VUE

PROCESSO DE COMPILAÇÃO

Versão 1.1 – 31/07/2018 – Bernardo Meyer

O Vue

O **Vue** é uma poderosa framework que implementa um **MVC** (***Model View Controller***) de forma mais eficiente e original do que os modelos ***Angular***, ***React*** e ***jquery***, para citar os mais usados. Para prover um modelo persistente de reação a mudanças de parâmetros, o Vue executa um processo de compilação dos contextos definidos na instanciação do seu objeto.

Neste trabalho descrevemos o processo de compilação e suas estruturas de apoio, bem como os produtos deste processo. Para isto, vamos utilizar o navegador de Internet e o console de erros e debug.

O Processo mount

O processo de compilação do framework VUE é disparado pelo comando:

app.$mount(‘.todoapp’)

onde:

app é a variável onde foi instanciado o objeto Vue em questão (***var app = new VUE({ … })***)
.todoapp é o seletor de classe CSS3 da raiz do contexto Vue (poderia ser ‘#app’ para a tag cujo id é “app”, ou outros tipos de seletores CSS)

A função que procede à compilação vem em cascata de chamadas de pilha:

- compileToFunctions(template,…)
- compile(template,…)
- baseCompile(template, …)
- var ast = parse(template.trim(), options)
- parseHTML(template, options) testa as opções possíveis de elementos HTML
  - Caracter inicial “<”:
    - comment
    - conditionalComment
    - doctypeMatch
    - endTagMatch
    - startTagMatch
  - Texto:
    - options.chars()
    - parseText(text, ...)

Nosso exemplo de compilação é o que consta da página a seguir.

VUE - PROCESSO DE COMPILAÇÃO

```
1  <html>
2    <head>
3      <title>VueJs Instance</title>
4      <script type = "text/javascript" src = "vue.js"></script>
5    </head>
6    <body>
7      <div id = "app">
8        <button v-on:click = "displayparams">Click ME</button>
9        <h2>O evento {{ evento }} vai acontecer em {{ local }}</h2>
10     </div>
11     <script type = "text/javascript">
12       var vm = new Vue({
13         el: '#app',
14         data: {
15           evento: "Open tura",
16           local : "Espaço eventual"
17         },
18         methods : {
19           displayparams : function(event) {
20             this.evento = "Abertura";
21             this.local = "Porta";
22             console.log(event);
23           return this;
24           }
25         },
26       });
27     </script>
28   </body>
29 </html>
```

VUE - PROCESSO DE COMPILAÇÃO

Todo este fonte HTML vem no parâmetro “***template***” da chamada das funções mencionadas na estrutura hierárquica da compilação.

Vamos analisar o trecho inicial:

** ...**

O tipo de trecho para compilação é detectado pela ER (expressão regular) “**startTagOpen**”

/^<((?:[a-zA-Z_][\w\-\.]*\:)?[a-zA-Z_][\w\-\.]*)/

da função **parseStartTag**, pois o caracter inicial desta template é “<”, não seguido de “/”. Os valores devolvidos em formato Array pelo “match” são:

```
1.0:"<div"
2.1:"div"
3.index:0
4.input:"<div id="app">⍰ <button v-on:click="displayparams">Click ME</button>⍰ <h2>O
evento {{ evento }} vai acontecer em {{ local }}</h2>⍰ </div>"
5.length:2
```

Através destes resultados é alimentado um objeto match, da seguinte forma:

{tagName:"div",attrs:Array(0),start:0}

```
1.attrs:[]
2.start:0
3.tagName:"div"
4.__proto__:Object
```

Ou seja, foi detectado o nome da tag e inicializados os atributos como uma matriz vazia (“[]”).

É feito um avanço no caracter analisado pelo parse até depois do “v” (4 caracteres), para tentar detectar atributos.

Os atributos são detectados pela ER “attribute”:

/^\s*([^\s"'<>\/=]+)(?:\s*(=)\s*(?:"([^"]*)"+|'([^']*)'+|([^\s"'=<>`]+)))?/

da função match de Regexp, que devolve o Array “attr”:

```
1.0:" id="app""
2.1:"id"
3.2:"="
4.3:"app"
5.index:0
6.length:6
7.__proto__:Array(0)
```

Se houvesse mais atributos, eles seriam devolvidos a cada passada do loop de rastreio da tag na mesma estrutura. O rastreio é interrompido quando é detectado o caracter “>”. O parse avança sobre este caracter.

A função parseStartTag retorna o Array “match”:

```
attrs:Array(1)
       0:Array(6)
              0:" id="app""
              1:"id"
              2:"="
              3:"app"
              index:0
              length:6
              __proto__:Array(0)
       length:1
       __proto__:Array(0)
start:0
tagName:"div"
__proto__:Object
```

O compilador passa a manipular esta resposta através da função “**handleStartTag**”. Esta função testa se a tag detectada pode ser uma tag de abertura HTML, e a coloca na pilha “stack”:

```
0:
       attrs:Array(1)
              0:{name:"id", value:"app"}
              length:1
              __proto__:Array(0)
       lowerCasedTag:"div"
       tag:"div"
       __proto__:Object
length:1
__proto__:Array(0)
```

Ou seja, a memória do parse está cuidadosamente armazenada.

Criação do elemento da Template

Ao final da função handleStartTag, o compilador vai criar um elemento de template VUE, invocando a função createASTElement:

var element = **createASTElement**(tag, attrs, currentParent);

Os parâmetros de entrada tem os seguintes conteúdos:

tag - “div”
attrs:

```
1.0:{name: "id", value: "app"}
2.length:1
3.__proto__:Array(0)
```

currentParent – undefined

O retorno da função (element) é:

```
attrsList:Array(1)
        0:{name:"id",value:"app"}
        length:1
        __proto__:Array(0)
attrsMap:
        id:"app"
        __proto__:Object
children:[]
parent:undefined
tag:"div"
type:1
__proto__:Object
```

É muito conveniente ter uma estrutura redundante, pois pode-se referir aos seus elementos em loop (formato Array) às propriedades (***name***, ***value***) ou através da chave (“**id**”).

Ao final do tratamento da startTag, a pilha “**stack**” é alimentada com o produto desta passada:

```
0:
        attrs:Array(1)
                0:{name:"id",value:""app""}
                length:1
                __proto__:Array(0)
        attrsList:Array(1)
                0:{name:"id", value:"app"}
                length:1
                __proto__:Array(0)
        attrsMap:
                id:"app"
                __proto__:Object
        children:[]
        parent:undefined
        plain:false
        tag:"div"
        type:1
        __proto__:Object
length:1
__proto__:Array(0)
```

Tratamento da tag BUTTON com evento

Ao passar pela ER de startTag, o match Regexp devolve a seguinte estrutura na variável **start**:

```
1.0:"<button"
2.1:"button"
3.index:0
4.length:2
5.__proto__:Array(0)
```

Esta é inserida em uma outra, com o fim de produzir redundância útil no processamento, de nome “match”:

```
attrs:Array(1)
        0:Array(6)
                0:" v-on:click="displayparams""
                1:"v-on:click"
                2:"="
                3:"displayparams"
                index:0
                length:6
                __proto__:Array(0)
        length:1
        __proto__:Array(0)
start:24
tagName:"button"
__proto__:Object
```

