版本无差异

由于 JavaScript 是解释执行的脚本，而并没有真正的“编译”能力，所以在 Qomo 中所谓的“集成(Build)”其实是指两个过程：

👉 **Link**：连接。将各个代码块有序地组合成单一的 .js 文件。

👉 **Compress**：压缩。清除代码中的空格等，使代码尽可能地变得短小以缩短载入时间。

其结果是生成一个单独的 .js 文件。这个生成的单独的 .js 文件是可以任意命名的，为了方便，下面都称其为“Qomo.js”^①。

Qomo 的代码压缩依赖于一份名为 jsmin 的开源代码，目前 Qomo 只使用了它的二级压缩能力（这是安全的压缩级别）。Qomo 项目计划在将来的代码中使用自己的一个名为 ParseLite 的工具来实现更高性能的压缩，这份代码已经包括在公共框架库中。Qomo 还提供了一个名为 ProtectCodeContext() 对象，它用于安全地移除注释、字符串（提供保护点和还原点）。

Qomo 的代码连接依赖一组配置项，Build/Default.Config.js 是它初始时的定义。这份定义用于描述哪些模块或组件是允许由 \$import() 装入的。Default.Config.js 的定义可以满足用户开发的一般性需求：除了调试、分析等特殊功能之外，它装载所有的模块。

如果用户在装载 Qomo.js 之前就定义一份自己的特殊配置，那么 Default.Config.js 中的配置项就会被替代。因此，Builder 系统的关键就在于：在装载 Qomo 之前设定一份用于 Build 的配置。在浏览器环境中初始化 Builder 系统的代码如下：

```
<!-- 1. 装入一份可定制的配置文件 -->
<script src="Qomo.Config.js"></script>

<!-- 2. 修改上述 Qomo.Config.js 中的配置项 -->
<script>
    $QomoConfig.set('Building', true);
    // more...
</script>

<!-- 3. 装入 Qomo.js -->
<script src="Qomo.js"></script>
```

^① 在浏览器环境中的 JavaScript 没有写文件的能力，因此 Qomo 代码包中的工具事实只是生成了结果的代码块，用户可以手工复制到一个 .js 文件中去。而且在 Mozilla 环境中，这个文件名只能命名为 Qomo.js——如果你必须要使用一个别的名字，那么你需要自行修改 \$import() 函数中的一个名为 _SYS_TAG 的正则表达式。

这样的代码被在 `Build/Builder.html` 这个文件中。与上面的示例不同的是，这个文件在第二步中将检测自己是否被其它窗口打开，并且是否能够从父窗口中得到一个已经定义过的 `$QomoConfig`；如果能找到这样一份配置，将调用 `trySyncConfig()` 将配置信息同步到当前环境中。

因此 `Builder.html` 是一个可以被复用（使用对话框窗体形式打开）的工具，所有链接、压缩的代码都包括在该网页的 JavaScript 中。而在 Qomo 中，打开该窗体的是一个名为 `T_CustomBuilder.html` 的网页，后者只是用一个界面来填写 `$QomoConfig` 配置信息，并传给 `Builder.html` 而已。

“代码连接”这一过程的所有秘密都与这些无关，连接过程事实上被封装在 `Qomo.js` 中第二个装载的文件——`Build/Building.js` 的内部。应当留意的是，系统内核的装载文件 `system.js` 也位置 `Building.js` 之后，因此 `Building.js` 可以初始整个系统的环境，以便监视其后的全部装载过程。

`Building.js` 描述了整个 Builder 系统的框架。它的基本实现方案是捕获 `$import()` 中的 `transitionUrl()` 与 `bodyDecode()` 过程。为了避免这替换后的例程被其它模块改写，它还重写了 `$import.set()` 和 `$import.get()` 方法。`Building.js` 还装载了 `CompatInlineParser.js`，这个工具用于分析 Qomo 面向浏览器的兼容性框架，并使用前面提及过的 `Compat/CompatInline.js` 来替换它。最后，`Building.js` 还内联了 `FinalBuilder.js` 这个文件——事实上，这个文件才是整个 Builder 系统的精要之处：它根据 Qomo 使用 `$import()`、`$inline()` 加载文件的顺序，向一个 `builded[]` 数组中添加添加代码，从而完成整个 link 过程。

`Building.js` 为整个系统初始化了一个 `$builder` 对象，并且在单元结束部分调用 `$builder.start()` 以启动 Build 过程。根据刚才的说明可知：在此后所有系统装载的代码都会被收集在其内部的 `builded[]` 数组中。这个 Build 过程结束于 `$builder.finish()`，该方法返回所有数组 `builded.join()` 的结果——也就是所有被链接的代码文本。

所以在 `Builder.html` 中，紧随 `$builder.finish()` 之后的，就是装入压缩代码、初始化界面，以及压缩链接后的代码。而这些细节，就与集成(build)框架没有多大的关系了。

第七章 一般性的动态函数式语言技巧

相对于某些特性更纯粹的语言，JavaScript 中看起来非常技巧化的处理方法，事实上是充分利用了函数式、原型继承、动态语言等等复合的语言特性的“一般性技术”。由于涉及了过多的、不同分类系统下的语言特性，这些技巧与其它语言不相容，或者看起来颇为另类。然而在作为语言的研究者，应有能力剖析这些技术，并从中找到所谓技巧的根本。

本章用条目的方式，归纳一些在 JavaScript 中常见的技巧。除了指出这些技术的涉及的语言特性（其绝大多数技巧的特性是复合的）之外，也给出它的应用、限制与更加复杂的变化。

1.1. 消除代码的全局变量名占用

说明：

解决函外执行的代码（行）可能占用大量全局变量名的问题。

示例背景：

一段代码运行在全局闭包中时，会占用一些全局标识符名。当后续代码使用到相同标识符名时，可能导致不可预测的逻辑错误。因此可以将这些代码“包装”在一个匿名函数闭包中，使用“声明即运行”的技术，既完成代码执行，又将代码的副作用限制在函数内部的有限范围之内。

示例代码：

```
1  /**
2   * 1. 基本示例
3   */
4  void function() {
5      // ...
6  }();
7
8  /**
9   * 2. 带参数的示例
10  */
11 void function(x, y, z) {
12     //...
```



```

13     }(10, 20, 300);
14
15     /**
16      * 3. 执行后不会被释放的示例
17      */
18     void function() {
19         // 将一个内部的 (引用类型的) 变量赋给全局变量或其成员
20         String.prototype.aMethod = function() {
21             // ...
22         }
23     }();

```

示例说明：

我们在 1.4.1.3 函数调用语句中讲到过上述的例子，主要是从词法分析的角度来解释它。但是，这个例子应当被称为“声明即调用的匿名函数”。它主要解决的问题是：用闭包隔离代码并执行。

如同模式中的单例一样，这种函数只被创建一个函数实例，并立即执行。如果可能，它也将在执行后立即被释放和回收。由于函数本身具有“声明逻辑相关的连续代码”的作用，因此你也可以用这种技巧来（结构化地）组织代码并为它（这个匿名函数）添加有意义的注释。这比将代码行分散在全局闭包的各处要好。

如果在这种匿名函数内部，将一些函数内引用赋值给了外部的、全局的变量引用，例如某些内部对象、DOM 对象的成员，那么这个匿名函数在执行后并不会被释放（示例 3）。这会导致匿名函数的生存周期依赖于该外部变量——被重写或删除。因此如果的确有这样的需求，应当将更多的外部操作集中在同一个匿名函数中，并减少该函数中无关的逻辑代码，例如在同一个“声明即调用的匿名函数”内为 String.prototype 扩展多个原型方法。

参考：

1.2. 一次性的构造器

说明：

只使用一次的构造器函数。

示例背景：

一个构造器函数在使用一次之后，标识符即被重写。但可以保证“<实例>.constructor”正确的指向构造器，且允许使用 new 运算来创建更多的实例。

示例代码：

```
1  /**
2   * 1. 基本示例：使用标识符重写
3   */
4   Instance = function() { ... }
5   Instance.prototype = {
6     // 直接量声明
7     ...,
8     // 维护原型：置构造器属性
9     constructor: Instance
10  }
11  Instance = new Instance();
12  Instance2 = new Instance.constructor();
13
14  /**
15   * 2. 使用匿名函数
16   */
17  Instance = new function() {
18    // or arguments.callee.prototype
19    var proto = this.constructor.prototype;
20
21    // 维护原型 proto
22    // ...
23  }
24
25  /**
26   * 3. 在示例 2 中使用参数
27   */
28  Instance = new function(x,y) {
29    //...
30  }(1,2);
```

示例说明：

示例 1 使用了重写标识符的方法来解释了这一技巧的主要目的，也同时说

明了如何有效地在重写原型时维护 `constructor` 属性。当对象系统正确维护 `constructor` 属性时，我们就总是可以使用 “`new <实例>.constructor()`” 来创建更多的实例，而不必依赖于构造器函数的名字。

