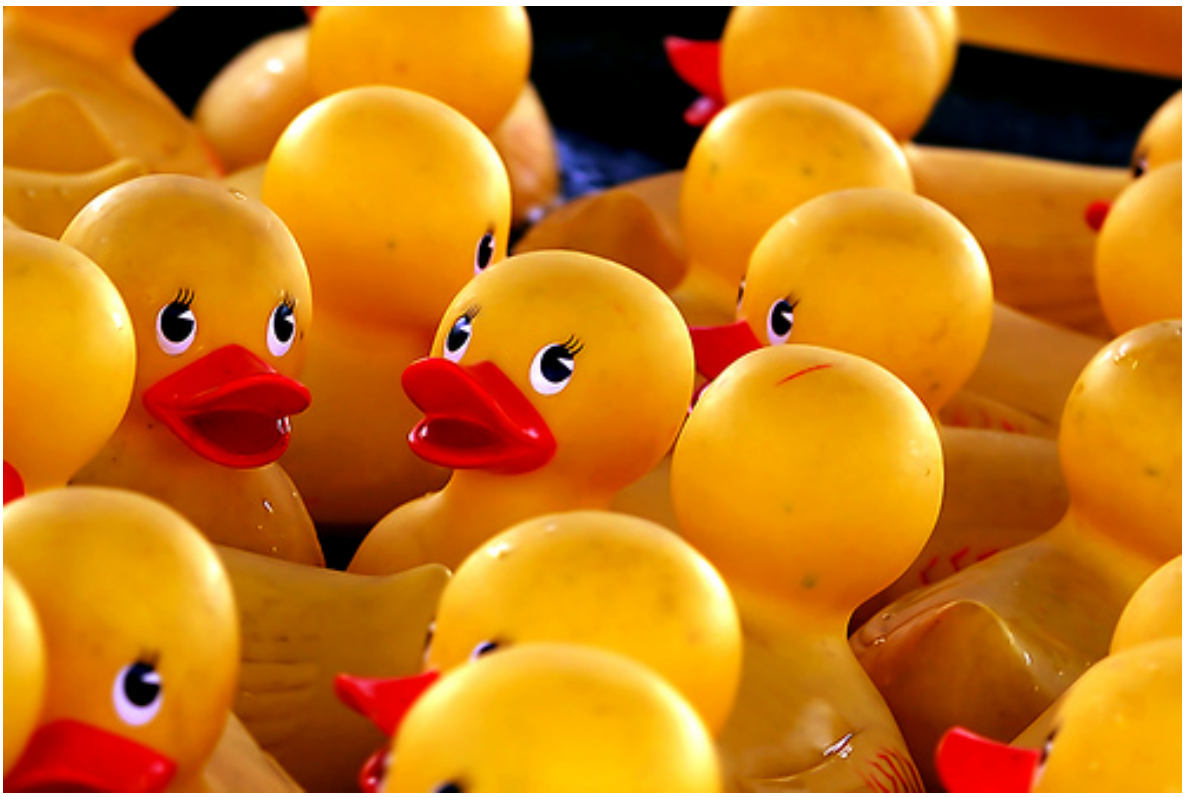
动态函数式语言精髓

《JavaScript 语言精髓与编程实践》精简版



周爱民 著

InfoQ企业软件开发丛书

目录

世界需要一种什么样的语言？
精简版·序

要有光

我从未停止过对语言的思考。

曾经很长的一段时间里，在临入睡前我的脑海中总会响起一种声音“我解决了语言问题”，而睡醒时，我仍觉得自己是无知小儿。编程十余年，我写过《Delphi 源代码分析》，我从中看到了一门语言如何从代码变成操作系统中可以运行的程序。我也写过《大道至简——软件工程实践者的思想》，我在其中说“语言不过是（工程的）细微末节”，而成书之后，我便又投入了新的、当前你所看的这本书的撰写之中。

语言于我，是一个死结。我一直在寻求尽头，或展望于将来，或求源于过往。我在一道大河的中间，前后观望，时而俯首所得的，不过是一掬破碎的倒影。

倒影中，还是我的迷惘。

我也在实现着一种语言，我用 JavaScript 来做这件事情。我选择它只是因为熟悉，以及它足够的表现力。我用它来做面向对象、面向切面、面向接口编程等等的尝试，也用它来模拟操作系统的调度机制，或构架业务系统的技术框架。在另一些代码中，我也看到过用 JavaScript 代码来模拟 CPU 的指令流水线，或者实现真实的虚拟机……但我认为没有必要向您去解说一种语言是何等的强大（或者专业、优秀与特异），完全没有必要。

因为，它不过是一种语言。一种语言只是一种思想的表现，而不是思想本身。

什么才是决定语言的未来的思想呢？或者我们也可以换个角度来提出这个问题：世界需要一种什么样的语言？

特性众多、适应性强，就是将来语言的特点吗？我们知道现在的 C#与 JAVA 都在向这条道路前进。与特定的系统相关，就是语言的出路吗？例如曾经的 VC++，以及它面向不同的平台的版本。当然，在类似的领域中，还有 C，以及汇编等等……

我们回顾这样的例举，其实都是在特定环境下的特定语言，所不同的无非是环境的大小。这其实也是程序员的心病：我们到底选 Windows 平台，还是 Java 平台，或者 Linux 系统，再或者是……我们总是在不同的厂商及其支持的平台中选择，而最终这种选择又决定了我们所使用的语言。这与喜好无关，也与语言的好坏无关，不过是一种趋利的选择罢了。所以，也许你是在使用着的只是一种“并不那么‘好’”，以及并不能令你那么开心地编程的语言。你越发辛勤地工作，越发地为这些语言摇旗鼓噪，你也就离语言的真相越来越远。

当然，这也不过是一种假设。但是，真相不都是从假设开始的么？

语言有些很纯粹，有些则以混杂著称。如果编程世界只有一种语言，无论它何等复杂，也必因毫无比较而显得足够纯粹。所以只有在多种语言之间比较，才会有纯粹或混杂这样的效果：纯粹与混杂总是以一种或多种分类法为背景来描述的。我们了解这些类属概念的标准、原则，也就回溯到了种种语言的本实：它是什么、怎么样，以及如何工作。这本书，将这些分类回溯到两种极端的对立：命令式与说明式，动态与静态。我讲述了除开静态语言（一般

是指类似 C、C++、Delphi 等的强类型、静态、编译型语言）之外的其它三种类型。正是从根底里具有这三种类型的特性，所以 JavaScript 具有令人相当困扰的混合语言特性。分离它们，并揭示将它们混沌一物的方法与过程，如历经涅磐。在这一经历中，这本书就是我的所得。

多年以来，我在我所看不见的黑暗与看得见的梦境中追寻着答案。这本书是我最终的结论，或结论面前的最后一层表象：我们需要从纯化的语言中领悟到我们编程的本质，并以混杂的语言来创造我们的世界。我看到：局部的、纯化的语言可能带来独特的性质，而从全局来看，世界是因为混杂而变得有声有色。如果上帝不说“要有光”，那么我们将不能了解世象之表；而世象有了表面，便有了混杂的色彩，我们便看不见光之外的一切事物。我们依赖于光明，而事实是光明遮住了黑暗。

如同你现在正在使用的那一种、两种或更多种语言，阻碍了你看到你的未来。

语言

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，它（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。但我希望你由此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，甚至比你我的生命都顽强。但它只是愚人的器物，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。

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 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……但是在重新开局之前，你是否注意到：在上述过程中，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。就如同一副新牌只有四套 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。当你从牌盒里把它们拿出来的时候，无论它们是以哪种方式分类的，这副牌都不混乱。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去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，但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，都在寻求纯质银弹[1]或混合毒剂[2]。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起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。

更进一步地说，前者在产生新的分类法以试图让武器变得完美，后者则通过混淆不同的分类法，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其他分类法混在一起，于是变成了我们这本书所要讲述的“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）[3]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。

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[4]。

---

[1] 参见《人月神话》，美国弗雷德里克•布鲁克斯（Frederick P. Brooks, Jr.）著。
[2] 参见《蓝精灵》，比利时皮埃尔•居里福特（Pierre Culliford，Peyo）著。
[3] 保罗•格雷厄姆是计算机程序语言Arc的设计者，著有多本关于程序语言，以及创业方面的书籍。
[4] 这是一本软件工程方面的书，但往往被人看成是医学书籍或有人希望从中求取养生之道。

其实这很冤枉。我想表达的意思不过是：如果你想把一副牌理顺，最好的法子，是回到它的分类法上，要么从A到K整理，要么按四个花色整理[5]。毕竟，两种或更多种分类法作用于同一事物，只会使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）它们。

特性与技巧

无论哪种语言（或其他工具）都有其独特的特性，以及借鉴自其他语言的特性。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，但却是其他语言的思想库。