Criando o elemento template

Após a chamada da função **processElement**, é devolvida uma estrutura **element** da seguinte forma:

```
attrsList:Array(1)
        0:{name:"v-on:click",value:"displayparams"}
        length:1
        __proto__:Array(0)
attrsMap:{v-on:click:"displayparams"}
children:[]
events:
        click:{value:"displayparams"}
        __proto__:Object
hasBindings:true
parent:{type:1,tag:"div",attrsList:Array(1),attrsMap:{…},parent:undefined,…}
plain:false
tag:"button"
type:1
__proto__:Object
```

O elemento **currentParent**, raiz da template, recebe a template deste BUTTON em sua propriedade children. Desta forma as partes da template vão sendo inseridas na hierarquia correta:

```
attrs:Array(1)
        0:{name:"id",value:""app""}
        length:1
        __proto__:Array(0)
attrsList:Array(1)
        0:{name:"id",value:"app"}
        length:1
        __proto__:Array(0)
attrsMap:{id:"app"}
children:Array(1)
        0:{type:1,tag:"button",attrsList:Array(1),attrsMap:{…},parent:{…},…}
        length:1
        __proto__:Array(0)
parent:undefined
plain:false
tag:"div"
type:1
__proto__:Object
```

Este último elemento é colocado na pilha “**stack**”, que fica da seguinte forma:

```
0:
        attrs:Array(1)
                0:{name:"id",value:""app""}
                length:1
                __proto__:Array(0)
        attrsList:Array(1)
                0:{name:"id",value:"app"}
                length:1
                __proto__:Array(0)
        attrsMap:
                id:"app"
                __proto__:Object
        children:Array(1)
                0:{type:1,tag:"button",attrsList:Array(1),attrsMap:{…},parent:{…},…}
                length:1
                __proto__:Array(0)
        parent:undefined
        plain:false
        tag:"div"
        type:1
        __proto__:Object
1:
        attrsList:Array(1)
                0:
                        name:"v-on:click"
                        value:"displayparams"
                        __proto__:Object
                length:1
                __proto__:Array(0)
        attrsMap:
                v-on:click:"displayparams"
                __proto__:Object
        children:[]
        events:
                click:{value:"displayparams"}
                __proto__:Object
        hasBindings:true
        parent:{type:1,tag:"div",attrsList:Array(1),attrsMap:{…},parent:undefined,…}
        plain:false
        tag:"button"
        type:1
        __proto__:Object
length:2
__proto__:Array(0)
```

A pilha contém a estrutura hierárquica na ordem de abertura e de fechamento dos níveis das tags, para o momento. Você vai observar que, quando as tags forem fechadas, os elementos serão desempilhados.

Análise de texto puro

Entre as tags de HTML podemos ter trechos de texto. Vejamos como eles são tratados. Os textos são detectados por exceção. Se nenhuma das condições de comentário HTML, comentário condicional, início de tag ou fim de tag for atendida, trata-se de texto plano (plain text).

Passando pela tag H2

A tag H2 de nossa template é detectada pela ER startTag, e o resultado da procura é devolvido na variável **match**:

```
attrs:Array(0)
       length:0
       __proto__:Array(0)
end:90
start:86
tagName:"h2"
unarySlash:""
__proto__:Object
```

A função **start** do método **parseHTML** do objeto principal Vue coloca em evidência a variável **currentParent**, que contém as estruturas da ***template*** em organização hierárquica.

Vejamos como ficou currentParent após a compilação da tag **H2**:

```
attrs:Array(1)
       0:{name:"id",value:""app""}
       length:1
       __proto__:Array(0)
attrsList:Array(1)
       0:{name:"id",value:"app"}
       length:1
       __proto__:Array(0)
attrsMap:{id:"app"}
children:Array(3)
       0:{type:1,tag:"button",attrsList:Array(1),attrsMap:{…},parent:{…},…}
       1:{type:3,text:" "}
       2:{type:1,tag:"h2",attrsList:Array(0),attrsMap:{…},parent:{…},…}
       length:3
       __proto__:Array(0)
parent:undefined
plain:false
tag:"div"
type:1
__proto__:Object
```

Compilação do texto com parâmetros “watch” do VUE

O VUE oferece a facilidade de alteração dinâmica de parâmetros no interior do conteúdo texto das tags do DOM. Vejamos agora a compilação deste tipo de trecho.

Esta compilação de texto invoca a função **parseText** do VUE. A detecção dos parâmetros VUE é feita pela **ER** tagRE. Esta função faz várias passadas usando esta ER. Na primeira o resultado é:

```
0:"{{ evento }}"
1:" evento "
index:9
input:"O evento {{ evento }} vai acontecer em {{ local }}"
length:2
__proto__:Array(0)
```

O VUE também armazena a expressão obtida em **token**, a ser incluída na função compilada:

```
0:""O evento ""
1:"_s(evento)"
length:2
__proto__:Array(0)
```

Na segunda passada, o resultado é:

```
0:"{{ local }}"
1:" local "
index:39
input:"O evento {{ evento }} vai acontecer em {{ local }}"
length:2
__proto__:Array(0)
```

e o token produzido é:

```
1.0:""O evento ""
2.1:"_s(evento)"
3.2:"" vai acontecer em ""
4.length:3
5.__proto__:Array(0)
```

A expressão completamente compilada fica armazenada em **rawTokens**:

```
0:"O evento "
1:{@binding: "evento"}
2:" vai acontecer em "
3:{@binding: "local"}
length:4
__proto__:Array(0)
```

Ao final temos a expressão completa compilada em **root**:

```
attrs:Array(1)
        0:{name:"id",value:""app""}
        length:1
        __proto__:Array(0)
attrsList:Array(1)
        0:{name:"id",value:"app"}
        length:1
        __proto__:Array(0)
attrsMap:
        id:"app"
        __proto__:Object
children:Array(3)
        0:
                attrsList:Array(1)
                        0:{name:"v-on:click",value:"displayparams"}
                        length:1
                        __proto__:Array(0)
                attrsMap:{v-on:click:"displayparams"}
                children:Array(1)
                        0:{type:3,text:"Click ME"}
                        length:1
                        __proto__:Array(0)
                events:
                        click:{value:"displayparams"}
                        __proto__:Object
                hasBindings:true
                parent:{type:1,tag:"div",attrsList:Array(1),attrsMap:{…},parent:undefined,…}
                plain:false
                tag:"button"
                type:1
                __proto__:Object
        1:{type:3,text:" "}
        2:
                attrsList:[]
                attrsMap:{}
                children:Array(1)
                        0:{type:2,expression:""O evento "+_s(evento)+" vai acontecer em
                        "+_s(local)",tokens:Array(4),text:"O evento {{ evento }} vai acontecer em
                        {{ local }}"}
                        length:1
                        __proto__:Array(0)
                parent:{type:1,tag:"div",attrsList:Array(1),attrsMap:{…},parent:undefined,…}
                plain:true
                tag:"h2"
                type:1
                __proto__:Object
        length:3
        __proto__:Array(0)
parent:undefined
plain:false
tag:"div"
type:1
__proto__:Object
```

A função **genElement** transforma toda esta estrutura lida no trecho de função:

```
_c('div',
        {attrs:{"id":"app"}},
        [
        _c('button',{on:{"click":displayparams}},[_v("Click ME")]),
        _v(" "),
        _c('h2',[_v("O evento "+_s(evento)+" vai acontecer em "+_s(local))])
        ]
)
```

