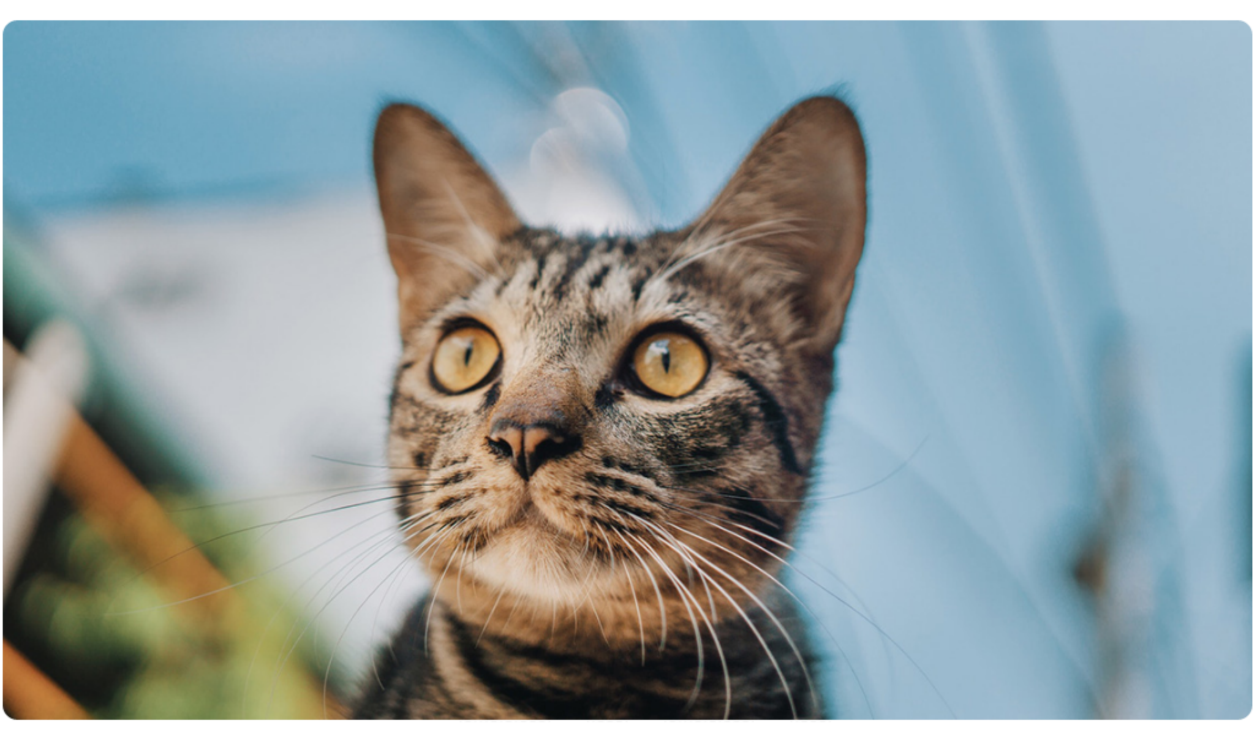
01 | 明确你的前端学习路线与方法

winter 2019-01-17



00:00 11:07

讲述：winter 大小：10.20M

你好，我是 winter。今天我们一起来聊聊前端的学习路线与方法。

在“开篇词”中，我和你简单回顾了前端行业的发展，到现在为止，前端工程师已经成为研发体系中的重要岗位之一。可是，与此相对的是，我发现极少或者几乎没有大学的计算机专业愿意开设前端课程，更没有系统性的教学方案出现。大部分前端工程师的知识，其实都是来自于实践和工作中零散的学习。

这样的现状就引发了一系列的问题。

首先是前端的基础知识，常常有一些工作多年的工程师，在看到一些我认为很基础的 JavaScript 语法的时候，还会惊呼“居然可以这样”。是的，基础知识的欠缺会让你束手束脚，更限制你解决问题的思路。

其次，技术上存在短板，就会导致前端开发者的上升通道不甚顺畅。特别是一些小公司的程序员，只能靠自己摸索，这样就很容易陷入重复性劳动的陷阱，最终耽误自己的职业发展。

除此之外，前端工程师也会面临技术发展问题带来的挑战。前端社区高度活跃，前端标准也在快速更新，这样蓬勃发展对技术来说无疑是好事，但是副作用也显而易见，它使得前端工程师的学习压力变得很大。

我们就拿 JavaScript 标准来说，ES6 中引入的新特性超过了过去十年的总和，新特性带来的实践就更多了，仅仅是一个 Proxy 特性的引入，就支持了 VueJS 从 2.0 到 3.0 的内核原理完全升级。

缺少系统教育 + 技术快速革新，在这样的大环境下，前端工程师保持自学能力就显得尤其重要了。

那么，前端究竟应该怎么学呢？我想，我可以简单分享一下自己的经验。

学习路径与学习方法

首先是**0 基础入门**的同学，你可以读几本经典的前端教材，比如《JavaScript 高级程序设计》《精通 CSS》等书籍，去阅读一些参考性质的网站也是不错的选项，比如MDN。

