## **createElement**

# **createElement**

Vue.js 利用 createElement 方法创建 VNode，它定义在 src/core/vdom/create-elemenet.js 中：

// wrapper function for providing a more flexible interface

// without getting yelled at by flow

export function createElement (

context: Component,

tag: any,

data: any,

children: any,

normalizationType: any,

alwaysNormalize: boolean

): VNode | Array<VNode> {

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)

}

createElement 方法实际上是对 \_createElement 方法的封装，它允许传入的参数更加灵活，在处理这些参数后，调用真正创建 VNode 的函数 \_createElement：

export function \_createElement (

context: Component,

tag?: string | Class<Component> | Function | Object,

data?: VNodeData,

children?: any,

normalizationType?: number

): VNode | Array<VNode> {

if (isDef(data) && isDef((data: any).\_\_ob\_\_)) {

process.env.NODE\_ENV !== '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()

}

// object syntax in v-bind

if (isDef(data) && isDef(data.is)) {

tag = data.is

}

if (!tag) {

// in case of component :is set to falsy value

return createEmptyVNode()

}

// warn against non-primitive key

if (process.env.NODE\_ENV !== 'production' &&

isDef(data) && isDef(data.key) && !isPrimitive(data.key)

) {

if (!\_\_WEEX\_\_ || !('@binding' in data.key)) {

warn(

'Avoid using non-primitive value as key, ' +

'use string/number value instead.',

context

)

}

}

// support single function children as default scoped slot

if (Array.isArray(children) &&

typeof children[0] === 'function'

) {

data = data || {}

data.scopedSlots = { default: children[0] }

children.length = 0

}

if (normalizationType === ALWAYS\_NORMALIZE) {

children = normalizeChildren(children)

} else if (normalizationType === SIMPLE\_NORMALIZE) {

children = simpleNormalizeChildren(children)

}

let vnode, ns

if (typeof tag === 'string') {

let Ctor

ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)

if (config.isReservedTag(tag)) {

// platform built-in elements

vnode = new VNode(

config.parsePlatformTagName(tag), data, children,

undefined, undefined, context

)

} else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {

// component

vnode = createComponent(Ctor, data, context, children, tag)

} else {

// unknown or unlisted namespaced elements

// check at runtime because it may get assigned a namespace when its

// parent normalizes children

vnode = new VNode(

tag, data, children,

undefined, undefined, context

)

}

} else {

// direct component options / constructor

vnode = createComponent(tag, data, context, children)

}

if (Array.isArray(vnode)) {

return vnode

} else if (isDef(vnode)) {

if (isDef(ns)) applyNS(vnode, ns)

if (isDef(data)) registerDeepBindings(data)

return vnode

} else {

return createEmptyVNode()

}

}

\_createElement 方法有 5 个参数，context 表示 VNode 的上下文环境，它是 Component 类型；tag 表示标签，它可以是一个字符串，也可以是一个 Component；data 表示 VNode 的数据，它是一个 VNodeData 类型，可以在 flow/vnode.js 中找到它的定义，这里先不展开说；children 表示当前 VNode 的子节点，它是任意类型的，它接下来需要被规范为标准的 VNode 数组；normalizationType 表示子节点规范的类型，类型不同规范的方法也就不一样，它主要是参考 render 函数是编译生成的还是用户手写的。

createElement 函数的流程略微有点多，我们接下来主要分析 2 个重点的流程 —— children 的规范化以及 VNode 的创建。

## **children 的规范化**

由于 Virtual DOM 实际上是一个树状结构，每一个 VNode 可能会有若干个子节点，这些子节点应该也是 VNode 的类型。\_createElement 接收的第 4 个参数 children 是任意类型的，因此我们需要把它们规范成 VNode 类型。

这里根据 normalizationType 的不同，调用了 normalizeChildren(children) 和 simpleNormalizeChildren(children) 方法，它们的定义都在 src/core/vdom/helpers/normalzie-children.js 中：

// The template compiler attempts to minimize the need for normalization by

// statically analyzing the template at compile time.

//

// For plain HTML markup, normalization can be completely skipped because the

// generated render function is guaranteed to return Array<VNode>. There are

// two cases where extra normalization is needed:

// 1. When the children contains components - because a functional component

// may return an Array instead of a single root. In this case, just a simple

// normalization is needed - if any child is an Array, we flatten the whole

// thing with Array.prototype.concat. It is guaranteed to be only 1-level deep

// because functional components already normalize their own children.

export function simpleNormalizeChildren (children: any) {

for (let i = 0; i < children.length; i++) {

if (Array.isArray(children[i])) {

return Array.prototype.concat.apply([], children)

}

}

return children

}

// 2. When the children contains constructs that always generated nested Arrays,

// e.g. <template>, <slot>, v-for, or when the children is provided by user

// with hand-written render functions / JSX. In such cases a full normalization

// is needed to cater to all possible types of children values.

export function normalizeChildren (children: any): ?Array<VNode> {

return isPrimitive(children)

? [createTextVNode(children)]

: Array.isArray(children)