A estrutura completa após a compilação é devolvida na variável **compiled**, ao final da função **compile**:

```
ast:
        attrs:Array(1)
                0:{name:"id",value:""app""}
                length:1
                __proto__:Array(0)
        attrsList:Array(1)
                0:{name:"id",value:"app"}
                length:1
                __proto__:Array(0)
        attrsMap:{id:"app"}
        children:Array(3)
                0:
                        attrsList:Array(1)
                                0:{name:"v-on:click",value:"displayparams"}
                                length:1
                                __proto__:Array(0)
                        attrsMap:{v-on:click:"displayparams"}
                        children:[{…}]
                        events:{click:{…}}
                        hasBindings:true
                        parent:{type:1,tag:"div",attrsList:Array(1),attrsMap:
                        {…},parent:undefined,…}
                        plain:false
                        static:false
                        staticRoot:false
                        tag:"button"
                        type:1
                        __proto__:Object
                1:{type:3,text:" ",static:true}
                2:
                        attrsList:[]
                        attrsMap:{}
                        children:Array(1)
                                0:{type:2,expression:""O evento "+_s(evento)+" vai acontecer
                                em "+_s(local)",tokens:Array(4),text:"O evento {{ evento }}
                                vai acontecer em {{ local }}",static:false}
                                length:1
                                __proto__:Array(0)
                        parent:{type:1,tag:"div",attrsList:Array(1),attrsMap:
                        {…},parent:undefined,…}
```

```
                        plain:true
                        static:false
                        staticRoot:false
                        tag:"h2"
                        type:1
                        __proto__:Object
                length:3
                __proto__:Array(0)
        parent:undefined
        plain:false
        static:false
        staticRoot:false
        tag:"div"
        type:1
        __proto__:Object
errors:[]
render:"with(this){return _c('div',{attrs:{"id":"app"}},[_c('button',{on:
{"click":displayparams}},[_v("Click ME")]),_v(" "),_c('h2',[_v("O evento "+_s(evento)+"
vai acontecer em "+_s(local))])])}"
staticRenderFns:[]
tips:[]
__proto__:Object
```

VUE - PROCESSO DE COMPILAÇÃO

No processamento da substituição de valores para os parâmetros watch do VUE, é feita uma estrutura Vnode:

```
asyncFactory:undefined
asyncMeta:undefined
children:Array(3)
       0:VNode {tag: "button", data: {…}, children: Array(1), text: undefined, elm:
       undefined, …}
       1:VNode {tag: undefined, data: undefined, children: undefined, text: " ", elm:
       undefined, …}
       2:VNode {tag: "h2", data: undefined, children: Array(1), text: undefined, elm:
       undefined, …}
       length:3
       __proto__:Array(0)
componentInstance:undefined
componentOptions:undefined
context:Vue {_uid: 0, _isVue: true, $options: {…}, _renderProxy: Proxy, _self: Vue, …}
data:
       attrs:{id: "app"}
       __proto__:Object
elm:undefined
fnContext:undefined
fnOptions:undefined
fnScopeId:undefined
isAsyncPlaceholder:false
isCloned:false
isComment:false
isOnce:false
isRootInsert:true
isStatic:false
key:undefined
ns:undefined
parent:undefined
raw:false
tag:"div"
text:undefined
child:(...)
__proto__:Object
```

As variáveis VUE “{{ … }}” são substituídas na template da seguinte forma:

- Com base na template física, obtida do rastreio do DOM, é montada a estrutura **ast**, como vimos anteriormente;
- Com base nesta estrutura **ast** é montada uma árvore DOM paralela, com a substituição dos parâmetros;
- A estrutura paralela é montada na árvore DOM, e a anterior é excluída;

E o que acontece quando forem necessárias novas substituições dos parâmetros, se verificamos, pelo debug dos navegadores, que os parâmetros “chave” VUE foram eliminados na estrutura paralela ?

Clicando no botão “Click me”

O botão mostrado dispara uma alteração nos parâmetros VUE. Vejamos como isto é feito.

Este botão dispara o método “***displayparams***” enumerado na seção **methods** na instanciação de um objeto VUE:

```
displayparams : function(event) {
        this.evento = "Abertura";
        this.local = "Porta";
        console.log(event);
    return this;
```

Vamos acompanhar o processamento deste click. Antes, porém, vamos fazer algumas considerações sobre o modelo **Vue** de estabelecimento de um contexto:

O processamento de variáveis (o termo correto seria propriedades dinâmicas de templates) é intermediado pela função ***proxy***. O método em relação ao objeto da variável é o **set**, ou seja, a propriedade de nome ***evento*** (não tem nada a ver com os eventos; é apenas um nome) vai receber um valor, ou terá seu valor modificado.

E na instância do objeto Vue corrente (**this**) a propriedade **evento** é referenciada como:

this.[“_data”][“evento”]

O set desta propriedade é executado pelo método proxySetter do objeto Proxy. Coisa semelhante ocorre com a propriedade **local**:

this.[“_data”][“local”]

Ao contrário do que seria natural esperar, o retorno da função não é feito imediatamente pelo ***return*** da função **displayparams**. Este return desvia o processamento para a função **createInvoker** do Vue.

O desvio é feito para **fn._withTask.fn._withTask** interiiormente à função **withMicroTask**, tendo como parâmetro **fn**, que aponta para a função **invoker** do Vue, tudo isto definido na inicialização do objeto Vue.

O retorno desta função é desviado para a função **flushCallbacks**(). Esta coloca funções numa pilha de callbacks.

VUE - PROCESSO DE COMPILAÇÃO

Definição de variáveis

As variáveis ou parâmetros VUE (nomes entre {{}}) são definidas como objetos formais dentro da estrutura vm do Vue.js.

A estrutura comum de objeto que recebe uma destas variáveis é **sharedPropertyDefinition**:

```
1.configurable:true
2.enumerable:true
3.get:ƒ proxyGetter()
        1.arguments:(...)
        2.caller:(...)
        3.length:0
        4.name:"proxyGetter"
        5.prototype:{constructor:ƒ}
        6.__proto__:ƒ()
        7.[[FunctionLocation]]:vue.js:3294
        8.[[Scopes]]:Scopes[3]
4.set:ƒ proxySetter(val)
        1.arguments:(...)
        2.caller:(...)
        3.length:1
        4.name:"proxySetter"
        5.prototype:{constructor:ƒ}
        6.__proto__:ƒ()
        7.[[FunctionLocation]]:vue.js:3297
        8.[[Scopes]]:Scopes[3]
5.__proto__:Object
```

Cujas funções são:

```
function proxySetter(val) {
   this[sourceKey][key] = val;
 }

function proxyGetter() {
   return this[sourceKey][key]
 }
```

Mas qual função sustenta estas definições ? A função **proxy**:

```
function proxy (target, sourceKey, key) {
        sharedPropertyDefinition.get = function proxyGetter () {
                return this[sourceKey][key]
                };
        sharedPropertyDefinition.set = function proxySetter (val) {
                this[sourceKey][key] = val;
                };
        Object.defineProperty(target, key, sharedPropertyDefinition);
        }
```

Neste caso, **this** é uma instância do objeto **Vue**.

Mas por quê falamos de Definição de Variáveis ? Porque esta é uma parte importante da chamada da função **initData**, invocada pela fase **initState** do Vue.

