深入响应式原理

大部分的基础内容我们已经讲到了，现在讲点底层内容。Vue.js 最显著的一个功能是响应系统 —— 模型只是普通对象，修改它则更新视图。这让状态管理非常简单且直观，不过理解它的原理也很重要，可以避免一些常见问题。下面我们开始深挖 Vue.js 响应系统的底层细节。

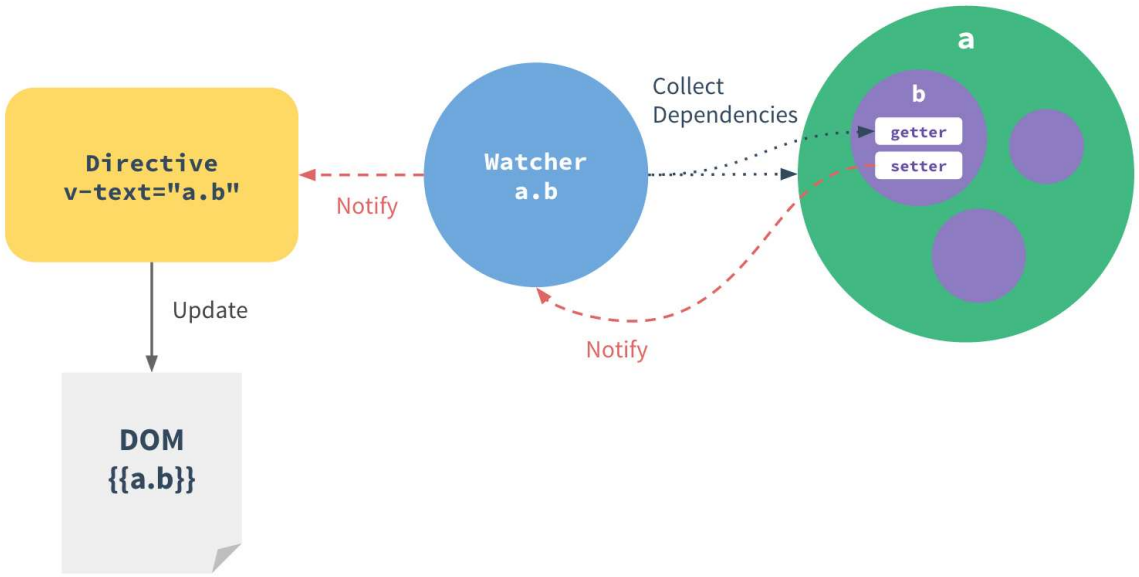
如何追踪变化

把一个普通对象传给 Vue 实例作为它的 `data` 选项，Vue.js 将遍历它的属性，用 **Object.defineProperty** 将它们转为 getter/setter。这是 ES5 特性，不能打补丁实现，这便是为什么 Vue.js 不支持 IE8 及更低版本。

用户看不到 getter/setters，但是在内部它们让 Vue.js 追踪依赖，在属性被访问和修改时通知变化。一个问题是在浏览器控制台打印数据对象时 getter/setter 的格式化不同，使用 `vm.$log()` 实例方法可以得到更友好的输出。

模板中每个指令/数据绑定都有一个对应的 **watcher** 对象，在计算过程中它把属性记录为依赖。之后当依赖的 setter 被调用时，会触发 watcher 重新计算，也就会导致它的关联指令更新 DOM。

自定义过滤器
混合
插件
构建大型应用
对比其它框架
加入 Vue 社区！



变化检测问题

受 ES5 的限制，Vue.js **不能检测到对象属性的添加或删除。**因为 Vue.js 在初始化实例时将属性转为 getter/setter，所以属性必须在 `data` 对象上才能让 Vue.js 转换它，才能让它是响应的。例如：

JS

```
var data = { a: 1 }
var vm = new Vue({
  data: data
})
// `vm.a` 和 `data.a` 现在是响应的
vm.b = 2
// `vm.b` 不是响应的
data.b = 2
// `data.b` 不是响应的
```

不过，有办法在实例创建之后**添加属性并且让它是响应的。**

对于 Vue 实例，可以使用 `$set(key, value)` 实例方法：

JS

```
vm.$set('b', 2)
// `vm.b` 和 `data.b` 现在是响应的
```

对于普通数据对象，可以使用全局方法 `Vue.set(object, key, value)` ：

JS

```
Vue.set(data, 'c', 3)
// `vm.c` 和 `data.c` 现在是响应的
```

有时你想向已有对象上添加一些属性，例如使用 `Object.assign()` 或 `_.extend()` 添加属性。但是，添加到对象上的新属性不会触发更新。这时可以创建一个新的对象，包含原对象的属性和新的属性：

JS

```
// 不使用 `Object.assign(this.someObject, { a: 1, b: 2 })`
this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 })
```