匿名函数同样具有 “一次性” 的特点，因此示例 2 做了一个相同功能的演示。在示例 2 中，依赖于：

👉 构造器函数的原型的 `constructor` 属性缺省指向自身

👉 实例(示例 2 中的 `this` 引用)从原型中继承 `constructor` 属性

这两个性质来找到匿名的构造器函数，并操作它的原型引用。此外也可以依赖参数对象的属性 `arguments.callee` 来找到该构造器，这是在匿名函数内部访问函数自身的标准方法。与示例 1 相同的是，如果示例 2 不是修改而是重写原型的话，那么它也需要维护好原型的 `constructor` 属性。

示例 3 说明在 `new` 中使用参数表的特殊性。这与函数调用不同的是，我们不必为这里的匿名函数后有参数表，而在使用一对括号来表明函数调用，例如：

```
new (<匿名函数>(1, 2));
```

这样的代码事实上是可能会导致语法异常的。因为这个参数表实际上是 `new` 运算符的一部分，而非构造器函数的一部分。因此它的语法含义事实上是：

```
new (<匿名函数>)(1, 2);
```

关于这部分的语义分析细节，请参见“1.5.1.1 使用构造器创建对象实例”。

参考：

1.3. 对象充当识别器

说明：

用一个对象来充当函数调用界面上，或针对特殊成员的识别器。利用了对象实例在全球唯一性的特性（包括直接量空白对象 “`{!={}}`”）。

示例背景：

我们在一个函数闭包内声明的局部变量，是会被外部代码直接访问到的，但能在该闭包中、更内层的子函数闭包中通过 `upvalue` 访问。因此我们可

以用该局部变量作为一个识别器，来辨识函数是在内部亦或外部调用。

除了在函数调用界面上的识别外，我们也可以利用这种技巧来弥补 in 运算不能有效甄别对象内部成员的问题。

示例代码：

```
1  /**
2   * 1. 用作函数调用界面上的识别器
3   */
4  var myFunc = function() {
5      var handle = {};
6      function _myFunc(x,y,z) {
7          if (arguments[0] === handle)
8              // 是内部调用...
9          else
10             // 是外部调用...
11         }
12
13         // 内部调用测试 (可传入更多参数)
14         _myFunc(handle, 1, 2, 3);
15
16         return _myFunc;
17     }
18     // 外部调用测试 (不能访问变量 handle)
19     myFunc(1, 2, 3);
20
21     /**
22     * 2. 用作对象成员识别器
23     */
24     var obj = function() {
25         var yes = {};
26         var tbl = { v1: yes, v2: yes, v3: yes };
27         return {
28             query: function(id) { return tbl[id] === yes }
29         }
30     }();
31     // 测试: 查询指定 id 是否存在
32     obj.query('v1');
33     obj.query('constructor');
```

示例说明：

与示例 1 相同的模式，被应用在 Qomo 的切面与多投事件特性的实现中。在这两个 Qomo 特性中，内部函数 `_myFunc()` 掌握着一些特殊的信息（例如更内部的变量或受保护的信息）。一方面，在某些有关 `_myFunc()` 的内部调用时需要索取该信息，另一方面，又要避免它的外部引用 `myFunc()` 在调用时索取该信息。这种情况下，示例 1 使用一个内 / 外部调用的识别器 `handle` 来识别函数调用的上下文环境。

另一个识别函数调用的上下文环境的方法，是直接检测 `arguments.callee.caller`，以判定是否来自某个函数调用。这种方法在 Qomo 中也有应用，但它不适合对批量调用者的识别，另外也存在限制：调用者函数的标识符必须能被 `_myFunc()` 访问。

示例 2 被应用在切面系统的实现中，用于检测某个标识名是否存在。示例 2 创建的内部检索表 `tbl` 是一个对象，而不是使用通常的索引数组。对对象的成员做检测的方法，通常是使用 `in` 运算。但是 `in` 运算并不排除 `constructor` 这种基本属性，以及通过原型继承来的属性，因此往往要花大量代码消除误判。在本示例中，通过识别器变量 `yes`，能有效地避免这些问题，并具有与 `in` 运算相同的性能。此外，在 `tbl` 表声明时，可以覆盖任何内部的或继承自原型的属性，这不会使运算效率或准确性受到影响。

本技巧在检测 `handle` 与 `yes` 等识别器变量时，必须使用 “`===`” 运算符。

参考：

1.4. 识别 `new` 运算进行的构造器调用

说明：

在一个函数中识别当前调用是使用 `new` 来进行的构造器调用，还是普通的函数或方法调用。

示例背景：

如果一个函数被设计为既可以用 `new` 运算来产生对象实例，又可以作为普通函数调用，或者可以作为对象方法调用，那么如何识别当前究竟在哪种环境下被调用呢？

示例代码：

```
1  /**
2   * 1. 识别当前函数是否使用 new 运算来构建实例
3   */
4  function MyObject() {
5      if (this instanceof arguments.callee) {
6          // 是使用 new 运算构建实例
7      }
8      else {
9          // 是普通函数调用
10     }
11 }
12 var obj = new MyObject();
13
14 /**
15  * 2. 识别当前函数是否使用作为方法调用
16  */
17 function foo() {
18     if (this === window) {
19         // 是普通函数调用
20     }
21     else {
22         // 是方法调用
23     }
24 }
25 obj.aMethod = foo;
26 obj.aMethod();
```

示例说明：

当使用 `new` 运算时，构建过程运行在函数自身的闭包中，而且新构建的对象总是构造器的一个实例。因此，示例 1 使用 `arguments.callee` 来找到这个构造器函数自身，并检测 `this` 对象是否是它的实例，从而可以检测是否运行在 `new` 运算的实例构建过程中。

但这种方法存在一个（唯一的）问题。如果用户代码试图将构造器函数作为它的实例的一个方法，那么示例 1 就会误判。例如：

```
obj.aMethod = MyObject;
obj.aMethod(); // <-- 这个方法调用会被识别为 new() 构建过程
```

除非用户代码愿意使用更为复杂的构造过程，在 `MyObject()` 中加入更多的

识别代码并与构造器原型等采用某种特别的约定，否则上述的误判问题是不可能解决的。

除了示例 1 所示的方法之外，用户代码也可以用一种简洁的方法来识别 `new` 运算中的构造器调用：

```
5     if (this.constructor === arguments.callee) {  
6         ...
```

这种方法依赖于构建器原型的 `constructor` 属性，因此要求 `MyObject` 的原型或该原型的 `constructor` 成员未被重写过。这种方法虽然有些限制，但仍然是常常用到的。

示例 2 利用了一条简单的规则：在普通函数调用中，`this` 引用指向宿主对象——在浏览器环境中是 `window`，如果是在其它环境中，请参考相关文档。

当用户代码使用 `aFunction.apply` 或 `aFunction.call` 来调用函数自身时，可能会显式地指定 `this` 引用为 `null`、`undefined` 等无意义的值。这种情况下，JavaScript 引擎内部会忽略该值，并以宿主对象作为 `this` 引用传入函数。因此我们不必额外地检测这些值。

参考：

1.5. 使用直接量及其包装类快速调用对象方法

说明：

这是一种简单的技术，用以调用某些对象方法。

示例背景：

由于 JavaScript 的类型识别并不严格，再加上用可使用 `apply()` 和 `call()` 来动态调用函数的特性，因此我们有些时候会在 `B` 类上应用 `A` 类的方法，以产生一些特殊的效果。

这种技巧最常见的是在一些数组方法上的应用。尽管我们也对字符串、数值以及正则表达式等使用相同的技巧，但事实上（在 JavaScript 规范中）他们的方法没什么可用性。

示例代码：

```
1  /**
2   * 示例 1: 用对象模拟数组并应用 (某些) 数组方法
3   */
4  var obj = {};
5  [].push.apply(obj, [1,2,3]);
6  alert(obj.length);
7
8  /**
9   * 示例 2: 检测 hasOwnProperty 是否被重写
10   * (限于 JScript)
11   */
12  var obj = {
13    hasOwnProperty: function() {} // 重写
14  }
15  // 显示 true, 表明该属性被重写
16  alert( {}.hasOwnProperty.call(obj, 'hasOwnProperty') );
```

示例说明：

示例 1 是最常见的用法，而且它在大多数时候被用来构造其它函数调用时的定制参数——这在“1.14 构造函数参数”会详细讲到。然而究其本质，不过是使用“ `[].XXX`”这种形式来取得数组（或其它直接量对象）方法的一个引用^①。

