Vue学习笔记

\_init

This.$options.data = mergeInstanceDataFn(){}

# Vue.prototype.\_callHook

\_callbackHook(‘init’)：

1. vue.$emit(‘pre-hook:init’)
2. 调用options.init，options.init是一个数组 <?>
3. Vue.$emit(‘hook:init’)

# \_initProps

dom.nodeType = 1 => Node.ELEMENT\_NODE

存在el且是元素节点才处理

# \_initMeta

# \_initData

Vue.\_data = this.$options.data? fn() : {} <2种情况>

Vue.\_runtimeData = this.$options.data?fn(): null

$options.\_data 和 $options.\_props 的key不能相同，可在props里配置default vaule <?>

->vue.prototype.\_proxy

# Vue.prototpye.\_proxy(key)

保留的属性（$, \_）不使用defineProperty包装，使用defineProperty代理 vm.key 的操作到 vm.\_data.key。不能直接在vm上访问自己定义的保留的属性options.\_inner\_var，可以通过vm.\_data.\_inner\_var访问，而且vm.\_data.\_inner\_var生成了getter和setter（在observe方法生成）

->observe(vm.\_data, vm)

# observe(value, vm)

**Object.isExtensible()** 是否可以添加属性。Object.prevetExtensions 只能阻止自己添加属性，不能阻止原型链上的修改，即其原型链上仍可添加属性

Value.\_\_ob\_\_ = value.\_\_ob\_\_ || new Observer(value)，每个Observer生成一个Dep

Value.\_\_ob\_\_.addVm(vm) -> value.\_\_ob\_\_.vms.push(vm)

Return value.\_\_ob\_\_

# Vue.prototype.\_initComputed

# Vue.prototype.$mount(el)

el = el || document.createElement(‘div’)

-> vue.prototype.\_compile(el)

# Vue.prototype.\_compile(el)

Original = el

->transclude(el, this.$options)

If(el有v-pre属性) return

rootLinker->compileRoot(el, options, contextOptions)

rootUnlinkFn = rootLinker(this, el, this.\_scope)

contentUnlinkFn = contentLinkFn? contentLinkFn(this, el): compile(el, optins)(this, el) 不是组件compoment则调用compile方法，编译el的内容

contentUnlinkFn值为调用compile方法返回的compositeLinkFn(this, el)的值，然后调用compositeLinkFn

在compositeLinkFn方法内会依次调用之前编译产生的linkFn

# transclude(el, options)

根据el的类型，处理并返回el

options.\_containerAttrs = extractAttrs(el) 获取el的属性数组

如果是template则生成el

如果有options.template，进行处理<?>

如果是fragement，生成v-start标记

返回el

# compileRoot(el, options, contextOpotins)

options.\_replaceAttrs

options.\_asCompoment

replacerLinkFn = compileDirectives(el.attributes, options) 该函数在rootLinkFn内通过linkAndCapture调用

Return rootLinkFn(vm, el, scope) {} 返回的方法会在compile返回的compositeLinkFn内调用

# RootLinkFn(vm, el, scope)

contextDirs = linkAndCapture (contextLinkFn, vm.\_context) link content scope dirs，组件才有

selfDires = linkAndCapture(replacerLinkFn, vm) link self，linkAndCapture方法调用传入的linker，并返回调用linker生成的指令dirs

return makeUnlinkFn(vm, selfDirs, context, contextDirs)

# madkUnlinkFn(vm, dirs., context, contextDirs)

teardownDirs(vm, dirs., destroying) 函数内部调用dirs[i].teardown() 删除指令，dirs为调用linker生成的指令

# compositeLinkFn(vm, el, host, scope, frag)

dirs = linkAndCapture() 调用makeChildLinkFn里的linkFns

<?>

# linkAndCapture(linker, vm)

apply alinker to a vm?element pair and capture the directives created during the process

dirs. = vm.\_directives.slice(originalDirCount)

for:

dirs.[i].\_bind()

return dirs

调用每个指定的\_bind方法，即调用指定义的 bind 方法

普通的{{var}}对应text指令

Derictive.\_bind()

/\*\*

\* Initialize the directive, mixin definition properties,

\* setup the watcher, call definition bind() and update()

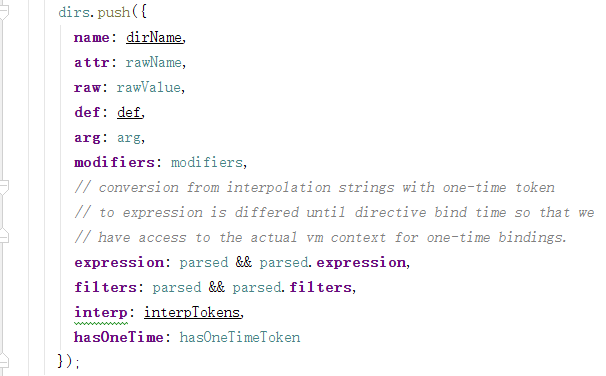
\* if present.

