

TURING

图灵原创

深入剖析Vue.js源码

刘博文◎著

# 深入浅出 Vue.js

360奇舞团团长月影 作序  
《JavaScript高级程序设计》译者李松峰 推荐

书 | 馆

09



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS

图书在版编目 (C I P) 数据

深入浅出Vue.js / 刘博文著. — 北京 : 人民邮电出版社, 2019.3 (2019.6重印)  
(图灵原创)  
ISBN 978-7-115-50905-5  
I. ①深… II. ①刘… III. ①网页制作工具—程序设计 IV. ①TP393.092.2  
中国版本图书馆CIP数据核字(2019)第037890号

内 容 提 要

本书从源码层面分析了 Vue.js。首先，简要介绍了 Vue.js；接着详细讲解了其内部核心技术“变化侦测”，并带领大家从 0 到 1 实现一个简单的“变化侦测”系统；然后详细介绍了虚拟 DOM 技术，其中包括虚拟 DOM 的原理及其 patching 算法；再后详细讨论了模板编译技术，其中包括模板解析器的实现原理、优化器的原理以及代码生成器的原理；最后详细介绍了其整体架构以及提供给我们使用的各种 API 的内部原理，同时还介绍了生命周期、错误处理、指令系统与模板过滤器等功能的原理。

本书适合前端开发人员阅读。

---

◆ 著 刘博文  
责任编辑 王军花  
责任印制 周昇亮  
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号  
邮编 100164 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
涿州市京南印刷厂印刷  
◆ 开本：800×1000 1/16  
印张：18.5  
字数：437千字 2019年3月第1版  
印数：7 001-9 000 册 2019年6月河北第4次印刷

---

定价：79.00元

读者服务热线：(010)51095183转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

# 序

近几年，JavaScript 的流行库和框架带有元编程（metaprogramming）的特征。所谓元编程，简单来说，是指框架的作者使用一种编程语言固有的语言特性，创造出相对新的语言特性，使得最终使用者能够以新的语法和语义来构建他们的应用程序，从而在某些领域开发中获得更好的开发体验。

早期的 jQuery 库之所以获得开发者们的认可，很大程度上是因为它独创的链式语法和隐式迭代语义。尽管 jQuery 仅仅通过巧妙设计 API 就能支持上述特性，并不依赖于编程语言赋予的元编程能力，但是毫无疑问，它以一种精巧的设计理念和思路，为 JavaScript 库和框架的设计者打开了一扇创新的大门。

今天的 Web 产品对构建用户界面的要求越来越高，jQuery 的方式不能满足构建复杂用户界面的需要，新的 UI 框架快速发展，其中一个最流行的框架就是 Vue.js。与 jQuery 相比，Vue.js 更强大，也具有更加明显的元编程特征。动态绑定属性和变化侦测、内置模板和依赖于模板语法的声明式渲染、可扩展的指令、支持嵌套的组件，这些原生 JavaScript 并不具备的特征和能力被一一融入，框架的使用者在使用 Vue.js 开发 Web 应用时，事实上获得了超越 JavaScript 原生语言特性的能力。

尽管 Vue.js 框架赋予开发者众多特性和能力，但它仍然是使用原生 JavaScript 实现的应用框架。JavaScript 自身提供了许多元编程特性，比如从 ES5 就开始支持的属性访问器（property accessor），ES6 支持的代理（proxy），还有标准提案已经处于 Stage 3 阶段的装饰器（decorator）。基于这些语言特性，我们能够比较方便地扩展新的语言特性，将这些特性融入应用框架，从而使得应用开发者能够更加得心应手地使用框架开发出优雅、简洁的应用程序模块。

如何设计 API 和如何使用元编程思想将新特性融入到框架中，是现代 JavaScript 框架设计的两个核心，Vue.js 更侧重于后者。理解元编程思想有助于深刻理解 Vue.js 的本质。而理解元编程思想本身最好的方法又是通过深入研究 Vue.js 的源码，因为元编程思想一旦涉及具体实现，不仅仅是使用 JavaScript 提供的特性来扩展能力那么简单，这其中有许多细节需要考虑，比如要做到向下兼容，那么就要对一些特性的实现方式做出取舍，一些语言能力可以通过书写向下兼容代码来弥补，而另一些则需要通过编译机制来做到，还有一些则必须舍弃；同样，基于性能考虑，一些特性也可能需要做出一定的修改或妥协。这些问题不仅在框架设计和实现的过程中会遇到，而

## 2 序 一

且在具体实现应用程序的过程中也会遇到。因此，通过学习 Vue.js，我们不仅能够掌握设计应用程序框架的一般性技巧，还可以在实现应用程序时运用其中的具体设计思想和方法论。

本书的作者刘博文是我的同事，也是奇舞团的一员，后来由于业务变动，博文所在的团队从奇舞团独立了出去，但是同为 360 的前端团队，我们也始终保持着项目合作和技术交流。很早就听到博文要写这样一本书，当时我很高兴，我一直鼓励大家写书，因为这种创作既能使自己成长，又能使读者获益。我自己也写过技术类的书，深知技术创作的不易，要把 Vue.js 这样的流行框架讲透也着实需要一番苦功。有时候，作为朋友，我会和博文开玩笑，说他的书再不出版，Vue.js 3.0 版本就要发布了，但这仅仅是玩笑，我不愿意博文因为要赶出版时间而草草了事，那样就无法真正做到“深入浅出”，毕竟这不是一本 Vue.js 的使用手册，而是真正能够透过 Vue.js 的设计思路去学习元编程思想，并将这种思想运用于程序开发中的书。只有这样，读者才能真正从这本书中获益。我想，在这一点上，博文没有让我失望，我也希望这本书没有让你们失望。

月影  
360 奇舞团团长  
2019 年 2 月 1 日

## 序二

“奇舞团”办公地点在“南瓜屋”7层，导航前端在“南瓜屋”8层。2017年某一天，我去8层的时候路过导航前端工位，梁超看到我，高兴地说：“李老师，博文正在写书呢。”我脱口而出：“谁是博文，给哪个出版社写？”由此我便认识了博文，也知道了他是王军花（本书策划编辑）发掘的作者。当时听到这个消息我也很兴奋，知道是在给图灵写书，而我又在图灵待过几年，熟悉图灵的“套路”，就忍不住当场给博文分享了一些选题和写作思路。听着我滔滔不绝地讲“写书经”，博文频频点头，好像很受启发的样子。

2018年年初，360 W3C 工作组成立，博文加入了 Web 性能工作组。于是几乎每周的例会上，我都会问问博文新书写作和出版的进度。时值年末，这本书终于要出版了。而这时候，我因为支持智能音箱项目临时搬到了 11 层，开发、联调非常繁忙。11月 16 日下午，博文突然在微信上问我能不能帮他写个序。我说：“你能不能先给我看看书稿？”然后博文把我加到了他 GitHub 的私有仓库。