之所以该技巧会经常在数组方法中使用，也因为数组是一种比较常用的数据结构。而且作为索引数组，正是 JavaScript 中实现对象时使用的关联数组的一种有效补充，因此在许多对象并不太胜任的场合总会出现数组的身影。另一方面，也正由于 `Array()` 实质上是一种能自动维护 `length` 属性和索引数组下标的“较为特殊的对象(Object)”，所以它的大多数方法也能作用于一般性的（这里指用 `Object` 或其它构造器构造的）对象。

示例 2 的应用在“1.6.3 干净的对象”中有过更加详细的叙述。

对 `String`、`Boolean` 与 `Number` 类型的直接量使用该技巧会增加一些开销，这种开销会被隐式地浪费在一个包装类过程中。关于这一点，我们在“1.5 包装类：面向对象的妥协”中有过非常细致地解释。

^① 在更新的 Spider JavaScript 版本中，几乎所有的 `Array` 对象方法都作为 `Array` 对象自身的一个方法——例如可以写成 `Array.join`，这无论是从形式还是功能上，都与“ `[].join`”的写法没有区别。而 `Array.join` 的写法语义上更加明确，因此被更广泛地推荐——但这并不能用在 JScript 上，因此理论推荐与应用效果正好相反。

参考：

1.6.三天前是星期几？

说明：

简单而快捷的日期运算。

示例背景：

JavaScript 是基于毫秒值来表达时间的。更具体的说，一个 JavaScript 的日期对象的数值含义其实是“1970 年 1 月 1 日午夜间全球标准时间以来逝去的毫秒数”。基于这种认识，我们可以非常方便地做一些日期相关的运算。

示例代码：

```
1  /**
2   * 三天前是星期几？
3   */
4   alert(new Date(new Date() - 3*24*60*60*1000).getDay());
5
6  /**
7   * 两个日期值之间相差的小时数/天数/...
8   */
9   var d1 = new Date(2007, 1, 1, 12, 1, 30, 300);
10  var d2 = new Date(2006, 10, 15);
11  // 分别显示 1884 (小时), 78 (天)
12  alert(parseInt((d1-d2)/1000/60/60));
13  alert(parseInt((d1-d2)/1000/60/60/24));
```

示例说明：

日期对象的奥秘在它的 `valueOf` 方法（而不是其它那些复杂的 `getXXX` 或 `setXXX` 方法）上。日期对象可以直接使用数学运算符（加、减、乘、除等），并不是 JavaScript 引擎为该对象进行了什么特别的设计，而是因为 `valueOf()` 方法返回了一个数值——即上面所说的“JavaScript 的日期对象的数值含义”。

参考：

1.7.使用对象的值含义来构造复杂对象

说明:

通过 `valueOf`（对象的值含义）来构造具有相对复杂的逻辑的对象，或对它的实例进行值运算。

示例背景:

如同上一技巧所述的 `Date()`对象，其本身具有某种特定的含义，而它的值具有另一种（值）运算的能力。我们可以在 JavaScript 中简单地构造这种复杂的对象。

示例代码:

```
14  /**
15   * 1. 基本示例: 可以进行值运算的对象
16   */
17   var obj = {};
18   obj.valueOf = function() {
19     return 10;
20   }
21   // 显示 100
22   alert( 10*obj );
23
24  /**
25   * 2. 高级示例: CRC32 对象
26   */
27   var CRC32 = function() {
28     // 1. 创建查表法使用的 crc32 表
29     var $crc = new Array(256);
30     for (var j, k, l, i=0; i<256; i++) {
31       for (k=0, j=i; k<8; k++) {
32         l = ((j & 0xFFFFFFFF) / 2) & 0x7FFFFFFF;
33         j = (j & 1) ? l ^ 0xEDB88320 : l;
34       }
35       $crc[i] = j;
36     }
37
38     // 2. 计算 CRC32, 返回计算结果数值
39     function calc(str, base) {
```

```

40     var C = isNaN(base) ? 0xFFFFFFFF : +base;
41     var i = 0, cnt = str.length;
42     while (cnt--) {
43         C = $crc[(C ^ str.charCodeAt(i++)) & 0xFF] ^ (C >>> 8);
44     }
45     return C;
46 }
47
48 // 3. 返回一个构造器函数到 CRC32
49 return function(str) {
50     this.source = str;
51     this.toString = UIntToString; // 参见示例说明
52
53     var C, args=arguments;
54     this.valueOf = function() {
55         return (C!==undefined ? C : C=calc.apply(this, args))
56     }
57 }
58 }();
59 // 显示'EBE6C6E6', 'EBE6C6E6'
60 var s1 = 'Hello, world!';
61 var s2 = 'Hello, ', s3 = 'world!';
62 alert( new CRC32(s1) );
63 alert( new CRC32(s3, new CRC32(s2)) );

```

示例说明：

示例 1 展示了 `valueOf()` 的基本应用：当一个对象需要进行值运算（如同对日期对象进行的数学运算一样）时，脚本引擎会主动调用 `obj.valueOf()` 方法并用返回值参与运算。

示例 2 构造得相对复杂得多，但这主要是源于 CRC32 运算以及该对象自身设计的复杂。其中代码 36~44 行是该对象构造器的主体，它完成三个功能：

- 👉 将被运算的字符串暂存到 `source` 属性^①；
- 👉 重写 `toString()` 方法以使用字符串形式展示 CRC 值；
- 👉 重写 `valueOf()` 方法，使该对象可以用于值运算。

在第二步中，由于 CRC32 运算所得到的结果值是一个无符号 32 位整型数，

^① 如果正则表达式对象有一个 `source` 属性一样，这里的 `source` 属性只是让用户能获得原始数据的一个引用。需要注意的是，由于 `crc32` 运算可以基于一个既有值(参数 `base`)，因此该对象值可能并不是对 `source` 的直接 `crc` 运算结果。

这在 JavaScript 中不能正常地转换或显示，所以将 `toString()` 重写为方法 `UIntToString()`。该方法代码如下（缺省将显示为 16 进制字符串）：

```
function UIntToString(radix) {  
    var V = +this;  
    return (~V<0 ? 0xFFFFFFFF-V : V).toString(radix||16).toUpperCase();  
}
```

第三步是本示例的关键：重写 `valueOf()` 方法。与此前不同的是，这里的重写使用了一个作为缓存的变量 `C`，以使得 CRC 运算总是发生在其它的值运算过程中——显然，用户有可能为一个极大的字符串创建了 CRC32 对象，但并不使用它，这一处设计正是为了避免在构建时就进行 CRC 运算所带来的无谓开销。

那么何时会发生 CRC 运算呢？在第 49 行，`alert()` 会要求对象提供字符串值以显示，而 `toString()` 方法现在指向 `UIntToString()`，因此在该函数的下面这行代码中：

```
var V = ~this;
```

按位非运算符“~”会导致一次对 CRC32 对象（这里的 `this` 引用）的取值运算。而在第 50 行，为创建 `s2`、`s3` 创建 CRC32 对象的运算会先后发生（注意次序是 `s2` 在先，`s3` 在后），但并不立即进行 `crc` 运算。如同前一行代码一样，随后 `alert()` 在显示值时会通过 `UIntToString()` 触发对 `s3` 的 CRC 求值运算，并进入下面的代码：

```
function calc(str, base) {  
    var C = isNaN(base) ? 0xFFFFFFFF : +base;  
    // ...
```

这里的 `isNaN()` 将会触发对 `base`——也就是为字符串 `s2` 构建的 CRC32 对象——的取值，于是发生了对 `s2` 的 CRC 运算。接下来，“`+base`”这个一元运算表达式也会触发一次取值，不过 `base` 对象在 `isNaN()` 运算后就已经有了缓存，因此效率很高——这里使用“`+base`”来强制取 `base` 对象的值，是为了避免变量 `C` 指向 `base` 对象的引用，从而导致在后面的运算中频繁地调用 `valueOf()`——这也是 `UIntToString()` 中使用变量 `V` 的原因。

从表达式运算上讲，上面的 `calc()` 中仍然会因为访问参数 `base` 而导致两次 `base.valueOf()` 运算——尽管第二次调用时会使用缓存。这可以使用下面这种复杂而难解的代码来消除第二次 `valueOf()` 调用：

```
var C = isNaN(C = +base) ? 0xFFFFFFFF : C;
```

