# 07 内在(深入响应式原理)

## 目录(Catalog)



## 生词(New Words)

## 内容(Content)

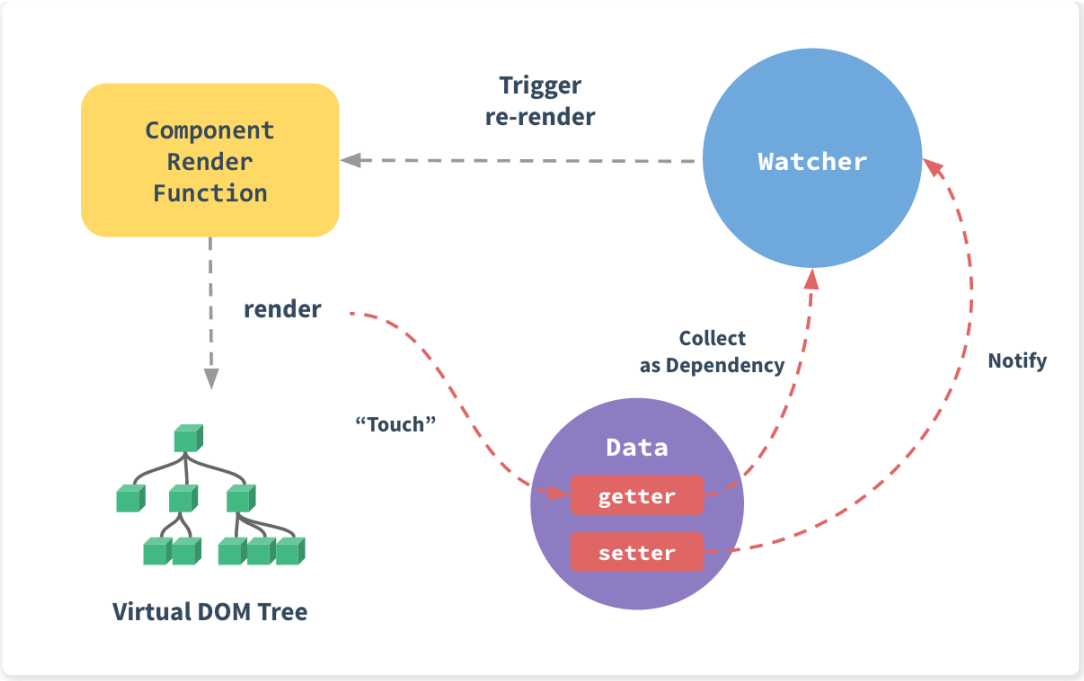
- 现在是时候深入一下了！Vue 最独特的特性之一，是其非侵入性的响应式系统。 数据模型仅仅是普通的 JavaScript 对象。而当你修改它们时，视图会进行更新。 这使得状态管理非常简单直接，不过理解其工作原理同样重要，这样你可以避开一些常见的问题. 在这个章节，我们将研究一下 Vue 响应式系统的底层的细节。

### 7.1 如何追踪变化

- 当你把一个普通的 JavaScript 对象传入 Vue 实例作为 `data` 选项，Vue 将遍历此对象所有的 property，并使用 `Object.defineProperty` 把这些 property 全部转为 **getter/setter**. `Object.defineProperty` 是 ES5 中一个无法 shim 的特性，这也就是 Vue 不支持 IE8 以及更低版本浏览器的原因。

  这些 `getter/setter` 对用户来说是不可见的，但是在内部它们让 Vue 能够追踪依赖， 在 property 被访问和修改时通知变更。 这里需要注意的是不同浏览器在控制台打印数据对象时对 getter/setter 的格式化并不同， 所以建议安装 **vue-devtools** 来获取对检查数据更加友好的用户界面.