Na inicialização do objeto Vue, a função **_init** invoca a função **initProxy(vm)** onde **vm** é uma instância do próprio Vue. Esta inicia um objeto **Proxy:**

```
[[Handler]]:Object
        has:ƒ has(target, key)
                arguments:(...)
                caller:(...)
                length:2
                name:"has"
                prototype:{constructor: ƒ}
                __proto__:ƒ ()
                [[FunctionLocation]]:vue.js:1930
                [[Scopes]]:Scopes[2]
        __proto__:Object
[[Target]]:Vue
[[IsRevoked]]:false
```

Na inicialização, o único método é “has”, cuja única finalidade é testar se um determinado objeto possui uma propriedade ou método:

```
function has (target, key) {
    var has = key in target;
    var isAllowed = allowedGlobals(key) || key.charAt(0) === '_';
    if (!has && !isAllowed) {
      warnNonPresent(target, key); // Mensagem de ausência
    }
    return has || !isAllowed
  }
```

Se existirem variáveis declaradas na porção data da instanciação do Vue, será invocada a função **initData(*vm*)**, onde vm é instância de Vue.

Seção Data do Vue

Na instanciação do Vue, temos a seção Data, que em nosso exemplo é:

```
data: {
   evento: "Open tura",
   local : "Espaço eventual"
}
```

O que pode passar despercebido do profissional de TI, que está conhecendo o **Vue**, é que esta seção é um objeto, e ele se esquecer das chaves (**{}**) que delimitam a seção.

VUE - PROCESSO DE COMPILAÇÃO

A função **initData** começa por recolher as chaves em um Objeto:

```
(2) ["evento", "local"]
        1.0:"evento"
        2.1:"local"
        3.length:2
        4.__proto__:Array(0)
```

Estas são as variáveis a serem controladas pelo Vue. Para cada uma delas é chamada a função **proxy**. Como já vimos, esta função define um método getter e um método setter para uma variável Vue.

O resultado é que são acrescidos na instância do Vue o getter e o setter desta variável:

```
        1.get local:ƒ proxyGetter()
                1.arguments:(...)
                2.caller:(...)
                3.length:0
                4.name:"proxyGetter"
                5.prototype:{constructor: ƒ}
                6.__proto__:ƒ ()
                7.[[FunctionLocation]]:vue.js:3294
                8.[[Scopes]]:Scopes[3]
        2.set local:ƒ proxySetter(val)
                1.arguments:(...)
                2.caller:(...)
                3.length:1
                4.name:"proxySetter"
                5.prototype:{constructor: ƒ}
                6.__proto__:ƒ ()
                7.[[FunctionLocation]]:vue.js:3297
                8.[[Scopes]]:Scopes[3]
```

Ou seja, quem vai cuidar da manutenção das variáveis Vue é a função **proxy**. Veremos como isto será posteriormente feito. Mas além da definição, é preciso construir uma infra-estrutura de “observação” do comportamento desta variável. Isto é feito pela função **observe**, que cria um objeto **Observer**:

```
{__ob__:Observer}
        evento:"Open tura"
        local:"Espaço eventual"
        __ob__:Observer{value:{…},dep:Dep,vmCount:0}
        get evento:ƒ reactiveGetter()
        set evento:ƒ reactiveSetter(newVal)
        get local:ƒ reactiveGetter()
        set local:ƒ reactiveSetter(newVal)
        __proto__:Object
```

Seus métodos acessores são:

```
  get: function reactiveGetter () {
    var value = getter ? getter.call(obj) : val;
    if (Dep.target) {
      dep.depend();
      if (childOb) {
        childOb.dep.depend();
        if (Array.isArray(value)) {
          dependArray(value);
        }
      }
    }
    return value
  },
  set: function reactiveSetter (newVal) {
    var value = getter ? getter.call(obj) : val;
    /* eslint-disable no-self-compare */
    if (newVal === value || (newVal !== newVal && value !== value)) {
      return
    }
```

Onde:

var property = Object.getOwnPropertyDescriptor(obj, key); // obj: objeto na forma key, value

e

```
 var getter = property && property.get;
 if (!getter && arguments.length === 2) {
            val = obj[key];
            }
```

O parâmetro ***arguments.length*** testado acima, como tendo que ser igual a 2, é o número de argumentos válidos, ou seja, que não provocaram exceção na chamada da função **defineReactive**, chamada pelo **initRender**, quando da inicialização do objeto Vue.

O processo mount e as funções de compilação

Após as inicializações do Vue, é disparado o processo mount pelo método ***$mount*** do Vue. Este recebe como parâmetro o objeto **DOM** que estabelece o contexto Vue. Em nosso exemplo é a **DIV** de Id "***app***", armazenada em **template**:

```
<div id = "app">
  <button v-on:click = "displayparams">Click ME</button>
  <h2>O evento {{ evento }} vai acontecer em {{ local }}</h2>
</div>
```

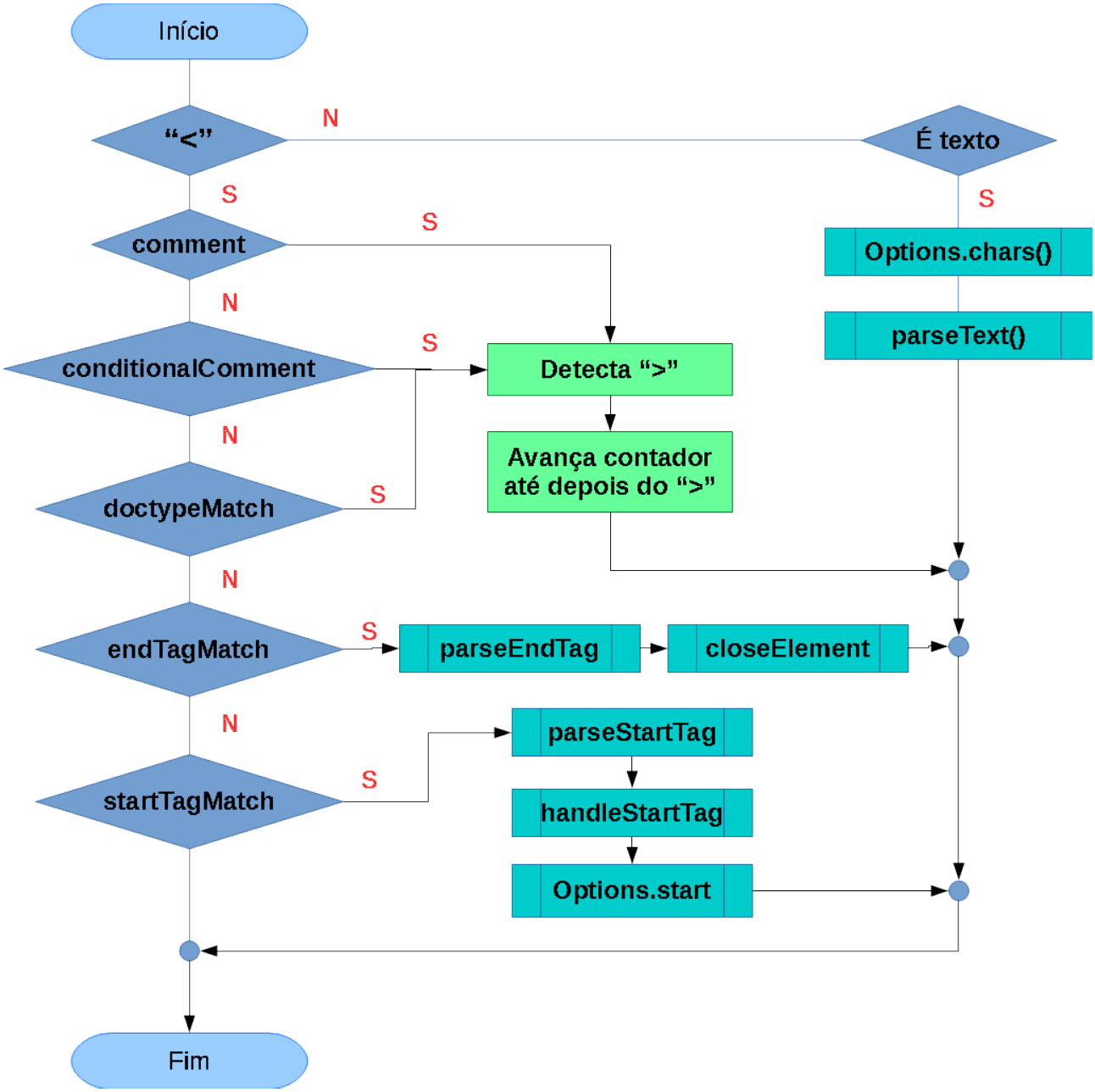
Esta passa pelos processos em cascata:

- compileToFunctions(template, …);
- compile(template, …);
- baseCompile(template, …);
- parse(template, …);
- parseHTML(template, …); // Ver detalhe no item "**O processo mount**" no início deste trabalho

O objetivo da compilação é construir uma ***estrutura de objeto*** capaz de armazenar esta ***template em DOM*** com os seus níveis hierárquicos e **variáveis VUE** para oferecer o comportamento dinâmico do padrão **MVC**.

VUE - PROCESSO DE COMPILAÇÃO

O núcleo funcional deste processo de compilação está na função **parseHTML**, cujo diagrama esquemático é dado a seguir:



VUE - PROCESSO DE COMPILAÇÃO

Versão 1.1 – 31/07/2018 – Bernardo Meyer

VUE - PROCESSO DE COMPILAÇÃO

Ao final da compilação das tags HTML, é obtido o produto **root**, onde "**root**", para o nosso exemplo, é:

```
attrs:Array(1)
       0:{name: "id", value: ""app""}
       length:1
       __proto__:Array(0)
attrsList:Array(1)
       0:{name: "id", value: "app"}
       length:1
       __proto__:Array(0)
attrsMap:{id: "app"}
children:Array(3)
       0:
              attrsList:Array(1)
                     0:{name: "v-on:click", value: "displayparams"}
                     length:1
                     __proto__:Array(0)
              attrsMap:{v-on:click: "displayparams"}
              children:Array(1)
                     0:{type: 3, text: "Click ME"}
                     length:1
                     __proto__:Array(0)
              events:{click: {…}}
              hasBindings:true
              parent:{type: 1, tag: "div", attrsList: Array(1), attrsMap: {…}, parent:
              undefined, …}
              plain:false
              tag:"button"
              type:1
              __proto__:Object
       1:{type: 3, text: " "}
       2:
              attrsList:[]
              attrsMap:{}
              children:Array(1)
                     0:{type: 2, expression: ""O evento "+_s(evento)+" vai acontecer em
                     "+_s(local)", tokens: Array(4), text: "O evento {{ evento }} vai
                     acontecer em {{ local }}"}
                     length:1
                     __proto__:Array(0)
              parent:{type: 1, tag: "div", attrsList: Array(1), attrsMap: {…}, parent:
              undefined, …}
              plain:true
              tag:"h2"
              type:1
              __proto__:Object
       length:3
       __proto__:Array(0)
parent:undefined
plain:false
tag:"div"
```

```
type:1
__proto__:Object
```

Os nodos do tipo texto ou atributos **bind** de modo texto são convertidos em funções VUE. Isto pode ser observado em uma das propriedades **children** da estrutura apresentada:

```
""O evento "+_s(evento)+" vai acontecer em "+_s(local)", tokens: Array(4), text: "O evento
{{ evento }} vai acontecer em {{ local }}"
```

Esta estrutura compilada também passa pela função **generate**, que vai fazer outras transformações para funções VUE. A função **generate** chama **genElement** ou constrói uma função "**_c(**".

A função **genElement** gera trechos específicos para cada tipo de atributo dos nodos correspondentes às tags HTML (for, if, once, etc), e a função **genComponent** gera os trechos para os componentes.

Para os nodos filhos é chamada a função **genChildren**, que pode chamar **genNode**, que pode chamar **genElement**. No caso de nodos texto é chamada a função genText, sendo construída uma função "**_v(**".

No caso da nossa tag **button** é construído um trecho, pela função genElement, como a seguir:

"**_c(**'button',{on:{"click":***displayparams***}},[**_v(**"Click ME"**)**])"

Para a tag H2 a construção é:

"**_c(**'h2',[**_v(**"O evento "+**_s(evento)**+" vai acontecer em "+**_s(local))**])"

A geração final deste código VUE é:

"**_c(**'div',{attrs:{"id":"app"}},[**_c(**'button',{on:{"click":***displayparams***}},[**_v(**"Click ME"**)**]),**_v(**" "**)**,**_c(**'h2',[**_v(**"O evento "+**_s(evento)**+" vai acontecer em "+**_s(local))**])])"

No retorno da função **baseCompile**, o código é acrescido de outro inicial, ficando desta forma:

"**with(this)**{**return _c(**'div',{attrs:{"id":"app"}},[**_c(**'button',{on:{"click":***displayparams***}},[**_v(**"Click ME"**)**]),**_v(**" "**)**,**_c(**'h2',[**_v(**"O evento "+**_s(evento)**+" vai acontecer em "+**_s(local))**])])}"

VUE - PROCESSO DE COMPILAÇÃO

No retorno da função **compile**, a estrutura **ast** devolvida é a seguinte:

```
ast:
        attrs:Array(1)
                0:{name: "id", value: ""app""}
                length:1
                __proto__:Array(0)
        attrsList:Array(1)
                0:{name: "id", value: "app"}
                length:1
                __proto__:Array(0)
        attrsMap:{id: "app"}
        children:Array(3)
                0:{type: 1, tag: "button", attrsList: Array(1), attrsMap: {…}, parent: {…},
                …}
                1:{type: 3, text: " ", static: true}
                2:{type: 1, tag: "h2", attrsList: Array(0), attrsMap: {…}, parent: {…}, …}
                length:3
                __proto__:Array(0)
        parent:undefined
        plain:false
        static:false
        staticRoot:false
        tag:"div"
        type:1
        __proto__:Object
errors:[]
render:"with(this){return _c('div',{attrs:{"id":"app"}},[_c('button',{on:
{"click":displayparams}},[_v("Click ME")]),_v(" "),_c('h2',[_v("O evento "+_s(evento)+"
vai acontecer em "+_s(local))])])}"
staticRenderFns:[]
tips:[]
__proto__:Object
```

Quando esta função **compile** retorna à função **compileToFunctions**, é chamada a função **createFunction**, que coloca a forma final da função VUE resultante da estrutura **Ast**:

```
(function(){
with(this){return _c('div',{attrs:{"id":"app"}},[_c('button',{on:{"click":displayparams}},[_v("Click
ME")]),_v(" "),_c('h2',[_v("O evento "+_s(evento)+" vai acontecer em "+_s(local))])])}
})
```

Uso da função de renderização

Por que fazer uma função para a renderização ?

