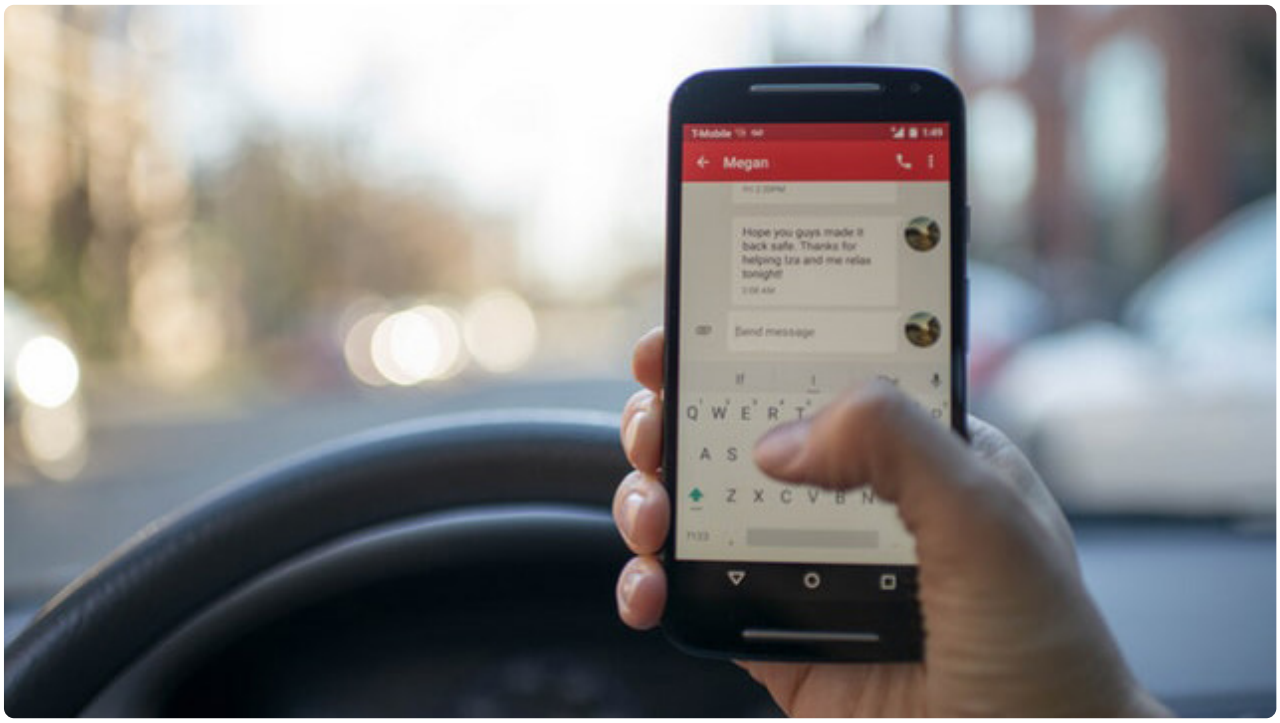
36 Vue核心——响应式原理源码级解析

更新时间：2020-08-28 10:43:16



> 只有在那崎岖的小路上不畏艰险奋勇攀登的人,才有希望达到光辉的顶点。——马克思

我们从本节开始，进入 Vue 的学习。

轮子变了，战术不变、思路不变——我们仍然是结合实际的面试场景、结合对命题思路的统计和归纳，划分出大的思路来复习。

Vue 面试，抓两个大头——响应式原理、nextTick，精准打击、深刻理解。辅以必要的关键知识点扫盲，确保知识结构没有硬伤。

但正如我在复习 React 基础时所提醒大家的，框架类题目有思路不难，覆盖全面却不可能。最重要的，还是“方法论”，是举一反三的能力。因此，在复习的过程中，我会结合之前的读者反馈，针对一些话题给到大家一些方法论上建议，也欢迎同学们来跟我探讨。

