

# VUE

# Variáveis

# Reativas

Bernardo Meyer

## VUE – Conceito de variável reativa

O framework VUE implementa o conceito de **Variável Reativa**. O que é uma variável reativa ?

Durante a execução de um programa, no interior da memória do dispositivo eletrônico programável, uma variável de programa só tem seu valor modificado se estiver à esquerda de uma atribuição. Na primeira fase da Tecnologia de Informação, existia um programa em execução e uma console de inspeção das variáveis deste programa. Os resultados também eram apresentados na console.

Na segunda fase de programação, separou-se o programa do Banco de dados.

Depois, na terceira fase da Tecnologia de Informação, existia uma interface permanente de apresentação – não mais uma console de múltipla atribuição – da informação, da forma mais dinâmica possível. Para alterar o valor das variáveis, era preciso que um evento ocorresse. Mas, mesmo assim, o valor exibido na interface poderia estar desatualizado em relação ao valor da memória, pois era preciso um esforço de programação para sincronizar o calculado com o exibido.

E agora, na quarta fase da Tecnologia da Informação, frameworks que operam na interface foram desenvolvidas para sincronizar o calculado com o exibido. Tal é o caso do VUE, entre outros, como **Angular, React, Kendo e jQuery**. Poderíamos deixar o **jQuery** em uma posição intermediária entre a terceira e quarta fase, pois ele não implementa uma interface reativa tão eficiente quanto os demais.

Em outras palavras, na interface reativa, se um **INPUT** altera uma variável a ele acoplada, a alteração deste **INPUT** se reflete no programa a ele associado, e vice-versa, caso tenha sido especificado. Da mesma forma, se uma **tag HTML** apresenta um conteúdo dependente do valor desta variável, este conteúdo também será modificado.

Vamos analisar a implementação deste modelo durante o debug de um exemplo bem simples:

```
<body>
  <div id="app">{{ message }}</div>
  <script type="text/javascript">
    new Vue({
      el: "#app",
      data: { message: "Hello World!" }
    });
  </script>
</body>
```

## Inicialização

Na instanciação do modelo **Vue**, que consta da **DIV#app** (DIV cujo Id é “app”), entre outras coisas que acontecem, é coletada a variável Vue “message”. Isto é feito pela função `initData`, segundo o seguinte percurso:

### Instanciação: `new Vue`

Chama `this._init(options)`

`options` contém a seção `data`, com os valores iniciais das variáveis

Chama `Vue.prototype._init`

Função definida dentro de `initMixin(Vue)`

Chama `resolveConstructorOptions (Options)`

Chama `mergeOptions` (une ***instanceData*** com ***defaultData***):

Chama `initProxy(vm /* instancia de Vue */)`

Coloca `Proxy(vm, handlers)` em `vm._renderProxy`

```
> vm._renderProxy
< ▼ Proxy {_uid: 0, _isVue: true, $options: {...}, _renderProxy: Proxy} ⓘ
  ▼ [[Handler]]: Object
    ▶ has: f has(target, key)
    ▶ __proto__: Object
    ▶ [[Target]]: Vue
    [[IsRevoked]]: false
```

Figura 1: Objeto Proxy básico preparado para receber variáveis.

Chama `initLifecycle(vm)`

Chama `initEvents(vm)`

Chama `initRender(vm)`

Chama `callHook(vm, 'beforeCreate')`

Chama `initInjections(vm)`

## VUE – Conceito de variável reativa

Chama initState(vm)

Chama initData(vm)

Chama getData(data, vm)

Chama mergedInstanceDataFn

Retorna instanceData com defaultData (que redonda em message:

“Hello World”

Chama proxy(target, sourceKey, key)

O proxy define uma estrutura básica de objeto para as variáveis:

```
> sharedPropertyDefinition
< ▼ {enumerable: true, configurable: true, get: f, set: f}
  configurable: true
  enumerable: true
  ▶ get: f noop(a, b, c)
  ▶ set: f noop(a, b, c)
  ▶ __proto__: Object
```

Figura 2: Protótipo de variável Proxy

Ao final da função, o protótipo fica da seguinte forma:

```
> sharedPropertyDefinition
< ▼ {enumerable: true, configurable: true, get: f, set: f}
  configurable: true
  enumerable: true
  ▶ get: f proxyGetter()
  ▶ set: f proxySetter(val)
  ▶ __proto__: Object
```

Figura 3: A função noop do protótipo é substituída por proxyGetter e proxySetter, respectivamente em get e set.