两周来，我利用空暇时间大致浏览了一遍书稿。无奈时间紧迫，大部分章节来不及细读。一是因为公司项目开发进度必须保证，二是自己还有一个字体服务的项目在并行迭代。虽然大部分内容未曾细读，但仅就仔细读过的几章而言，着实让我受益匪浅。我想，等到手头的项目开发告一段落之后，一定要抽时间重新研读两遍。没错，这本书至少要读两遍以上。

浏览书稿的时候，我也在回忆第一次跟博文分享“写书经”的情景。当时我说，要想让技术书畅销，一是读者定位必须是新手，因为新手人数众多；二是要注重实用，书中的例子最好能立即照搬到项目上。然而，这本书的读者定位显然不是新手，而且书中的源码分析似乎也不能直接套用到项目上。其实这也是没办法的事，因为博文写这本书的初衷就是把自己研究 Vue.js 源码的心得分享出来。就 Vue.js 源码分析而言，这本书确确实实是非常棒的。反正我是爱不释手。

这本书取名“深入浅出”是名副其实的。因为它确实有相当的深度，而且语言真的浅显易懂。最重要的是，与其他源代码分析类的技术书连篇累牍地堆砌、照搬项目源代码的做法截然不同，这本书里很少看到超过一页的代码片段。所有代码片段明显都被作者精心筛选、编排过，而且层层递进，加上了“新增”“修改”之类的注释。再辅以明白浅显的文字和配图，原本隐晦、抽象、艰深的代码逻辑，瞬间变得明白易懂，让人不时有“原来如此”之叹，继而“拍手称快”！

毋庸置疑，Vue.js 是一个优秀的前端框架。一个优秀的前端框架如果没有一本优秀的解读著

## 2 序二

作，确实是一大缺憾。应该说，本书正是一本优秀的 Vue.js 源码解读专著。全书从一个新颖的“入点”——“变化侦测”切入，逐步过渡到“虚拟 DOM”和“模板编译”，最后展开分析 Vue.js 的整体架构。如果想读懂这本书，读者不仅要有一些 Vue.js 的实际使用经验，而且还要有一些编译原理（比如 AST）相关的知识储备，这样才能更轻松地理解模板解析、优化与代码生成的原理。译者（比如 AST）相关的知识储备，这样才能更轻松地理解模板解析、优化与代码生成的原理。虽然借鉴了 Vue.js 官方文档，但作者更注重实现原理的分析，弥补了文档的不足。

虽然本书不是写给新手看的，但鉴于 Vue.js 在国内的用户基数巨大，我对它的销量还是很乐观的。这些年来，前端行业一直在飞速发展。行业的进步，导致对从业人员的要求也不断攀升。放眼未来，虽然仅仅会用某些框架还可以找到工作，但仅仅满足于会用一定无法走得更远。随着越来越多“聪明又勤奋”的人加入前端行列，能否洞悉前沿框架的设计和实现将会成为高级人才与普通人才的“分水岭”。

“欲穷千里目，更上一层楼。”我衷心希望博文这本用心之作，能够帮助千千万万的 Vue.js 用户从“知其然”跃进到“知其所以然”的境界。最后想说一句，有心购买本书的读者大可不必纠结于 Vue.js 的版本问题。因为优秀源代码背后的思想是永恒的、普适的，跟版本没有任何关系。早一天读到，早一天受益，仅此而已。

李松峰

360 奇舞团高级前端开发工程师  
前端 TC 委员、W3C AC 代表  
《JavaScript 高级程序设计》译者

2018 年 12 月 2 日

# 前　　言

时至今日，Vue.js 就像曾经的 jQuery，已经成为前端工程师必备的技能。不可否认，它可以帮助极大地提高我们的开发效率，并且很容易学习。

这就造成了一个很普遍的现象，大部分前端工程师对框架以及第三方周边插件的关注程度越来越高，甚至把自己全部的关注点都放在了框架上。

在我看来，这多少有点亚健康，不是很利于前端工程师的技术成长。因为我发现大家关注框架时，更多的是关注其用法（包括框架自身、第三方插件和 UI 组件库等）、奇淫技巧和最佳实践等。

而我希望大家拿出一部分精力去关注框架所解决的问题以及它是如何解决这些问题的。这有助于我们提升自己的技术和解决问题的能力。

大家在使用 Vue.js 开发项目时，不免总会遇到一些奇奇怪怪的问题，而我们是否能很快解决这些问题以及理解这些问题为什么会发生，主要取决于对 Vue.js 原理的理解是否足够深入。

## 本书目的

所有技术解决方案的终极目标都是在解决问题，都是先有问题，然后有解决方案。解决方案可能并不完美，也可能有很多种。

Vue.js 也是如此，它解决了什么问题？如何解决的？解决问题的同时都做了哪些权衡和取舍？

本书将带领大家透过现象看到 Vue.js 的本质，通过本书，我们将学会：

- Vue.js 的响应式原理，理解为什么修改数据视图会自动更新；
- 虚拟 DOM (Virtual DOM) 的概念和原理；
- 模板编译原理，理解 Vue.js 的模板是如何生效的；
- Vue.js 整体架构设计与项目结构；
- 深入理解 Vue.js 的生命周期，不同的生命周期钩子之间有什么区别，不同的生命周期之间 Vue.js 内部到底发生了什么；
- Vue.js 提供的各种 API 的内部实现原理；
- 指令的实现原理；

- 过滤器的实现原理；
- 使用 Vue.js 开发项目的最佳实践。

## 组织结构

本书共分四篇，全方位讲解了 Vue.js 的内部原理。

- 第一篇：共 3 章，详细讲解了 Vue.js 内部核心技术“变化侦测”，并一步一步带领大家从 0 到 1 实现一个简单的“变化侦测”系统。
- 第二篇：共 3 章，详细介绍了虚拟 DOM 技术，其中包括虚拟 DOM 的原理及其 patching 算法。
- 第三篇：共 4 章，详细介绍了模板编译技术，其中包括模板解析器的实现原理、优化器的原理以及代码生成器的原理。
- 第四篇：这是本书占比最大的一部分，详细介绍了 Vue.js 的整体架构以及提供给我们使用的各种 API 的内部原理。同时还对 Vue.js 的生命周期、错误处理、指令系统与模板过滤器等功能的原理进行了介绍。在本书最后一章，我们为大家提供了一些使用 Vue.js 开发项目的最佳实践，这些内容中一大部分是 Vue.js 官网提供的，还有一小部分是我自己总结的。

在撰写本书时，Vue.js 的最新版本是 2.5.2，所以本书中的代码参考该版本进行撰写。如果你想对照源码来阅读本书，可以在 GitHub 上找出该版本的源码。此外，关于本书的任何意见和建议，都可以在这里讨论：<https://github.com/berwin/Blog/issues/34>。关于本书的微信群，也请参见这个页面。

## 致谢

这本书的诞生我要感谢很多人。我曾幻想过如果有一天自己能出版一本技术书，那该有多好，但从来没有想到这一天来得这么快，我更想象不到这一天会在我 23 岁时发生。在我看来，这件事不可能发生在我的身上，但它确确实实发生了。