这一技巧利用了函数非惰性求值、闭包初始化局部变量（的标识符），以

及赋值运算将产生副作用、两次重写这些特性，尽管更优化，但代码也更难阅读——尤其它复合使用了动态、函数式以及过程式的特性——因此并不是我乐于推荐的代码。

参考：

1.8. 控制字符串替换过程的基本模式

说明：

构造一个与替换过程相关的对象，用于控制字符串替换过程。

示例背景：

使用一个数组中的数据，或使用一个对象中的成员去替换字符串的子串。这些子串使用正则表达式来指定与检索。

示例代码：

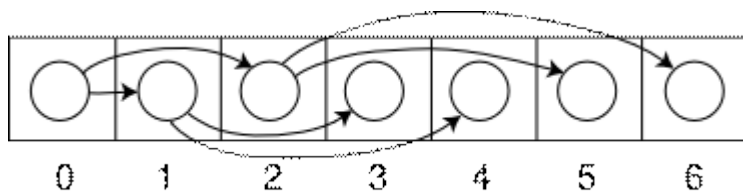
```
1  /**
2   * 普通示例：使用数组的序列来控制替换过程
3   */
4   var aString = '.....';
5   var aArray = [1,2,3,4,5,6];
6   var rx = /\./g;
7   var newStr = aString.replace(rx, function(arr) {
8       var pos = 0;
9       return function() {
10           return arr[pos++];
11       }
12   }(aArray));
13   // 测试，输出'123456'
14   alert(newStr);
15
16  /**
17   * 高级示例：设计通用的替换器①
18   */
19   function Replacer(obj) {
```

^① 这个示例来自于 jslib 项目的一个 tips 库。

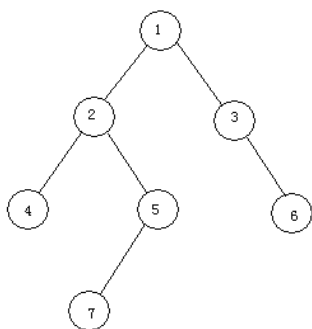
利用数组的特性实现二叉树。

示例背景：

这是一个纯粹的关于数据结构的示例^①。该示例利用 JavaScript 的关联数组的特性，构造一个可以快速存取与遍历的二叉树。它的基本原理是：一个树节点（设序列为 x ）的左、右两个子节点，可以使用数组中序列为 $2x+1$ 和 $2x+2$ 两个元素来保存。其结构如下图所示：



本例代码先说明以下二叉树在该数组中的存储，再解释对该结构的深度与广度优先的遍历：



示例代码：

```
1 // 使用数组构建的二叉树②
2 var aBinTree = [1,2,3,4,5, ,6, , ,7];
3
4 /**
5  * 示例 1: 深度优先遍历
6  */
7 void function(tree, x) {
8     x in tree && (alert(tree[x]),
9         arguments.callee.call(this, tree, x*2+1),
10        arguments.callee.call(this, tree, x*2+2))
11 }
```

^① 这一设想来自于“9 JavaScript Tips You May Not Know”（Ayman Hourieh's Blog, <http://aymanh.com/>）。

^② 这是一种简省的、但只适用于 JScript 的声明方法。在 SpiderMonkey JavaScript 中，那些未填值的数组元素也有下标，并将导致后面算法中的 `in` 运算出现误判。因此在 SpiderMonkey JavaScript 等引擎中，只能用对下标元素逐一赋值的形式来构建该二叉树。

```

11     )
12 } (aBinTree, 0);
13
14 /**
15  * 示例 2: 广度优先遍历
16  */
17 void function(tree, x) {
18     if (x in tree) {
19         var queue = { left:0, '0': x, length:1, leaf: 0 };
20         with (queue) {
21             do {
22                 x=queue[left++], alert (tree[x]), leaf=2*x,
23                 ++leaf in tree && (queue[length++] = leaf); // left
24                 ++leaf in tree && (queue[length++] = leaf); // right
25             }
26             while (left<length);
27         }
28     }
29 } (aBinTree, 0);
30
31 /**
32  * 示例 3: 遍历完整树(无序①)
33  */
34 for (var x in aBinTree) {
35     alert (aBinTree[x])
36 }

```

示例说明:

示例 1 采用经典的递归方案来实现深度优先遍历，它主要利用了 JavaScript 的函数参数“非惰性求值”的特性。因此递归将首先发生在第 9 行的 `call()` 调用中，且等到所有由此发生的递归全部返回后，第 10 行的 `call()` 调用才开始进行。而这正是深度优先算法所需要的。

示例 2 使用先入先出队列来实现广度优先的遍历。通常情况下，该算法应被实现为如下的形式：

```

...
with ([x]) {
    do {
        x=shift(), alert (tree[x]), leaf=2*x;

```

^① JavaScript 中使用 `in` 运算列举成员时，是否以成员的外部顺序列举是未被约定的。


```

    ++leaf in tree && push(leaf); // left
    ++leaf in tree && push(leaf); // right
  }
  while (length);
  ...

```

——在这个算法中，语句：

```
with ([idx]) ...
```

构建了一个数组直接量并打开了它的对象闭包，因此在该闭包中可以直接检测它的 `length` 属性以及调用 `push()` 和 `shift()` 方法。

然而 `shift()` 的使用将导致数组重排，所以会降低性能。因此在示例中构建了对象 `'queue'` 来模拟它，`queue` 的数字属性 (`0..n`) 模拟了队列下标：`queue[0..n]` 指向二叉树结点的序列 `x`。在初始时，它只有下标 `0`：

```
var queue = { left:0, '0': x, length:1, leaf: 0 };
```

随后的代码逻辑上与上面的使用数组的示例是一致的。不过由于不使用 `shift()` 来移除队列左侧的结点，因此它的效率较高。

示例 3 说明如何列举全树，不过由于 `for..in` 的实现机制的缘故，列举是无序的。如果需要一个有序的、广度优先的、全树的列举，那么可以使用增量的 `for` 语句——但不要忘记数组的某些下标是空值。

参考：

1.10. 将函数封装为方法

说明：

将函数封装为方法，使之在调用时使用特定的 `this` 引用的简单方法。

示例背景：

这一技巧是 `Qomo` 中多投事件对象的基础，它使用闭包来持有函数以及该函数所引用的 `this` 对象。除了演示如何使普通函数具有对象方法的性质之外，它也展示了 JavaScript 闭包的基本性质：当闭包内部被外部引用时，闭包不被释放。

示例代码:

```
37  aObj = {info: 'hello'};
38  aFunc = function() { alert(this.info) };
39
40  /**
41   * 示例 1: 一般性的工具函数
42   */
43  function getMethod(obj, func) {
44      return function() {
45          return func.apply(obj, arguments);
46      }
47  }
48  // 测试, 显示'hello'
49  m = getMethod(aObj, aFunc);
50  m();
51
52  /**
53   * 示例 2: 为函数原型添加的方法
54   */
55  Function.prototype.bind = function(obj) {
56      var method = this;
57      return function() {
58          return method.apply(obj, arguments);
59      }
60  }
61  // 测试, 显示'hello'
62  m2 = aFunc.bind(aObj);
63  m2();
64
65  /**
66   * 示例 3: 在示例 2 中使用对象闭包的方案
67   */
68  Function.prototype.bind = function(obj) {
69      with ( {instance:obj, method:this } ) return function() {
70          return method.apply(instance, arguments);
71      }
72  }
73  // 测试, 显示'hello'
74  m3 = aFunc.bind(aObj);
75  m3();
```

示例说明：

示例 1 的要点在于第 6 行代码使用的标识符 `obj` 与 `func`，它们来自于在第 4 行中由 `getMethod()` 函数声明的形式参数。当第 1 2 行的 `getMethod()` 执行之后，第 5 行返回的一个匿名函数将被外部变量 `m` 引用。这将导致“`getMethod()` 当前实例的一个闭包”在执行不被立即销毁。因此，尽管 `getMethod()` 函数执行结束了，但是它的形式参数传入实际值 `aObj` 与 `aFunc` 仍然是有效的。于是当第 1 3 行代码执行时，流程回到第 6 行代码，这时就调用到了 `aFunc.apply(aObj, arguments)`。

需要留意的是，第 1 3 行代码执行结束后，上述的闭包仍然是不被销毁的。该闭包的生存周期决定于变量 `m` 的生存周期——例如显式的使用“`delete m`”来销毁它。此外，由于 `getMethod()` 闭包通过形式参数也执有了 `aObj` 与 `aFunc` 的引用，因此在该闭包销毁之前重写 `aObj` 与 `aFunc` 标识符，并不会导致旧变量立即销毁。

示例 2 来自于 `Prototype JavaScript framework`。该示例与示例 1 在本质上是相同的，但由于它是作为 `Function` 的方法调用的，因此不必显性的传入 `aFunc` 参数——在第 1 9 行代码中，它变成了 `this` 引用。但也正是由于 `aFunc` 现在可以通过 `this` 引用取得，但在 2 0 ~ 2 2 行代码所声明的匿名函数的上下文中，不能通过 `this` 来获得，所以它创建了创建了局部变量 `method` 来暂存它。

