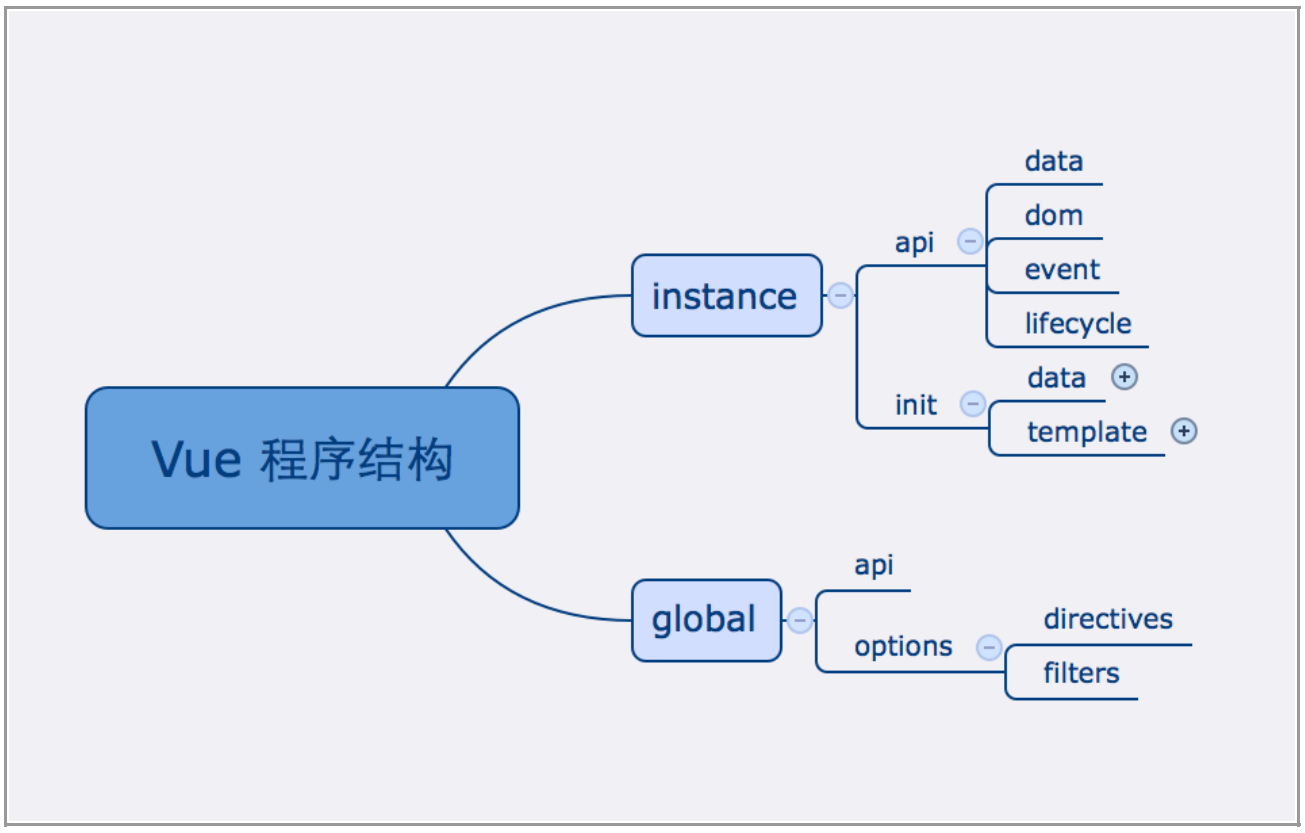
Vue.js 源码学习笔记

最近饶有兴致的又把最新版 Vue.js 的源码学习了一下，觉得真心不错，个人觉得 Vue.js 的代码非常之优雅而且精辟，作者本身可能无 (bu) 意 (xie) 提及这些。那么，就让我来吧：）