这一切都要感谢王军花老师，是她给了我这个机会。最初她找到我，问我有没有兴趣写一本深入介绍 Vue.js 的书时，我的内心很挣扎。因为这可以实现我的一个梦想，但我又担心自己写不好，觉得自己不够资格出版一本书。最终经过激烈的思想斗争后，我决定接受这个挑战，做一些让自己佩服自己的事。

不止是感谢军花老师给我这个机会，我还非常感谢她前前后后跟进这本书，包括书的进度以及与我一起审校和修改这本书等很多事情，非常感谢！

其次我要感谢我的领导 LC（梁超）和肆爷（何炼），当他们听说我要写一本书时，给了我很大的帮助和支持。本书没有拖稿，我按时写完了所有章节，这一切都是源于他们对我的大力支持。

持。如果没有他们，我想我也没有办法按时交稿，非常感谢！

同时我也非常感谢李松峰老师，在开始写这本书时，李老师给我讲了很多写作方面的技巧，并且教我怎样写一本好书，怎么写出阅读体验良好的书。并且在这本书写完之后，李老师还答应给这本书写序，真的非常感谢！

我更要感谢我的父母，感谢你们对我多年的养育之恩，辛辛苦苦把我养大。如今，我虽有了一份稳定的工作，但回家的次数却越来越少。我很愧疚不能在你们身边工作，不能经常陪在你们身边。现在，我出版了一本书，不知道你们会不会为我感到骄傲。

我还要感谢堂姐王砚天，在写作期间给了我很多精神鼓励与支持，并且给我买了很多好吃的。

除此之外，我要感谢第一批内测读者（刘冰晶、姚向阳、周延博、王建兵、陈凤），感谢你们的阅读以及给我提供的宝贵修改意见，非常感谢！

最后，我要感谢正在阅读这部分的你，感谢你阅读本书，感谢你对我的支持，谢谢！

# 目 录

<b>第 1 章</b>	<b>Vue.js 简介</b>	1
1.1	什么是 Vue.js	1
1.2	Vue.js 简史	2
<b>第一篇 变化侦测</b>		
<b>第 2 章</b>	<b>Object 的变化侦测</b>	6
2.1	什么是变化侦测	6
2.2	如何追踪变化	7
2.3	如何收集依赖	7
2.4	依赖收集在哪里	8
2.5	依赖是谁	10
2.6	什么是 Watcher	10
2.7	递归侦测所有 key	12
2.8	关于 Object 的问题	13
2.9	总结	14
<b>第 3 章</b>	<b>Array 的变化侦测</b>	16
3.1	如何追踪变化	16
3.2	拦截器	17
3.3	使用拦截器覆盖 Array 原型	18
3.4	将拦截器方法挂载到数组的属性上	19
3.5	如何收集依赖	21
3.6	依赖列表存在哪儿	22
3.7	收集依赖	23
3.8	在拦截器中获取 Observer 实例	24
3.9	向数组的依赖发送通知	25
3.10	侦测数组中元素的变化	26
3.11	侦测新增元素的变化	27
3.11.1	获取新增元素	27
3.11.2	使用 Observer 侦测新增元素	28
3.12	关于 Array 的问题	29
3.13	总结	29
<b>第 4 章</b>	<b>变化侦测相关的 API 实现原理</b>	31
4.1	vm.\$watch	31
4.1.1	用法	31
4.1.2	watch 的内部原理	32
4.1.3	deep 参数的实现原理	36
4.2	vm.\$set	38
4.2.1	用法	38
4.2.2	Array 的处理	39
4.2.3	key 已经存在于 target 中	40
4.2.4	处理新增的属性	40
4.3	vm.\$delete	41
4.3.1	用法	42
4.3.2	实现原理	42
4.4	总结	45
<b>第二篇 虚拟 DOM</b>		
<b>第 5 章</b>	<b>虚拟 DOM 简介</b>	48
5.1	什么是虚拟 DOM	48
5.2	为什么要引入虚拟 DOM	51
5.3	Vue.js 中的虚拟 DOM	51
5.4	总结	53
<b>第 6 章</b>	<b>vNode</b>	54
6.1	什么是 vNode	54

## 2 目录

6.2 VNode 的作用.....	55	第 9 章 解析器.....	93
6.3 VNode 的类型.....	56	9.1 解析器的作用.....	93
6.3.1 注释节点.....	57	9.2 解析器内部运行原理.....	94
6.3.2 文本节点.....	57	9.3 HTML 解析器.....	99
6.3.3 克隆节点.....	57	9.3.1 运行原理.....	100
6.3.4 元素节点.....	58	9.3.2 截取开始标签.....	101
6.3.5 组件节点.....	59	9.3.3 截取结束标签.....	107
6.3.6 函数式组件.....	59	9.3.4 截取注释.....	108
6.4 总结.....	59	9.3.5 截取条件注释.....	108
<b>第 7 章 patch .....</b>	<b>60</b>	9.3.6 截取 DOCTYPE.....	109
7.1 patch 介绍.....	60	9.3.7 截取文本.....	109
7.1.1 新增节点.....	61	9.3.8 纯文本内容元素的处理.....	112
7.1.2 删除节点.....	62	9.3.9 使用栈维护 DOM 层级.....	114
7.1.3 更新节点.....	63	9.3.10 整体逻辑.....	114
7.1.4 小结.....	63	9.4 文本解析器.....	117
7.2 创建节点.....	64	9.5 总结.....	121
7.3 删除节点.....	67	<b>第 10 章 优化器 .....</b>	<b>122</b>
7.4 更新节点.....	68	10.1 找出所有静态节点并标记.....	125
7.4.1 静态节点.....	68	10.2 找出所有静态根节点并标记.....	127
7.4.2 新虚拟节点有文本属性.....	69	10.3 总结.....	129
7.4.3 新虚拟节点无文本属性.....	69	<b>第 11 章 代码生成器 .....</b>	<b>130</b>
7.4.4 小结.....	70	11.1 通过 AST 生成代码字符串.....	131
7.5 更新子节点.....	72	11.2 代码生成器的原理.....	134
7.5.1 更新策略.....	72	11.2.1 元素节点.....	134
7.5.2 优化策略.....	77	11.2.2 文本节点.....	136
7.5.3 哪些节点是未处理过的.....	82	11.2.3 注释节点.....	137
7.5.4 小结.....	83	11.3 总结.....	137
7.6 总结.....	86	<b>第四篇 整体流程</b>	
<b>第三篇 模板编译原理</b>		<b>第 12 章 架构设计与项目结构 .....</b>	<b>140</b>
<b>第 8 章 模板编译 .....</b>	<b>88</b>	12.1 目录结构.....	140
8.1 概念.....	88	12.2 架构设计.....	143
8.2 将模板编译成渲染函数.....	89	12.3 总结.....	145
8.2.1 解析器.....	90	<b>第 13 章 实例方法与全局 API 的实现原理 .....</b>	<b>146</b>
8.2.2 优化器.....	91	13.1 数据相关的实例方法.....	147
8.2.3 代码生成器.....	91		
8.3 总结.....	92		