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。直到昨天[6]，JavaScript的创建者还在小心翼翼地增补着它的语言特性。而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。

所谓技巧，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，是因为难以控制，并且容易破坏代码的可读性。

技巧的使用取决于具体的目标，以及“需要、能够”维护这个代码的人对技巧的理解。这包括：

- 技巧是一种语言特性，还是仅特定版本所支持或根本就是 BUG；
- 技巧是不是唯一可行的选择，有没有不需要技巧的实现；
- 技巧是为了实现功能，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该技巧，那么连代码本身都失去了价值，更何论技巧存在这份代码中的意义呢？

所以本书中的例子的确要用到许多“技巧”，但我一方面希望读者能明白，这是语言内核或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和理论，以及活用的方法。

然而对于很多人来说，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，本书所讲述的一切，都只不过是“正常的方法”；在其他类型的一些语言中，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。

所以你最好能换一个角度来看待本书中讲述的“方法”。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

[5] 不过这都将漏掉了两张王牌。这正是问题之所在，因为如果寻求“绝对一分为二的方法”，那么应该分为“王牌”和“非王牌”。但这往往不被程序员或扑克牌玩家们采用，因为极端复杂性才是他们的毕生目标。

[6] 在JavaScript 2——这种把银弹涂上毒剂以试图用单发手枪击杀恐龙的构想发布之前的“昨天”。

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。

这本书

本电子书是由电子工业出版社出版的《JavaScript 语言精髓与编程实践》一书的迷你版本。但本电子书不包括该纸质书中的有关 JavaScript 语言的绝大部分内容，而只是摘选了其中关于语言范型的论述，并由此组织成文。故本电子书定名为《动态函数式语言精髓》。

本书的部分内容曾以电子文档的形式发布为《主要程序设计语言范型综论与概要》。

本电子书由作者周爱民先生亲自摘选编撰，由 InfoQ 中文网站独家在线发布。在此，感谢电子工业出版社、博文视点资讯有限公司（武汉分部）予以许可。

导读

《JavaScript 语言精髓与编程实践》这本书，最初的名字是叫《动态函数式语言精髓与编程实践》的，这才是我写那本书的原意。确切地说，我并非是想讨论 JavaScript 作为一种语言工具的用法或特性。我更多地是希望用一种简洁的语言来讨论动态语言、函数式语言。而为了给这些语言范型以参照，以及讲述多范型如何“杂凑”在一起，我也讨论了 JavaScript 中的命令式语言特性。

因此，事实上《JavaScript 语言精髓与编程实践》一书是假 JavaScript 语言之力，讨论了我们常用的、主要的程序设计范型。而这也是该书难读的根源——很少有人会以一门确切的语言来讨论多种语言范型。尤其在使用 JavaScript 的开发者群体里，深入了解该语言的本来就不多，研究语言特性的就更少了。

在本电子迷你书中，我摘引了《JavaScript 语言精髓与编程实践》一书有关语言讨论的关键章节（主要在第三、四、五章），从 JavaScript 无关的角度综论这些语言的产生发展，以及特性的概要。作为纯语言学范畴的讨论，读者在本摘引中不需要过多地了解 JavaScript，也不必深究某种语言的细节，作参考文论来读，便是不错。

本文主要有“命令式语言”、“函数式语言”和“动态语言”三个部分，均精减自《JavaScript 语言精髓与编程实践》一书的相应内容。本文最末一个部分，对几种语言作一综述，一孔之见而已。

命令式语言

《JavaScript 语言精髓与编程实践》：第 3 章

1、命令式语言的发展综论

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。

本节将首先讨论程序的本质，并从这个本质出发，以另一种分类法对程序语言做出分类：命令式语言和说明式语言。《JavaScript 语言精髓与编程实践》一书基于该分类法讨论 JavaScript 的非函数式语言特性，有关内容组织如表 3-1 所示。

表 3-1　基于程序本质的分类

分类	子类	章节
命令式	冯·诺依曼（结构化编程）	第 3.2 节
	面向对象（面向对象编程）	第 3.3 ~ 3.4 节
说明式	函数式	第 4 章
	（其他）	

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。简单地说，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，对结构的解释包括三个部分：程序的控制结构、组织结构和数据结构。所谓控制结构，即是顺序、分支和循环这三种基本程序逻辑；所谓组织结构，即是指表达式、语句行、语句块、过程、单元、包等；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。

那么命令式语言与结构化编程在概念上有多大的相关性呢？事实上它们并不是同一层面上的概念，前者讲的是运算范型（表达为语言），后者讲的是一种程序设计与开发的方法。因此在结构化编程的整个知识域中，其实仅有“数据结构”与“命令式语言（这一编程范型）”在同一层面上。而所谓“数据结构”，即是命令式语言所关注的“存储”。

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他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——这里是特指“数据结构（或类型系统）”。表 3-2 说明在 Intel 计算机体系中“数据结构”上的简单抽象。

表 3-2 “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（*注 1）	array, string, struct, ...
有关系的数据片断	存储地址	指针、结构、引用	pointer,tree, ...

*注 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”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）”。

命令式语言在运算上也基于上述的“存储结构”来进行算法设计。例如表检索，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”，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，仍旧是表 3-2 中对“数据表现形式”的设定。

像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”。正是大师们在“数据的结构”上的不懈努力，成就了C/Pascal这样的结构化编程语言[7]、Windows、Linux/Unix这些伟大的操作系统，以及Oracle、MS SQL Server等这些数据库系统[8]。

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）、高级语言（3GL），以及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。

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（结构类型）”。但是一方面，结构体

[7] 结构化程序设计中的“结构”并不是语言概念中的“结构类型（struct）”，二者没有必然的联系。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。这一分析、设计的过程被称为结构化，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），来自于上述分析、设计阶段的数据建模。

[8] 结构化程序设计绝不是“数据结构”一言可概之的，但这里我们重在强调语言特性，而非编程方法的历史与演进。

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，除了关系型数据库之外，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。

抽象层次过低带来的问题至少包括三个方面。

其一，结构体与实体直接相关，并且将这种相关性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。它只是忠实地反映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 `TRoom` 这个结构体应用于一个实际系统中：对于工程辅助设计（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，其他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。换而言之，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，以最终写出《某某系统未公开文档技术大全》之类的著作之外，并未解决根本问题。

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和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，或者使其他算法被应用到这个旧的结构。

结构体的设计直接面向存储，正是这种过低的抽象层次使重用性大大地降低。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构造（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，那是靠另外的一套法则去保障的，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，泛型系统像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。

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”带来的其他问题。这是因为，这三点正是“面向对象”所解决的主要问题：

- 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。这些可见性标识见表 3-3。

表 3-3　面向对象系统常见的可见性设定

可见性	含义	备注
published	已发布，面向特殊系统（例如 IDE）的	（*注 1）
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	（*注 2）
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	（*注 2）
private	私有，访问限于该成员所属的类型	

*注 1：在 Delphi 中，该可见性仅面向可视化组件库、RTTI 和 IDE。

*注 2：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”）的细节。

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）。简单地说，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。

下面用一个较为复杂的示例，综合说明面向对象系统的这三种特性。

```
/**
 * programming language: Delphi
 * (以下形式代码中，斜体字表明一个系统的或外部的处理例程)
 */

// 步骤 1. 基类及其表达的运算逻辑
type
```

```
  // 封装性：值的表达形式，以及它与其他值的计算方法被封装在类的内部，是外部不关心的逻辑
  TCalcData = class(TObject)
    function GetValue: integer; abstract;
    function CalcValue(y: integer): double; abstract;
    // ... 与运算类型相关的、上述方法的不同版本（overload;）

    function GetResult: double;
  end;

  // 多态性：Machine 负责处理的都是 TCalcData，而不必关心真实的子类类型
  TCalcMachine = class(TObject)
  privated
    FLastObject: TCalcData;
    property LastObject: TCalcData read get_last write set_last;
  public
    function calc(obj: TCalcData): TCalcData;
  end;

// 步骤 2. TCalcData 的子类，表达各自子系统对数据的理解
type
  // 继承性：对象系统如何继承以及在子类中如何实现，与(其他的)外部逻辑是无关的
  TIntegerData = class(TCalcData)
    ...
  end;

  TDoubleData = class(TCalcData)
    ...
  end;

// 步骤 3. 由步骤 1 所决定的算法逻辑
function TCalcMachine.calc(obj: TCalcData): TCalcData;
begin
  Result := create_data_instance(LastObject.CalcValue(obj.GetValue));
  LastObject := Result;
end;

// 步骤 4. 外部业务逻辑（假设外部系统总是能显示 double 值），IO 操作等
var x, y: TCalcData;
var mac: TCalcMachine;
...

repeat
  x := get_data_instance(get_data_from_input_source);
  y := mac.calc(x);
  echo_data_to_output_dest(y.getResult);
until (query_total);
...
```