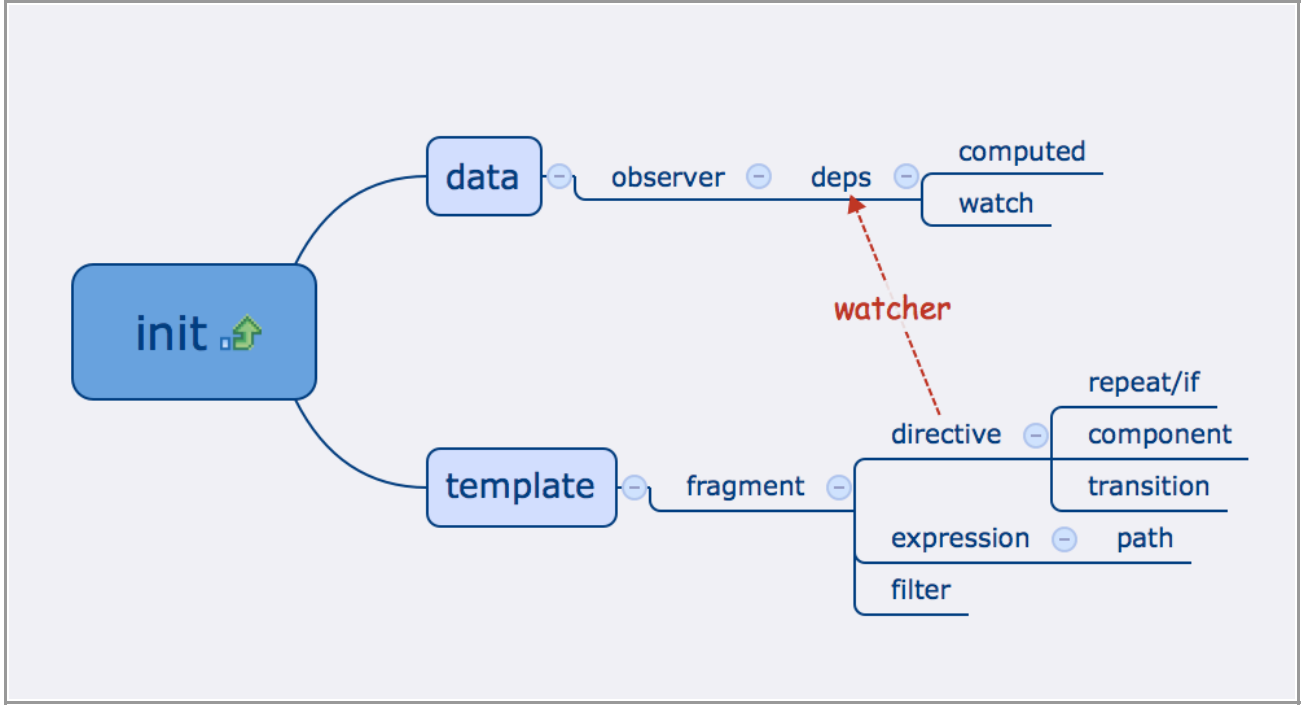
程序结构梳理



Vue.js 是一个非常典型的 MVVM 的程序结构，整个程序从最上层大概分为

1. 全局设计：包括全局接口、默认选项等
2. vm 实例设计：包括接口设计 (vm 原型)、实例初始化过程设计 (vm 构造函数)

这里面大部分内容可以直接跟 Vue.js 的官方 API 参考文档对应起来，但文档里面没有且值得一提的是构造函数的设计，下面是我摘出的构造函数最核心的工作内容。



整个实例初始化的过程中，重中之重就是把数据 (Model) 和视图 (View) 建立起关联关系。Vue.js 和诸多 MVVM 的思路是类似的，主要做了三件事：

1. 通过 observer 对 data 进行了监听，并且提供订阅某个数据项的变化的能力
2. 把 template 解析成一段 document fragment，然后解析其中的 directive，得到每一个 directive 所依赖的数据项及其更新方法。比如 `v-text="message"` 被解析之后 (这里仅作示意，实际程序逻辑会更严谨而复杂)：

    1. 所依赖的数据项 `this.$data.message`，以及
    2. 相应的视图更新方法 `node.textContent = this.$data.message`