13.2 事件相关的实例方法.....	147	14.7 初始化状态.....	215
13.2.1 <code>vm.\$on</code> .....	148	14.7.1 初始化 <code>props</code> .....	216
13.2.2 <code>vm.\$off</code> .....	149	14.7.2 初始化 <code>methods</code> .....	224
13.2.3 <code>vm.\$once</code> .....	152	14.7.3 初始化 <code>data</code> .....	225
13.2.4 <code>vm.\$emit</code> .....	153	14.7.4 初始化 <code>computed</code> .....	228
13.3 生命周期相关的实例方法.....	154	14.7.5 初始化 <code>watch</code> .....	238
13.3.1 <code>vm.\$forceUpdate</code> .....	154	14.8 初始化 <code>provide</code> .....	241
13.3.2 <code>vm.\$destroy</code> .....	155	14.9 总结 .....	241
13.3.3 <code>vm.\$nextTick</code> .....	159		
13.3.4 <code>vm.\$mount</code> .....	169		
13.4 全局 API 的实现原理 .....	178	<b>第 15 章 指令的奥秘 .....</b>	242
13.4.1 <code>Vue.extend</code> .....	178	15.1 指令原理概述 .....	242
13.4.2 <code>Vue.nextTick</code> .....	182	15.1.1 <code>v-if</code> 指令的原理概述 .....	243
13.4.3 <code>Vue.set</code> .....	183	15.1.2 <code>v-for</code> 指令的原理概述 .....	243
13.4.4 <code>Vue.delete</code> .....	183	15.1.3 <code>v-on</code> 指令 .....	244
13.4.5 <code>Vue.directive</code> .....	184	15.2 自定义指令的内部原理 .....	246
13.4.6 <code>Vue.filter</code> .....	185	15.3 虚拟 DOM 钩子函数 .....	250
13.4.7 <code>Vue.component</code> .....	186	15.4 总结 .....	251
13.4.8 <code>Vue.use</code> .....	188		
13.4.9 <code>Vue.mixin</code> .....	189	<b>第 16 章 过滤器的奥秘 .....</b>	252
13.4.10 <code>Vue.compile</code> .....	190	16.1 过滤器原理概述 .....	253
13.4.11 <code>Vue.version</code> .....	190	16.1.1 串联过滤器 .....	254
13.5 总结 .....	191	16.1.2 滤器接收参数 .....	254
<b>第 14 章 生命周期 .....</b>	192	16.1.3 <code>resolveFilter</code> 的内部原理 .....	255
14.1 生命周期图示 .....	192	16.2 解析过滤器 .....	256
14.1.1 初始化阶段 .....	193	16.3 总结 .....	258
14.1.2 模板编译阶段 .....	194		
14.1.3 挂载阶段 .....	194	<b>第 17 章 最佳实践 .....</b>	259
14.1.4 卸载阶段 .....	194	17.1 为列表渲染设置属性 <code>key</code> .....	259
14.1.5 小结 .....	194	17.2 在 <code>v-if/v-if-else/v-else</code> 中使用 <code>key</code> .....	259
14.2 从源码角度了解生命周期 .....	195	17.3 路由切换组件不变 .....	260
14.3 <code>errorCaptured</code> 与错误处理 .....	199	17.3.1 路由导航守卫 <code>beforeRouteUpdate</code> .....	261
14.4 初始化实例属性 .....	203	17.3.2 观察 <code>\$route</code> 对象的变化 .....	261
14.5 初始化事件 .....	204	17.3.3 为 <code>router-view</code> 组件添加属性 <code>key</code> .....	262
14.6 初始化 <code>inject</code> .....	208	17.4 为所有路由统一添加 <code>query</code> .....	262
14.6.1 <code>provide/inject</code> 的使用方式 .....	208	17.4.1 使用全局守卫 <code>beforeEach</code> .....	263
14.6.2 <code>inject</code> 的内部原理 .....	210	17.4.2 使用函数劫持 .....	263

4 目 录

---

17.5 区分 Vuex 与 props 的使用边界	264	17.10.9 JS/JSX 中的组件名	
17.6 避免 v-if 和 v-for 一起使用	264	大小写	274
17.7 为组件样式设置作用域	266	17.11 自闭合组件	275
17.8 避免在 scoped 中使用元素选择器	267	17.12 prop 名的大小写	276
17.9 避免隐性的父子组件通信	268	17.13 多个特性的元素	276
17.10 单文件组件如何命名	268	17.14 模板中简单的表达式	276
17.10.1 单文件组件的文件名的 大小写	268	17.15 简单的计算属性	277
17.10.2 基础组件名	269	17.16 指令缩写	278
17.10.3 单例组件名	270	17.17 良好的代码顺序	278
17.10.4 紧密耦合的组件名	270	17.17.1 组件/实例的选项的顺序	278
17.10.5 组件名中的单词顺序	271	17.17.2 元素特性的顺序	280
17.10.6 完整单词的组件名	272	17.17.3 单文件组件顶级元素的 顺序	281
17.10.7 组件名为多个单词	273	17.18 总结	282
17.10.8 模板中的组件名大小写	273		

## 第 1 章

# Vue.js 简介

# 1

在过去的 10 年时间里，网页变得更加动态化和强大了。通过 JavaScript，我们已经可以把很多传统的服务端代码放到浏览器中。身为一名前端工程师，我们所面临的需求变得越来越复杂。

当应用程序开始变复杂后，我们需要频繁操作 DOM。由于缺乏正规的组织形式，我们的代码变得非常难以维护。

这本质上是命令式操作 DOM 的问题，我们曾经用 jQuery 操作 DOM 写需求，但是当应用程序变复杂后，代码就像一坨意大利面一样，有点难以维护。我们无法继续使用命令式操作 DOM，所以 Vue.js 提供了声明式操作 DOM 的能力来解决这个问题。

通过描述状态和 DOM 之间的映射关系，就可以将状态渲染成 DOM 呈现在用户界面中，也就是渲染到网页上。

## 1.1 什么是 Vue.js

Vue.js，通常简称为 Vue，是一款友好的、多用途且高性能的 JavaScript 框架，能够帮助我们创建可维护性和可测试性更强的代码。它是目前所有主流框架中学习曲线最平缓的框架，非常容易上手，其官方文档也写得非常清晰、易懂。

它是一款渐进式的 JavaScript 框架。关于什么是渐进式，其实一开始我琢磨了好久，后来才弄懂，就是说如果你已经有一个现成的服务端应用，也就是非单页应用，可以将 Vue.js 作为该应用的一部分嵌入其中，带来更加丰富的交互体验。

如果希望将更多业务逻辑放到前端来实现，那么 Vue.js 的核心库及其生态系统也可以满足你的各种需求。和其他前端框架一样，Vue.js 允许你将一个网页分割成可复用的组件，每个组件都有自己的 HTML、CSS 和 JavaScript 来渲染网页中一个对应的位置。