在这个示例中，步骤 1 中 TCalcData 与 TCalcMachine 的类设定决定了系统如何计算数据，该计算方法实现在步骤 3 中。但是，步骤 1、步骤 3 与步骤 2 之间并不存在逻辑上的相关性，因为步骤 2 的作用在于通过继承性扩展系统，而不影响既有系统的逻辑。至于步骤 4，是在确定的“对象系统＋对象系统间的逻辑”之外进行的系统 IO 操作，这些操作与既有对象系统也是无关的。如此一来，我们把“运算数据的表达”、“数据间的运算规则”、“Machine 如何计算”以及“应用与外部系统如何交互”这些逻辑都分离开了。

我们看到，“对象”无疑是比“结构体”更高层次的数据抽象（结构）。它的基础，正是“结构确定（步骤 1），则算法确定（步骤 3）”。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（步骤 2），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：如果对象基类的抽象不合理，或者继承

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。

“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“类继承树”的映射。由此可见，在面向对象系统中，对象基类及其继承树是对数据抽象的表达，而这种抽象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。

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）”，等等。而在具体语言中，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，例如 Pascal 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度确定了。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。

这个问题的根源仍然在于抽象程度过低：我们确定了运算目标（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` 与 `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能力，而不暴露这种能力的实现方法和基于的数据特性——这里用“数据体”而不是“对象”，是因为 Interface 并不关注“接口系统实现者”的数据结构特性——例如使用“对象 / 类类型”来实现接口，或用“结构类型”来实现接口（尽管具体的语言中，这是与确定的数据类型相关的）。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，可能只是如下的 Interface（体系描述中不应强调交互的数据类型）：

```
ISearch = interface
  function 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，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：在远在银河系之外的星系，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想要的东西。

这就是Web Services。Web Servcies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这些虚头八脑[9]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。

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，因此可以毫无疑问地说，“面向对象编程”也是一种命令式语言。这有两

[9] 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”？

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《程序设计语言概念》（COPL，Concepts of Programming Language）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，上述语言中的“对象”仍然是基于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（既是设计，也是实现），便是在这种限制下的产物。

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”，后者关注“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（见表 3-4 中字体加粗的部分）：

表 3-4　程序设计语言的分类及其与计算机语言的关系

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**	（*注 1）
			数据流式	**Id, Val**	
			逻辑式	**Prolog, VisiCalc**	
		命令式	**冯·诺依曼**	**Fortran, Pascal, Basic, C, ...**	（*注 2）
			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	

*注 1：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。

*注 2：一般概念下的命令式语言，或者称为结构化程序设计语言。

不过，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 2005 年 12 月，Christopher Diggins[10]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 年 2 月）。与此同时，IOP作为一种新的理念，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“在共同的规约描述下的（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 并不是一种命令式语言，因为它缺乏“基于结构”这样的基本特性；IOPL 更像是一种说明式语言，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，实际上不单单是陈述架构，也陈述了实现算法。

[10] O’Reilly 2005 年出版的《C++ Cookbook》一书的作者，是一种支持IOP的Heron语言的创建者。

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）”，却是更接近“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[11]：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”。

2、语法及作用域问题

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 foo() {
  // 代码行
  // 代码行
}
```

最后，更多的函数、语句与代码块则构成了文件——而这些，通常也称为代码的物理结构。这个物理结构中，文件通常对应于“单元（unit）”，后者是部署概念上的逻辑结构。

不同语言对“单元（unit）”以上级别的模块的解释与处理都不相同。例如对于 “程序（program）”的解释与处理，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。而 JavaScript 没有与此类似的约定，脚本引擎对载入的每一块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，或引用了一个从未声明过的变量。而且 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。

更高层次上的、部署概念的逻辑结构还有包、命名空间等。一些语言内置了处理这些概念的语法，例如 Java 或 C#，另一些则完全没有实现，或约定为某种规则。

回到代码的组织结构，我们通常可以简单地解释为“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。

结构化语言中，代码块的语法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：

[11] 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。

```
/**
 * 示例 1：代码块一与代码块二平行
 */

// 代码块一
if (true) {
  // ...
}

// 代码块二
while (true) {
  // ...
}
```

```
/**
 * 示例 2：代码块一嵌套代码块二
 */

// 代码块一
if (true) {
  // ...

  // 代码块二
  while (true) {
    // ...
  }
}
```

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这两个代码块之间的相互作用关系。这种关系是通过 “语法作用域的级别”来控制的（以 JavaScript 为例，其语法作用域有四种：表达式、语句/批语句、函数、全局，也相应存在“等级 1~4”这四种“语法作用域的级别”）。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，一般处理成语法上的违例，或者理解为“平行”的关系。

第一个规则的应用是常见的。例如 `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` 语句可以包含 `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。至于第二个规则，在我们写函数时就已经经常使用了：

```
function foo() {
  // ...
  if (true) {
    // ...
  }
}
```

但是，对于第三个规则，我们就需要较为详细地说明一下。在下面的例子中，示例 3 与示例 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，从而在示例 3 中将形式上的嵌套关系理解为平行关系：

```
/**
 * 示例 3：该示例在效果上与示例 4 等效
 */

// 代码块一
if (true) {
  // ...

  // 代码块二
  function foo() {
    // ...
  }
}
```

```
/**
 * 示例 4
 */

// 代码块一
if (true) {
  // ...
}

// 代码块二
function foo() {
  // ...
}
```

当这些作用域关系一旦存在，代码在形式上就可以最后被理解为“块的顺序执行”。这与术语“命令式”存在惊人的一致性。“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。”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。

我们在前面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代码“微缩”一下，你就会发现，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。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。

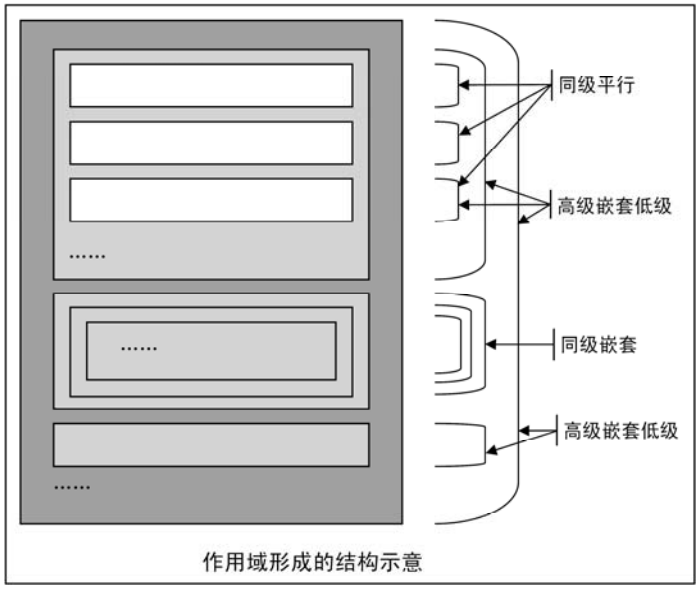
最自然的想法当然是“GOTO”，但 GOTO 语句带来的灾难与它解决的问题一样多。于是，更进一步的想法是：如果我们对上面的代码做足够的抽象（例如分析它们的语法作用域），并对每一个抽象设计一些类似 GOTO 功能的语句，那么必然得到足够的灵活性，而又避免了 GOTO 的滥用。

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 continue、break、return 和 throw 等等。

语法及其作用域带来了构造命令式语言系统的全部思想，这包括：

- 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。

从与语言和语法无关的、形式化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下图所示的形式。



作用域形成的结构示意

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。因此，尽管在理论上Bohm和Jacopini早已证明过类似这样的灵活性不是必须的[12]，E. W. Dijkstra则更进一步地指出灵活性对系统带来的危害[13]，然而在既存的语言中（即使已经声称“消灭了GOTO

[12] Böhm, Corrado, and Jacopini Guiseppe. "Flow diagrams, Turing machines and languages with only two formation rules." Communication of ACM, 9(5):366-371, May 1966.

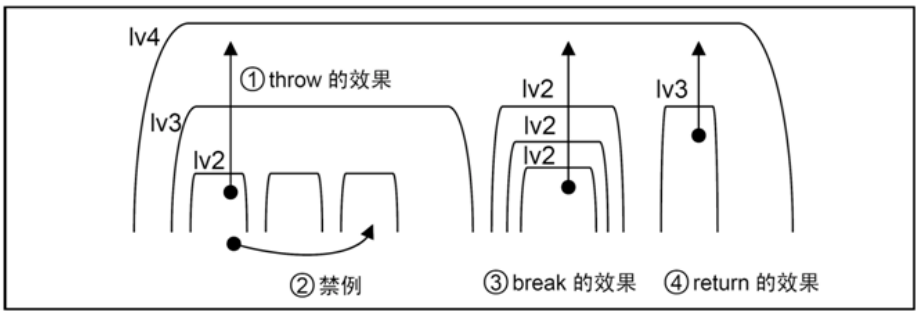
[13] Edsger W. Dijkstra. "Letters to the editor: Go to statement considered harmful". Communications of the ACM, 11(3):147–148, March 1968.