示例 3 与示例 2 相比的一项特别的改进在于 `with(...)` 语句的使用，它在该语句中创建了一个对象直接量并使用它的成员引用 `this` 与 `obj`——这使用了在下一小节中会讲述到的一个技巧。表面上看来，这与示例 2 的方案并没有本质的不同，但是由于添加了一层闭包，使得最后返回给外部变量 `m3` 的函数并没有持有 `bind()` 方法闭包中的任何数据——而在示例 2 中持有了变量 `method`。因此从理论上来说，示例 3 通过创建一个对象闭包来使引擎得到了一次优化、消除 `bind()` 的函数实例及其闭包的机会^①。

参考：

^① 在这个例子中，我只能说引擎得到了这样的一个机会，我无法证明该闭包是否真的被回收（我想在当前各种引擎中应该没有实现这样的优化）；我也不确信“对象闭包”相对于“函数闭包”是否更加轻量。所以在这里的论述只是理论性的，读者可以据此分析多层闭包中的引用情况，以及尝试类似的优化方案。

1.11. 使用 with 语句来替代函数参数传递

说明:

在一段代码中使用某个参数，但不使用函数(及其形式参数)来传递该值。

示例背景:

函数闭包可以为闭包内的代码保存形式参数的实际值。对象闭包也具有类似的特性。本例使用这种特性来初始化对象原型^①。

示例代码:

```
1 // 0. 设有某个构造器
2 function Bezier() {
3 }
4
5 /**
6  * 1. 一般示例: 使用函数个形式来初始化构造器原型
7  */
8 void function(b, p) {
9     b.point = ...
10    p.xxxxxx = ...
11    p.yyyyyy = ...
12 } (Bezier, Bezier.prototype);
13
14 /**
15  * 2. 高级示例: 直接量定义、值引用、对象声明与销毁
16  */
17 with ({b: Bezier, p: Bezier.prototype}) {
18     b.point = .....
19     p.xxxxxx = .....
20     p.yyyyyy = .....
21 }
```

示例说明:

一般示例所展示的是通常的方案。在这个方案中，我们传入了 Bezier 与 Bezier.prototype，以替代形式参数 b、p。

^① 本例来自于 <http://jsfromhell.com/> 中一段构造 bezier 曲线对象的代码。

在高级示例中，我们构造了一个对象直接量，上述两个变量（和成员）成为了该对象直接量的两个属性：

```
{  
  b: Bezier,  
  p: Bezier.prototype  
}
```

该对象被声明后立即作为 `with()` 语句的一个直接量表达式，然后 `with` 打开了该对象的对象闭包，并执行接下来的代码。在这些代码中可以存取 `b`、`p` 这些属性。

与一般示例不同的是，这里的 `b`、`p` 是对象的成员而且形式参数。当然，在本质上来说，这并没有特别的不同——它们受限于对象闭包的生存周期，而前者则受限于函数闭包的生存周期。

参考：

1.12. 使用对象闭包来重置重写

说明：

对象的某些成员被重写后如何重置到原始状态。

示例背景：

某些情况下，我们不希望对象的某些成员被重写；或者我们可能需要一种能重写、重置的特性，那么我们可以在 `with()` 语句所打开的对象闭包中来暂存原始状态。

示例来自于 Qomo 的 AOP 系统。Qomo 的早期版本使用函数的 `upvalue` 来保存原始值，而后来的版本则采用了本例所述的技巧。主要实现的功能是：当 `Aspect.combine()` 方法将把 `OnIntroduction` 等事件句柄替换为合并到的对象的同名句柄，而 `Aspect.uncombine()` 需要将它恢复到原始的状态。

示例代码：

```
1  /**  
2   * 旧版本：使用 _self 暂存原始状态，并在 uncombine() 中通过 upvalue 存取它  
3   */
```

```

4    // TAspect 类的实例构造函数
5    this.Create = {
6        var _self = {
7            OnIntroduction: this.OnIntroduction,
8            OnBefore: this.OnBefore,
9            OnAfter: this.OnAfter,
10           OnAround: this.OnAround
11        }
12        this.uncombine = function() {
13            this.OnIntroduction = _self.OnIntroduction;
14            this.OnBefore = _self.OnBefore;
15            this.OnAfter = _self.OnAfter;
16            this.OnAround = _self.OnAround;
17        }
18    }
19
20    /**
21     * 新版本: 使用对象闭包
22     */
23    this.Create = {
24        with (this) {    // 对象<this>的闭包
25            this.uncombine = function() {
26                this.OnIntroduction = OnIntroduction;
27                this.OnBefore = OnBefore;
28                this.OnAfter = OnAfter;
29                this.OnAround = OnAround;
30            }
31        }
32    }

```

示例说明:

旧版本利用了 `this.Create()` 作为一个函数闭包可以在退出后保存值的特性, 使得在调用 `this.uncombine()` 方法时能够回到 `this.Create()` 方法内部, 并通过 `upvalue` 取得 `_self` 变量。而 `_self` 变量总是保存着 `this.Create()` 方法调用时暂存的原始事件句柄, 因此 `this.uncombine()` 能够通过该变量来重置。

新版本的代码看起来简单得多。关键在于: `with` 闭包保存了一个隐式的 `this` 引用, 而在第 26~29 行中赋值运算的右边的标识符正是缺省引用了该 `this` 引用——即第 24 行中的 `this` 引用。作为 JavaScript 对象系统方法调用时的基本特性, 26~29 行中赋值运算左侧的 `this`, 是“当前方法”调用时的所属对象, 而

并非上述的隐式的 `this` 引用。

需要强调的是，对象闭包在使用后是否释放，同样取决于闭包中是否有成员被其它外部对象引用。以上例来说，由于 `this.uncombine()` 方法内部引用了 `OnIntroduction` 等成员，因此在退出 `with(){...}` 语句块后，由 `with` 语句打开的对象闭包并不会被销毁。从这个角度上来说，这种方案并不会比旧版本的方案节省任何东西。

最后对技巧“1.10 将函数封装为方法”略为作出一些补充。根据本例的经验，至少在形式上看来，`Function.prototype.bind` 可以采用一种无需创建对象直接量的方法实现：

```
Function.prototype.bind = function(obj) {  
  with ( this ) return function() {  
    return apply(obj, arguments);  
  }  
}
```

但这除了节省一个对象之外，并没有如同前面设想的那样“使引擎得到了一次优化……的机会”。因为 `with(this)` 打开的闭包引用了当前环境中的 `this`，且内层的匿名函数闭包引用了 `bind()` 函数闭包的形式参数 `obj`，所以实质上是虽然节省了一个变量名 `method`，但添加了一层（不可优化的）对象闭包，效率反而打了折扣。

参考：

1.13. 构造函数参数

说明：

通过数组方法，动态地构造函数调用参数。

示例背景：

当我们确知一个函数的入口参数，或当前函数具有确定的形式参数时，我们可以使用 `aFunction.call()` 方法来调用它。但是当上述条件不具备时，我们就需要使用 `aFunction.apply()` 方法了。这种情况下，我们需要使用一个参数对象或构造一个数组作为被调用函数的实际参数传入。

示例代码:

```
1  示例代码:
2  /**
3   * 1. 普通示例: 直接使用当前函数的 arguments 对象
4   */
5  Array.prototype.pushList = function() { /* args... */
6      this.push.apply(this, arguments);
7  }
8
9  /**
10   * 2. 高级示例: 使用数组作为实际参数表
11   */
12  Array.prototype.findAndInsert = function(item, list) {
13      this.splice.apply(this, [
14          item = this.indexOf(item)<0 ? this.length : item,
15          1].concat(list)
16      );
17  }
18
19  /**
20   * 3. 高级示例: 直接操作 arguments 并复用之
21   */
22  var aPosition=1, x=100, y=200, z=300;
23  function foo(v1, v2, v3) {
24      var args = function() {
25          return [].splice.call(arguments, aPosition, 0, x, y, z), arguments;
26      }.apply(this, arguments)
27
28      // 数据 x,y,z 被插入到参数对象中间, 显示: 1,100,200,300,2,3
29      alert([].slice.apply(args));
30      // 原始参数没有受到影响, 显示: 1,2,3
31      alert([].slice.apply(arguments));
32  };
33  foo(1,2,3);
```

示例说明:

在示例 1 中, 我们简单地使用参数 `arguments` 作为 `push()` 函数的实际参数表 (这里 `pushList()` 只作为一个使用 `apply()` 方法的示例而已, 并没有什么特别的意义)。

示例 2 中，`findAndInsert()` 需要先在当前数据中查找到 `item` 元素并删除它，并将 `list` 数组插入到该元素位置；如果没有找到 `item`，则插入到数组末尾。这可以使用 `splice()` 方法来实现，但它的参数表与当前方法 `findAndInsert()` 的参数表存在较大的差异。这时我们就可以使用数组连接的方式来构造参数。