如果要构建一个大型应用，就需要先搭建项目，配置一些开发环境等。Vue.js 提供了一个命令行工具，它让快速初始化一个真实的项目工程变得非常简单。

我们甚至可以使用 Vue.js 的单文件组件，它包含各自的 HTML、JavaScript 以及带作用域的 CSS 或 SCSS。我本人在使用 Vue.js 开发项目时，通常都会使用单文件组件。单文件组件真的是

一个非常棒的特性，它可以使项目架构变得非常清晰、可维护。

## 1.2 Vue.js 简史

2013年7月28日，有一位名叫尤雨溪，英文名叫Evan You的人在GitHub上第一次为Vue.js提交代码。这是Vue.js的第一个提交(commit)，但这时还不叫Vue.js。从仓库的package.json文件可以看出，这时的名字叫作Element，后来被更名为Seed.js。

2013年12月7日，尤雨溪在GitHub上发布了新版本0.6.0，将项目正式改名为Vue.js，并且把默认的指令前缀变成v-。这一版本的发布，代表Vue.js正式问世。

2014年2月1日，尤雨溪将Vue.js 0.8发布在了国外的Hacker News网站，这代表它首次公开发行。听尤雨溪说，当时被顶到了Hacker News的首页，在一周的时间内拿到了615个GitHub的star，他特别兴奋。

从这之后，经过近两年的孵化，直到2015年10月26日这天，Vue.js终于迎来了1.0.0版本的发布。我不知道当时尤雨溪的心情是什么样的，但从他发布版本时所带的格言可以看出，他心里一定很复杂。

那句话是：

“The fate of destruction is also the joy of rebirth.”

翻译成中文是：

毁灭的命运，也是重生的喜悦。

并且为1.0.0这个版本配备了一个代号，叫新世纪福音战士(Evangelion)，这是一部动画片的名字。事实上，Vue.js每一次比较大的版本发布，都会配一个动画片的名称作为代号。

2016年10月1日，这一天是祖国的生日，但同时也是Vue.js 2.0发布的日子。Vue.js 2.0的代号叫攻壳机动队(Ghost in the Shell)。

同时，这一次尤雨溪发布这个版本时所带的格言是：

“Your effort to remain what you are is what limits you.”

翻译成中文是：

保持本色的努力，也在限制你的发展。

在开发Vue.js的整个过程中，它的定位发生了变化，一开始的定位是：

“Just a view layer library”

就是说，最早的Vue.js只做视图层，没有路由，没有状态管理，也没有官方的构建工具，只有一

个库，放在网页里就直接用。

后来，他发现 Vue.js 无法用在一些大型应用上，这样在开发不同大小的应用时，需要不停地切换框架以及思维模式。尤雨溪希望有一个方案，有足够的灵活性，能够适应不同大小的应用需求。

所以，Vue.js 就慢慢开始加入了一些官方的辅助工具，比如路由（Router）、状态管理方案（Vuex）和构建工具（vue-cli）等。

加入这些工具时，Vue.js 始终维持着一个理念：“这个框架应该是渐进式的。”

这时 Vue.js 的定位是：

The Progressive Framework

翻译成中文，就是渐进式框架。

所谓渐进式框架，就是把框架分层。

最核心的部分是视图层渲染，然后往外是组件机制，在这个基础上再加入路由机制，再加入状态管理，最外层是构建工具，如图 1-1 所示。

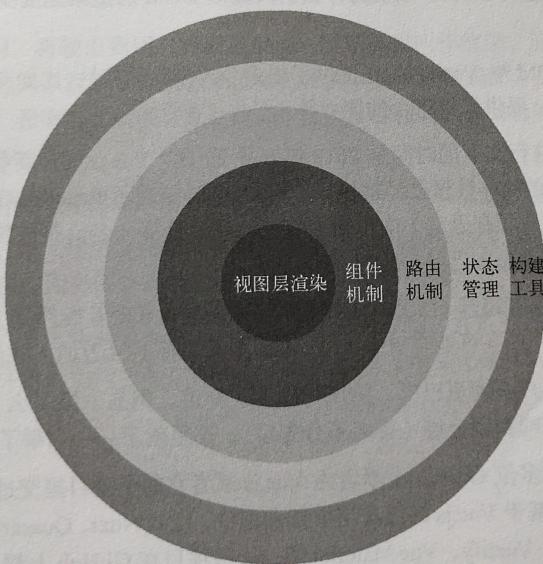


图 1-1 框架分层

所谓分层，就是说你既可以只用最核心的视图层渲染功能来快速开发一些需求，也可以使用一整套全家桶来开发大型应用。Vue.js 有足够的灵活性来适应不同的需求，所以你可以根据自己 的需求选择不同的层级。

Vue.js 2.0 与 Vue.js 1.0 之间内部变化非常大，整个渲染层都重写了，但 API 层面的变化却很小。可以看出，Vue.js 是非常注重用户体验和学习曲线的，它尽量让开发者用起来很爽，同时在应用场景上，其他框架能做到的 Vue.js 都能做到，不存在其他框架可以实现而 Vue.js 不能实现这样的问题，所以在技术选型上，只需要考虑 Vue.js 的使用方式是不是符合口味，团队来了新同学能否快速融入等问题。由于无论是学习曲线还是 API 的设计上，Vue.js 都非常优雅，所以它具有很强的竞争力。

Vue.js 2.0 引入了非常多的特性，其中一个明显的效果是 Vue.js 变得更轻、更快了。

Vue.js 2.0 引入了虚拟 DOM，其渲染过程变得更快了。虚拟 DOM 现在已经被网上说烂了，但是我想说的是，不要人云亦云。Vue.js 引入虚拟 DOM 是有原因的。事实上，并不是引入虚拟 DOM 后，渲染速度变快了。准确地说，应该是 80% 的场景下变得更快了，而剩下的 20% 反而变慢了。

任何技术的引入都是在解决一些问题，而通常解决一个问题的同时会引发另外一个问题，这种情况更多的是做权衡，做取舍。所以，不要像网上大部分人那样，成天说因为引入了虚拟 DOM 而变快了。我们要透过现象看本质，本书的目的也在于此。

关于为什么引入虚拟 DOM，以及为什么引入虚拟 DOM 后渲染速度变快了，第 5 章会详细介绍。

除了引入虚拟 DOM 外，Vue.js 2.0 还提供了很多令人激动的特性，比如支持 JSX 和 TypeScript，支持流式服务端渲染，提供了跨平台的能力等。

到目前，我写下这行文字的时间是 2018 年 6 月 29 日，Vue.js 的最新版本是 2.5.16。就在前几天，它在 GitHub 上的 star 数量已经超过了 10 万，同时超越了 React 在 GitHub 上的 star 数量。在 GitHub 上所有项目（所有语言）中排进了前五，目前是第 4 名，挤进前三指日可待。可能你在读这行文字的时候，Vue.js 已经挤进前三了。

目前，Vue.js 每个月有超过 115 万次 NPM 下载，Chrome 开发者插件有 17.4 万周活跃用户（这是 2017 年 5 月的数据，现在可能会更多），这表示每天都有 17.4 万的人在使用它开发应用。