理解 Vue 响应式原理，我个人比较推荐的方法是结合源码来看。

到底要不要读源码

既然说到源码，那咱们干脆来聊聊读源码这个事情吧。

在过去这段时间里，我其实也收到过一部分同学的来信/微信消息，这部分同学很焦虑：一方面，他们急于完成面试所需的全盘知识点复习；另一方面，他们对自己的知识深度不够自信。出于“提高知识深度”的目的，他们觉得应该去读读源码；但碍于时间的限制，又不舍得划这么大一块“蛋糕”给这单一的一条知识线。思前想后，只剩下头疼和纠结。

源码要不要读？要读！不仅要读，还要系统地读、精细地读，一些涉及框架核心思想的模块，应该反复去读！但是——人，可以是灵活的，我们在面对不同的人生情境、不同的挑战时，应该有能力去调节自己的战术。

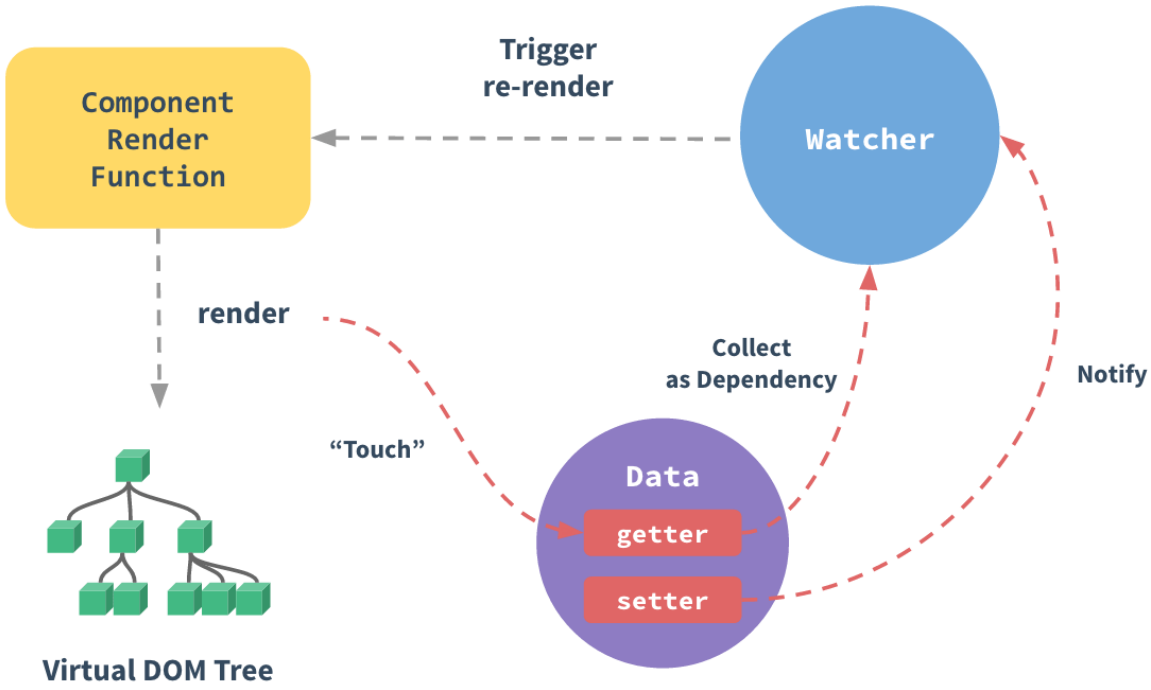
读源码，这是一个长期的、战略层面的考虑——通过系统阅读某一个框架的源码，你会对其设计模式和编码风格有更加深入的理解，这间接地也会提高你的水平。从这个角度来说，我推荐你读。

但是如果你是处在一个时间短、任务重、目标紧急的面试周期里，实话说，我真不想眼看你这么“挥霍”时间——对面试来说，重要的是“全局观”。

举个例子，假如你把 React 源码完全读了一遍，在 React 面试中拿到了100分（咱们先且不论读完源码就能拿框架类面试题100分这个假设其实是过分理想的），却因此忽略了对网络知识的复习，导致在这个环节只拿了40分；和你通读各类面试专题解读&面经后，有针对性地攻克少数重点知识，在框架层面和网络知识两块各拿到 70 分相比，面试官一定会毫不犹豫地选择后者！所以说最重要不是要你短期去成为某方面的专家，而是要求你尽快地解决自己的知识盲点！从这个角度来说，如果你是为了面试去大量读源码，实在是不合适。