Chama observe(data, true), onde data contém ‘message: “Hello World”’

A função produz o objeto Observer:

```
< ▼ {__ob__: Observer} ⓘ
  message: "Hello World!"
  ▶ __ob__: Observer {value: {_-}, dep: Dep, vmCount: 0}
  ▶ get message: f reactiveGetter()
  ▶ set message: f reactiveSetter(newVal)
  ▶ __proto__: Object
```

A diferença entre o objeto **Proxy** e o **Observer** é a de que os métodos **get** e **set** invocam funções mais elaboradas (**reactiveGetter** e **reactiveSetter**). A forma final do **observer** é:

```
▼ Observer {value: {_-}, dep: Dep, vmCount: 1} ⓘ
  ▶ dep: Dep {id: 2, subs: Array(0)}
  ▼ value:
    message: "Hello World!"
    ▶ __ob__: Observer {value: {_-}, dep: Dep, vmCount: 1}
    ▶ get message: f reactiveGetter()
    ▶ set message: f reactiveSetter(newVal)
    ▶ __proto__: Object
  vmCount: 1
  ▶ __proto__: Object
```

## Uso da variável

Repetindo: a seção `data` de `options`, definida na instância do objeto `Vue`, contém os valores INICIAIS das variáveis. No corpo do `HTML` esta variável poderá ou não ser utilizada. E para isto, será necessária a compilação do contexto onde ela será usada (`DIV#app`).

Mostraremos o processo em dois momentos:

- Quando o contexto é compilado;
- Quando o contexto é montado e renderizado.

## Compilação do trecho texto

Para não complicar, mostraremos o momento do `parse` sobre o trecho `"{{ message }}"`.

A compilação é disparada pelo processo `Vue.$mount`, até chegar em `parseHTML`.

`"{{ message }}"` é colocada na variável `text`. É chamado o método `options.chars(text)`. Que é método de `parseHTML`, o qual chamamos. Este método (`options.chars`) chama `parseText`.

É feita uma operação `Regex` com `tagRE` como expressão regular sobre `text`, até que se esgotem as variáveis `Vue`, resultando em:

```
▼ (2) [{"{{ message }}", " message ", index: 0, input: "{{ message }}"}, {"_s(message)", "_s(message)", index: 0, input: "_s(message)"}]
```

---

Figura 4: Exemplo com uma só variável para tornar a explicação compreensível.

Ao final da análise deste trecho, duas estruturas recebem valores: `tokens` e `rawTokens`. Ambas são arrays:

```
tokens
▼ [{"_s(message)"}]
  0: "_s(message)"
  length: 1
  ▶ __proto__: Array(0)
rawTokens
▼ [{"..."}]
  0: {@binding: "message"}
  length: 1
  ▶ __proto__: Array(0)
```

---

Em `tokens` vemos a chamada de função a ser usada na renderização.

## VUE – Conceito de variável reativa

O produto da compilação é a seguinte estrutura:

```
> root
< ▼ {type: 1, tag: "div", attrsList: Array(1), attrsMap: {...}, parent: undefined, ...} ⓘ
  ▼ attrs: Array(1)
    ▶ 0: {name: "id", value: "'app'"}
      length: 1
    ▶ __proto__: Array(0)
  ▶ attrsList: [{}]
  ▶ attrsMap: {id: "app"}
  ▼ children: Array(1)
    ▶ 0: {type: 2, expression: "_s(message)", tokens: Array(1), text: "{{ message }}" }
      length: 1
    ▶ __proto__: Array(0)
  parent: undefined
  plain: false
  tag: "div"
  type: 1
  ▶ __proto__: Object
```

Está registrado que a tag de contexto é **DIV** (tag: “div”), cujo **id** é “app” e que os nós filhos são **children**, com a expressão “*message*”.