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。这遵循着一个简单的基本原则：

- 高级别的流程变更子句（或语句）可以跨越低级别的作用域，反之则不成立。

这里要进一步强调的是：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（注意在域作用形式上是前一图例的倒置）[14]如下图所示。



结构化程序设计语言中的流程控制设计

3、变量作用域及生存周期问题

我们为什么讨论语法作用域呢？因为语言中存在一个问题，就是在程序执行过程中的“变量（内存分配）”与语法语义上的“变量（词法标识）”是否一致。例如说一个语法作用域表明“变量 A”是一个（函数内的）局部变量，而实际运行过程中，这个“变量 A”却指向了全局变量，那么就显然存在了二者的不一致。

通常语法作用域与变量作用域是一致的，这称为“静态语义”。这意味着无论是程序员，还是编译器，都可以从代码上下文中分析出一个变量标识符的作用域效果。在语法作用域与变量作用域一致的情况下，变量与其数据类型、内存分配等都是可以在执行前预知的，所以也称为“静态绑定”。

但是，JavaScript 却并不是一种静态语义的语言（这个与后面要讲的动态语言特性有关），所以事实上用 JavaScript 语言书写的代码的作用域是不确定的。动态作用域绑定的一个事实（或者效果）是：语法作用域与变量(数据)作用域不一致。关于这一点，在上一个小节中已有一些叙述。JavaScript 是现今不多见的、仍然保持了动态作用域绑定特性的语言，因此这种要把作用域区分为两种情况来讨论的，也不太多见。

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（和其他语法结构）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，后者是实际的占有[15]。

---

[14] 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。

[15] 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

“语法作用域”与“变量作用域”二者之间的区别，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。正如圈地与割据并不是等义的——我们可以将一个区域划分为九块却只有其中六块有人占领，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，并没有“语句”这个级别[16]。

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
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 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[17]：

```
{
  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
```

综合上述对第三方语言的考察，我们可以看出：在本书中讨论的JavaScript并没有“语句”级别的变量作用域[18]——它没有提供类似上述示例的语法效果。从这里我们也看到了两点事实：

- 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是：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

非实现方式）上的“语法作用域”、“变量作用域”来命名它们。

[16] JavaScript 1.5 以上版本中开始有了非常丰富的语言特性，通过let关键开始支持表达式与语句级别的变量作用域。但对于ECMA Script来说，则只对这些特性只有不完整的描述。至于“符合”ECMA Script规范的JScript，则完全不支持语句级别的变量作用域。

[17] 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。

[18] JavaScript高版本中的`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。

会逸出到同级的其他结构中去。

现在你也许会问，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，因为二者都是从同一个角度来看同一个问题。但是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被释放。在 JavaScript 中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”这个机制时采用的方法所决定的，是实际实现中的一种选择，而不是语法语义上的约定。

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 在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——我们在形式上并没有改变了它的语法作用域（位置）、变量作用域（可见性），但事实上它的生存周期却被改变了。

4、原型继承的基本原理与实质

“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”概念的异乎寻常的多种解释中并不常用的一种，因为其他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[19]。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。

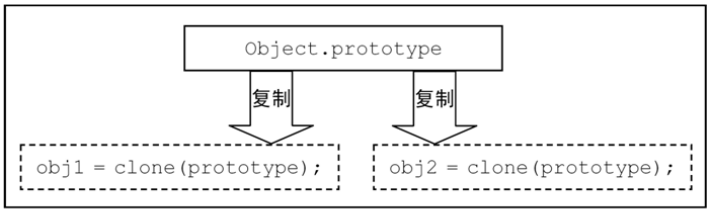
所谓原型其实也是一个对象实例。原型的含义是指：如果构造器有一个原型对象 A，则由该构造器创建的实例（Instance）都必然复制自 A。这里的“复制”就存在了多种可能性，由此引申出了动态绑定和静态绑定等问题。但我们先不考虑“复制”如何被实现，而只需先认识到：由于实例复制自对象 A，所以实例必然继承了 A 的所有属性、方法和其他性质。

“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”在本质上的不同。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

[19] 也因此JavaScript被称为“无类语言”。

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。

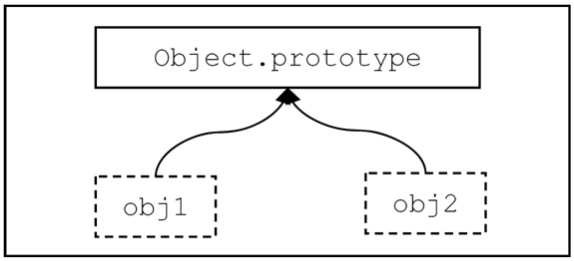
基于原型的对象的“构建过程”可以被简单地映射为“复制”。如下图所示：



原型继承的实质是“复制”

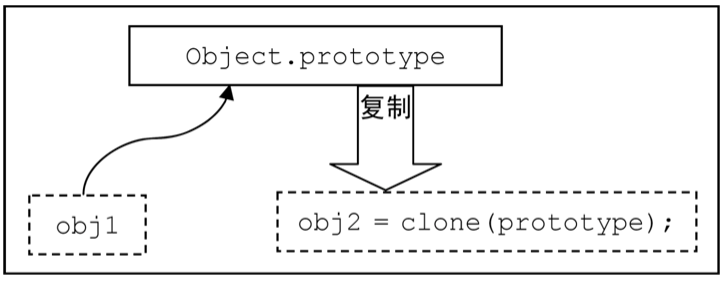
但这个图例假设每构造一个实例，都从原型中复制出一个实例来，新的实例与原型占用了相同的内存空间。这虽然使得 obj1、obj2 与它们的原型“完全一致”，但也非常不经济——内存空间的消耗会急速增加。

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下图所示：



使用写时复制机制的原型继承

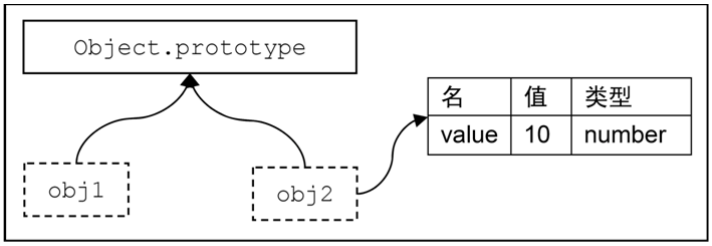
这时，我们只要在系统中指明 obj1 和 obj2 等同于它们的原型，而读取的时候只需要顺着指示去读原型即可。当需要写对象（例如 obj2）的属性时，我们就复制一个原型的映像出来，并使以后的操作指向该映像就行了。这大致就变成了下图所示的情况：



“写操作”在使用写时复制机制的原型继承中的效果

这种方式的优点是我们只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。但此后就不再有这种开销了，因为访问映像与访问原型的效率是一致的。不过对于经常进行写操作的系统来说，这种法子并不比上一种法子经济。

而 JavaScript 采用了另一种法子：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映像中。这样一来，在初始构造该对象时，局面仍与上图“使用写时复制机制的原型继承”一致，但写对象属性（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。如图：



JavaScript 使用读遍历机制实现的原型继承

这样，由于 obj2 仍然是一个指向原型的引用，因此在操作过程中也没有与原型相同大小的对象实例创建出来。这样，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。而在读的时候，则可以顺着继承链，从 obj2、Object.prototype 中读取属性值。显然这种基于原型继承的存取规则，其实是与“对象是什么”没有关系的，它基于上述的数据结构约定。不过，由此带来的唯一一个问题是：存取实例中的属性，比存取原型中的属性效率要高。而且随着原型继承的层次变得越深，则存取原型中的属性的效率就越来越差。

与原型继承相关的，就是通过原型修改来影响对象系统。简单地说，就是要使上述的 obj1、obj2（以及更多 object）的特性发生变化，事实上我们可以通过修改 Object.prototype 来影响它们。而如果 JavaScript 是一种“静态的语言”，那么通过这一过程（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。

真的是这样吗？

综合前面所述内容，我们可以明确地说：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、接口及实例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、好像、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")）”[20]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。

所以原型继承的实质其实是从无到有的一个过程。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。

[20] Chistophe Dony, Jacques Malenfan, Daniel Bardou, "Classifying Prototype-based Programming Languages".