A função é necessária pelo fato de que, após a primeira substituição dos parâmetros, a estrutura da template no DOM é perdida. E como esta função pode se transformar em HTML ?

Durante um debug de uma página que utilize o Vue, usando o console, em um breakpoint antes de mount.call (…) ser executado, coloque este código em uma variável:

codigo = “with(this){return _c('div',{attrs:{"id":"app"}},[_c('button',{on:{"click":displayparams}},[_v("Click ME")]),_v(" "),_c('h2',[_v("O evento "+_s(evento)+" vai acontecer em "+_s(local))])])}”

Gere uma função a partir deste código:

var a = new Function(codigo)

Execute:

a.call(**this**)

Por que, se a função é anônima, precisamos fornecer o parâmetro **this**? Porque as funções “**_c**”, “**_s**” e “**_v**”, são como métodos do Vue. O parâmetro **this** que representa a instância Vue é passado para o **this** da chamada “with(**this**)”.

Estas funções estão descritas no item “**Funções de renderização**”, mais a frente.

O resultado da chamada é um objeto do tipo Vnode com todas as informações da template:

```
asyncFactory:undefined
asyncMeta:undefined
children:Array(3)
        0:VNode
                asyncFactory:undefined
                asyncMeta:undefined
                children:[VNode]
                componentInstance:undefined
                componentOptions:undefined
                context:Vue {_uid: 0, _isVue: true, $options: {…}, _renderProxy: Proxy,
                _self: Vue, …}
                data:
                        1.on:{click: ƒ}
                        2.__proto__:Object
                elm:undefined
                fnContext:undefined
                fnOptions:undefined
                fnScopeId:undefined
                isAsyncPlaceholder:false
                isCloned:false
                isComment:false
                isOnce:false
                isRootInsert:true
                isStatic:false
                key:undefined
                ns:undefined
                parent:undefined
                raw:false
                tag:"button"
                text:undefined
                child:(...)
                __proto__:Object
        1:VNode {tag: undefined, data: undefined, children: undefined, text: " ", elm:
        undefined, …}
        2:VNode
                asyncFactory:undefined
                asyncMeta:undefined
                children:Array(1)
                        0:VNode {tag: undefined, data: undefined, children: undefined, text:
                        "O evento Open tura vai acontecer em Espaço eventual", elm:
                        undefined, …}
                        length:1
                        __proto__:Array(0)
                componentInstance:undefined
                componentOptions:undefined
                context:Vue {_uid: 0, _isVue: true, $options: {…}, _renderProxy: Proxy,
                _self: Vue, …}
                data:undefined
                elm:undefined
                fnContext:undefined
```

```
                fnOptions:undefined
                fnScopeId:undefined
                isAsyncPlaceholder:false
                isCloned:false
                isComment:false
                isOnce:false
                isRootInsert:true
                isStatic:false
                key:undefined
                ns:undefined
                parent:undefined
                raw:false
                tag:"h2"
                text:undefined
                child:(...)
                __proto__:Object
        length:3
        __proto__:Array(0)
componentInstance:undefined
componentOptions:undefined
context:Vue {_uid: 0, _isVue: true, $options: {…}, _renderProxy: Proxy, _self: Vue, …}
data:{attrs: {…}}
elm:undefined
fnContext:undefined
fnOptions:undefined
fnScopeId:undefined
isAsyncPlaceholder:false
isCloned:false
isComment:false
isOnce:false
isRootInsert:true
isStatic:false
key:undefined
ns:undefined
parent:undefined
raw:false
tag:"div"
text:undefined
child:(...)
__proto__:Object
```

Funções de Renderização

Como visto anteriormente, o Vue transforma o modelo DOM em um modelo de funções. Citamos três delas: _c, _s e _v. Todas são definidas pela função **initRender**.

A função "**_c**" remete o Vue à função:

function (a, b, c, d) { return createElement(vm, a, b, c, d, false); }

E createElement se define da seguinte forma:

```
function createElement (
  context,
  tag,
  data,
  children,
  normalizationType,
  alwaysNormalize
) {
  if (Array.isArray(data) || isPrimitive(data)) {
    normalizationType = children;
    children = data;
    data = undefined;
  }
  if (isTrue(alwaysNormalize)) {
    normalizationType = ALWAYS_NORMALIZE;
  }
  return _createElement(context, tag, data, children, normalizationType)
}
```

Onde:

```
function _createElement (
  context,
  tag,
  data,
  children,
  normalizationType
) {
  if (isDef(data) && isDef((data).__ob__)) {
    "development" !== 'production' && warn(
      "Avoid using observed data object as vnode data: " + (JSON.stringify(data)) + "\n" +
      'Always create fresh vnode data objects in each render!',
      context
    );
    return createEmptyVNode()
  }
```

VUE - PROCESSO DE COMPILAÇÃO

Onde:

```
var createEmptyVNode = function (text) {
  if ( text === void 0 ) text = '';
  var node = new VNode();
  node.text = text;
  node.isComment = true;
  return node
};
```

A função “**_s**” remete o Vue à função toString:

```
function toString (val) {
  return val == null ? '': typeof val === 'object' ? JSON.stringify(val, null, 2) : String(val)
}
```

Esta não é a função **toString** do javascript, e sim do objeto Vue, pois é estabelecida dentro de seu contexto.

A função “**_v**” remete o Vue à função createTextVNode:

```
function createTextVNode (val) {
  return new VNode(undefined, undefined, undefined, String(val))
}
```

VUE - PROCESSO DE COMPILAÇÃO

Onde:

```
var VNode = function VNode ( tag, data, children, text, elm, context,
  componentOptions, asyncFactory) {
  this.tag = tag;
  this.data = data;
  this.children = children;
  this.text = text;
  this.elm = elm;
  this.ns = undefined;
  this.context = context;
  this.fnContext = undefined;
  this.fnOptions = undefined;
  this.fnScopeId = undefined;
  this.key = data && data.key;
  this.componentOptions = componentOptions;
  this.componentInstance = undefined;
  this.parent = undefined;
  this.raw = false;
  this.isStatic = false;
  this.isRootInsert = true;
  this.isComment = false;
  this.isCloned = false;
  this.isOnce = false;
  this.asyncFactory = asyncFactory;
  this.asyncMeta = undefined;
  this.isAsyncPlaceholder = false;
};
```

Disparando a ação de substituição de valores

Na instanciação do objeto Vue, é acionada a função displayparams, que altera os valores das variáveis **evento** e **local** do objeto.

Vejamos o que ocorre, com o auxílio do debug do navegador, colocando um breakpoint na linha posterior a “displayparams : function(event)”, ao clicar do botão “Click ME”.

O processamento para na linha “***this.evento = "Abertura"***”. O debug realça a palavra “this”. Neste momento você deve pressionar o botão de avanço “*skip into next function call*” e não o “*skip over*”.