Com este código, o Vue chama a função **generate(ast, options)**, onde ast é a estrutura mostrada anteriormente (**root**). Esta acaba chamando a função **genElement** para a porção “div”, determinante do contexto. O resultado é:

```
> code
< "_c('div',{attrs:{"id":"app"}},[_v(_s(message))])"
```

Este é o código em que foi transformado o contexto HTML onde o **Vue** vai agir. Mas a função **generate** ainda acrescenta mais um trecho, antes do retorno:

```
“with(this){return _c('div',{attrs:{"id":"app"}},[_v(_s(message))])}”
```

## VUE – Conceito de variável reativa

A função compile, após o retorno daquelas que chamou, devolve uma estrutura AST, como segue:

```
▼ ast:
  ▶ attrs: [{}]
```

---

```
  ▶ attrsList: [{}]
```

---

```
  ▶ attrsMap: {id: "app"}
```

---

```
  ▶ children: [{}]
```

---

```
    parent: undefined
```

---

```
    plain: false
```

---

```
    static: false
```

---

```
    staticRoot: false
```

---

```
    tag: "div"
```

---

```
    type: 1
```

---

```
  ▶ __proto__: Object
```

---

```
▶ errors: []
```

---

```
render: "with(this){return _c('div',{attrs:{"id":"app"}},[_v(_s(message))])}"
```

---

```
▶ staticRenderFns: []
```

---

```
▶ tips: []
```

---

```
▶ __proto__: Object
```

Quem faz a atualização dos valores são os Watchers, atualizados pela função mountComponent:

```
▼ Watcher {vm: Vue, sync: false, lazy: false, user: false, deep: false, ...} ⓘ
```

---

```
  active: true
```

---

```
▶ cb: f noop(a, b, c)
```

---

```
  deep: false
```

---

```
▶ depIds: Set(0) {}
```

---

```
▶ deps: []
```

---

```
  dirty: false
```

---

```
  expression: "function () {# vm._update(vm._render(), hydrating);# }"
```

---

```
▶ getter: f ()
```

---

```
  id: 1
```

---

```
  lazy: false
```

---

```
▶ newDepIds: Set(0) {}
```

---

```
▶ newDeps: []
```

---

```
  sync: false
```

---

```
  user: false
```

---

```
▶ vm: Vue {_uid: 0, _isVue: true, $options: {}, _renderProxy: Proxy, _self: Vue, ...}
```

## VUE – Conceito de variável reativa

A função `getter` está definida, nesta estrutura, como:

```
▼ getter: f ()
  arguments: (...)
  caller: (...)
  length: 0
  name: "updateComponent"
  ▶ prototype: {constructor: f}
  ▶ __proto__: f ()
  [[FunctionLocation]]: vue.js:2787
  ▶ [[Scopes]]: Scopes[3]
  id: 1
  lazy: false
```

onde se define o local como a linha 2787 da biblioteca Vue utilizada:

```
2787 -   updateComponent = function () {
2788         vm._update(vm._render(), hydrating);
2789     };
```

O processamento continua em **Vue.prototype.\_render**. O elemento chave da renderização é a chamada:

```
vnode = render.call(vm._renderProxy, vm.$createElement);
```

Quando esta função é chamada, `vm._renderProxy` aponta para:

```
1 (function() {
2   with(this){return _c('div',{attrs:{"id":"app"}},[_v(_s(message))])}
3 })
```

E quem é o `this` neste contexto ? É o objeto Proxy:

```
▼ Proxy {_uid: 0, _isVue: true, $options: {...}, _renderProxy: Proxy, _self: Vue, ...}
  ▼ [[Handler]]: Object
    ▶ has: f has(target, key)
    ▶ __proto__: Object
  ▶ [[Target]]: Vue
  [[IsRevoked]]: false
```



## VUE – Conceito de variável reativa

onde Target é:

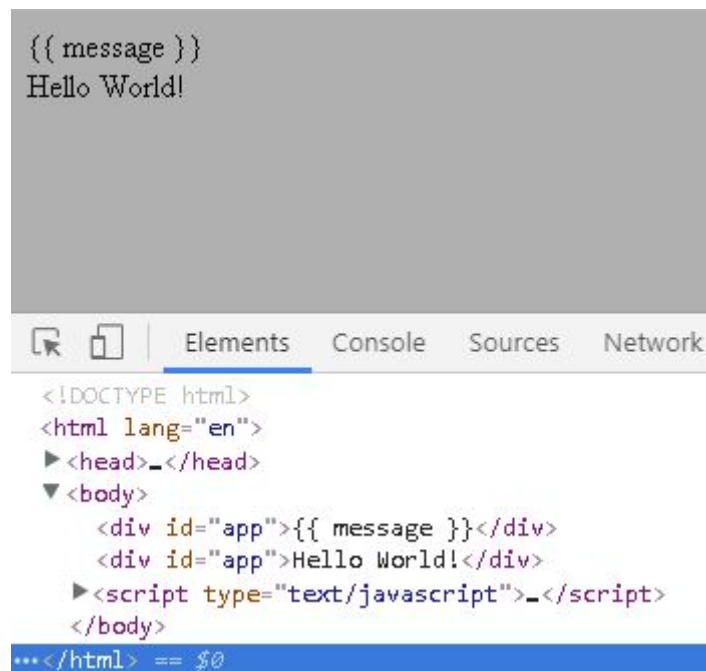
```
▼ [[Target]]: Vue
  $attrs: (...)
  ▶ $children: []
  ▶ $createElement: f (a, b, c, d)
  ▶ $el: div#app
  $listeners: (...)
  ▶ $options: {components: {-}, directives: {-}, filters: {-}, el: "#app", _base: f, -}
  $parent: undefined
  ▶ $refs: {}
  ▶ $root: Vue {_uid: 0, _isVue: true, $options: {-}, _renderProxy: Proxy, _self: Vue, -}
  ▶ $scopedSlots: {}
  ▶ $slots: {}
  $vnode: undefined
  message: (...)
  ▶ _c: f (a, b, c, d)
  ▶ _data: {_ob__: Observer}
  _directInactive: false
  ▶ _events: {}
  _hasHookEvent: false
  _inactive: null
  _isBeingDestroyed: false
  _isDestroyed: false
  _isMounted: false
  _isVue: true
  ▶ _renderProxy: Proxy {_uid: 0, _isVue: true, $options: {-}, _renderProxy: Proxy, _self: Vue, -}
  ▶ _self: Vue {_uid: 0, _isVue: true, $options: {-}, _renderProxy: Proxy, _self: Vue, -}
  _staticTrees: null
  _uid: 0
  _vnode: null
  ▶ _watcher: Watcher {vm: Vue, sync: false, lazy: false, user: false, deep: false, -}
  ▶ _watchers: [Watcher]
  $data: (...)
  $isServer: (...)
  $props: (...)
  $ssrContext: (...)
  ▶ get $attrs: f reactiveGetter()
  ▶ set $attrs: f reactiveSetter(newVal)
  ▶ get $listeners: f reactiveGetter()
  ▶ set $listeners: f reactiveSetter(newVal)
  ▶ get message: f proxyGetter()
  ▶ set message: f proxySetter(val)
  ▶ __proto__: Object
  [[IsRevoked]]: false
```

## VUE – Conceito de variável reativa

Na renderização são chamadas as funções:

`_s()` invoca a função `toString`;  
`_v()` invoca a função `createTextVNode`;  
`_c()` invoca a função `createElement`;

O efeito desta função de renderização é gerar um nó a mais no document de nossa página:



```

{{ message }}
Hello World!

<!DOCTYPE html>
<html lang="en">
  <head>_</head>
  <body>
    <div id="app">{{ message }}</div>
    <div id="app">Hello World!</div>
    <script type="text/javascript">_</script>
  </body>
</html> == $0
```

Este processo é efetuado pela função **patch**, chamada por **Vue.update**, chamada por **updateComponent**, chamada pelo **get** de **Watcher**, chamada por **mountComponent**, que é chamada por **Vue.\$mount**. O primeiro nó DIV contém a template como digitada pelo programador. O segundo nó DIV possui o produto da renderização. Posteriormente, ele é removido, fornecendo o resultado desejado, que é a frase **“Hello World”** em nosso navegador.

Chegamos ao resultado que desejávamos.