Vue.js 在国内的用户有阿里巴巴、腾讯、百度、新浪、网易、饿了么、滴滴出行、360、美团、苏宁、58、哔哩哔哩和掘金等（排名不分先后），这里就不一一列举了。

在社区上，有 300 多位 GitHub 贡献者为 Vue.js 或者它的子项目提交过代码。社区项目也非常活跃，社区上有很多基于 Vue.js 的更高层框架和组件，比如 Nuxt、Quasar Framework、Element、iView、Muse-UI、Vux、Vuetify、Vue Material 等，这些项目在 GitHub 上都是几千个 star 的项目。

说了这么多，我想说的是，Vue.js 已是一名前端工程师必备的技能。而想深入了解 Vue.js 内部的核心技术原理，就来阅读本书吧。

## 第一篇

# 变化侦测

Vue.js 最独特的特性之一是看起来并不显眼的响应式系统。数据模型仅仅是普通的 JavaScript 对象。而当你修改它们时，视图会进行更新。这使得状态管理非常简单、直接。不过理解其工作原理同样重要，这样你可以回避一些常见的问题。——官方文档

从状态生成 DOM，再输出到用户界面显示的一整套流程叫作渲染，应用在运行时会不断地进行重新渲染。而响应式系统赋予框架重新渲染的能力，其重要组成部分是变化侦测。变化侦测是响应式系统的核心，没有它，就没有重新渲染。框架在运行时，视图也就无法随着状态的变化而变化。

简单来说，变化侦测的作用是侦测数据的变化。当数据变化时，会通知视图进行相应的更新。

正如文档中所说，深入理解变化侦测的工作原理，既可以帮助我们在开发应用时回避一些很常见的问题，也可以在应用程序出问题时，快速调试并修复问题。

本篇中，我们将针对变化侦测的实现原理做一个详细介绍，并且会带着你一步一步从 0 到 1 实现一个变化侦测的逻辑。学完本篇，你将可以自己实现一个变化侦测的功能。

# Object 的变化侦测

# 2

大部分人不会想到 Object 和 Array 的变化侦测采用不同的处理方式。事实上，它们的侦测方式确实不一样。在这一章中，我们将详细介绍 Object 的变化侦测。

## 2.1 什么是变化侦测

Vue.js 会自动通过状态生成 DOM，并将其输出到页面上显示出来，这个过程叫渲染。Vue.js 的渲染过程是声明式的，我们通过模板来描述状态与 DOM 之间的映射关系。

通常，在运行时应用内部的状态会不断发生变化，此时需要不停地重新渲染。这时如何确定状态中发生了什么变化？

变化侦测就是用来解决这个问题的，它分为两种类型：一种是“推”（push），另一种是“拉”（pull）。

Angular 和 React 中的变化侦测都属于“拉”，这就是说当状态发生变化时，它不知道哪个状态变了，只知道状态有可能变了，然后会发送一个信号告诉框架，框架内部收到信号后，会进行一个暴力比对来找出哪些 DOM 节点需要重新渲染。这在 Angular 中是脏检查的流程，在 React 中使用的是虚拟 DOM。

而 Vue.js 的变化侦测属于“推”。当状态发生变化时，Vue.js 立刻就知道了，而且在一定程度上知道哪些状态变了。因此，它知道的信息更多，也就可以进行更细粒度的更新。

所谓更细粒度的更新，就是说：假如有一个状态绑定着好多个依赖，每个依赖表示一个具体的 DOM 节点，那么当这个状态发生变化时，向这个状态的所有依赖发送通知，让它们进行 DOM 更新操作。相比较而言，“拉”的粒度是最粗的。

但是它也有一定的代价，因为粒度越细，每个状态所绑定的依赖就越多，依赖追踪在内存上的开销就会越大。因此，从 Vue.js 2.0 开始，它引入了虚拟 DOM，将粒度调整为中等粒度，即一个状态所绑定的依赖不再是具体的 DOM 节点，而是一个组件。这样状态变化后，会通知到组件，组件内部再使用虚拟 DOM 进行比对。这可以大大降低依赖数量，从而降低依赖追踪所消耗的内存。

Vue.js之所以能随意调整粒度，本质上还要归功于变化侦测。因为“推”类型的变化侦测可以随意调整粒度。

## 2.2 如何追踪变化

2

关于变化侦测，首先要问一个问题，在JavaScript（简称JS）中，如何侦测一个对象的变化？

其实这个问题还是比较简单的。学过JavaScript的人都知道，有两种方法可以侦测到变化：使用`Object.defineProperty`和ES6的`Proxy`。

由于ES6在浏览器中的支持度并不理想，到目前为止Vue.js还是使用`Object.defineProperty`来实现的，所以书中也会使用它来介绍变化侦测的原理。

由于使用`Object.defineProperty`来侦测变化会有很多缺陷，所以Vue.js的作者尤雨溪说日后会使用`Proxy`重写这部分代码。好在本章讲的是原理和思想，所以即便以后用`Proxy`重写了这部分代码，书中介绍的原理也不会变。

知道了`Object.defineProperty`可以侦测到对象的变化，那么我们可以写出这样的代码：

```

01 function defineReactive (data, key, val) {
02   Object.defineProperty(data, key, {
03     enumerable: true,
04     configurable: true,
05     get: function () {
06       return val
07     },
08     set: function (newVal) {
09       if(val === newVal){
10         return
11       }
12       val = newVal
13     }
14   })
15 }
```

这里的函数`defineReactive`用来对`Object.defineProperty`进行封装。从函数的名字可以看出，其作用是定义一个响应式数据。也就是在这个函数中进行变化追踪，封装后只需要传递`data`、`key`和`val`就行了。

封装好之后，每当从`data`的`key`中读取数据时，`get`函数被触发；每当往`data`的`key`中设置数据时，`set`函数被触发。

## 2.3 如何收集依赖

如果只是把`Object.defineProperty`进行封装，那其实也没什么实际用处，真正有用的是收集依赖。

现在我要问第二个问题：如何收集依赖？

思考一下，我们之所以要观察数据，其目的是当数据的属性发生变化时，可以通知那些曾经使用了该数据的地方。

举个例子：

```
01 <template>
02   <h1>{{ name }}</h1>
03 </template>
```

该模板中使用了数据 name，所以当它发生变化时，要向使用了它的地方发送通知。

**注意** 在 Vue.js 2.0 中，模板使用数据等同于组件使用数据，所以当数据发生变化时，会将通知发送到组件，然后组件内部再通过虚拟 DOM 重新渲染。

对于上面的问题，我的回答是，先收集依赖，即把用到数据 name 的地方收集起来，然后等属性发生变化时，把之前收集好的依赖循环触发一遍就好了。

总结起来，其实就一句话，在 getter 中收集依赖，在 setter 中触发依赖。

## 2.4 依赖收集在哪里

现在我们已经有了很明确的目标，就是要在 getter 中收集依赖，那么要把依赖收集到哪里去呢？

思考一下，首先想到的是每个 key 都有一个数组，用来存储当前 key 的依赖。假设依赖是一个函数，保存在 window.target 上，现在就可以把 defineReactive 函数稍微改造一下：

```
01 function defineReactive (data, key, val) {
02   let dep = [] // 新增
03   Object.defineProperty(data, key, {
04     enumerable: true,
05     configurable: true,
06     get: function () {
07       dep.push(window.target) // 新增
08       return val
09     },
10     set: function (newVal) {
11       if(val === newVal){
12         return
13       }
14       // 新增
15       for (let i = 0; i < dep.length; i++) {
16         dep[i](newVal, val)
17       }
18       val = newVal
19     }
20   })
21 }
```

```
20     })
21 }
```