\*/

# compileDirectives(attrs, options)

遍历每个属性，对属性进行处理。对各种v属性进行处理。

根据属性生成指令dirs[]，每个dirs元素都是一个指令信息对象。指令根据属性名从options.derictives里找。



While:

Tokens = parseText(value)

无各种v属性，options.\_containerAttrs = options.\_replacerAttrs = null

If(dirs.length) {  
 return makeNodeLinkFn(dirs)

}

# Compile(el, options, partial)

nodeLinkFn = compileNode(el, options) 如果 是partial || !options.\_asCompoent

childLinkFn = compileNodeList(el.childNodes, options)

return compositeLinkFn(vm, el, host, scope, frag)

# compileNode(node, options)

nodeType=1 && !script => return compileElement(node, options) 元素节点而且不是script

nodeTpe = 3 => compileTextNode(node, options) 文本节点

# compileElement(node, options)

处理各种值的绑定，解析指令

Normal direcrtives:

linkFn = compileDirectives(attrs, options)

return linkFn

# compileNodeList(nodeList, options)

*Compile a node list and return a childLinkFn*

linkFns = []

for nodeList:

nodeLinkFn = compileNode(node, options)

childLinkFn = compileNodeList(node.childNodes, options)

return linkFns.length? makeChildLinkFn(linkFns): null

在makeChildLinkFn返回的方法里调用linkFns

# makeChildLinkFn(linkFns)

*Make a child link function for a node's childNodes*

# compileTextNode(node, options)

*Compile a textNode and return a nodeLinkFn*

tokens = parseText(node.wholeText) 解析成指令？

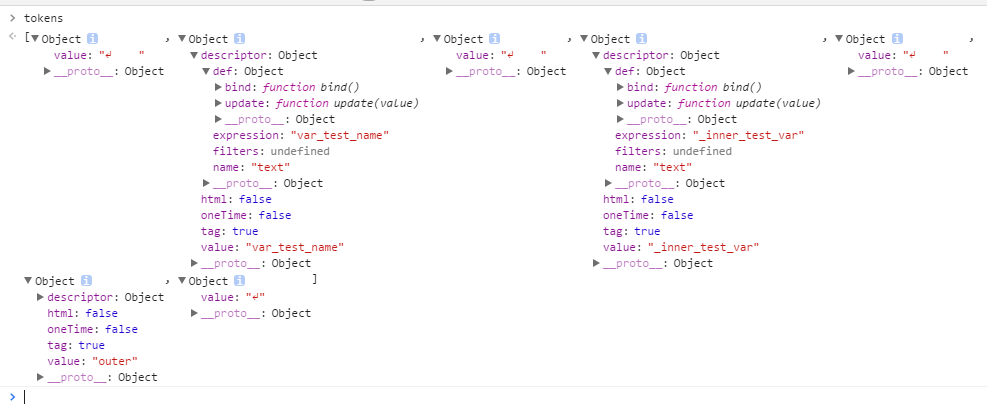
"

{{ var\_test\_name }}

{{ \_inner\_test\_var }}

{{ outer }}

"



Frag = document.createDocumentFragement()

For tokens:

El = token.tag? processTextToken(token, options): document.createTextNode(token.value)

Frag.appendChild(el)

对每个{{var}}内的表达式进行处理 或 创建文本节点

Token.value 为 vm.data中对就的值（表达式），空文本节点就是’’，

{{var}}解析出的token的tag为true

Return makeTextNodeLinkFn(tokens, frag, options)

# makeTextNodeLinkFn(tokens, frag)

*Build a function that processes a textNode*

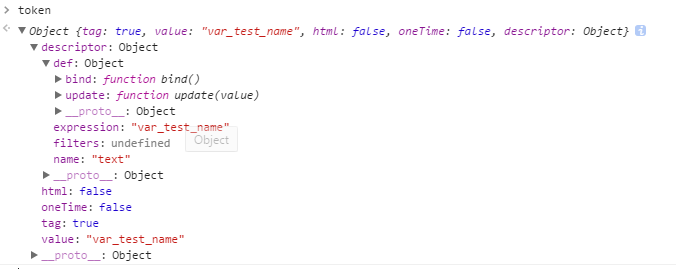
return textNodeLinkFn(vm, el, host, scope)

# textNodeLinkFn(vm, el, host, scope)

for:

->vm.\_bindDir(token.descriptor, node, host, scope)

Replace(el, fragClone)



# Vm.prototype.\_bindDir(descriptor, node, host, scope, frag)

this.\_directives.push(new Directive(descriptor, this, node, host, scope, frag))

<?>

# ProcessTextToken(token, options)

!token.html:

el = document.createTextNode(‘ ‘) 不是html，就创建文本节点

setTokenType(‘text’)

return el

# Observer.prototype.walk(obj)

对每个属性调用convert，在defineReactive中为每个属性生成一个Dep，

->Observer.prototype.convert(key, obj[key])

# Observer.prototype.convert(key, value)

->defineReactive(this.value, key, value)，this.value为new Observer传入的对象，key为其中的一个属性

# defineReactive(obj, key, val)

为每个属性生成一个Dep，定义getter: reativeGetter，setter: reactiveSetter

\_、$开头的保留属性也像其它属性一样处理

->observe(val)