  每个组件实例都对应一个 **watcher** 实例，它会在组件渲染的过程中把 "接触" 过的数据 property 记录为依赖。之后当依赖项的 setter 触发时，会通知 watcher, 从而使它关联的组件重新渲染。




## 7.2 检测变化的注意事项

- 由于 JavaScript 的限制， **Vue 不能检测**数组和对象的变化。 尽管如此我们还是有一些办法来回避这些限制并保证它们的响应性。

### 7.2.1 对于对象

- Vue 无法检测 property 的添加或移除。由于 Vue 会在初始化实例时对 property 执行 getter/setter 转化，所以 property 必须在 `data` 对象上存在才能让 Vue 将它转换为响应式的。例如:

```
var vm = new Vue({
    data:{
        a:1
    }
})
// `vm.a` 是响应式的

vm.b = 2
// `vm.b` 是非响应式的
```

对于已经创建的实例, Vue 不允许动态添加根级别的响应式 property. 但是, 可以使用 `Vue.set(object, propertyName, value)` 方法向嵌套对象添加响应式 property. 例如, 对于:

```
Vue.set(vm.someObject, 'b', 2);

// - 例如:
Vue.set(vm.data, 'b', 2);
```

您还可以使用 `vm.$set` 实例方法, 这也是全局 `Vue.set` 方法的别名:

```
this.$set(this.someObject, 'b', 2);
```

有时你可能需要为已有对象赋值多个新 property, 比如使用 `Object.assign()` 或 `_.extend()`. 但是, 这样添加到对象上的新 property 不会触发更新. 在这种情况下, 你应该用原对象与要混合进去的对象的 property 一起创建一个新的对象:

```
// - 代替 `Object.assign(this.someObject, {a: 1, b: 2})`
this.someObject = Object.assign({}, this.someObject, {a: 1, b: 2});
```

### 7.2.2 对于数组

- Vue 不能检测以下数组的变动:
  - (1) 当你利用索引直接设置一个数组项时，例如: `vm.items[indexOfItem] = newValue`
  - (2) 当你修改数组的长度时，例如： `vm.items.length = newLength`

  举个例子:

```
var vm = new Vue({
    data: {
        items: ['a', 'b', 'c']
    }
})
vm.items[1] = 'x' // 不是响应性的
vm.items.length = 2 // 不是响应性的
```

  为了解决第一类问题，以下 2 种方式都可以实现和 `vm.items[indexOfItem] = newValue` 相同的效果， 同时也将在响应式系统内触发状态更新:

```
// Vue.set
Vue.set(vm.items, indexOfItem, newValue)
```

```
// Array.prototype.splice
vm.items.splice(indexOfItem, 1, newValue)
```

  你也可以使用 `vm.$set` 实例方法, 该方法是全局方法 `Vue.set` 的一个别名:

```
vm.$set(vm.items, indexOfItem, newValue)
```

  为了解决第二类问题, 你可以使用 `splice`:

```
vm.items.splice(newLength);
```

## 7.3 声明响应式属性

- 由于 Vue 不允许动态添加根级响应式 property， 所以你必须在初始化实例前声明所有根级响应式 property，哪怕只是一个空值:

```
var vm = new Vue({
    data: {
        // 声明 message 为一个空值字符串
        message: ''
    },
    template: '<div>{{ message }}</div>'
})
// 之后设置 `message`
vm.message = 'Hello!'
```

如果你未在 `data` 选项中声明 `message`，Vue 将警告你渲染函数正在试图访问不存在的 property。

这样的限制在背后是有其技术原因的，它消除了在依赖项跟踪系统中的一类边界情况，也使 Vue 实例能更好地配合类型检查系统工作. 但与此同时在代码可维护性方面也有一点重要的考虑：`data` 对象就像组件状态的结构 (schema)。提前声明所有的响应式 property， 可以让组件代码在未来修改或给其他开发人员阅读时更易于理解。