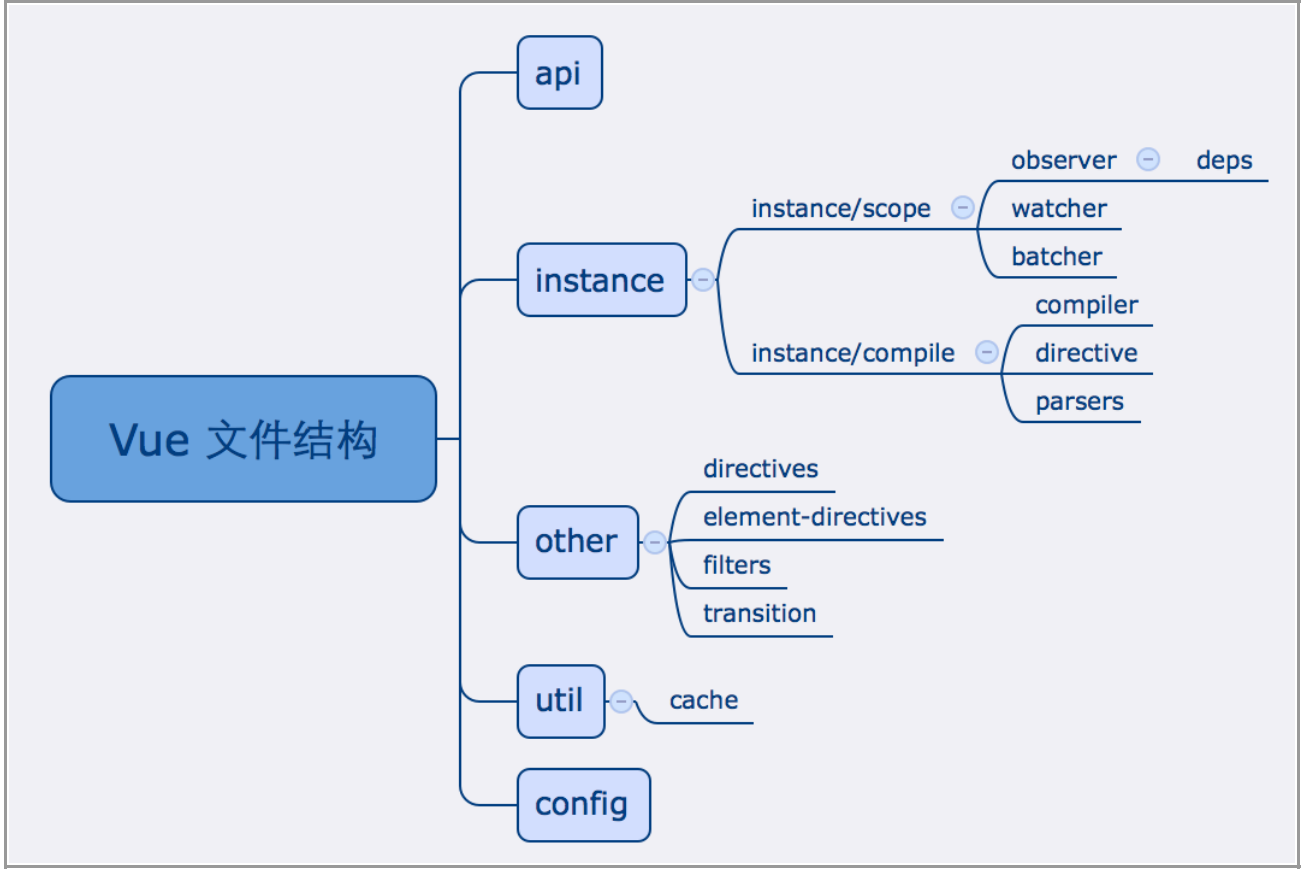
3. 通过 watcher 把上述两部分结合起来，即把 directive 中的数据依赖订阅在对应数据的 observer 上，这样当数据变化的时候，就会触发 observer，进而触发相关依赖对应的视图更新方法，最后达到模板原本的关联效果。

所以整个 vm 的核心，就是如何实现 observer, directive (parser), watcher 这三样东西

文件结构梳理

Vue.js 源代码都存放在项目的 `src` 目录中，我们主要关注一下这个目录 (事实上 `test/unit/specs` 目录也值得一看，它是对应着每个源文件的测试用例)。

`src` 目录下有多个并列的文件夹，每个文件夹都是一部分独立而完整的程序设计。不过在我看来，这些目录之前也是有更立体的关系的：



- 首先是 `api/*` 目录，这几乎是最“上层”的接口封装，实际的实现都埋在了其它文件夹里
- 然后是 `instance/init.js`，如果大家希望自顶向下了解所有 Vue.js 的工作原理的话，建议从这个文件开始看起
    - `instance/scope.js`：数据初始化，相关的子程序 (目录) 有 `observer/*`、`watcher.js`、`batcher.js`，而 `observer/dep.js` 又是数据观察和视图依赖相关联的关键
    - `instance/compile.js`：视图初始化，相关的子程序 (目录) 有 `compiler/*`、`directive.js`、`parsers/*`
- 其它核心要素：`directives/*`、`element-directives/*`、`filters/*`、`transition/*`
- 当然还有 `util/*` 目录，工具方法集合，其实还有一个类似的 `cache.js`
- 最后是 `config.js` 默认配置项

篇幅有限，如果大家有意“通读” Vue.js 的话，个人建议顺着上面的整体介绍来阅读赏析。

接下来是一些自己觉得值得一提的代码细节

一些不容错过的代码/程序细节

`this._eventsCount` 是什么?

一开始看 `instance/init.js` 的时候，我立刻注意到一个细节，就是 `this._eventsCount = {}` 这句，后面还有注释

```
31  // events bookkeeping                                  instance/init.js
32  this._events = {}           // registered callbacks
33  this._eventsCount = {}      // for $broadcast optimization
34  this._eventCancelled = false // for event cancellation
35
```