我们必须先找到一个能够返回数组的方法，这包 `String.split()`、`Array.slice()` 以及 `Array.concat()` 等等方法——但是 `Array.push()` 等方法就不适用。我们最后要的效果是通过这些方法得到一个符合函数参数序列的数组（例如示例 2 中的 `splice()` 方法的“2~任意多个”参数）。

示例 2 中采有了一个数组直接量：

```
[item = this.indexOf(item)<0 ? this.length : item, 1]
```

并使用它的方法来连接当前函数传入的 `list` 数组：

```
[<上述数组直接量>].concat(list);
```

这个运算的返回结果正是 `splice()` 所需的参数表。

除此之外，我们也可以使用一些数组操作从 `arguments` 中得到参数。例如：

```
// 得到整个 arguments (相当于将它转换为数组)
[].slice.call(arguments);

// 得到 arguments 数组的部分
[].slice.call(arguments, start, [end]);
```

并将一个（或一些数据）连接在它的前面、后面或中间部分：

```
// 连接到前面
[x,y,z].concat([].slice.call(arguments));

// 连接到后面
[].slice.call(arguments).concat(x,y,z);

// 插入数据到中间
[].slice.call(arguments, 0, aPosition).concat([x,y,z],
[.slice.call(arguments, aPosition));
```

这些（表达式）运算的结果都返回数组，因此可以用于 `apply()`。事实上，如果更好的利用函数式语言的特性，那么最后这种情况可以有更加特别的处理方法。本小节的示例 3 用于说明这种特别的技巧。

示例 3 中声明了一个内嵌的、匿名的函数，但并没有用函数调用运算符“`()`”来执行它，而是直接使用了它的 `apply()` 方法——第 26 行。在第 25 行中，

这个匿名函数使用了与其它例子类似的技巧：使用数组方法来操作 `arguments` 对象。但这里使用的是 `splice()` 方法，它直接操作数组（以及这里的 `arguments` 对象），它的返回值被忽略掉了，因为接下来是一个连续运算符“`,`”。所以 `return` 子句就返回了最后的这个运算元：`arguments`。而在返回之前，它是正好经过了 `splice()` 方法的处理。

我们应该留意到 25~26 行中用到了三处 `arguments` 对象。其中 25 行中的 `arguments` 是同一个引用（匿名函数执行时参数对象），而 26 行中的，却是当前函数 `foo()` 执行时的参数对象——因此它与第 31 行中的 `arguments` 是同一个引用。之所以第 25 行的 `splice()` 运算并不会影响到它，是因为在使用 `apply()` 方法时，JavaScript 引擎将传入参数表的一个备份，而不是该参数表的一个引用——尽管在 `apply()` 函数的接口声明：

```
Function.prototype.apply([thisObj[, argArray]])
```

中，`argArray` 是一个引用类型的对象或数组。因为在外部的调用进即已发生了复制，因此在内部（第 25 行）就没必须保护 `arguments` 了。并且，我们也可以传出该对象，让他为外部的代码使用。这里 `arguments` 就仅仅是一个标识符，可供外部的 `foo()` 函数复用，并遵循有关函数、闭包的所有引用计数规则。

最后强调一下，我们没有办法“创建”一个 `arguments` 实例。因此若非在 `apply()` 中使用数组，那么唯一的解决方法就是复用其它函数闭包中的 `arguments` 对象了^①。

参考：

1.14. 使用更复杂的表达式来消减 IF 语句

说明：

更加复杂的条件表达式来进行 IF 语句消减。

示例背景：

除了三元表达式之外，我们也能够使用复杂的表达式运算来消减 IF 语句。

^① 尽管 `arguments` 看来非常简单，简单到可用对象直接量来模拟，但无论什么方法也逃不过 `apply()` 的检测。因此我们只能在“创建数组”和“复用其它函数的 `arguments`”两个方案中选择其一。

示例代码：

```
1  /**
2   * 基本示例
3   */
4  if (aBool) {
5      doExpress
6  };
7  /* 可以写作: */
8  aBool && doExpress;
9
10 /**
11  * 高级示例: RegExp.exec() 连续使用中的多种写法
12  */
13 rx.lastIndex = aPos;
14 match = rx.exec(str);
15
16 /* 写法包括: */
17 // 1. 将赋值包括在连续运算之中
18 (rx.lastIndex=aPos, match=rx.exec(str))
19 // 2. 使用函数参数的运算能力
20 match = rx.exec(str, rx.lastIndex=aPos)
21 // 3. 使用逻辑运算
22 match = (void rx.lastIndex=aPos) || rx.exec(str)
23 // 4. 使用连续运算 (较推荐)
24 match = (rx.lastIndex=aPos, rx.exec(str))
```

示例说明：

基本示例展示了用布尔运算来构造等价的条件语句的方法。这主要是利用了布尔运算短路特性。另外，在 JavaScript 中布尔运算并不严格地返回 true/false，而是返回布尔短路中的最后一次表达式值。例如：

```
s = 0 || false || 'test' || 'hello';
```

右边的布尔运算将返回 'test'。这是因为，在第三个运算元（单值表达式）时得到一个有值的结果而发生布尔运算短路，于是整个右边的运算就返回了值 'test'。如果将这样的表达式用在 if() 等语句中：

```
if (0 || false || 'test' || 'hello') {
    ...
}
```

引擎会在解释 if 语句时检测 (...) 中的表达式值，并判断结果是真值或假值（这

里也并不一定是布尔值 `true/false`)。

高级示例演示了一组更为复杂的应用，这些应用利用了 JavaScript 表达式运算、函数调用等的多种特性，例如写法 2 中例用了 JavaScript 非惰性求值的特性，以保证 `rx.exec(str)` 之前总会发生 “`rx.lastIndex=aPos`” 运算——尽管该运算的结果值对 `exec()` 方法来说没有意义。

这个示例所述技巧来自于 Qomo 项目组为 Narcissus 提供优化方案的过程中。在 Narcissus 实现中，它使用如下方法来扫描源代码以分析语法：

```
var rx = /^\/((?:\\\.|^[^\/])+)\/([gimy]*)/;
if (this.scanOperand &&
    (match = rx.match(inputText))) {
    ...
}
else if ...
```

请注意 Narcissus 使用的正则表达式总是自字符串头部开始检索的（正则表达式以 `^...` 开始）。尽管这是出于 JavaScript 语法分析的必要，但这也导致它的 `inputText` 必须不断地截取自原始代码。这段代码是这样的：

```
getInput = function() {
    this.source.substring(this.cursor); // cursor 指向当前分析到的代码位置
}
...
inputText = sourceText.getInput();
```

`substring()` 的频繁使用导致效率急剧下降：在处理大段代码时，算法复杂度是指数级的。Qomo 项目组为解决这个问题而提供的解决方案是：

- 👉 不使用 `substring()`，而直接使用原始的代码字符串 `sourceText`
- 👉 因为上一条规则，我们要修改原来的正则表达式，去掉开始检索的特性
- 👉 基于上一条规则，我们可以使用 `rx.exec()`，并在调用前改写 `rx.lastIndex`
- 👉 为了保证匹配总是发生在指定位置的开始处（即相当于 `^...` 正则表达式的效果），我们需要在 `rx.exec()` 后复核匹配位置是否等于开始位置

不过如果我们只是满足上述条件，那么代码可能进行下面这样的大调整：

```
var rx = /\//((?:\\\.|^[^\/])+)\/([gimy]*)/;
if (this.scanOperand) {
    rx.lastIndex = this.cursor;
    match = rx.exec(sourceText);
    if (match && match.index == this.cursor) { // 保证匹配是在指定位置开始处
        ...
    }
}
```

```

    }
}
if (!match) {
    // ...
}

```

正是由于这里有一条“`rx.lastIndex = this.cursor`”语句，使得我们不能使用连续的 `if .. else if ..` 运算。代码将因此被分隔成许多的、没有连续特性的片断。而且与 Narcissus 原始代码差异很大，使得难于同步维护。

因此除此之外，我们还不希望对原始代码做大量的修改。为此我们构造了更加复杂的表达式，来使得代码结构与原始代码保持一致。这种结构如下：

```

var rx = /^\/((?:\\\.|^[^\/])+)\/([gimy]*)/;
if (this.scanOperand &&
    (match = (rx.lastIndex=this.cursor, rx.exec(sourceText))) &&
    (match.index == this.cursor)) {
    token.type = REGEXP;
    token.value = new RegExp(match[1], match[2]);
}
else if ...