## 7.4 异步更新队列

### 前置知识:

1. 异步更新的简单叙述: `../../../JS-book-learning/JavaScript知识集合/JavaScript掘金小册/前端性能优化原理与实践/渲染篇 4：千方百计——Event Loop 与异步更新策略.md`
    - **注意:**

      (1) 这个文章中: 关于 "一个完整的 Event Loop 过程" 可以概括为以下阶段: 内部的部分叙述是不正确的, 当 JS 引擎解析到 `script` 时会产生一个 全局上下文, 这里说遇到 `script` 标签会被推入到宏任务队列, 我认为这个叙述是有问题, 详解: `../../../JS-book-learning/《深入理解JavaScript系列》--汤姆大叔/11-执行上下文/11-执行上下文(环境).md`

      (2) 此文章中对宏任务和微任务的执行机制叙述也有问题, 宏任务是一个一个执行的, 微任务是一队一队执行的, 那么一队到底有多少个? 这里的讲解也不详细. 关于 此部分知识的讲解请见: `../../../JS-book-learning/JavaScript知识集合/详解_执行栈-任务队列-事件循环.md` 内的 `#### 7. 宏任务(macro-task)和微任务(micro-task)`
2. Vue 的异步任务默认情况下都是用 `Promise` 来包装的，也就是是说它们都 `micro-task`.

- 可能你还没有注意到, Vue 在更新 DOM 是是异步执行的. 只要侦听到数据变化, Vue 将开启一个队列(Callback Task), 并缓冲在同一事件循环中发生的所有数据变更. 如果同一个 watcher 被多次触发, 只会被推入到队列中一次. 这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的. 然后, 在下一个事件循环 `tick` (标记/滴答) 中, Vue 刷新队列并执行实际(已去重的) 工作. Vue 在内部对异步队列尝试使用原生的 `Promise.then`, `MutationObserver` 和 `setImmediate`, 如果执行环境不支持, 则会采用 `setTimeout(fn, 0)` 代替.

  例如, 当你设置 `vm.data.b = 'New B'`, 该组件不会立即重新渲染. 当刷新队列时, 组件会在下一个事件循环 `tick` 中更新. 多数情况下你不需要关心这个过程, 但是如果你想基于更新后的 DOM 状态来做点什么, 这就可能会有些棘手. 虽然 Vue.js 通常鼓励开发人员使用 "数据驱动" 的方式思考, 避免直接接触 DOM, 但是有时我们必须要这么做. 为了在数据变化之后等待 Vue 完成

更新 DOM, 可以在数据变化之后理解使用 `Vue.nextTick(callback)`. 这样回调函数将在 DOM 更新完成之后被调用. 例如:

```
<div id="example">{{ message }}</div>

<script>
    var vm = new Vue({
        el: '$example',
        data: {
            message: '123'
        }
    });
    vm.message = 'new message';     // - 更该数据
    vm.$el.textContent === 'new message'    // false
    Vue.nextTick(function() {
        vm.$el.textContent === 'new message'    // true
    })
</script>
```

在组件内使用 `vm.$nextTick()` 实例方法特别方便, 因为它不需要全局 `Vue`, 并且回调函数的 `this` 将自动绑定到当前的 Vue 实例上. 例如:

```
<template>
    <div>
        <p>{{ message }}</p>
    </div>
</template>
<script>
    export default {
        data(){
            return {
                message: '未更新'
            }
        },
        methods: {
            updateMessage; function() {
                this.message = '已更新';
                console.log(this.$el.textContent)   // '未更新'
                this.$nextTick(function() {
                    console.log(this.$el.textContent);  // '已更新'
                })
            }
        }
    }
</script>
```

因为 `$nextTick` 返回一个 `Promise` 对象, 所以你可以使用新的 **ES2017 async/await** 语法完成同样的事情:

```
methods: {
    updateMessage: async function() {
        this.message = '已更新';
        console.log(this.$el.textContent);  // '未更新'
        await this.$nextTick();
        console.log(this.$el.textContent);  // '已更新'
    }
}
```