> *for $broadcast optimization*

非常好奇，然后带着疑问继续看了下去，直到看到 `api/events.js` 中 `$broadcast` 方法的实现，才知道这是为了避免不必要的深度遍历：在有广播事件到来时，如果当前 vm 的 `_eventsCount` 为 `0`，则不必向其子 vm 继续传播该事件。而且这个文件稍后也有 `_eventsCount` 计数的实现方式。

api/events.js

```
153 /**
154  * Modify the listener counts on all parents.
155  * This bookkeeping allows $broadcast to return early when
156  * no child has listened to a certain event.
157  *
158  * @param {Vue} vm
159  * @param {String} event
160  * @param {Number} count
161  */
162
163 var hookRE = /^hook:/
164 function modifyListenerCount (vm, event, count) {
165   var parent = vm.$parent
166   // hooks do not get broadcasted so no need
167   // to do bookkeeping for them
168   if (!parent || !count || hookRE.test(event)) return
169   while (parent) {
170     parent._eventsCount[event] =
171       (parent._eventsCount[event] || 0) + count
172     parent = parent.$parent
173   }
174 }
```

api/events.js

```
113 /**
114  * Recursively broadcast an event to all children instances.
115  *
116  * @param {String} event
117  * @param {...*} additional arguments
118  */
119
120 exports.$broadcast = function (event) {
121   // if no child has registered for this event,
122   // then there's no need to broadcast.
123   if (!this._eventsCount[event]) return
124   var children = this.$children
125   for (var i = 0, l = children.length; i < l; i++) {
126     var child = children[i]
127     child.$emit.apply(child, arguments)
128     if (!child._eventCancelled) {
129       child.$broadcast.apply(child, arguments)
130     }
131   }
132   return this
133 }
```

这是一种很巧妙同时也可以在很多地方运用的性能优化方法。

数据更新的 diff 机制

前阵子有很多关于视图更新效率的讨论，我猜主要是因为 virtual dom 这个概念的提出而导致的吧。这次我详细看了一下 Vue.js 的相关实现原理。

实际上，视图更新效率的焦点问题主要在于大列表的更新和深层数据更新这两方面，而被热烈讨论的主要是前者 (后者是因为需求小还是没争议我就不得而知了)。所以这

里着重介绍一下 directives/repeat.js 里对于列表更新的相关代码。

```
/**
 * Diff, based on new data and old data, determine the
 * minimum amount of DOM manipulations needed to make the
 * DOM reflect the new data Array.
 *
 * The algorithm diffs the new data Array by storing a
 * hidden reference to an owner vm instance on previously
 * seen data. This allows us to achieve O(n) which is
 * better than a levenshtein distance based algorithm,
 * which is O(m * n).
 *
 * @param {Array} data
 * @param {Array} oldVms
 * @return {Array}
 */

diff: function (data, oldVms) {
```

directives/repeat.js

首先 diff(data, oldVms) 这个函数的注释对整个比对更新机制做了个简要的阐述，大概意思是先比较新旧两个列表的 vm 的数据的状态，然后差量更新 DOM。

```
// First pass, go through the new Array and fill up
// the new vms array. If a piece of data has a cached
// instance for it, we reuse it. Otherwise build a new
// instance.
for (i = 0, l = data.length; i < l; i++) {
  obj = data[i]
  raw = converted ? obj.$value : obj
  primitive = !isObject(raw)
  vm = !init && this.getVm(raw, i, converted ? obj.$key : null)
  if (vm) { // reusable instance
    vm._reused = true
    vm.$index = i // update $index
    // update data for track-by or object repeat,
    // since in these two cases the data is replaced
    // rather than mutated.
    if (idKey || converted || primitive) {
      if (alias) {
        vm[alias] = raw
      } else if (_.isPlainObject(raw)) {
        vm.$data = raw
      } else {
        vm.$value = raw
      }
    }
  } else { // new instance
    vm = this.build(obj, i, true)
    vm._reused = false
  }
  vms[i] = vm
  // insert if this is first run
  if (init) {
    vm.$before(end)
  }
}
```

directives/repeat.js

第一步：便利新列表里的每一项，如果该项的 vm 之前就存在，则打一个 _reused 的标 (这个字段我一开始看 init.js 的时候也是困惑的..... 看到这里才明白意思)，如果不存在对应的 vm，则创建一个新的。

```
282     // Second pass, go through the old vm instances and
283     // destroy those who are not reused (and remove them
284     // from cache)                          directives/repeat.js
285     var removalIndex = 0
286     var totalRemoved = oldVms.length - vms.length
287     for (i = 0, l = oldVms.length; i < l; i++) {
288       vm = oldVms[i]
289       if (!vm._reused) {
290         this.uncacheVm(vm)
291         vm.$destroy(false, true) // defer cleanup until removal
292         this.remove(vm, removalIndex++, totalRemoved, inDoc)
293       }
294     }
```

第二步：便利旧列表里的每一项，如果 `_reused` 的标没有被打上，则说明新列表里已经没有它了，就地销毁该 vm。

```
295     // final pass, move/insert new instances into the
296     // right place.
297     var targetPrev, prevEl, currentPrev
298     var insertionIndex = 0                  directives/repeat.js
299     for (i = 0, l = vms.length; i < l; i++) {
300       vm = vms[i]
301       // this is the vm that we should be after
302       targetPrev = vms[i - 1]
303       prevEl = targetPrev
304         ? targetPrev._staggerCb
305           ? targetPrev._staggerAnchor
306           : targetPrev._blockEnd || targetPrev.$el
307         : start
308       if (vm._reused && !vm._staggerCb) {
309         currentPrev = findPrevVm(vm, start, this.id)
310         if (currentPrev !== targetPrev) {
311           this.move(vm, prevEl)
312         }
313       } else {
314         // new instance, or still in stagger.
315         // insert with updated stagger index.
316         this.insert(vm, insertionIndex++, prevEl, inDoc)
317       }
318       vm._reused = false
319     }
```