5、原型继承的问题与继承方式的选择

JavaScript 的原型继承依赖关系数组和函数式语言两方面特性；另外，原型重写与原型链维护等问题还涉及到动态语言特性。所以尽管原型是一种简单的对象系统实现，但在 JavaScript 中，具有相当迷人（这里具有双重含义）的性质。

它存在什么问题呢？除了在《JavaScript 语言精髓与编程实践》中提到的：

- 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，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，以及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——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、以 Web 浏览器端为主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运算）。

类抄写是弥补这一缺陷的有效方法，它与原型继承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。

除此之外，原型继承时中的“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如此。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。由此带来的问题实际上是较为严重的，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。这其实又走回了老路：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 本身的优点也正是它的缺点。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，因此应该更深入地学习 JavaScript 中不同语言的精髓。正是前者导致业界热推的所有 AJAX 实现方案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。

函数式语言

通常来讲，函数式语言被认为是基于“数学函数”的一种语言。当我们开始用数学领域中的抽象概念来解释函数式语言时，问题被放大（或缩小、聚焦）为下面两个描述[21]：

- 数学函数是集合 A（称为定义域）中成员到集合 B（称为值域）中成员的映射；
- 函数式程序设计是通过数学函数的定义、应用的说明和求值完成运算过程的。

第一句话基本上等于什么都没说，它的含义完全等同于“函数＝从问题中找到答案”。而第二句话的“定义和应用说明”基本上等于第一句话，所以相当于说：函数式程序设计是“计算函数”——还是等于什么也没有说。

但是这些古怪的文字的确是在阐述函数式语言的精髓。为了减轻你的痛苦（但绝非轻视你的智商），我换个说法来陈述它们：如果表达式“1+1=2”中的“+”被理解为求值函数，那么所谓函数式语言，就是通过连续表达式运算求值的语言；既然上面的表达式可以算出结果“=2”，那么函数式语言自然也可以通过不停地求值找到问题的答案。

1、函数式语言基础

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。

其中的原因之一，在于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，不应该这样写代码[22]：

```
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好之苦并幡然醒悟。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))
    ((ATOP lis2) NIL)
    ((equal_lists (CAR lis1) (CAR lis2))
      (equal_lists (CDR lis1) (CDR lis2)))
    (T NIL)
  )
)
```

[21] 基本概念引用自《程序设计语言原理》（Robert W. Sebesta著），但并未复录原文的概念陈述。
[22] 《程序设计实践》Brian W. Kernighan和Rob Pike著，裘宗燕译。

然而答案是：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）。由此看来：不同语言中所谓的“良好风格”看起来是并没有统一标准的。

所以说，语言风格的好坏只是判断“是否连续运算”的一个并不要重要的方面。

2、为什么常见的语言不赞同连续求值

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。在这一体系的程序设计观念中，我们应这样写代码：

```
var desktop = new Destktop();
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，然后再参与 CPU 运算。于是我们得到了结论，汇编语言应该这样写：

```
MOV EAX, $0044C8B8
CALL @InitExe
```

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•诺依曼体系下，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。

综合上一小节的讨论来看，一方面，冯•诺依曼体系对存储的理解从根本上规范了我们

的代码风格；另一方面，语言环境是风格限定与编程习惯形成的重要前提。

因此对于一种语言来说，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。从形式上讲，如果我们以过程式代码的风格来看 LISP 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“这样一种”复杂的代码风格，如同冯·诺依曼体系需要“那样一种”风格一样。

3、函数式语言的渊源

上述 LISP（这种函数式语言）的代码风格所表达的基本语言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。在连续运算过程中，无需中间变量来“寄存”。因此从理论上来说：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？或者说，这些影响到函数式语言的代码风格的特性是什么呢？要了解这一问题的实质，需要更远地回溯“函数式”语言的起源。我们得先回答一个问题：

这种语言是如何产生的呢？

1930 年前后，在第一台电子计算机还没有问世之前，有四位著名的人物展开了对形式化运算系统的研究。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。

在 1936 年，图灵提出了现在称为“图灵机”的形式系统。图灵机概念中提出了通过 0、1 运算系统来解决复杂问题。接下来，在 1939 年，阿坦纳索夫研制成功第一台电子计算机ABC，其中采用了电路开合来代表 0、1，运用电子管和电路执行逻辑运算。再接下来，在 1945 年，冯·诺依曼等人基于当时计算机系统ENIAC（Electronic Numerical Integrator And Computer，电子数字积分计算机）的研究成果，提出了EDVAC体系设计[23]，以及其上的编码程序、纸带存储与输入。该设计方案完全实现了图灵的科学预见与构思[24]。

我们现在最常见的通用编程环境，就是构架于冯·诺依曼在 EDVAC 中的设计，该设计包括五大部件：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 I/O 则依赖于 0、1 存储。因此基于冯·诺依曼体系架构的程序设计语言，必然面临这样的物理环境——具有存储系统（例如内存、硬盘等）的计算机体系，并依赖存储（这里指内存）进行运算。后来有人简单地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”。因此相对于基于图灵机模型提出的运算范型，阿隆左·丘奇所提出的运算系统更加趋近“运算才是本质”观点。

这是一种被称为Lambda演算的形式系统。这个系统本质上就是一种虚拟的机器的编程语

[23] 《存储程序通用电子计算机方案——EDVAC（Electronic Discrete variable Automatic Computer，离散变量自动电子计算机）》是一份设计方案，而非（当时的）物理实现。EDVAC方案直到 1950 年以后才被实现。

[24] 电子计算机的历史一直存在很多争议，如今这些争议已经被澄清。这一部分的文字请参见袁传宽教授在《人物》杂志 2007 年 10 月和 11 月期中的一组文章《计算机世界第一人——艾兰·图灵》和《被遗忘的计算机之父——阿坦纳索夫》。

言——而不是虚拟的机器，它的基础是一些以函数为参数和返回值的函数[25]。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。大约在冯·诺依曼等人的EDVAC报告提出的十年之后，一位 MIT 的教授John McCarthy[26]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。该语言其实就是对阿隆左·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（类似 Lambda 演算的）连续运算的。

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说明式语言？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。如同函数式与命令式语言，这些分类之间存在着本质的差异。

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（在物理特性上的、完美的）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实现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.NET或JVM的、能够进行某些函数式运算的虚拟机环境[27]。

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，还具有一些其他的性质。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。

首先，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。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

[25] 《函数式编程另类指南》（Functional Programming For The Rest of Us），Vyacheslav Akhmechet著，lihaitao译。
[26] John McCarthy被称为人工智能之父，是 1971 年（第 6 届）图灵奖得主。
[27] 自.NET 3.0 开始，C#开始支持Lambda表达式特性；而JVM中，则要等到Java 7 以后。

函数 B 的指针。但是这样的指针显然可能指向另一个进程空间的地址，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。同时，基于地址指针值进行的运算，也带来了“内存访问违例”的隐患。

当 JavaScript 中的函数作为参数时，也是传递引用的，但并没有地址概念。由于彻底地杜绝了地址运算，也就没了上述的隐患。“函数调用”实质上是一个普通的运算符，因此所谓“传入参数”可以被理解为运算元。由此的结论是，（作为“传入参数”的）函数只有运算元的含义而没有地址含义，“函数参数”与普通参数并没有什么特别不同。

其次，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，但与函数式语言也存在根本不同。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，在函数执行结束后，所占用的栈被释放。因此函数内的数据不可能被保存。

而在 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。在函数内保持数据的特性被称为“闭包（Closure）”，其显而易见的好处是：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）赋给实例的方法就可以使用这些数据进行运算；而在多个实例间，由于数据存在于不同的闭包中，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。

第三，运算对函数外无副作用，也是函数式语言应当实现的一种特性。“无副作用”这一特性的含义在于：

- 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他数据的值（例如全局变量）；
- 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。

对于一个封闭系统来说，也是可以做到该封闭系统对外无副作用的。例如对象系统可以作为一个独立系统，一个对象实例的方法也可具有与此相当的特性：既不必影响该对象之外的其它对象，也不必直接影响对象的成员。当把“不在方法内修改对象成员”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）向外暴露系统，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。

3、从运算式语言到函数式语言

1、运算式语言

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，其基本运算模型就是：

- （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 开始引入的“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。因此我们应该认识到，函数式语言中所谓的“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。

对于运算的过程来说，显然表达一种连续运算的方法并不必是自然语言中的语句，也可以是数学运算中的表达式。从这个角度来看，我们事实上也可以将通用语言中的语句改写成表达式，例如用三元表达式来替代条件分支语言，用函数递归来替代循环语句。事实上，在 JavaScript 中的对象构造、函数与方法的调用等，本质上都是表达式运算，而非语句。

而当我们考察语言中的各种运算的结果类型时，我们会得到一个令人惊讶的结论：所有的运算都产生“值类型”的结果值。正因为“运算都产生值类型的结果”，且“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”[28]。

我们知道，计算机其实只能表达值数据。任何复杂的现象（例如界面、动画或模拟现实），在运算系统看来其实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，至于如何展示，则是另一个物理系统（或其他运算系统）来负责的事情。所以运算的实质其实是值的运算。至于像“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。

综合上面的叙述，我们可以说：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”。

所以我们可以有一个语言系统，它满足说明式语言的两个特性：一是陈述运算，二是求值，仅此就可以完成我们上述的所有计算需求。这种类型的语言在现实环境中也是有应用的，例如在 Internet Explorer 浏览器中的 CSS 就支持这样一种表达式，这其实是一种：

- 消灭了语句的、
- 用表达式来运算求值的

JavaScript 语言的简化版本。类似的语言，被称为表达式语言（Expression Language，EL）。表达式语言具有充备的程序设计能力，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，我们将称之为“运算式语言（范型）”，以强调它是“通过运算求值来实现程序设计”的编程范型[29]——事实上也存在这样的翻译。

2、函数在运算式语言中的价值

[28] 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。此外，“如何消灭逻辑语句结构”的问题，我们会在下一节中予以详述。

[29] 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、性质和功能的语言，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。

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，根据我们前面讲述的“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（如果“函数”是满足函数式语言的三个特性的话）。

不过一些语言中的函数并不能胜任这个工作，例如当在 JavaScript 中的函数就不完全满足函数式语言的三个特性。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。

另一方面，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。但是如果我们真的要这样去做，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——在代码（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。

所以，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，并更好地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。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。

3、重新认识“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正式地解释过“函数式”是什么意思，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，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？

产生这些问题的症结其实在于“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。所以如果仅凭“JavaScript 中函数是第一型的”就推论出“JavaScript 是函数式语言”（或类似的某种语言是函数式语言）的话，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，必须明确这个“函数”的含义。这其中，“Vyacheslav Akhmechet”[30]对有一个有趣的解释：

> “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。

[30] 《函数式编程另类指南》的作者。

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）”，而不是在我们现在的通用语言（我指的是像 C、Pascal 这样的命令式语言）中讲到的 function。

4、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;
}
```