也有一些数组相关的问题，之前已经在**列表渲染**中讲过。

初始化数据

尽管 Vue.js 提供了 API 动态地添加响应属性，还是推荐在 `data` 对象上声明所有的响应属性。

不这么做：

JS

```
var vm = new Vue({
  template: '<div>{{msg}}</div>'
})
// 然后添加 `msg`
vm.$set('msg', 'Hello!')
```

这么做：

JS

```
var vm = new Vue({
  data: {
    // 以一个空值声明 `msg`
    msg: ''
  },
  template: '<div>{{msg}}</div>'
})
// 然后设置 `msg`
vm.msg = 'Hello!'
```

这么做有两个原因：

1. `data` 对象就像组件状态的模式（schema）。在它上面声明所有的属性让组件代码更易于理解。

2. 添加一个顶级响应属性会强制所有的 watcher 重新计算，因为它之前不存在，没有 watcher 追踪它。这么做性能通常是可以接受的（特别是对比 Angular 的脏检查），但是可以在初始化时避免。

异步更新队列

Vue.js 默认**异步**更新 DOM。每当观察到数据变化时，Vue 就开始一个队列，将同一事件循环内所有的数据变化缓存起来。如果一个 watcher 被多次触发，只会推入一次到队列中。等到下一次事件循环，Vue 将清空队列，只进行必要的 DOM 更新。在内部异步队列优先使用 `MutationObserver` ，如果不支持则使用 `setTimeout(fn, 0)` 。

例如，设置了 `vm.someData = 'new value'` ，DOM 不会立即更新，而是在下一次事件循环清空队列时更新。我们基本不用关心这个过程，但是如果想在 DOM 状态更新后做点什么，这会有帮助。尽管 Vue.js 鼓励开发者沿着数据驱动的思路，避免直接修改 DOM，但是有时确实要这么做。为了在数据变化之后等待 Vue.js 完成更新 DOM，可以在数据变化之后立即使用 `Vue.nextTick(callback)` 。回调在 DOM 更新完成后调用。例如：

HTML

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

JS

```
var vm = new Vue({
  el: '#example',
  data: {
    msg: '123'
  }
})
vm.msg = 'new message' // 修改数据
vm.$el.textContent === 'new message' // false
Vue.nextTick(function () {
  vm.$el.textContent === 'new message' // true
})
```

`vm.$nextTick()` 这个实例方法比较方便，因为它不需要全局 `Vue` ，它的回调的 `this` 自动绑定到当前 Vue 实例：

JS

```
Vue.component('example', {
  template: '<span>{{msg}}</span>',
  data: function () {
```

```
    return {
      msg: 'not updated'
    }
  },
  methods: {
    updateMessage: function () {
      this.msg = 'updated'
      console.log(this.$el.textContent) // => 'not updated'
      this.$nextTick(function () {
        console.log(this.$el.textContent) // => 'updated'
      })
    }
  }
})
```

计算属性的奥秘

你应该注意到 Vue.js 的计算属性**不是**简单的 getter。计算属性持续追踪它的响应依赖。在计算一个计算属性时，Vue.js 更新它的依赖列表并缓存结果，只有当其中一个依赖发生了变化，缓存的结果才无效。因此，只要依赖不发生变化，访问计算属性会直接返回缓存的结果，而不是调用 getter。

为什么要缓存呢？假设我们有一个高耗计算属性 `A` ，它要遍历一个巨型数组并做大量的计算。然后，可能有其它的计算属性依赖 `A` 。如果没有缓存，我们将调用 `A` 的 getter 许多次，超过必要次数。

由于计算属性被缓存了，在访问它时 getter 不总是被调用。考虑下例：

JS

```
var vm = new Vue({
  data: {
    msg: 'hi'
  },
  computed: {
```

```
    example: function () {
      return Date.now() + this.msg
    }
  }
})
```

计算属性 `example` 只有一个依赖： `vm.msg` 。 `Date.now()` **不是** 响应依赖，因为它跟 Vue 的数据观察系统无关。因而，在访问 `vm.example` 时将发现时间戳不变，除非 `vm.msg` 变了。

有时希望 getter 不改变原有的行为，每次访问 `vm.example` 时都调用 getter。这时可以为指定的计算属性关闭缓存：

JS

```
computed: {
  example: {
    cache: false,
    get: function () {
      return Date.now() + this.msg
    }
  }
}
```

现在每次访问 `vm.example` 时，时间戳都是新的。**但是，只是在 JavaScript 中访问是这样的；数据绑定仍是依赖驱动的。**如果在模块中这样绑定计算属性 `{{example}}` ，只有响应依赖发生变化时才更新 DOM。

← **组件**

自定义指令 →