第三步：整理新的 vm 在视图里的顺序，同时还原之前打上的 `_reused` 标。就此列表更新完成。

顺带提一句 Vue.js 的元素过渡动画处理 (`v-transition`) 也设计得非常巧妙，感兴趣的自己看吧，就不展开介绍了

组件的 [keep-alive] 特性

directives/component.js

```
bind: function () {
  if (!this.el.__vue__) {
    // create a ref anchor
    this.anchor = _.createAnchor('v-component')
    _.replace(this.el, this.anchor)
    // check keep-alive options.
    // If yes, instead of destroying the active vm when
    // hiding (v-if) or switching (dynamic literal) it,
    // we simply remove it from the DOM and save it in a
    // cache object, with its constructor id as the key.
    this.keepAlive = this._checkParam('keep-alive') != null
    // wait for event before insertion
    this.readyEvent = this._checkParam('wait-for')
    // check ref
    this.refID = _.attr(this.el, 'ref')
    if (this.keepAlive) {
      this.cache = {}
    }
```

directives/component.js

```
/**
 * Instantiate/insert a new child vm.
 * If keep alive and has cached instance, insert that
 * instance; otherwise build a new one and cache it.
 *
 * @param {Object} [data]
 * @return {Vue} - the created instance
 */

build: function (data) {
  if (this.keepAlive) {
    var cached = this.cache[this.ctorId]
    if (cached) {
      return cached
    }
  }
```

Vue.js 为其组件设计了一个 [keep-alive] 的特性，如果这个特性存在，那么在组件被重复创建的时候，会通过缓存机制快速创建组件，以提升视图更新的性能。代码在 directives/component.js。

数据监听机制

如何监听某一个对象属性的变化呢？我们很容易想到 Object.defineProperty 这个 API，为此属性设计一个特殊的 getter/setter，然后在 setter 里触发一个函数，就可以达到监听的效果。

```
/**
 * Convert a property into getter/setter so we can emit
 * the events when the property is accessed/changed.
 *
 * @param {String} key
 * @param {*} val
 */

p.convert = function (key, val) {
  var ob = this
  var childOb = ob.observe(val)
  var dep = new Dep()
  if (childOb) {
    childOb.deps.push(dep)
  }
  Object.defineProperty(ob.value, key, {
    enumerable: true,
    configurable: true,
    get: function () {
      if (ob.active) {
        dep.depend()
      }
      return val
    },
    set: function (newVal) {
      if (newVal === val) return
      // remove dep from old value
      var oldChildOb = val && val.__ob__
      if (oldChildOb) {
        oldChildOb.deps.$remove(dep)
      }
      val = newVal
      // add dep to new value
      var newChildOb = ob.observe(newVal)
      if (newChildOb) {
        newChildOb.deps.push(dep)
      }
      dep.notify()
    }
  })
}
```

不过数组可能会有点麻烦，Vue.js 采取的是对几乎每一个可能改变数据的方法进行 prototype 更改：

observer/array.js

```
;[
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
.forEach(function (method) {
  // cache original method
  var original = arrayProto[method]
  _.define(arrayMethods, method, function mutator () {
    // avoid leaking arguments:
    // http://jsperf.com/closure-with-arguments
    var i = arguments.length
    var args = new Array(i)
    while (i--) {
      args[i] = arguments[i]
    }
    var result = original.apply(this, args)
    var ob = this.__ob__
    var inserted
    switch (method) {
      case 'push':
        inserted = args
        break
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    if (inserted) ob.observeArray(inserted)
    // notify change
    ob.notify()
    return result
  })
})
```

但这个策略主要面临两个问题：

1. 无法监听数据的 `length`，导致 `arr.length` 这样的数据改变无法被监听
2. 通过角标更改数据，即类似 `arr[2] = 1` 这样的赋值操作，也无法被监听

为此 Vue.js 在文档中明确提示不建议直接角标修改数据

```
50 ▼ /**
51    * Swap the element at the given index with a new value
52    * and emits corresponding event.
53    *                                          observer/array.js
54    * @param {Number} index
55    * @param {*} val
56    * @return {*} - replaced element
57    */
58
59 ▼ _.define(
60     arrayProto,
61     '$set',
62 ▶   function $set (index, val) {…
67     }
68 )
69
70 ▼ /**
71    * Convenience method to remove the element at given index.
72    *
73    * @param {Number} index
74    * @param {*} val
75    */
76
77 ▼ _.define(
78     arrayProto,
79     '$remove',
80 ▶   function $remove (index) {…
89     }
90 )
```