为了让代码简洁些，我们可以写成这样（所谓简洁是指忽略函数声明的部分）：

```
// 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 {y;
    return x *
  }
}
// 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，那么可不可以像示例 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。换而言之，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；
- 对于任何表达式来说，运算符必须写在前面，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。

这一过程中我们只是换用了新的符号标识系统。新的代码应该写成这样：

```
// 示例 6: 新的代码风格
(if #f + *) 3 4)
```

到这里，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。

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。

4、函数式语言

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）。

所以其实我们一直在讲述的就是函数式语言，而所谓“运算式语言范型”无非是我们偷梁换柱的一个名词罢了。

类似的，我们讨论到的一些特性，也就正是函数式语言的特性集：

- 在函数外消除语句，只使用表达式和函数，通过连续求值来组织代码；
- 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，函数等义于表达式运算符，其参数是运算元，返回运算结果；
- 函数严格强调无副作用。

要让 JavaScript 中的“函数（function）”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。而“函数既可以是运算符，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”。

但从另一个角度——“函数/表达式运算的效果”来看，这一切就变成了“所有的东西都

是值”[31]。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；可以直接参与表达式运算；可以作为其他函数的参数传入，或者作为结果值传出。这一切，既可以解释为“高阶函数”的特性，作为“值的特性”来解释，也是一样的。

这些特性通常被概而言之为：“函数是第一型”。“第一型（first-class data types）”通常是指基础类型——在语言中用来组织、声明其他复合类型的基本元素，它在语言 / 语法解释器级别存在，无需用户代码重述的类型。更加直观地说，它表现为如下特性[32]：

- 能够表达为匿名的直接量（立即值）；
- 能被变量存储；
- 能被其他数据结构存储；
- 有独立而确定的名称（如语法关键字）；
- 可（与其他数据实体）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；
- ……

所有这些特性的要点在于：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。

[31] 从面向对象的观点看来则是“所有的东西都是对象”，这事实上是一种哲学观念上的统一。
[32] 引自：http://en.wikipedia.org/wiki/First-class_object。

动态语言

程序最终可以被表达为数据（结构）和逻辑（算法）两个方面，命令式和说明式（以及函数式）语言是从程序的这两个本质方面来进行的分类。而所谓“语言”，其实（从与计算机系统无关的角度来看，）是包括“语法、语义和语用”三个方面的。具体地在计算机系统中实现某种语言时，如果语言陈述时无法确定、而必须在计算机执行时才能确定这三者之间的关系，我们称该语言是动态语义的（反之则称为静态语义），例如对于 JavaScript 代码“a+b”，我们并不能确定是字符串连接还是数值求和。

是哪些因素导致这三者的关系不能静态确定呢？如同自然语言一样，上述“a+b”要有确定的含义，至少有两方面的限定因素：其一是“a、b、+”这三个标识符的指称确定，其二是该语句所在的上下文环境确定。然而遗憾的是，这两个方面在 JavaScript 中都是不确定的——所以 JavaScript 是完全动态的语言，其“标识符指称不确定”表现为：动态类型、（动态）重写和（动态存取的）数据结构三方面；其“上下文环境不确定”表现为动态的变量 / 语法作用域，也涉及我们在前面讲述过的闭包作用域问题。

1、动态语言概要

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，只有在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。当我们以代码的静态语义来看待所谓“变量”时，它其实只是一个标识符。当标识符被赋以一种含义或性质时——更普适的说法是“当事物A与B存在关联时”，我们称为绑定。由此而来的概念是：SNOBOL与APL是一种在标识符上动态绑定“数据类型”与“存储位置”含义的语言。换成现在通常的概念，即是动态类型绑定和动态数据绑定。在这种概念中，变量可以理解为一个无类型指针（没有类型含义的、指向自由地址的标识），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内容以及可能的数据类型[33]。

尽管《程序设计语言概念》（COPL，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[34]，但除开针对某种直接确指的语言，“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。

COM 体系中的 variants 是另一种形式的动态类型系统，它不是通过语言解释的层面，而是通过系统结构来支持的。《程序设计语言概念》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”（p147），而事实上 COM 设计理念打破了这一规则，基于对类型的高度抽象与统一（我是指 IDL 对类型系统的规定），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。

---

[33] 《程序设计语言概念》中称之为“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。

[34] SNOBOL 4 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）。

2、动态执行系统的起源

自从第一份能够被有意义地书写于其他介质——泛指计算机存储系统之外——的代码出现以来，一个关键问题就被提了出来：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——这里主要是指单纯解释执行的语言系统则用一个执行环境来读入并执行这份代码。

对于解释执行的系统来说，显然我们不必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[35]，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。

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（JIT，Just In Time）编译器。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，因此实时性较好；而编译结果是本机的机器码，因此执行效率也很高[36]。

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 1.x 的各个实现引擎都没有即时编译系统（例如 DMonkey 的所谓编译，只是保存代码的语法解释树），但是基于 ECMA Script Edition 4（即 JavaScript 2.0）规范的引擎，却基本都采用了“虚拟执行环境＋即时编译系统＋语言引擎（自宿主＋语言）”的结构来实现。

从技术上来讲，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”——例如利用我们前面讲到的 COM 变体的某些特性。

尽管在早期，通常以“动态类型绑定和动态存储绑定”作为对动态语言特性的基本约定，但在《JavaScript 语言精髓与编程实践》这本书中，也将“动态执行”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，需要上一小节所述的解释系统的支持。无论这种解释器是直接面向代码文本的，还是面向中间代码的，它都必须能够维护原始代码中的、全局的符号系统（例如公布的对象成员名）。因为这些运行时读入的、动态执行的代码使用的，是原始代码中的以及当前（装载时的）环境下的符号系统。事实上一些在程序中嵌入的动态执行引擎（脚本引擎）需要在装载时为既有对象系统或 RTL 库初始化一套标识符，其根源也就在这里。

3、脚本系统的起源

事实上人们很早就习惯于使用“动态执行”的方法来操作计算机系统了，甚至连 DOS 批处理都具有这种“动态执行”的特性：命令行外壳（DOS Shell，comand.com 或 cmd.exe）其实可以看作上述的解释器，批处理则是可以动态装载并执行的代码——包含某种语法规则下的代码行（批处理语句和 DOS 命令）。

正如你此时所想的，早期的 Shell、批处理或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）；

---

[35] 中间代码（Intermediate Code）经常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），就有与它对应的MSIL OpCode。一些并不使用中间语言机制的，也在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。

[36] 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。

- 脚本解释和执行环境。

——所以脚本系统最早并不是作为“程序设计语言”的面貌出现的。如前所述的，批处理是一种提供动态执行能力的脚本语言[37]，因为它们的确具有语言全部要素：关键字、逻辑语句或语法、声明和处理过程（函数或命令）。从语言的角度上来看，批处理也具有更加专业的称谓：Shell脚本。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 批处理要强大得多。

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—1968 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，多路信息与计算服务）计划。在 20 世纪 60 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 成为 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，着手编写了一套操作系统[38]，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”[39]。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[40]。这就是脚本类语言（Shell）的最初起源[41]。

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（它的创建者后来将它正式命名为“样式扫描和处理语言”）。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。

正是因为AWK与Shell这两种早期的脚本语言系统，使得许多介绍“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”。但如果真的要从“功用”的角度来讨论，那么Shell及脚本语言最早受到的影响应该来自于 1960 年的IBM 360 系统中[42]，该系统中提供了一个任务控制语言（JCL，Job Control Languge），其基本思想是“用于控制其他程序（used of control other programms）”。

---

[37] 准确地说，“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”。

[38] 可见偏执也是一种生产力。

[39] 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）。

[40] 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”。

[41] 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。

[42] 参见《CSCI: 4500/6500 Programming Languages - Scripting Languages Chapter 13》（.pdf），Maria Hybinette。

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。

简单地说，你可以将 Pascal、C、PROLOG 这些语言等全部实现成“脚本语言”，但此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的确更适用于实现动态语言，并强调“脚本化”并非 JavaScript 这种语言（以及其他动态语言）的本质特征。同样，下面这些与脚本化相关的特性，也疏离于其语言的本质。

- 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[43]。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 Embeded、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语言等。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程序，可以直接从命令行或系统中装载脚本并执行[44]，而无需依赖宿主页面。

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），但并没有真正的编译器。
- 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。重写是一种实现，其效果便是前面提到的“动态绑定”。

其他的一些来自于动态语言系统自身定义的特性如下。

- 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[45]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。

这些内容将在随后的章节中予以详述。

2、动态执行

[43] 例如一个基于SpiderMonkey JavaScript的开源项目jslib，即是用C/C++语言扩展引擎的内置对象系统，以便使用该语言实现较大型的软件系统。
[44] 例如WSH中的WScript.exe，Java实现的Riho引擎和SpiderMonkey JavaScript等引擎都有各自的Shell程序。
[45] 作为类型系统完备性的必须，JavaScript中将`undefined`也视为类型。但这不是通常的、强类型含义中的“类型”。

动态语言的执行系统，基本可以分为动态装载与动态执行两个阶段。但某些系统中可能将两个阶段合而为一。例如 DOS 批处理中的 `call` 命令即装入并执行另一个批处理：