Vue 响应式原理，你所需要知道的

首先，各位再熟悉不过的，一定是 Vue 官方提供的这张示意图了：



我们以这张图为基础，先帮助大家重新捋一遍响应式的机制。在这个基础上，再去做更进一步的分析。

注意我们图中有三个关键角色：Watcher、Data、和 Render，它们之间会上演这样的故事：

Vue 会对传入的 data 做处理：为每一个属性添加 getter 和 setter。在这个过程中，涉及到了 **Object.defineProperty** 这个方法。

同时每一个 Vue 组件实例，都对应着一个 watcher 实例；这个 watcher 实例仿佛一个跟踪狂，它的目光永远跟随着 data： 由于 render 函数的执行依赖于数据的读取，因此渲染时必定会读取 data 属性进而触发其对应的 getter 方法。getter 方法被调用后，会通知到 watcher，watcher 就会把这些 getter 方法被触发的属性记录为“依赖”——这一过程，就是大家常常听到的“**依赖收集**”过程。

如果 data 发生了更新，也就是说被“写”了，此时对应属性的 setter 方法就会被触发。setter 也会去通知 watcher，告诉它“我改变了”。watcher 拿到消息后，立刻跑去告诉 render：“data 变了，你也给我跟着变！”。由此去触发一个 re-render 的过程、与数据更新相关的组件会重新渲染。

响应式原理-源码层面的分析

通过上面这一通讲，咱们搞清楚了 Vue 响应式原理的机制。不过，单单搞清楚“机制”还不够——大家知道，Vue 相关的面试考点中，基础知识点居多。相对 React 来说，Vue 的理论性难点并不密集。

从另一个角度看，我们也可以理解为：Vue 的难点和重点是非常突出的、容易定位。比如说如果你让一个面试官只能出一道 Vue 面试题的话，他一定想都不想就直接问你响应式原理了。正是因为这个考察点如此地“没有悬念”、人人都能说上几句，导致了面试官们越来越倾向于抬高自己的标准——能说清楚，这还不够。既然机制没有区分度，那就问源码。所以，响应式原理相关的源码知识，各位需要引起重视。

这里，我会为大家提取出源码中的考点，抽离出源码中最契合面试需求的部分来作解析。

Object.defineProperty

`**Object.defineProperty()**` 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性， 并返回这个对象。

它的调用形式如下：

```
Object.defineProperty(obj, prop, descriptor)
```

其中第一个入参，是我们操作的目标对象；第二个入参，是我们需要修改的属性的名称；第三个入参，是一个描述符，用来描述你到底要对这个目标属性做什么。

我们在 Vue 响应式原理中涉及到的“描述符”，就是 getter/setter 方法：

> getter 方法： 一个给属性提供 getter 的方法，实际方法名为“get”；如果没有 getter 则为 `undefined`。当访问该属性时，该方法会被执行，方法执行时没有参数传入，但是会传入`this`对象（由于继承关系，这里的`this`并不一定是定义该属性的对象）。**默认为 undefined**。
>
> setter 方法： 一个给属性提供 setter 的方法，实际方法名为“set”；如果没有 setter 则为 `undefined`。当属性值修改时，触发执行该方法。该方法将接受唯一参数，即该属性新的参数值。**默认为 undefined**。

我们来看一个例子：

```
const obj = {
  name: 'xiuyan',
  career: 'coder'
}
Object.defineProperty(obj, 'career', {
  // getter 方法
  get() {
    console.log('尝试读取修言的工作')
  },
  // setter 方法
  set(newCareer) {
    console.log(`修言的工作换成了${newCareer}`)
  },
});
```

在这个例子中，我们定义了 career 这个属性的 getter 和 setter 方法，当我们尝试读取其 career 属性时：

```
> obj.career
  尝试读取修言的工作                                    VM946:8
```

就会触发 getter 方法，输出对应的文字。

如果我尝试修改 career 属性，那么 setter 方法就会被触发：

```
> obj.career = 'writer'
  修言的工作换成了writer                                VM946:12
< "writer"
```