同时 Vue.js 提供了两个额外的“糖方法” `$set` 和 `$remove` 来弥补这方面限制带来的不便。整体上看这是个取舍有度的设计。我个人之前在设计数据绑定库的时候也采取了类似的设计 (一个半途而废的内部项目就不具体献丑了)，所以比较认同也有共鸣。

path 解析器的状态机设计

首先要说 `parsers` 文件夹里有各种“财宝”等着大家挖掘！认真看一看一定不会后悔的

`parsers/path.js` 主要的职责是可以把一个 JSON 数据里的某一个“路径”下的数据取出来，比如：

```
var path = 'a.b[1].v'
var obj = {
  a: {
    b: [
      {v: 1},
      {v: 2},
      {v: 3}
    ]
  }
}
parse(obj, path) // 2
```

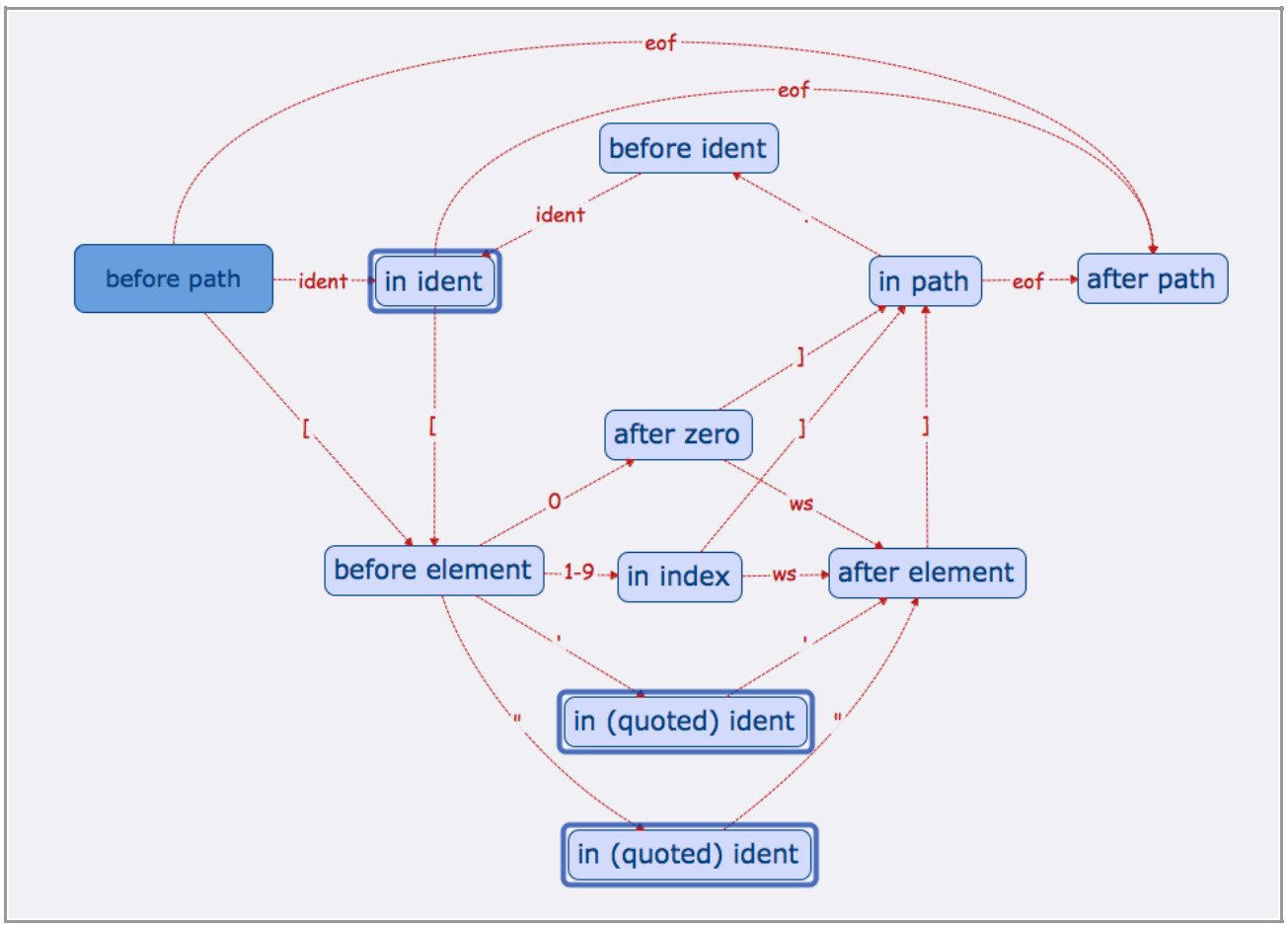
所以对 `path` 字符串的解析成为了它的关键。Vue.js 是通过状态机管理来实现对路径的解析的：

```
10 ▼ var pathStateMachine = {                    parsers/path.js
11 ▼   'beforePath': {
```

```
    'ws': ['beforePath'],
    'ident': ['inIdent', 'append'],
    '[': ['beforeElement'],
    'eof': ['afterPath']
  },
  'inPath': {
    'ws': ['inPath'],
    '.': ['beforeIdent'],
    '[': ['beforeElement'],
    'eof': ['afterPath']
  },
  'beforeIdent': {
    'ws': ['beforeIdent'],
    'ident': ['inIdent', 'append']
  },
  'inIdent': {
    'ident': ['inIdent', 'append'],
    '0': ['inIdent', 'append'],
    'number': ['inIdent', 'append'],
    'ws': ['inPath', 'push'],
    '.': ['beforeIdent', 'push'],
    '[': ['beforeElement', 'push'],
    'eof': ['afterPath', 'push'],
    ']': ['inPath', 'push']
  },
  'beforeElement': {
    'ws': ['beforeElement'],
    '0': ['afterZero', 'append'],
    'number': ['inIndex', 'append'],
    "'": ['inSingleQuote', 'append', ''],
    '"': ['inDoubleQuote', 'append', ''],
    'ident': ['inIdent', 'append', '*']
  },
  'afterZero': {
    'ws': ['afterElement', 'push'],
    ']': ['inPath', 'push']
  },
  'inIndex': {
    '0': ['inIndex', 'append'],
    'number': ['inIndex', 'append'],
    'ws': ['afterElement'],
    ']': ['inPath', 'push']
  },
  'inSingleQuote': {
    "'": ['afterElement'],
    'eof': 'error',
```

```
58        'else': ['inSingleQuote', 'append']
59      },
60      'inDoubleQuote': {
61        '"': ['afterElement'],
62        'eof': 'error',
63        'else': ['inDoubleQuote', 'append']
64      },
65      'afterElement': {
66        'ws': ['afterElement'],
67        ']': ['inPath', 'push']
68      }
69    }
```