这里我们新增了数组 dep，用来存储被收集的依赖。

然后在 set 被触发时，循环 dep 以触发收集到的依赖。

2

但是这样写有点耦合，我们把依赖收集的代码封装成一个 Dep 类，它专门帮助我们管理依赖。

使用这个类，我们可以收集依赖、删除依赖或者向依赖发送通知等。其代码如下：

```
01  export default class Dep {
02    constructor () {
03      this.subs = []
04    }
05
06    addSub (sub) {
07      this.subs.push(sub)
08    }
09
10   removeSub (sub) {
11     remove(this.subs, sub)
12   }
13
14   depend () {
15     if (window.target) {
16       this.addSub(window.target)
17     }
18   }
19
20   notify () {
21     const subs = this.subs.slice()
22     for (let i = 0, l = subs.length; i < l; i++) {
23       subs[i].update()
24     }
25   }
26 }
27
28 function remove (arr, item) {
29   if (arr.length) {
30     const index = arr.indexOf(item)
31     if (index > -1) {
32       return arr.splice(index, 1)
33     }
34   }
35 }
```

之后再改造一下 defineReactive：

```
01  function defineReactive (data, key, val) {
02    let dep = new Dep() // 修改
03    Object.defineProperty(data, key, {
04      enumerable: true,
05      configurable: true,
06      get: function () {
```

```

07      dep.depend() // 修改
08      return val
09    },
10    set: function (newVal) {
11      if(val === newVal){
12        return
13      }
14      val = newVal
15      dep.notify() // 新增
16    }
17  })
18 }

```

此时代码看起来清晰多了，这也顺便回答了上面的问题，依赖收集到哪儿？收集到 Dep 中。

## 2.5 依赖是谁

在上面的代码中，我们收集的依赖是 window.target，那么它到底是什么？我们究竟要收集谁呢？

收集谁，换句话说，就是当属性发生变化后，通知谁。

我们要通知用到数据的地方，而使用这个数据的地方有很多，而且类型还不一样，既有可能是模板，也有可能是用户写的一个 watch，这时需要抽象出一个能集中处理这些情况的类。然后，我们在依赖收集阶段只收集这个封装好的类的实例进来，通知也只通知它一个。接着，它再负责通知其他地方。所以，我们要抽象的这个东西需要先起一个好听的名字。嗯，就叫它 Watcher 吧。

现在就可以回答上面的问题了，收集谁？Watcher！

## 2.6 什么是 Watcher

Watcher 是一个中介的角色，数据发生变化时通知它，然后它再通知其他地方。

关于 Watcher，先看一个经典的使用方式：

```

01 // keypath
02 vm.$watch('a.b.c', function (newVal, oldVal) {
03   // 做点什么
04 })

```

这段代码表示当 data.a.b.c 属性发生变化时，触发第二个参数中的函数。

思考一下，怎么实现这个功能呢？好像只要把这个 watcher 实例添加到 data.a.b.c 属性的 Dep 中就行了。然后，当 data.a.b.c 的值发生变化时，通知 Watcher。接着，Watcher 再执行参数中的这个回调函数。

好，思考完毕，写出如下代码：

```

01  export default class Watcher {
02    constructor (vm, expOrFn, cb) {
03      this.vm = vm
04      // 执行 this.getter(), 就可以读取 data.a.b.c 的内容
05      this.getter = parsePath(expOrFn)
06      this.cb = cb
07      this.value = this.get()
08    }
09
10   get() {
11     window.target = this
12     let value = this.getter.call(this.vm, this.vm)
13     window.target = undefined
14     return value
15   }
16
17   update () {
18     const oldValue = this.value
19     this.value = this.get()
20     this.cb.call(this.vm, this.value, oldValue)
21   }
22 }

```

2

这段代码可以把自己主动添加到 `data.a.b.c` 的 Dep 中去，是不是很神奇？

因为我在 `get` 方法中先把 `window.target` 设置成了 `this`，也就是当前 `watcher` 实例，然后再读一下 `data.a.b.c` 的值，这肯定会触发 `getter`。

触发了 `getter`，就会触发收集依赖的逻辑。而关于收集依赖，上面已经介绍了，会从 `window.target` 中读取一个依赖并添加到 `Dep` 中。

这就导致，只要先在 `window.target` 赋一个 `this`，然后再读一下值，去触发 `getter`，就可以把 `this` 主动添加到 `keypath` 的 `Dep` 中。有没有很神奇的感觉啊？

依赖注入到 `Dep` 中后，每当 `data.a.b.c` 的值发生变化时，就会让依赖列表中所有的依赖循环触发 `update` 方法，也就是 `Watcher` 中的 `update` 方法。而 `update` 方法会执行参数中的回调函数，将 `value` 和 `oldValue` 传到参数中。

所以，其实不管是用户执行的 `vm.$watch('a.b.c', (value, oldValue) => {})`，还是模板中用到的 `data`，都是通过 `Watcher` 来通知自己是否需要发生变化。

这里有些小伙伴可能会好奇上面代码中的 `parsePath` 是怎么读取一个字符串的 `keypath` 的，下面用一段代码来介绍其实现原理：

```

01 /**
02  * 解析简单路径
03  */
04 const bailRE = /^[^\w.$]/
05 export function parsePath (path) {
06   if (bailRE.test(path)) {
07     return

```

```

08     }
09     const segments = path.split('.')
10    return function (obj) {
11      for (let i = 0; i < segments.length; i++) {
12        if (!obj) return
13        obj = obj[segments[i]]
14      }
15    return obj
16  }
17 }

```

可以看到，这其实并不复杂。先将 `keypath` 用 `.` 分割成数组，然后循环数组一层一层去读数据，最后拿到的 `obj` 就是 `keypath` 中想要读的数据。