如果你至少已经**有了 1 年以上的工作经验**，希望在技术上有一定突破，那么，这个专栏就可以是你技术进阶的一个选项了。

在这个专栏中，我希望传达的不仅仅是具体的知识点，还有体系架构和学习方法。我希望达到三个目标：

- 带你摸索出适合自己的前端学习方法；
- 帮助你建立起前端技术的知识架构；
- 让你理解前端技术背后的核心思想。

在开始具体的知识讲解之前，这篇文章中，我想先来谈两个前端学习方法。

第一个方法：建立知识架构

第一个方法是建立自己的知识架构，并且在这个架构上，不断地进行优化。

我们先来讲讲什么叫做知识架构？我们可以把它理解为知识的“目录”或者索引，**它能够帮助我们把零散的知识组织起来，也能够帮助我们发现一些知识上的盲区。**

当然，知识的架构是有优劣之分的，最重要的就是逻辑性和完备性。

我们来思考一个问题，如果我们要给 JavaScript 知识做一个顶层目录，该怎么做呢？

如果我们把一些特别流行的术语和问题，拼凑起来，可能会变成这样：

- 类型转换；
- this 指针；
- 闭包；
- 作用域链；
- 原型链；
- ……

这其实不是我们想要的结果，因为这些知识点之间，没有任何逻辑关系。它们既不是并列关系，又不是递进关系，合在一起，也就没有任何意义。这样的知识架构，无法帮助我们去发现问题和理解问题。

如果让我来做，我会这样划分：

- 文法
- 语义
- 运行时

为什么这样分呢，因为对于任何计算机语言来说，必定是“用规定的文法，去表达特定语义，最终操作运行时的”一个过程。

这样，JavaScript 的任何知识都不会出现在这个范围之外，这是知识架构的完备性。我们再往下细分一个层级，就变成了这个样子：

- 文法
  - 词法
  - 语法
- 语义
- 运行时
  - 类型
  - 执行过程

我来解释一下这个划分。

文法可以分成词法和语法，这来自编译原理的划分，同样是完备的。语义则跟语法具有一一对应关系，这里暂时不区分。

对于运行时部分，这个划分保持了完备性，**我们都知道：程序 = 算法 + 数据结构，那么，对运行时来说，类型就是数据结构，执行过程就是算法。**

当我们再往下细分的时候，就会看到熟悉的概念了，词法中有各种直接量、关键字、运算符，语法和语义则是表达式、语句、函数、对象、模块，类型则包含了对象、数字、字符串等……

这样逐层向下细分，知识框架就初见端倪了。在顶层和大结构上，我们通过逻辑来保持完备性。如果继续往下，就需要一些技巧了，我们可以寻找一些线索。

比如在 JavaScript 标准中，有完整的文法定义，它是具有完备性的，所以我们可以根据它来完成，我们还可以根据语法去建立语义的知识架构。实际上，因为 JavaScript 有一份统一的标准，所以相对来说不太困难。

如果是浏览器中的 API，那就困难了，它们分布在 w3c 的各种标准当中，非常难找。但是我们要想找到一些具有完备性的线索，也不是没有办法。我喜欢的一个办法，就是用实际的代码去找：for in 遍历 window 的属性，再去找它的内容。

我想，学习的过程，实际上就是知识架构不断进化的过程，通过知识架构的自然延伸，我们可以更轻松地记忆一些原本难以记住的点，还可以发现被忽视的知识盲点。

建立知识架构，同样有利于面试，没人能够记住所有的知识，当不可避免地谈到一个记不住的知识，如果你能快速定位到它在知识架构中的位置，把一些相关的点讲出来，我想，这也能捞回不少分。（关于前端具体的知识架构，我会在 02 篇文章中详细讲解。）

第二个方法：追本溯源

第二个方法，我把它称作追本溯源。

有一些知识，背后有一个很大的体系，例如，我们对比一下 CSS 里面的两个属性：

- opacity；
- display。

虽然都是“属性”，但是它们背后的知识量完全不同，opacity 是个非常单纯的数值，表达的意思也很清楚，而 display 的每一个取值背后都是一个不同的布局体系。我们要讲清楚 display，就必须关注正常流（Normal Flow）、关注弹性布局系统以及 grid 这些内容。

还有一些知识，涉及的概念本身经历了各种变迁，变得非常复杂和有争议性，比如 MVC，从 1979 年至今，概念变化非常大，MVC 的定义几乎已经成了一段公案，我曾经截取了 MVC 原始论文、MVP 原始论文、微软 MSDN、Apple 开发者文档，这些内容里面，MVC 画的图、箭头和解释都完全不同。

这种时候，就是我们做一些考古工作的时候了。追本溯源，其实就是关注技术提出的背景，关注原始的论文或者文章，关注作者说的话。

操作起来也非常简单：翻翻资料（一般 wiki 上就有）找找历史上的文章和人物，再顺藤摸瓜翻出来历史资料就可以了，如果翻出来的是历史人物（幸亏互联网的历史不算悠久），你也可以试着发封邮件问问。

这个过程，可以帮助我们理解一些看上去不合理的东西，有时候还可以收获一些趣闻，比如 JavaScript 之父 Brendan Eich 曾经在 Wikipedia 的讨论页上解释 JavaScript 最初想设计一个带有 prototype 的 scheme，结果受到管理层命令把它弄成像 Java 的样子（如果你再挖的深一点，甚至能找到他对某位“尖头老板”的吐槽）。

根据这么一句话，我们再去看看 scheme，看看 Java，再看看一些别的基于原型的语言，我们就可以理解为什么 JavaScript 是现在这个样子了：函数是一等公民，却提供了 new this instanceof 等特性，甚至抄来了 Java 的 getYear 这样的 Bug。

结语

今天我带你探索了前端的学习路径，并提出了两个学习方法：你要试着建立自己的知识架构，除此之外，还要学会追本溯源，找到知识的源头。

这个专栏中，我并不奢望通过短短的 40 篇专栏，事无巨细地把前端的所有知识都罗列清楚，这本身是 MDN 这样的参考手册的工作。但是，我希望通过这个专栏，把前端技术背后的设计原理和知识体系讲清楚，让你能对前端技术产生整体认知，这样才能够在未来汹涌而来的新技术中保持领先的状态。

在你的认识中，前端知识的结构是怎样的？欢迎留言告诉我，我们一起讨论。