咋一看很头大，不过如果再稍微梳理一下：



也许看得更清楚一点了，当然也能发现其中有一点小问题，就是源代码中 `inIdent` 这个状态是具有二义性的，它对应到了图中的三个地方，即 `in ident` 和两个 `in (quoted) ident`。

实际上，我在看代码的过程中顺手提交了这个 bug，作者眼明手快，当天就进行了修复，现在最新的代码里已经不是这个样子了：

new parsers/path.js

```
// actions
var APPEND = 0
var PUSH = 1

// states
var BEFORE_PATH = 0
var IN_PATH = 1
var BEFORE_IDENT = 2
var IN_IDENT = 3
var BEFORE_ELEMENT = 4
var AFTER_ZERO = 5
var IN_INDEX = 6
var IN_SINGLE_QUOTE = 7
var IN_DOUBLE_QUOTE = 8
var IN_SUB_PATH = 9
var AFTER_ELEMENT = 10
var AFTER_PATH = 11
var ERROR = 12

var pathStateMachine = []

pathStateMachine[BEFORE_PATH] = {
  'ws': [BEFORE_PATH],
  'ident': [IN_IDENT, APPEND],
  '[': [BEFORE_ELEMENT],
  'eof': [AFTER_PATH]
}

pathStateMachine[IN_PATH] = {
  'ws': [IN_PATH],
  '.': [BEFORE_IDENT],
  '[': [BEFORE_ELEMENT],
  'eof': [AFTER_PATH]
}

pathStateMachine[BEFORE_IDENT] = {
  'ws': [BEFORE_IDENT],
  'ident': [IN_IDENT, APPEND]
}

pathStateMachine[IN_IDENT] = {
  'ident': [IN_IDENT, APPEND],
  '0': [IN_IDENT, APPEND],
  'number': [IN_IDENT, APPEND],
  'ws': [IN_PATH, PUSH],
  '.': [BEFORE_IDENT, PUSH],
  '[': [BEFORE_ELEMENT, PUSH],
  'eof': [AFTER_PATH, PUSH]
}

pathStateMachine[BEFORE_ELEMENT] = {
  'ws': [BEFORE_ELEMENT],
  '0': [AFTER_ZERO, APPEND],
  'number': [IN_INDEX, APPEND],
  "'": [IN_SINGLE_QUOTE, APPEND, ''],
  '"': [IN_DOUBLE_QUOTE, APPEND, ''],
  'ident': [IN_SUB_PATH, APPEND, '*']
}

pathStateMachine[AFTER_ZERO] = {
  'ws': [AFTER_ELEMENT, PUSH],
  ']': [IN_PATH, PUSH]
}

pathStateMachine[IN_INDEX] = {
  '0': [IN_INDEX, APPEND],
  'number': [IN_INDEX, APPEND],
  'ws': [AFTER_ELEMENT],
  ']': [IN_PATH, PUSH]
}

pathStateMachine[IN_SINGLE_QUOTE] = {
  "'": [AFTER_ELEMENT],
  'eof': ERROR,
  'else': [IN_SINGLE_QUOTE, APPEND]
}

pathStateMachine[IN_DOUBLE_QUOTE] = {
  '"': [AFTER_ELEMENT],
  'eof': ERROR,
  'else': [IN_DOUBLE_QUOTE, APPEND]
}

pathStateMachine[IN_SUB_PATH] = {
  'ident': [IN_SUB_PATH, APPEND],
  '0': [IN_SUB_PATH, APPEND],
  'number': [IN_SUB_PATH, APPEND],
  'ws': [AFTER_ELEMENT],
  ']': [IN_PATH, PUSH]
}

pathStateMachine[AFTER_ELEMENT] = {
  'ws': [AFTER_ELEMENT],
  ']': [IN_PATH, PUSH]
}
```