```
// a.bat
echo now execute a.bat
call b.bat

// b.bat
echo now exec b.bat
dir *.exe
```

而JavaScript就将装载[46]与执行[47]分成两个阶段。对于后者来说，动态执行的对象是一个字符串格式的“代码文本”，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。

我们首先来观察 JavaScript 的执行系统（而不是动态执行系统），它的代码总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。

由于代码总要运行在一个闭包中，所以构建动态执行系统时，也要为 `eval()`来准备一个闭包环境。然而这成了一个焦点问题：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？对于 JavaScript 来说，答案是“两种都有可能”。当然，对于大多数“稳定、安全而有效”的动态语言来说，应该是只准备“当前函数闭包”这样一种的。只有这样，动态执行的效果才明确，才可以让程序员来预期。

反过来说，正是由于JavaScript具有不明确的“eval()执行环境”的设定，所以它会有一些“奇特的、甚至是负面的影响”，例如动态执行可能会导致变量作用域发生变化[48]：

```
var i = 100;
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，因此也有机会通过（由变量 `ctx` 传入的）代码来修改它。例如：

```
myFunc('var i = 10;');
```

代码的执行效果将是显示：

```
value is: 100
value is: 10
```

JavaScript的代码的不可编译——这一为人垢病的性质也是由动态执行导致的。总的来说，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，而非某种支持动态执行的虚拟机环境中的中间代码[49]。矛盾在于：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。某些极端的情况下，一旦引擎全面支持动态执行，则编译过程必须以明码形式保存标识符表。与此相关的技术和解决方案包括：

---

[46] 动态装载在JavaScript中是由宿主提供的能力。例如在WSH中提供了File System Object（FSO，文件系统对象）来装载本地文件，而在浏览器环境中提供了XMLHttpRequest，或者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。

[47] 动态执行在JavaScript中主要是由eval()方法带来的效果。不过不同的语言系统也存在差异，例如使用evaluate()、exec()或execute()方法。

[48] 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）。

[49] 在JavaScript 2 的各种实现引擎中，基本都支持编译成中间代码并运行于一个面向动态语言的虚拟机环境，例如在JScript.NET（以及DLR）和ActionMonkey中。但这是因为ECMAScript Edition 4 规范对动态执行（以及其他诸多方面）作出非常多的限制，与JavaScript 1.x中相比来说，JavaScript 2 已经是“非常不动态”了。

- 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 应用环境中，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。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。

3、重写

重写是 JavaScript 语言中的灵魂，也是其所有语言特性之间的粘合剂，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。但在另一面，重写也是动态语言系统的灾难，例如上一小节“执行”中的示例代码，就包括了一个标识符重写，这导致了作用域的改变。

重写是一个代码执行期的行为。在拥有动态执行能力的系统中，引擎在代码的语法分析期既无法对重写行为进行任何的预期，也不能对其进行限制。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。

JavaScript 中针对原型和构造器的重写，会影响到重写前所创建实例的一些重要特性——例如继承性的识别。经常见到的情况是：重写导致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。与此相似的，显式的构造器修改也会导致继承关系丢失，同时还可能导致语法声明与语句含义不一致。

对于语法声明与语句含义这个问题，我们可以从“（自然语言中的）语言学”的角度加以解释。语言学将语言分为语法、语义、语用三个部分，当这三个部分一致时，语言含义明确并且可以在不同语言之间直接对译，否则将因为存在人为的、主观的模糊而无法理解。语法，可以是（书面）记述法或（口头）讲述法，而语义则是通过这个语法来了解到的含义。例如“咬死了猎人的狗”从语法上没有任何问题，但语义就不明确：我们不能确知是“狗”死了，还是“猎人”死了。同样的，语用就与环境有关，例如“吃了吗”，在早上问的时候，或许就可以直接理解为“早安”。

所以为了让计算机能够“准确地”理解我们说的“（代码）语言”，我们在设计语言时，总是强调它具有完全的、绝对的语法与语义一致性，而且还要避免在不同的上下文中存在歧义（语用）。我们在本章开始时就例举的“a+b”就没有确定的含义，因为 JavaScript 的语法设计就违背了上述原则。同样地，重写也带来了问题：它使语法声明的效果，与语句执行的含义变得不一致。不过也使我们在 JavaScript 得到了“足够的”，以及“难于驾驭的”灵活性。

所以，一些重写特性（例如构造器重写）就与代码装入它的时序（这里可以对应于自然语言中的语境，或相关的语用问题）有关，我们通常要求一些代码或程序包在引擎中最先装入。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。

最后，不同的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，例如宿主（或应用执行环境）的限制。

4、包装类，以及“一切都是对象”

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，以及其他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50。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”。不过，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。

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。而且，元数据在进行对象系统的相关运算时，是不会有包装行为的，例如：

```
// 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
```