Surpreendentemente, o processamento salta para a função “***proxySetter (val)***” que representa o método **set** definido pela função **proxy** que, como dissemos anteriormente, controla as variáveis Vue. Como esta é uma variável “reativa”, ela invoca o método **set** definido pela função **reactiveSetter**. Reproduzimos esta função a seguir:

```
function reactiveSetter (newVal) {
    var value = getter ? getter.call(obj) : val;
    /* eslint-disable no-self-compare */
    if (newVal === value || (newVal !== newVal && value !== value)) {
          return
          }
    /* eslint-enable no-self-compare */
    if ("development" !== 'production' && customSetter) {
          customSetter();
          }
    if (setter) {
          setter.call(obj, newVal);
          } else {
          val = newVal;
          }
    childOb = !shallow && observe(newVal);
    dep.notify();
  }
```

O valor da variável é ajustado por **setter.call** ou simplesmente pela atribuição **val = newVal**. Posteriormente, é invocada a função **observe:**

```
function observe (value, asRootData) {
  if (!isObject(value) || value instanceof VNode) {
    return
  }
  var ob;
  if (hasOwn(value, '__ob__') && value.__ob__ instanceof Observer) {
    ob = value.__ob__;
  } else if (
    shouldObserve &&
    !isServerRendering() &&
    (Array.isArray(value) || isPlainObject(value)) &&
    Object.isExtensible(value) &&
    !value._isVue
  ) {
    ob = new Observer(value);
  }
  if (asRootData && ob) {
    ob.vmCount++;
  }
  return ob
}
```

Como esta variável não é do tipo objeto, o retorno se dá imediatamente pelo primeiro **return**, sem retornar nenhum valor.

A função retorna para **proxySetter**, que conclui a linha "**this[sourceKey][key] = val**", onde **this** é a instância do objeto Vue, **sourceKey** é "**_data**" e key é "**evento**", ou seja, as variáveis Vue estão declaradas no próprio objeto Vue, no seu atributo "**_data**", como vemos na reprodução do conteúdo de Vue, abaixo:

```
_data:
        evento:"Abertura"
        local:"Espaço eventual"
        __ob__:Observer {value: {…}, dep: Dep, vmCount: 1}
        get evento:ƒ reactiveGetter()
        set evento:ƒ reactiveSetter(newVal)
        get local:ƒ reactiveGetter()
        set local:ƒ reactiveSetter(newVal)
        __proto__:Object
```

Observe que, realmente, os métodos de acesso às variáveis são **reactiveGetter** e **reactiveSetter**, como dissemos e constatamos pelo debug.

A seguir, o processamento retorna para **displayParams**, onde novamente o processamento realça a palavra **this**. Agora a variável **local** sofre a alteração, como ocorreu com **evento**. Na linha seguinte de reactiveSetter, encontramos a chamada **dep.notify()**. Esta função corresponde a um método do objeto **Dep**, que acessa os **Watchers** que monitoram dinamicamente as variáveis, chamando o seu método update:

```
Watcher.prototype.update = function update () {
        /* istanbul ignore else */
        if (this.lazy) {
                this.dirty = true;
        } else if (this.sync) {
                this.run();
                } else {
                queueWatcher(this);
                }
        };
```

Por que isto ocorre desta forma ? Porque alteramos os valores das variáveis nos **proxies**, mas a renderização com os novos valores precisa ser refeita. O mais importante nesta função é a chamada de **queueWatcher**:

```
function queueWatcher (watcher) {
	var id = watcher.id;
	if (has[id] == null) {
		has[id] = true;
		if (!flushing) {
			queue.push(watcher);
		} else {
		// if already flushing, splice the watcher based on its id
		// if already past its id, it will be run next immediately.
		var i = queue.length - 1;
		while (i > index && queue[i].id > watcher.id) {
			i--;
			}
		queue.splice(i + 1, 0, watcher);
		}
	// queue the flush
	if (!waiting) {
		waiting = true;
		nextTick(flushSchedulerQueue);
		}
	}
}
```

Vejamos o que esta função faz. Para nosso exemplo, ela desvia o processamento para o comando **queue.push(watcher)**. Vejamos a estrutura do parâmetro **watcher**:

```
active:true
cb:ƒ noop(a, b, c)
deep:false
depIds:Set(2) {3, 4}
deps:Array(2)
	0:Dep {id: 3, subs: Array(1)}
	1:Dep {id: 4, subs: Array(1)}
	length:2
	__proto__:Array(0)
dirty:false
expression:"function () { vm._update(vm._render(), hydrating); }"
getter:ƒ ()
id:1
lazy:false
newDepIds:Set(0) {}
newDeps:[]
sync:false
user:false
value:undefined
vm:Vue {_uid: 0,_isVue: true, $options: {…}, _renderProxy: Proxy, _self: Vue, …}
__proto__:Object
```

A chamada **queue.push** coloca o Watcher na pilha queue. A seguir é ativada uma flag de espera (waiting) de atualizações e emitido o comando de **nextTick**:

nextTick(flushSchedulerQueue)

cuja definição é:

```
function nextTick (cb, ctx) {
	var _resolve;
	callbacks.push(function () {
	if (cb) {
		try {
			cb.call(ctx);
		} catch (e) {
			handleError(e, ctx, 'nextTick');
			}
	} else if (_resolve) {
_		resolve(ctx);
		}
	});
	if (!pending) {
		pending = true;
		if (useMacroTask) {
			macroTimerFunc();
		} else {
			microTimerFunc();
		}
	}
	// $flow-disable-line
	if (!cb && typeof Promise !== 'undefined') {
		return new Promise(function (resolve) {
_			resolve = resolve;
			})
		}
	}
```

Logo de início é colocada uma função anônima no topo da pilha de callbacks:

```
function () {
	if (cb) {
		try {
			cb.call(ctx);
		} catch (e) {
			handleError(e, ctx, 'nextTick');
			}
	} else if (_resolve) {
_		resolve(ctx);
		}
	}
```

Se a pilha de callbacks estiver vazia, a variável **pending** estará em estado ***false***. Se isto acontecer ela é ativada em ***true***. Em seguida, em nosso caso, é acionada a função **macroTimerFunc** que ordena a utilização do objeto MessageChannel da API dos navegadores. Este tipo de manipulação de mensagem é usado, por exemplo, para enviar mensagens quando se tem uma ou mais IFRAMES em uma página HTML.

No retorno das atribuições, o modelo Vue faz com que dysplayparams retorne para o comando ***fns.apply(null, arguments)*** da função **createFnInvoker**. Isto foi definido durante a inicialização do objeto Vue. O processamento retorna à função withMacroTask pela qual passamos anteriormente em **nextTick** :

```
function withMacroTask (fn) {
  return fn._withTask || (fn._withTask = function () {
    useMacroTask = true;
    var res = fn.apply(null, arguments);
    useMacroTask = false;
    return res
  })
}
```

Agora useMacroTask retorna ao valor **false**. A função retorna para flushCallBacks, para processar a pilha de retorno de funções do Vue:

```
function flushCallbacks () {
        pending = false;
        var copies = callbacks.slice(0);
        callbacks.length = 0;
        for (var i = 0; i < copies.length; i++) {
                copies[i]();
                }
        }
```

Anteriormente colocamos uma função anônima na pilha de callbacks. É anulada a flag de pendências (**pending**). A função anônima, nossa conhecida, é chamada:

```
function () {
        if (cb) {
                try {
                        cb.call(ctx);
                } catch (e) {
                        handleError(e, ctx, 'nextTick');
                        }
        } else if (_resolve) {
    _           resolve(ctx);
                }
        }
```