? normalizeArrayChildren(children)

: undefined

}

simpleNormalizeChildren 方法调用场景是 render 函数当函数是编译生成的。理论上编译生成的 children 都已经是 VNode 类型的，但这里有一个例外，就是 functional component 函数式组件返回的是一个数组而不是一个根节点，所以会通过 Array.prototype.concat 方法把整个 children 数组打平，让它的深度只有一层。

normalizeChildren 方法的调用场景有 2 种，一个场景是 render 函数是用户手写的，当 children 只有一个节点的时候，Vue.js 从接口层面允许用户把 children 写成基础类型用来创建单个简单的文本节点，这种情况会调用 createTextVNode 创建一个文本节点的 VNode；另一个场景是当编译 slot、v-for 的时候会产生嵌套数组的情况，会调用 normalizeArrayChildren 方法，接下来看一下它的实现：

function normalizeArrayChildren (children: any, nestedIndex?: string): Array<VNode> {

const res = []

let i, c, lastIndex, last

for (i = 0; i < children.length; i++) {

c = children[i]

if (isUndef(c) || typeof c === 'boolean') continue

lastIndex = res.length - 1

last = res[lastIndex]

// nested

if (Array.isArray(c)) {

if (c.length > 0) {

c = normalizeArrayChildren(c, `${nestedIndex || ''}\_${i}`)

// merge adjacent text nodes

if (isTextNode(c[0]) && isTextNode(last)) {

res[lastIndex] = createTextVNode(last.text + (c[0]: any).text)

c.shift()

}

res.push.apply(res, c)

}

} else if (isPrimitive(c)) {

if (isTextNode(last)) {

// merge adjacent text nodes

// this is necessary for SSR hydration because text nodes are

// essentially merged when rendered to HTML strings

res[lastIndex] = createTextVNode(last.text + c)

} else if (c !== '') {

// convert primitive to vnode

res.push(createTextVNode(c))

}

} else {

if (isTextNode(c) && isTextNode(last)) {

// merge adjacent text nodes

res[lastIndex] = createTextVNode(last.text + c.text)

} else {

// default key for nested array children (likely generated by v-for)

if (isTrue(children.\_isVList) &&

isDef(c.tag) &&

isUndef(c.key) &&

isDef(nestedIndex)) {

c.key = `\_\_vlist${nestedIndex}\_${i}\_\_`

}

res.push(c)

}

}

}

return res

}

normalizeArrayChildren 接收 2 个参数，children 表示要规范的子节点，nestedIndex 表示嵌套的索引，因为单个 child 可能是一个数组类型。 normalizeArrayChildren 主要的逻辑就是遍历 children，获得单个节点 c，然后对 c 的类型判断，如果是一个数组类型，则递归调用 normalizeArrayChildren; 如果是基础类型，则通过 createTextVNode 方法转换成 VNode 类型；否则就已经是 VNode 类型了，如果 children 是一个列表并且列表还存在嵌套的情况，则根据 nestedIndex 去更新它的 key。这里需要注意一点，在遍历的过程中，对这 3 种情况都做了如下处理：如果存在两个连续的 text节点，会把它们合并成一个 text 节点。

经过对 children 的规范化，children 变成了一个类型为 VNode 的 Array。

## **VNode 的创建**

回到 createElement 函数，规范化 children 后，接下来会去创建一个 VNode 的实例：

let vnode, ns

if (typeof tag === 'string') {

let Ctor

ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)

if (config.isReservedTag(tag)) {

// platform built-in elements

vnode = new VNode(

config.parsePlatformTagName(tag), data, children,

undefined, undefined, context

)

} else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {

// component

vnode = createComponent(Ctor, data, context, children, tag)

} else {

// unknown or unlisted namespaced elements

// check at runtime because it may get assigned a namespace when its

// parent normalizes children

vnode = new VNode(

tag, data, children,

undefined, undefined, context

)

}

} else {

// direct component options / constructor

vnode = createComponent(tag, data, context, children)

}

这里先对 tag 做判断，如果是 string 类型，则接着判断如果是内置的一些节点，则直接创建一个普通 VNode，如果是为已注册的组件名，则通过 createComponent 创建一个组件类型的 VNode，否则创建一个未知的标签的 VNode。 如果是 tag 一个 Component 类型，则直接调用 createComponent 创建一个组件类型的 VNode 节点。对于 createComponent 创建组件类型的 VNode 的过程，我们之后会去介绍，本质上它还是返回了一个 VNode。

## **总结**

那么至此，我们大致了解了 createElement 创建 VNode 的过程，每个 VNode 有 children，children 每个元素也是一个 VNode，这样就形成了一个 VNode Tree，它很好的描述了我们的 DOM Tree。

回到 mountComponent 函数的过程，我们已经知道 vm.\_render 是如何创建了一个 VNode，接下来就是要把这个 VNode 渲染成一个真实的 DOM 并渲染出来，这个过程是通过 vm.\_update 完成的，接下来分析一下这个过程。

****任务****

请仔细阅读本文档，为下面学习视频内容做准备。