```

这就是本技巧所展示的高级示例的应用价值：`rx.exec()`总会在 `lastIndex` 赋值后才被执行，整个表达式会返回 `rx.exec()`的结果值到变量 `match`。

参考：

1.15. 利用钩子函数来扩展功能

说明：

用简单的方法来实现钩子函数，以及以相同的技巧来扩展原型方法或原始函数。

示例背景：

钩子函数，是指模拟原始函数的功能与接口，在原始函数执行前、执行后增加一些特殊功能或检测代码的技术。在更接近系统层次上的技术中，它被称为 HOOK，而在更面向应用的高级开发，一般使用子类继承或切面技术来实现

相同的功能。

本技巧通过动态特性中的“重写”，来实现钩子以及脱钩(unhook)技术。

示例代码：

```
25  /**
26   * 1. 基本示例
27   */
28   function myFunc() { ... }
29
30   myFunc = function(foo) {
31     return function() {
32       // 钩子的功能代码
33       // ...
34       return foo.apply(this, arguments);
35     }
36   } (myFunc);
37
38  /**
39   * 2. 高级示例：对原型方法进行扩展且不影响原有功能
40   */
41   Number.prototype.toString = function(foo) {
42     return function(radix, length) {
43       var result = foo.apply(this, arguments).toUpperCase();
44       // 钩子的功能代码
45       ...
46     }
47   } (Number.prototype.toString);
48
49  /**
50   * 3. 高级示例：脱钩 (unhook)，以及只执行一次的钩子
51   */
52   myFunc = function(foo) {
53     return function() {
54       // 脱钩 (unhook)
55       myFunc = foo;
56
57       // 钩子的功能代码
58       // ...
59       return foo.apply(this, arguments);
60     }
61   } (myFunc);
```

示例说明：

这个技巧充分考虑了标识符重写时赋值运算符的优先级，以及重写对引用计数的影响。对于下面的表达式模式来说：

```
myFunc = function(foo) {  
  return function() { ... }  
}(myFunc);
```

赋值表达式右边的一个函数调用运算，该调用的入口参数表获得了 `myFunc` 的一个引用并暂存为形式参数 `foo`。在完成函数调用运算之后，发生赋值运算并产生重写效果，这时 `myFunc` 的引用因为标识符重写而减一。然而由于形式参数 `foo` 持有了 `myFunc` 的一个引用，因此尽管 `myFunc` 标识符被重写，但该函数实例却并不会被销毁。所以，在内层的匿名函数——也就是钩子函数中，可以通过 `upvalue` 该问 `foo` 参数，以调用原始的代码。

在这个过程中，关键的地方在于：

- 👉 赋值运算符的优先级非常低（仅高于连续运算符“,”逗号与语句分隔符“;”分号），所以它两边的表达式总是被更优先的运算。因此，标识符 `myFunc` 能在被重写之前，向内部匿名函数传入一个引用；
- 👉 函数闭包包括该函数通过形式参数建立的数据或引用；
- 👉 函数（包括匿名函数）在执行后并不销毁也不重置闭包内数据，它的生存周期取决于函数实例是否仍外引用。

上例的 1~3 所示的模式中，该匿名函数有一个内嵌的匿名函数，被作为返回值由全局标识符（`myFunc`）或对象成员（`Number.prototype.toString`）持有了引用，因此整个闭包总是不被销毁。

在示例 3 中，通过 `unhook` 过程可以清除外部标识符对这个闭包的引用。在示例代码的行 31 中：

```
myFunc = foo;
```

同样是通过重写，来清除 `myFunc` 标识符对“当前函数闭包”的引用。这样，在函数执行完之后，不但完成了“脱钩”、“只执行一次”的行为，也完成了对闭包引用计数的维护——如果没有别的引用，该函数实例会被引擎自动回收。

参考：

1.16. 安全的字符串

说明：

在函数内安全的使用字符串参数。

示例背景：

JavaScript 中的字符串是最常见用的数据类型之一。但在复杂的环境中，许多运算却总是由字符串运算导致的。

示例代码：

```
62  /**
63   * 示例一：一般性方法
64   */
65   function doTest(aStr) {
66     aStr = aStr.toString();
67     ...
68   }
69
70  /**
71   * 示例二：使用表达式运算(隐式转换)
72   */
73   function doTest(aStr) {
74     // 或
75     // aStr += '';
76     aStr = aStr + '';
77     ...
78   }
79
80  /**
81   * 示例三：使用变量声明来避免重写
82   */
83   function doTest(aStr) {
84     var aStr = aStr + '';
85     ...
86   }
```

示例说明：

如示例一，在大多数情况下，我们都可以使用 `toString()` 方法来获得数据的字

符串形式，然后访问字符串方法或属性，例如 `match()`、`charAt()` 和 `length`。

但是在更为复杂的环境中，上述的方法就可能导致错误。例如在浏览器环境中，很多对象（包括 DOM 对象）就没有 `toString()` 方法，因此示例二是较为安全的方法。示例二使用表达式运算，只要字符串能被转换为字符串形式——无论是 JavaScript 引擎调用 `toString()` 方法，还 COM/ActiveX 内部通过变体类型转换，都可在表达式运算过程中隐式地转换为字符串（参见“1.7.3.1 运算导致的类型转换”）。

但是示例二中对函数的形式参数 `aStr` 是重用的，也就是说代码：

```
// 或
// aStr += '';
aStr = aStr + '';
```

是重写 `aStr` 这个变量的，这一过程将可能触发宿主环境中的变量重写的“副作用”。例如浏览器环境下的 `location` 对象，对它的赋值操作就意味着加载指定地址的网页（参见“1.6.7 闭包中的标识符(变量)特例”）。

示例三是一个更为安全的版本。示例三声明了一个局部变量 `aStr`，而按照 JavaScript 的约定，在处理第 23 行代码时，左侧使用新声明的局部变量 `aStr`，右侧的 `aStr` 则是函数的形式参数。因此该行代码并不导致重写，从而避免了示例二的问题（参见“1.4.4.2 宿主环境的限制”）。

参考：

附录一：相关资料及参考章节

需要修改 / 统一的内容：

1、全局内统一“关键字”与“保留字”。

一、《JavaScript 权威指南》

这本“巨书”讲述了 JavaScript 语言特性以及它在 Web 环境中应用的方方面面，其“第一部分：JavaScript 的核心”（约前 200 页）与本书所述的内容有些是重叠的，下面列举在内容上具有一定关联的章节或段落。

章节 11.4 词法作用域和嵌套函数中，讲述的是词法作用域与动态作用域相关的问题：

JavaScript 的函数是词法上的作用域，而不是动态作用域。

这个问题被本书分解成如下问题讲述：

- 模块化的层次：语法作用域
- 模块化的效果：变量作用域
- 闭包

章节 11.5 Function() 构造函数和函数直接量中，讲述函数直接量声明与函数对象的构造过程：

（每次使用函数直接量，并不创建一个新的函数对象，但）可能需要一个新闭包来捕捉定义函数的词法作用域的差别。

关于这里“可能”存在的运行期效果，在本书中通过如下章节予以详述：

- 什么是函数实例与函数引用
- 在运行期，每个函数实例有一个闭包

使用构造函数 Function() 创建的函数不使用词法作用域，相反的，它们总是被当作顶级函数来编译。

关于这种效果，在本书中通过如下章节予以详述：

- 函数对象的闭包及其效果

章节 11.2 使用值与使用引用、4.4 基本类型和引用类型中的相关陈述，在本书中基本并入到如下章节：

- 第六章 JavaScript 的动态语言特性
- JavaScript 的语法：变量声明
- JavaScript 的语法：表达式运算

章节 4.3.1 没有块级作用域，在本书被作为如下章节的主要内容加以讨论：

- 级别 2：语句

章节 4.6 作为属性的变量，在本书被作为如下章节的主要内容加以讨论：

- 函数闭包与调用对象

二、《JavaScript 权威指南》章节 11.4 词法作用域和嵌套函数

附录二：术语表

(referer: http://rubycn.ce-lab.net/man/appendix_dic.html)

标识符(identifier)

关键字(keyword)

标签(label)

符号(symbol)

标记, 记号(token)

数据类型(type, data type)

无类型(untype)

变量(variable)

声明(declare)

赋值(assignment)

未赋值变量(unassigned variable)

未声明变量(undeclared variable)

值(values)

直接量(literal)

常量, 常数, 常值(constant)

指数计数法(也有称科学计数法)

定点计数法(犀牛, 亦即是带小数点的计数法)

语句(statement)

语句块(statement block)

复合语句(compound statement)

简单语句, 单行语句(single line statement)

条件(condition)

表达式(expression)

运算符, 操作符(operator)

一元运算符(unary operator), 单目运算符

二元运算符(binary operator)

三元运算符(ternary operator)

运算符优先级(precedence)

运算元, 操作数(operator)

正则表达式(regular expression)

数组(array)

关联数组(associative array)

索引数组

多维数组()

动态数组(dynamic array)与矢量(vector), 动态数组也常称为“变长数组”, 表明它在编译语言中的一个特性: 数组是变长、动态分配内存的, 但数组的引用是不变的。

元素(element)

索引, 下标(index)

异常(exception)

错误(error)