VUE - PROCESSO DE COMPILAÇÃO

O objeto **cb** é uma função (flushSchedulerQueue).

```
function flushSchedulerQueue () {
  flushing = true;
  var watcher, id;

  // Sort queue before flush.
  // This ensures that:
  // 1. Components are updated from parent to child. (because parent is always
  //    created before the child)
  // 2. A component's user watchers are run before its render watcher (because
  //    user watchers are created before the render watcher)
  // 3. If a component is destroyed during a parent component's watcher run,
  //    its watchers can be skipped.
  queue.sort(function (a, b) { return a.id - b.id; });

  // do not cache length because more watchers might be pushed
  // as we run existing watchers
  for (index = 0; index < queue.length; index++) {
    watcher = queue[index];
    id = watcher.id;
    has[id] = null;
    watcher.run();
    // in dev build, check and stop circular updates.
    if ("development" !== 'production' && has[id] != null) {
      circular[id] = (circular[id] || 0) + 1;
      if (circular[id] > MAX_UPDATE_COUNT) {
        warn(
          'You may have an infinite update loop ' + (
            watcher.user
              ? ("in watcher with expression \"" + (watcher.expression) + "\"")
              : "in a component render function."
          ),
          watcher.vm
        );
        break
      }
    }
  }

  // keep copies of post queues before resetting state
  var activatedQueue = activatedChildren.slice();
  var updatedQueue = queue.slice();

  resetSchedulerState();

  // call component updated and activated hooks
  callActivatedHooks(activatedQueue);
  callUpdatedHooks(updatedQueue);

  // devtool hook
  /* istanbul ignore if */
  if (devtools && config.devtools) {
    devtools.emit('flush');
  }
}
```

VUE - PROCESSO DE COMPILAÇÃO

Versão 1.1 – 31/07/2018 – Bernardo Meyer

A linha a analisar, para nossos propósitos, é:

watcher.run():

```
Watcher.prototype.run = function run () {
  if (this.active) {
    var value = this.get();
    if (
      value !== this.value ||
      // Deep watchers and watchers on Object/Arrays should fire even
      // when the value is the same, because the value may
      // have mutated.
      isObject(value) ||
      this.deep
    ) {
      // set new value
      var oldValue = this.value;
      this.value = value;
      if (this.user) {
        try {
          this.cb.call(this.vm, value, oldValue);
        } catch (e) {
          handleError(e, this.vm, ("callback for watcher \"" + (this.expression) + "\""));
        }
      } else {
        this.cb.call(this.vm, value, oldValue);
      }
    }
  }
};
```

É averiguado se o **Watcher** está ativo (**this**). Em nosso caso, está, e é chamado o método **get** de **Watcher**:

```
Watcher.prototype.get = function get () {
  pushTarget(this);
  var value;
  var vm = this.vm;
  try {
    value = this.getter.call(vm, vm);
  } catch (e) {
    if (this.user) {
      handleError(e, vm, ("getter for watcher \"" + (this.expression) + "\""));
    } else {
      throw e
    }
  } finally {
    // "touch" every property so they are all tracked as
    // dependencies for deep watching
    if (this.deep) {
      traverse(value);
    }
    popTarget();
    this.cleanupDeps();
  }
  return value
};
```

Este método de watcher chama a sua função ***getter***. A função getter de Watcher é **updateComponent**:

```
updateComponent = function () {
      vm._update(vm._render(), hydrating);
    };
```

VUE - PROCESSO DE COMPILAÇÃO

Esta usa o método **_render** de Vue como parâmetro, que é, portanto, executada:

```
Vue.prototype._render = function () {
    var vm = this;
    var ref = vm.$options;
    var render = ref.render;
    var _parentVnode = ref._parentVnode;

    // reset _rendered flag on slots for duplicate slot check
    {
      for (var key in vm.$slots) {
        // $flow-disable-line
        vm.$slots[key]._rendered = false;
      }
    }

    if (_parentVnode) {
      vm.$scopedSlots = _parentVnode.data.scopedSlots || emptyObject;
    }

    // set parent vnode. this allows render functions to have access
    // to the data on the placeholder node.
    vm.$vnode = _parentVnode;
    // render self
    var vnode;
    try {
      vnode = render.call(vm._renderProxy, vm.$createElement);
    } catch (e) {
      handleError(e, vm, "render");
      // return error render result,
      // or previous vnode to prevent render error causing blank component
      /* istanbul ignore else */
      {
        if (vm.$options.renderError) {
          try {
            vnode = vm.$options.renderError.call(vm._renderProxy, vm.$createElement, e);
          } catch (e) {
            handleError(e, vm, "renderError");
            vnode = vm._vnode;
          }
        } else {
          vnode = vm._vnode;
        }
      }
    }
    // return empty vnode in case the render function errored out
    if (!(vnode instanceof VNode)) {
      if ("development" !== 'production' && Array.isArray(vnode)) {
        warn(
          'Multiple root nodes returned from render function. Render function ' +
          'should return a single root node.',
          vm
        );
      }
      vnode = createEmptyVNode();
    }
    // set parent
    vnode.parent = _parentVnode;
    return vnode
  };
}
```

O processamento acaba recaindo sobre a linha marcada. Era neste ponto que queríamos chegar. A função **_renderProxy** a executar é o produto da compilação do HTML em modelo Ast, como já vimos anteriormente:

```
(function(){
with(this){return _c('div',{attrs:{"id":"app"}},[_c('button',{on:{"click":displayparams}},[_v("Click ME")]),_v(" "),_c('h2',[_v("O evento "+_s(evento)+" vai acontecer em "+_s(local))])])}
})
```

O processamento retorna para a chamada do **_update,** que invoca o método _update do Vue, declarado na função **lifecycleMixin**:

```
Vue.prototype._update = function (vnode, hydrating) {
  var vm = this;
  if (vm._isMounted) {
    callHook(vm, 'beforeUpdate');
  }
  var prevEl = vm.$el;
  var prevVnode = vm._vnode;
  var prevActiveInstance = activeInstance;
  activeInstance = vm;
  vm._vnode = vnode;
  // Vue.prototype.__patch__ is injected in entry points
  // based on the rendering backend used.
  if (!prevVnode) {
    // initial render
    vm.$el = vm.__patch__(
      vm.$el, vnode, hydrating, false /* removeOnly */,
      vm.$options._parentElm,
      vm.$options._refElm
    );
    // no need for the ref nodes after initial patch
    // this prevents keeping a detached DOM tree in memory (#5851)
    vm.$options._parentElm = vm.$options._refElm = null;
  } else {
    // updates
    vm.$el = vm.__patch__(prevVnode, vnode);
  }
  activeInstance = prevActiveInstance;
  // update __vue__ reference
  if (prevEl) {
    prevEl.__vue__ = null;
  }
  if (vm.$el) {
    vm.$el.__vue__ = vm;
  }
  // if parent is an HOC, update its $el as well
  if (vm.$vnode && vm.$parent && vm.$vnode === vm.$parent._vnode) {
    vm.$parent.$el = vm.$el;
  }
  // updated hook is called by the scheduler to ensure that children are
  // updated in a parent's updated hook.
};
```

Logo nas primeiras instruções é chamado o “***hook***” **beforeUpdate**. Prosseguindo chegamos à linha marcada:

vm.$el = vm.__patch__(prevVnode, vnode)

que chama a função **patchVnode**. E esta função chama **updateChildren**.

A função **updateChildren** percorre um a um os nodos da estrutura em árvore armazenada (**oldCh**) para comparar com os nodos da estrutura gerada pela renderização (**newCh**). A função encontra o local correto para colocar a estrutura, pois ela pode ter sido deslocada em função de movimentos dos nodos. Cada nó filho é atualizado (patched) pela função **patchVnode**.

VUE - PROCESSO DE COMPILAÇÃO

Versão 1.1 – 31/07/2018 – Bernardo Meyer