Observer、Dep 和 Watcher 的关系

在源码层次，大家需要把握好这三个角色：

- Observer：处理 data 的家伙。它会给 data 安装 getter 和 setter，这些安装上的逻辑会联动 Dep 去完成依赖收集和更新的派发；
- Dep：实际通知 Watcher 的人。在 getter 和 setter 逻辑中，正是通过调度 Dep 来完成信息的收集、以及和 Watcher 间的通信；
- Watcher：Watcher 被通知之后，就会通知 render、进而触发重渲染了。

Observer

Observer 的作用是遍历所有的属性，给它们安装上 getter/setter 方法：

```
class Observer {
  constructor() {
    // 具体逻辑在 observe 函数里
    observe(this.data);
  }
}

function observe (data) {
  // 取出所有的 key
  const keys = Object.keys(data);
  // 遍历所有属性
  for (let i = 0; i < keys.length; i++) {
    // 绑定 getter/setter 方法
    defineReactive(obj, keys[i]);
  }
}
```

这里我们看到，具体的绑定操作是在 defineReactive 里做的：

```
function defineReactive (obj, key, val) {
  // 定义一个 Dep 对象，它的作用正如我们上文所说
  const dep = new Dep();

  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get() {
      // 收集依赖、关联到 watcher
      dep.depend();
      return val;
    },
    set(newVal) {
      if (newVal === val) return;
      // 感知更新、通知 watcher
      dep.notify();
    }
  });
}
```

在 defineReactive 里面，每一个 getter/setter 里面都出现了 Dep 实例。正如我们前面所介绍的一样，实际收集信息和通知 watcher 的工作是 Dep 来做的。**每一个属性都对应一个单独的 Dep 实例**。

在 getter 方法里面，调用了 dep 的 depend 方法，这个方法有什么玄机呢？我们来看看 Dep 的结构：

Dep

Dep 的角色，宛如一个“工具人”，它是 Watcher 和 Observer 之间的纽带，是“通信兵”：

```
class Dep {
  constructor () {
    // 存储 Watcher 实例的数组
    this.subs = []
  }

  // 将 watcher 实例添加到 subs 中（这个方法在 Watcher 类的实现里会用到）
  addSub (sub: Watcher) {
    this.subs.push(sub)
  }

  // 收集依赖
  depend() {
    // Dep.target 实际上就是当前 Dep 对应的 watcher，我们下文会提及
    if (Dep.target) {
      // 把当前的 dep 实例关联到组件对应的 watcher 上去
      Dep.target.addDep(this)
    }
  }

  // 通知 watcher 对象发生更新
  notify () {
    const subs = this.subs.slice()
    // 这里 subs 的元素是 watcher 实例，逐个调用 watcher 实例的 update 方法
    for (let i = 0, l = subs.length; i < l; i++) {
      subs[i].update()
    }
  }
}
```

在 Dep 内部，会维护一个 watcher 队列。

depend 方法在每次 getter 触发时都会把 watcher 实例和 dep 实例做一次关联。

在 setter 触发时，dep 实例便会逐个通知每一个和自己有关联的 watcher：我对应的属性发生了更新！进而调度 watcher 实例的 update 方法，实现视图更新。

Watcher

```
class Watcher {

  constructor() {
    ...
    // Dep 的 target 属性是有赋值过程的^_^，它是组件对应的 watcher 对象
    Dep.target = this
      ...
  }

  addDep (dep: Dep) {
    ...
    // 把当前的 watcher 推入 dep 实例的 watcher 队列（subs）里去
    dep.addSub(this)
    ...
  }

  update() {
    // 更新视图
  }
}
```

这里需要大家注意一点：宏观上看，咱们说“收集依赖”，是指 watcher 去收集自己所依赖的数据属性；不过从实现上来看，实际上是把 watcher 对象推入了 dep 实例的队列里，更像是 dep 在“收集” watcher。

其实，不管是谁来维护队列、谁“收集”谁，其本质目的都是建立起 dep 和 watcher 间的关联，达到 dep 发生变化后可以立刻通知到 watcher 的目的。

}

35 React16题眼：理解Fiber思想

37 Vue核心——nextTick原理源码级解析