而且状态机标识由字符串换成了数字常量，解析更准确的同时执行效率也会更高。

一点自己的思考

首先是视图的解析过程，Vue.js 的策略是把 element 或 template string 先统一转换成 document fragment，然后再分解和解析其中的子组件和 directives。我觉得这里有一定的性能优化空间，毕竟 DOM 操作相比之余纯 JavaScript 运算还是会慢一些。

然后是基于移动端的思考，Vue.js 虽确实已经非常非常小巧了 (min+gzip 之后约 22 kb)，但它是否可以更小，继续抽象出常用的核心功能，同时更快速，也是个值得思考的问题。

第三我非常喜欢通过 Vue.js 进行模块化开发的模式，Vue 是否也可以借助类似 web components + virtual dom 的形态把这样的开发模式带到更多的领域，也是件很有意义的事情。

总结

Vue.js 里的代码细节还不仅于此，比如：

- `cache.js` 里的缓存机制设计和场景运用 (如在 `parsers/path.js` 中)
- `parsers/template.js` 里的 `cloneNode` 方法重写和对 HTML 自动补全机制的

兼容
- 在开发和生产环境分别用注释结点和不可见文本结点作为视图的“占位符”等等

自己也在阅读代码，了解 Vue.js 的同时学到了很多东西，同时我觉得代码实现只是 Vue.js 优秀的要素之一，整体的程序设计、API 设计、细节的取舍、项目的工程考量都非常棒！

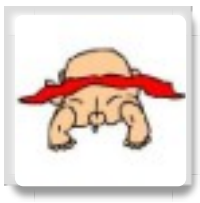
总之，分享一些自己的收获和代码的细节，希望可以帮助大家开阔思路，提供灵感。

已有 5 条评论 »

傅小黑 July 26th, 2015 at 11:48 pm

赞

回复



chaoren1641 July 28th, 2015 at 11:32 am

32个赞

回复

kid August 7th, 2015 at 11:05 am

码下以后仔细看

回复

comver September 27th, 2015 at 03:37 am

学习了！

回复

cxczy November 16th, 2015 at 05:27 pm

总结的很好，Vue如此小巧却又这么多优点

回复

添加新评论 »

称呼

电子邮件

网站

提交评论 (请至少包含一个汉字，且汉字不能比日本字少)

问卷

- 暂无

广告

我发起的开源项目
H5Slides
Zorro
欢迎了解

最新文章

- Vue.js 1.0.0 发布了!
- 如何成为一名卓越的前端工程师
- 手机淘宝前端的图片相关工作流程梳理
- 如何让办公室政治最小化
- Vue.js 源码学习笔记
- 从原型到发布——“团队时间线” 1.0 开发心得
- Vue + webpack 项目实践
- 用 Koa 写服务体验
- webcomponents 笔记 之 配置管理
- 14}, {15

最近回复

- tty228: 一直安装不成功- - 不知道是不是百度统计换了新版的原因，审视元素能看到h.gif但是统计一直提示检...
- tty228: 谢谢博主，拿去用了~~
- cxczy: 总结的很好，Vue如此小巧却又这么多优点
- Randy: 用多了之后你会发现 require css 是很方便的东西，因为用 style-loader 和其它...
- elevensky: 大爱
- 云库网: 字体有点大，css命名还是标准加习惯为好
- 影乐: 名字好有意思
- 常某某: = =我总觉得你的网站设计很奇怪。或许是相邻两块之间并无色差。而且宽度很小。总感觉两边空出很多来。特...
- 小四: 棒棒哒
- SuperZhang: 你的字确实很大....我也是用手动添加的. 我是来保持队形的

归档

- October 2015
- August 2015
- July 2015
- June 2015
- March 2015
- January 2015
- October 2014
- September 2014
- January 2014
- December 2013
- October 2013
- September 2013
- July 2013
- June 2013
- May 2013
- March 2013
- February 2013
- January 2013
- December 2012
- November 2012
- September 2012
- August 2012
- July 2012
- June 2012
- May 2012
- April 2012
- March 2012

其它

- 登录
- Valid XHTML
- Typecho

链接

- 傲游浏览器
- 我的Github

{"theme":"我被拍平了","designer":"@勾三股四"}