字符串(string)

unicode 字符串(unicode string)

转义序列(escape sequence)

函数(function)

参数(argument)

外部局部变量(external local variable, upvalue)

对于函数 A 的子函数 B 来说, 函数 A 的局部变量, 是函数 B 的外部局部变量。

类(class)

内建类(member class)

继承(inheritance)

多态(polymorphism)

动态绑定(dynamic bind)

封装(encapsulation)

对象(object)

实例(instance)

原型(prototype)

构造器, 构造函数(constructor, constructor function)

属性(property)

用户定义属性

预定义属性 (内部属性 / 方法 / 成员)

事件(event)

事件句柄, 事件处理器, 事件处理代码(event handle)

特性, 性质, 属性(attribute)

特性(feature)

引用(reference)

域, 成员, 字段(member, field)

方法(method)

类方法(class method)

虚方法(virtual method)

纯虚方法, 抽象方法(abstract method)

覆盖(override)

接口(interface)

实现(implementation)

切面(aspect)

切点, 接入点(join point)

织入(wearing)

观察者()

被观察者()

解释器(interpreter)

运行期(runtime)

宿主(host)

上下文(context)

执行环境, 执行上下文(execution context)

代码逻辑行、物理行以及逻辑结构与物理结构:

物理行是你在编写程序时所 看见 的。逻辑行是 Python 看见 的单个语句。

作用域(scope)

闭包(closure)

闭包包括相应函数原型的引用、环境(environment, 用来查找全局变量的表)的引用以及一个由所有 upvalue 引用组成的数组, 每个闭包可以保有自己的 upvalue 值。函数是编译期概念, 是静态的, 而闭包是运行期概念, 是动态的。

全局(global)

局部(local)

全局对象(global object)

全局变量(global variable)

局部变量(local variable)

lambda 运算

“ λ ” (lambda、兰姆达) 是希腊字母, 在计算系统中, 它表示一种运算型式。具体来说, λ 代表一个函数, 因此 lambda 运算不妨直接理解为函数运算。在运算表达式中, 一个运算符其实是一个函数的指代, 因此表达式运算的本质, 就变成了函数运算。

编程范型(Programming paradigm)

多范型语言(Multi-paradigm)

在早期，人们总是专注于某个种类的语言特性，并以命令式语言为主体（尽管 ISP、APL 与 SNOBOL4 等非命令式程序设计语言已经诞生）。但自从 J.Backus 在 1978 年图灵奖获奖讲演中指出了传统过程性语言的不足之后，人们开始把注意力转向研究其它风格、其它范型的程序设计语言。这个词是由 Bjarne Stroustrup 博士在其著作中提出的，最初用于表述 [C++](#) 可以以同时使用多种风格来写程序，比如面向对象和泛型编程。

JScript 词汇表

ASCII 字符集

美国标准信息交换编码（ASCII）的 7 位字符集，它被广泛地用来表示标准的美国键盘上的字母和符号。ASCII 字符集与 ANSI 字符集中的头 128 个字符（0 - 127）完全相同。

Automation 对象

通过 Automation 接口可以被其他应用程序或编程工具使用的对象。

按位比较

对两个数值表达式中相同位置上的位进行的逐位比较。

Boolean 表达式

一个值为 **true** 或者 **false** 的表达式。如果需要，非 Boolean 表达式也可以被转换为 Boolean 值，但是要遵循下列规则：

- 所有的对象都被当作 true。
- 当且仅当字符串为空时，该字符串被当作 false。
- **null** 和 **undefined** 被当作 false。
- 当且仅当数字为零时，该数字被当作 false。

字符编码

用来表示一个集合，诸如 ASCII 字符集，中特定字符的数字。

类

对象的形式定义。类的作用就相当于一个模板，在运行时可以根据此模板来创建对象的一个实例。类定义对象的各种属性并且定义各种方法以便控制对象的操作。

注释

程序员向代码中添加的文本，用来说明代码是如何工作的。在 JScript 中，一行注释通常以 `//` 开头。如果要创建多行的注释，请使用 `/*` 和 `*/` 定界符。

比较运算符

表示两个或更多的表达式或值之间关系的字符或符号。这些运算符包括小于 (`<`)、小于或等于 (`<=`)、大于 (`>`)、大于或等于 (`>=`)、不等 (`!=`) 和等于 (`==`)。

复合语句

用大括号 (`{}`) 括起来的语句序列。复合语句可以被用来在需要使用单行语句的地方完成多项任务。

构造函数

一种 JScript 函数，具有两个特殊的性质：

- 由 **new** 运算符来调用此函数。
- 通过 **this** 关键字将新创建对象的地址传递到此函数。

请使用构造函数来初始化新的对象。

表达式

关键字、运算符、变量以及文字的组合，用来生成字符串、数字或对象。一个表达式可以完成计算、处理字符、调用函数、或者验证数据等操作。

固有对象

固有对象是作为标准 JScript 语言一部分的一种对象。所有的脚本都可以使用这种对象。JScript 中的固有对象包括 Array, Boolean, Date, Function, Global, Math, Number, Object, RegExp, Regular Expression, 和 String。

本地时间

本地时间指的是脚本被执行的服务器或客户机上的时间。

区域设置

对应于给定语言或国家/地区的一系列信息。区域设置影响预定义编程术语所使用的语言并影响一些跟区域设置相关的设置。在下面两种情况下区域设置信息很重要：

- 代码区域设置影响术语，诸如关键字，所使用的语言并定义与区域设置相关的设置，诸如十进制分隔符和列表分隔符、日期格式、和字符排序的顺序等。
- 系统区域设置影响一些与区域设置相关的功能的执行方式，例如，显示数字或将字符串转换为日期等。可以在操作系统所提供的“控制面板”实用程序中调整系统“区域选项”的设置。

null

null 值指出一个变量中没有包含有效的数据。产生 **null** 的原因是：

- 对一个变量显式地赋值为 **null**。
- 包含 **null** 的表达式之间的任何操作。

数值表达式

数值表达式指的是任何值为数字的表达式。这种表达式的元素可以包括关键字、变量、文字和运算符的任意组合，只要此组合能够生成一个数字。在特定的情况下，如果可以的话，字符串也可以被转换为数字。

基本

一种数据类型，是 JScript 语言的一部分并通过值来进行操作。在 JScript 中，数字、Boolean、字符串和函数等数据类型都被看成 primitive。而对象和数组则不是基本数据类型。

属性

对象的一种命名的特性。属性定义了对象的一些特性，诸如大小、颜色和屏幕位置，或对象的状态，诸如可用的或禁止的。

运行时错误

代码运行过程中出现的错误。如果一条语句试图进行非法操作，那么就会导致一个运行时错误。

范围

定义一个变量、过程或对象的可见性。在函数中定义的变量仅在函数内部可见，在调用该函数时并不能保持其值。

字符串比较

两个字符序列之间的比较。除非在进行比较操作的函数中指出，所有的字符串比较操作都是二进制的。在英语中，二进制比较区分大小写；而文本比较则不区分。

字符串表达式

任何值为一个连续字符序列的表达式。一个字符串表达式的元素可以包括返回字符串的函数，字符串文字，**String** 对象，或者字符串变量。

未定义

在变量被创建之后和被赋给值之前分配给该变量的一个特殊值。

全球标准时间 (UTC)

全球标准时间指的是由世界时间标准设定的时间。原先也被称为格林威治标准时间或者 GMT。

用户定义对象

由用户在源代码中创建的对象。

变量

用于按名称来保存并操作值的位置。因为 JScript 的类型是自由的，在一个脚本过程中，单个变量可以保存不同的数据类型。

包装器

一种对象，被创建来为一些其他类型数据提供对象风格的接口。**Number** 和 **Boolean** 对象就是包装器对象的例子。

(转自: <http://python.cn/pipermail/python-chinese/2005-January/007276.html>)

静态类型定义语言

一种在编译时，数据类型是固定的语言。大多数静态类型定义语言强制这一点，它要求你在使用所有变量之前要声明它们的数据类型。**Java** 和 **C** 是静态类型定义语言。

动态类型定义语言

一种在执行期间才去发现数据类型的语言，与静态类型定义相反。**VBScript** 和 **Python** 是动态类型定义的，因为它们是在第一次给一个变量赋值的时候找出它的类型的。

强类型定义语言

一种总是强制类型定义的语言。**Java** 和 **Python** 是强制类型定义的。如果你有一个整数，如果不显示地进行转换，你不能将其视为一个字符串（在本章后面会有更多如何做的内容）。

弱类型定义语言

一种类型可以被忽略的语言，与强类型定义相反。**VBScript** 是弱类型定义的。在 **VBScript** 中，可以将字符串 `'12'` 和整数 `3` 进行连接得到字符串 `'123'`，然后可以把它看成整数 `123`，而不需要显式转换。