但是在做下面的运算时，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
```

综合上例中的步骤 3~5 可知：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[51]。这种包装类的方法，是JavaScript用来应对“可以调用值类型数据的方法，使它看起来像是对象调用”的处理技术。这与后来在.NET中出来的“装箱（boxing）”是一样的[52]，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。在 showDataType()调用结束后（准确地说是在该对象的生存周期结束时），在包装过程中临时创建的包装对象也将被清理掉。

---

[50] 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没有成为一种像Java一样“纯粹的”面向对象语言。

[51] 除此之外，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。

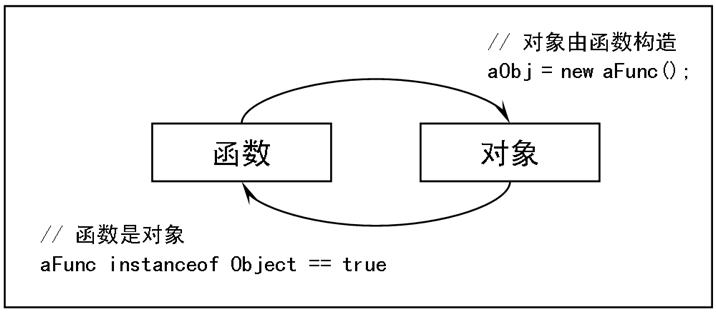
[52] JavaScript没有类似于“拆箱（unboxing）”的过程，因为它的函数形式参数不支持值的传出。

显然变量 n1 并不等同于“临时创建的包装对象”，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的。而同样的原因，无论我们如何修改这个“临时创建的包装对象”的成员，这种修改也会不影响到原来的变量 n1 的值。

但 function 在包装类的性质上却存在着一些特例，不过这些特例正好展现了两个重要的函数特性：

- typeof 对函数直接量（v1）与包装后的对象（v2）检测都返回'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。

这正是 JavaScript 统一函数式语言与面向对象编程（命令式语言范型）的核心思想，这种思想一方面表现为“一切都是对象”，另一方面表现为“函数是第一型”。这是一种简洁的、互证的、相互作用的二元关系：



JavaScript 统一语言范型的基本模型

5、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[53]。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作为下标变量来存取的数组。

有趣的是，关联数组与索引数组是互为补充的两个数据结构。在内存分配上，一般索引数组是连续的，而关联数组则以指针形式存储；在下标使用是，索引数组使用序数，而关联数组使用字符串来访问/遍历；使用效果上，索引数组通常用于数学运算，而关联数组用于检索或结构性组织……类似的，我们也看到一些语言在实现数据类型时，总是“不由自主地”使用性质相对的数组（或表、列表）来组织复合类型，二者相互补益。例如 Erlang 使用 tuple 来表示定长数组，使用 list 来表示变长数组，定长数组与变长数组在存储效率和扩展性上的表现就恰恰相反。这些都是有趣的语言设计。

而在 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，其他两种则是非序数的（number 在 JavaScript 中实现为浮点数）。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。更确切地说，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。

[53] 另一种常见的对数据的区分方法是动态数组与静态数组，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。

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。所以事实上，JavaScript 中的 Array 并不具有一般索引数组的连续存储特性，在矩阵运算等数学运算中也不会有什么特别的表现。

关联数组下标是非序数的，所以它看起来更像是一张“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。在 JavaScript 中，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，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。

6、值运算：类型转换的基础

我们观察 JavaScript 的元类型系统，其实它只有两类：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`。

我们曾讨论过“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。对于计算系统来说，引用类型的价值是：

- 标识一组值数据；
- 提供一组存取值数据的规则；
- 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[54]；
- 将引用类型转换为值类型（以参与运算）。

这样的两种情况。而这两种情况，就是一个语言中类型转换的全部了。

当然你也许会认为：C、C++以及 Delphi 等等语言中显然还有其它的类型转换方法，即使在一般的面向对象系统中，也还存在“as”、“is”这样的多态问题。是啊，既然如此，那么我们又怎么说“值类型与引用类型之间的问题”，就已经是语言中类型转换的全部了呢？

我们需要抛开复杂的语言、语法的表面，我们需要认识到一个语言本身的作用仍然不过是让机器系统理解我们的想法。而其最终的识别方法，仍然不过是一个简单的运算规则：计算什么，以及如何计算。这些问题，在计算机系统中永远都是值的问题，而绝不会是引用的

[54] `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。

问题。所谓引用，例如指针、表、结点、结构或对象，甚至我们说到的函数或闭包等等，都只是抽象概念，是不为计算机所知的。

在这些抽象概念的背后，是我们人（程序员或非程序员）描述世界的方法，这是主观的、可变的。对于计算机来说，它仅是理解那些原本设定不为变的东西。在前面所有的论述中，有一个东西是根本上未曾变化过的：值的含义，和运算的方法。例如说，我们理解的整型，在计算机系统中就是总线宽度；我们理解的乘法，就是一组开关运算，等等。

我们最终、最本质的语言设计目的，就是把人的、类似上述直接映射之外的、复杂的抽象，通过语言（语法、语义和语用的设定），变成计算机理解的数据序列与指令序列。这所有的序列只能是“值”，而没有所谓“引用”的问题。

所以我们在前面强调“引用类型的价值是”：标识、规范和传递(被标识的)数据。这个过程与 CPU 所理解的“运算”，是两回事。同样的，JavaScript 这门语言在设计时，便已深刻地领悟到了这一语言本质，并通过对包装类使所有值、引用都持有 valueOf()方法，用来向计算系统说明自己的“值含义”。这样一来，任何一个数据或结构都可以参与运算，而无有冲突。

同样的思路下，所有值、引用都持有 toString()方法，用来向外部系统说明自己的“形式化描述”——简单的说，就是字符串序列的描述。一方面，这是程序员所能理解的形式；另一方面，这也是 JavaScript 最初应用环境（WEB 浏览器）所能理解的形式——文本。

所以，JavaScript 语言（当然也可以是其它的语言）中，类型转换的本质只不过是将一个运算元“理解为何种‘值(value)’”，或“理解为何种‘内容(string)’”的问题。在这样的规则下，我们、以及计算系统将不难（或更难）理解下面这行代码的含义：

```
[] + []
```

它的含义是“两个空数组的并集”吗？其值是[ ]？是""？还是 NaN？

不，什么都不是。上面代码的输出存在任何的、与上下文相关的一种可能，例如 0：

```
// 重写 valueOf()
Array.prototype.valueOf = function() { return 0 }

// 输出: 0
alert([] + []);
```

或者是“Hello, World!”：

```
// 重写 toString()
Array.prototype.toString = function() {
  var args = arguments, argn = 0;
  return function() { return args[argn++] }
}("Hello, ", "World!");

// 输出: Hello, World!
alert([] + []);
```

综述

总体来说，语言以实现方法可分动态、静态的，或以本质不同分命令式或说明式的。《JavaScript 语言精髓与编程实践》讨论了这些复合特性的一个实现：JavaScript，涉及了除静态语言之外的其它所有语言特性。而源于 JavaScript 自身的限制，在对象系统、过程式、函数式和动态语言四个方面，这本书都未能完全覆盖。例如对象系统中的元类继承或动态语言中的持续，JavaScript 自身就不支持，所以也无从谈起。

语言本身只是表象，程序才是目标。语言表达了程序的算法和结构两个方面，其一或其全部。以语言本身来说，能表达二者之一，或（并）实现这二者即已是“完备”的了。然而语言还有外貌或外观的问题。在这个问题上的答案是“分块”。与之伴生的问题，就是机器系统对“块”的理解，与人对“块”的理解未必一致。看起来如此简单的哲学问题：一分成二就不是二了，而是三。因为存在了界面、界面上的观察者，观察者的审核者……等等如此，世界逐渐复杂起来。

所以语言的复杂性最终并不是语言问题。举例来说，现在的 C#或 Java 代码中，注释可以自身体系地成为一种“可编译成文档的语言”。那么这种“文档语言”是不是 C#或 Java 的一部分呢？再向前推进，这种“文档语言”具有“算法+结构”的特性吗？是一种完备的语言吗？需要完备吗？

未见得。我们未见得要这样讨论问题。语言发展的边界，是计算系统还是应用系统？这才是要最先明确的前提。如果是计算系统，有函数式的 LISP 或命令式的 FORTRAN 就足够了，我们不会再需要更多的“计算方法”。面对应用系统时，应用系统自身的复杂性——而不是语言本身“计算求值”的需求决定了一切。这意味着语言越来越复杂，且这越来越复杂的语言中的一行“简单的代码”就包含了一个极其复杂的逻辑。我们，将问题分域并定义，提供这个特定域中的解决方案，以及相关的语言和习惯性的陈述方法。而任何一个域和满足习惯的陈述方法都是一项伟大的创建。

但是，这不是语言本身推动的。面向应用与面向语言本质，这是两个方向。因此它们对“简单或复杂”的评判标准并不相同。要确切的说函数式“简单”或“好”，或 C#更简单更好，是要放在确定的背景上来讲的。而抛开好或不好的争论，我们看到的事实是：一切的程序设计语言都归源于说明或命令，归源于结构与算法，归源于编译和解释，或归源于……

所有归源的终点，是相互的借鉴与学习。这在这个终点上的表现，不是更纯粹，而是更混杂。这是无可避免的事，人们往往在极端追求的终点看到起点。

如同所来之处。

“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，而其他技术则没有。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

Eich 对 JavaScript 原始设计目标的回顾，让我们看到了一门语言最初的设定与它最终应用环境之间的关系与冲突，以及由这些关系与冲突带来的、开发人员对这个语言系统的种种反应。而另一方面，Eich 对语言标准化的反省，让我们看到了商业与技术的分歧、争端与平衡。当然，最终平衡的结果可能相当可笑，例如我们看到的 ECMAScript Harmony Project。但即使可笑，也有其存在的背景与原因，这才是真相的全部。

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

Wirth 成就了一门语言（Pascal）和一句名言（算法+结构＝程序），他对于语言的理解，在几十年之后看来，仍然相当精准。我不确定谁是更能“逢源”的语言（或语言范型），但我在不停地追寻那个问题的答案。谢谢 Wirth，我们在前行，无论这是否是您所设定的方向。

（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。但也有唯一的一种东西，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

Stroustrup 对“通用程序设计语言”的理解，是他几十年来从用户那里得到的最为宝贵的经验。真实可信的语言，是可以会话的、交流的以及表达我们所需的语言。从这个角度上来说，即使 LISP 已经具备了一种语言所需的全部要素，也只是一套违反人性的机器指令记述法——但这并不表明在 LISP 影响下发展起来的那些语言也一样违反人性。我们持以相同的“始终不渝的信念”，说着那些文法艰涩的话语，与语言一起成长。直到有一天，我们变得跟机器一样，或者反之。