## 2.7 递归侦测所有 key

现在，其实已经可以实现变化侦测的功能了，但是前面介绍的代码只能侦测数据中的某一个属性，我们希望把数据中的所有属性（包括子属性）都侦测到，所以要封装一个 `Observer` 类。这个类的作用是将一个数据内的所有属性（包括子属性）都转换成 `getter/setter` 的形式，然后去追踪它们的变化：

```

01 /**
02  * Observer 类会附加到每一个被侦测的 object 上。
03  * 一旦被附加上，Observer 会将 object 的所有属性转换为 getter/setter 的形式
04  * 来收集属性的依赖，并且当属性发生变化时会通知这些依赖
05 */
06 export class Observer {
07   constructor (value) {
08     this.value = value
09
10   if (!Array.isArray(value)) {
11     this.walk(value)
12   }
13 }
14
15 /**
16  * walk 会将每一个属性都转换成 getter/setter 的形式来侦测变化
17  * 这个方法只有在数据类型为 Object 时被调用
18 */
19 walk (obj) {
20   const keys = Object.keys(obj)
21   for (let i = 0; i < keys.length; i++) {
22     defineReactive(obj, keys[i], obj[keys[i]])
23   }
24 }
25 }
26
27 function defineReactive (data, key, val) {
28   // 新增，递归子属性
29   if (typeof val === 'object') {
30     new Observer(val)

```

```

31      }
32      let dep = new Dep()
33      Object.defineProperty(data, key, {
34        enumerable: true,
35        configurable: true,
36        get: function () {
37          dep.depend()
38          return val
39        },
40        set: function (newVal) {
41          if(val === newVal){
42            return
43          }
44          val = newVal
45          dep.notify()
46        }
47      })
48    })
49  }

```

在上面的代码中，我们定义了 `observer` 类，它用来将一个正常的 `object` 转换成被侦测的 `object`。

然后判断数据的类型，只有 `Object` 类型的数据才会调用 `walk` 将每一个属性转换成 `getter/setter` 的形式来侦测变化。

最后，在 `defineReactive` 中新增 `new Observer(val)` 来递归子属性，这样我们就可以把 `data` 中的所有属性（包括子属性）都转换成 `getter/setter` 的形式来侦测变化。

当 `data` 中的属性发生变化时，与这个属性对应的依赖就会接收到通知。

也就是说，只要我们将一个 `object` 传到 `Observer` 中，那么这个 `object` 就会变成响应式的 `object`。

## 2.8 关于 Object 的问题

前面介绍了 `Object` 类型数据的变化侦测原理，了解了数据的变化是通过 `getter/setter` 来追踪的。也正是由于这种追踪方式，有些语法中即便是数据发生了变化，`Vue.js` 也追踪不到。

比如，向 `object` 添加属性：

```

01  var vm = new Vue({
02    el: '#el',
03    template: '#demo-template',
04    methods: {
05      action () {
06        this.obj.name = 'berwin'
07      }
08    },
09    data: {

```

```

10     obj: {}
11   }
12 })

```

在 `action` 方法中，我们在 `obj` 上面新增了 `name` 属性，`Vue.js` 无法侦测到这个变化，所以不会向依赖发送通知。

再比如，从 `obj` 中删除一个属性：

```

01 var vm = new Vue({
02   el: '#el',
03   template: '#demo-template',
04   methods: {
05     action () {
06       delete this.obj.name
07     }
08   },
09   data: {
10     obj: {
11       name: 'berwin'
12     }
13   }
14 })

```

在上面的代码中，我们在 `action` 方法中删除了 `obj` 中的 `name` 属性，而 `Vue.js` 无法侦测到这个变化，所以不会向依赖发送通知。

`Vue.js` 通过 `Object.defineProperty` 来将对象的 `key` 转换成 `getter/setter` 的形式来追踪变化，但 `getter/setter` 只能追踪一个数据是否被修改，无法追踪新增属性和删除属性，所以才会导致上面例子中提到的问题。

但这也是没有办法的事，因为在 ES6 之前，`JavaScript` 没有提供元编程的能力，无法侦测到一个新属性被添加到了对象中，也无法侦测到一个属性从对象中删除了。为了解决这个问题，`Vue.js` 提供了两个 API——`vm.$set` 与 `vm.$delete`，第 4 章会详细介绍它们。

## 2.9 总结

变化侦测就是侦测数据的变化。当数据发生变化时，要能侦测到并发出通知。

`Object` 可以通过 `Object.defineProperty` 将属性转换成 `getter/setter` 的形式来追踪变化。读取数据时会触发 `getter`，修改数据时会触发 `setter`。

我们需要在 `getter` 中收集有哪些依赖使用了数据。当 `setter` 被触发时，去通知 `getter` 中收集的依赖数据发生了变化。

收集依赖需要为依赖找一个存储依赖的地方，为此我们创建了 `Dep`，它用来收集依赖、删除依赖和向依赖发送消息等。

所谓的依赖，其实就是 `Watcher`。只有 `Watcher` 触发的 `getter` 才会收集依赖，哪个 `Watcher`

触发了 getter，就把哪个 Watcher 收集到 Dep 中。当数据发生变化时，会循环依赖列表，把所有的 Watcher 都通知一遍。

Watcher 的原理是先把自己设置到全局唯一的指定位置（例如 `window.target`），然后读取数据。因为读取了数据，所以会触发这个数据的 getter。接着，在 getter 中就会从全局唯一的那个位置读取当前正在读取数据的 Watcher，并把这个 Watcher 收集到 Dep 中去。通过这样的方式，Watcher 可以主动去订阅任意一个数据的变化。

此外，我们创建了 Observer 类，它的作用是把一个 object 中的所有数据（包括子数据）都转换成响应式的，也就是它会侦测 object 中所有数据（包括子数据）的变化。

由于在 ES6 之前 JavaScript 并没有提供元编程的能力，所以在对象上新增属性和删除属性都无法被追踪到。

图 2-1 给出了 Data、Observer、Dep 和 Watcher 之间的关系。

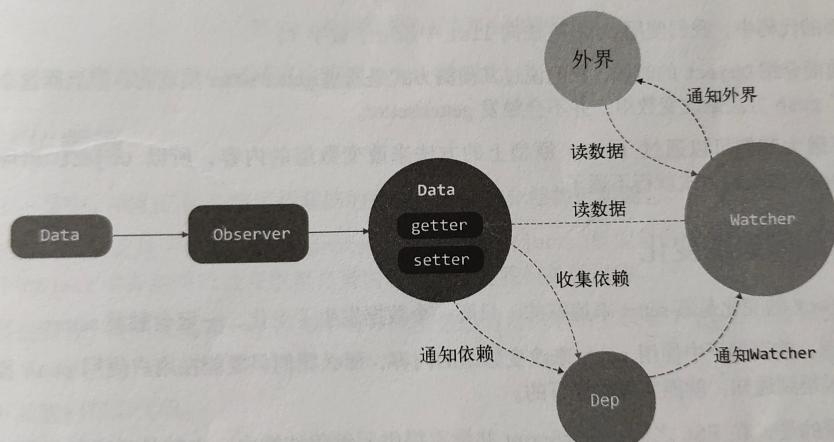


图 2-1 Data、Observer、Dep 和 Watcher 之间的关系

Data 通过 Observer 转换成了 getter/setter 的形式来追踪变化。

当外界通过 Watcher 读取数据时，会触发 getter 从而将 Watcher 添加到依赖中。

当数据发生了变化时，会触发 setter，从而向 Dep 中的依赖（Watcher）发送通知。

Dep 收到通知后，会向外界发送通知，变化通知到外界后可能会触发视图更新，也有可能触发用户的某个回调函数等。